

PYTHON PROGRAMMING

LECTURE 8: OBJECT-ORIENTED PROGRAMMING

goorm

**KAIST AI**  
Graduate School of AI

 **DAVIAN**

# Object-Oriented Programming

## 수강 신청 프로그래밍을 만들어보자!

- 방법 1: 절차 지향 프로그래밍
  - 교수용 프로그램과 학생용 프로그램을 따로 만듦
  - 수강 신청과정을 처음부터 끝까지 작성
- 방법 2: 객체 지향 프로그래밍
  - 수강신청을 하는 주체들(학생, 교수)의 행동과 데이터를 정의
  - 각 객체들을 엮어 유기적인 수강신청 API 작성

# OOP Example

## 학생의 객체를 만들어 보자!

### Class (설계도)

- Attribute (데이터, 속성)
  - 이름
  - 학번
  - 현재 수강중인 강의
- Method (행동)
  - 특정 강의를 수강하기
  - 특정 강의를 드롭하기

### Instance (객체)

- Attribute (데이터, 속성)
  - 조호준
  - 20194547
  - CS241
- Method (행동)
  - 특정 강의를 수강하기
  - 특정 강의를 드롭하기

## Class Example

### 학생의 틀(class)을 만들어 보자

```
class Student(object):  
    SCHOOL = 'KAIST'                                # 클래스 속성 (Class attribute)  
  
    def __init__(self, name: str, sid: int):          # 생성자  
        self.name = name                            # 속성 (Attribute)  
        self.sid = sid  
        self.classes = set()  
  
    # 클래스 함수 (Method)  
    def take_class(self, class_name: str) -> None:  
        self.classes.add(class_name)  
  
    def drop_class(self, class_name: str) -> None:  
        self.classes.discard(class_name)
```

## Class Example

### 학생 객체(Instance)를 만들고 써보자

```
# 클래스 생성
hojun_cho = Student('Hojun Cho', 20194547)

# 속성 출력
print(hojun_cho.name, "in", Student.SCHOOL)

# 메소드 실행
hojun_cho.take_class("CS101")
hojun_cho.take_class("CS202")
hojun_cho.drop_class("CS101")
```

## Class Declaration

### Class 선언부

```
class Student(object):
```

예약어

클래스  
이름

부모  
클래스

- 클래스 이름은 CamelCase가 관습적으로 사용됨
- 부모 클래스가 지정되지 않았을 시 object가 자동 상속 (python3)

## Class Attribute

### 클래스 속성

```
class Student(object):  
    SCHOOL = 'KAIST'          # 클래스 속성 (Class attribute)
```

- 클래스 전체가 공유하는 속성 값
- 모든 객체 (instance)가 같은 값을 참조
- 남용하면 스파게티 코드의 원인이 됨
- Class.attr 형태로 접근

```
# 속성 출력  
print(Student.SCHOOL)
```

# Method

## 클래스 함수 (Method)

```
class Student:
    # 클래스 함수 (Method)
    def take_class(self, class_name: str) -> None:
        self.classes.add(class_name)
```

- 각 객체에 적용이 가능함 함수
- 현재 수정하고자 하는 객체를 "self"로 지칭 (관습)
  - C와 Java에서의 "this"
  - 파이썬은 "self"를 첫번째 파라미터로 명시적으로 받음
- Class.method(instance, args, ...) 혹은 instance.method(args, ...)

```
# 메소드 실행
hojun_cho.take_class("CS101")
```



## Class Attribute

### 객체 속성

```
class Student(object):  
    def __init__(self, name: str, sid: int):           # 생성자  
        self.name = name                             # 속성 (Attribute)
```

- 각각의 객체가 개인적으로 가지는 값
- instance.attr 의 형태로 접근
- class 형태로 선언되어 나온 객체는 언제 어디서든 attribute 수정 가능

```
hojun_cho = Student('Hojun Cho', 20194547)  
hojun_cho.value = 10      # 존재하지 않는 속성에 값 부여가 가능  
print(hojun_cho.value)    # 그러나 권장하지 않으므로 가능하면 생성자에서 설정
```

## Magic Method: Initializer

- 메소드 이름이 "`__METHOD__`" 형태일 경우 특별한 Magic Method  
**생성자 (`__init__`)**

```
class Student:
    def __init__(self, name: str, sid: int):          # 생성자
        self.name = name                            # 속성 (Attribute)
        self.sid = sid
        self.classes = set()
```

- 객체를 생성할 때 호출됨
- 일반적으로 객체의 속성을 초기화 하는 데 사용
- `Class(args, ...)` 형태로 호출하여 객체 생성
- 거의 유일하게 정해진 Argument format이 없는 Magic Method

```
# 클래스 생성
hojun_cho = Student('Hojun Cho', 20194547)
```

## Magic Method: Destroyer

### 소멸자 (\_\_del\_\_)

```
class Student:
    def __del__(self):                # 소멸자
        self.classes.clear()
```

- 객체를 소멸할 때 호출됨
- 파이썬은 쓰레기 수거(Garbage Collection)로 메모리 관리
  - 객체가 어디에서도 참조되지 않을 때 객체가 소멸
  - 소멸 타이밍을 잡기 어려워 잘 사용되지 않음
- del 명령어
  - 변수 이름을 명시적으로 없애기 가능
  - 참조를 명시적으로 삭제하는 것이지 객체를 명시적으로 삭제하는 것이 아님

```
del hojun_cho    # 명시적으로 이름을 지우더라도 GC되지 않으면 사라지지 않는다!
```

### Indexing 메소드 (\_\_getitem\_\_, \_\_setitem\_\_)

```
class DoubleMapper(object):
    def __init__(self):
        self.mapping = {}

    def __getitem__(self, index):          # Indexing get
        return self.mapping.get(index, index * 2)

    def __setitem__(self, index, item):    # Indexing set
        self.mapping[index] = item

mapper = DoubleMapper()
print(mapper[10], mapper[1, 2])
mapper[10] = 15
print(mapper[10], mapper[1, 2])
```

- [ ] indexing을 재정의

## Magic Method: Length

### Length 메소드 (\_\_len\_\_)

```
class Dataset:
    def __init__(self, data, times=3):
        self.data = data
        self.times = times

    def __len__(self):
        # len(instance) 호출될 시 호출
        return len(self.data) * self.times

    def __getitem__(self, index):
        if index > len(self):
            raise IndexError()

        return self.data[index % len(self.data)]

dataset = Dataset([10, 2, 5, 2], times=5)
print(len(dataset))
```

## Magic Method: Typing

```
Class Student:
    def __init__(self, name: str, sid: int):
        self.name = name
        self.sid = sid

    def __str__(self):                # str 형변환
        return self.name + '_' + str(self.sid)

hojun_cho = Student("Hojun Cho", 20194547)
print(hojun_cho)                    # str 형태 출력
```

- 객체를 다른 타입으로 형 변환할때 호출
- 이외에도 \_\_int\_\_, \_\_float\_\_, \_\_bool\_\_ 등이 존재

## Magic Method: Comparison Operator

```
class Student:
    def __init__(self, name: str, sid: int):
        self.name = name
        self.sid = sid

    def __add__(self, other):
        return self.sid < other.sid # + 연산자를 재정의

students = [
    Student("Cho", 1234),
    Student("Ho", 4566),
    Student("Jun", 2267)
]

print(*[student.name for student in sorted(students)]) # Sorted 사용 가능
```

- $A < B$ 를 호출  $\rightarrow A.__lt__(B)$ 를 호출
- 이외에도 `__le__`, `__gt__`, `__ge__`, `__eq__`, `__ne__` 가 존재

## Magic Method: Arithmetic Operator

```
class MyComplex:
    def __init__(self, real, imaginary):
        self.real = real
        self.imaginary = imaginary

    def __str__(self):
        return str(self.real) + '+' + str(self.imaginary) + 'j'

    def __add__(self, other):
        # + 연산자 재정의
        # Out-place 연산
        return MyComplex(
            self.real + other.real,
            self.imaginary + other.imaginary
        )

a = MyComplex(3, -5)
b = MyComplex(-6, 7)
print(a + b)
```

- 이외에도 `__sub__`, `__mul__` 등이 존재
- In-place 버전인 `__iadd__` 가 존재 (이 경우 `self`를 직접 수정 필요)



## Magic Method: Callable

### 함수화 메소드 (\_\_call\_\_)

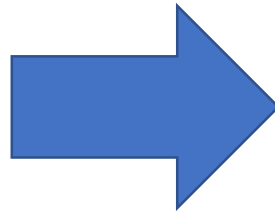
```
class AdditionNumber(object):  
    def __init__(self, number: int):           # 생성자  
        self.number = number  
  
    def __call__(self, number: int):           # 함수화 메소드  
        return number + self.number  
  
addition_5 = AdditionNumber(5)  
print(addition_5(10))                         # 객체를 함수처럼 사용
```

- 생성된 객체를 호출 가능하게 만듦
- instance(args, ...)가 instance.\_\_call\_\_(args, ...)를 호출

## Magic Method: Iterable

### 실제 for 문에서 일어나는 일은?

```
seq = [1, 2, 3, 4, 5]
for elem in seq:
    print(elem)
```



```
seq = list([1, 2, 3, 4, 5])
iterable = iter(seq)
while True:
    try:
        elem = next(iterable)
    except StopIteration:
        break
    print(elem)
```

### 실제 for 문에서 일어나는 일은?

```
seq = list([1, 2, 3, 4, 5])  
  
iterable = iter(seq)  
while True:  
    try:  
        elem = next(iterable)  
    except StopIteration:  
        break  
    print(elem)
```

- iter 내장함수
  - 해당 객체의 순환자 반환
  - `__iter__` 호출
- next 내장함수
  - 해당 순환자를 진행
  - `__next__` 호출
- 끝에서 StopIteration Exception

- Generator는 자동으로 `__iter__`와 `__next__`가 구현

# Magic Method: Context Manager

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

    def __enter__(self):
        self.classes = set()
        return self

    def __exit__(self, exc_type, exc_value, trace):
        self.classes.clear()

hojun_cho = Student("Hojun Cho", 20194547)
with hojun_cho:
    hojun_cho.classes.add('CS201')

with Student("Hojun Cho", 20194547) as hojun_cho:
    hojun_cho.classes.add('CS201')
```

- 소멸자 대용으로 특정 Block 입장/종료 시 자동으로 호출
- File description 등을 자동으로 닫고자 할 때 사용

# Property

```
class Circle(object):
    PI = 3.141592
    def __init__(self, raidus=3.):
        self.radius = raidus

    def get_area(self):
        return Circle.PI * self.radius ** 2

    def set_area(self, value):
        self.radius = (value / Circle.PI) ** .5

circle = Circle(5.)
print(circle.get_area())
circle.set_area(10)

print(circle.radius)
```



```
class Circle(object):
    PI = 3.141592
    def __init__(self, raidus=3.):
        self.radius = raidus

    @property
    def area(self):
        return Circle.PI * self.radius ** 2

    @area.setter
    def area(self, value):
        self.radius = (value / Circle.PI) ** .5

circle = Circle(5.)
print(circle.area)

circle.area = 10.
print(circle.radius)
```

- Property를 통해 Getter, Setter를 명시적 설정 가능
- Encapsulation 등에 활용

# Static & Class Method

```
class Number:
    Constant = 10

    @staticmethod
    def static_factory():
        obj = Number()
        obj.value = Number.Constant
        return obj

    @classmethod
    def class_factory(cls):
        obj = cls()
        obj.value = cls.Constant
        return obj

number_static = Number.static_factory()
number_class = Number.class_factory()
print(number_static.value, number_class.value)
```

파이썬에는 2가지 정적 함수 존재

- 객체에는 접근 불가능
- 일반적으로 Class.method 형태로 사용
- Static Method
  - staticmethod 꾸밈자 사용
  - 특별한 argument를 받지 않음
  - 일반적으로 class 내 유틸 함수로 사용
  - Class를 일종의 Namespace로 사용
- Class Method
  - Classmethod 꾸밈자 사용
  - 호출된 class인 **cls**를 받음 (self와 비슷)
  - factory 패턴에서 사용

**상속하면 차이가 발생**

## Three Elements of OOP

상속  
(Inheritance)

가시성,  
캡슐화  
(Visibiity)

다형성  
(Polymorphis  
m)

그러나 파이썬의 3원칙 구현방식은 다소 다르다

# Inheritance & Polymorphism

- 상속: 기존에 구현되어 있는 틀을 상속받아 새로운 틀을 만드는 것
  - 기존의 틀: 부모 Class, 새로운 틀: 자식 Class
  - 자식 Class에서는 부모의 기능을 이용할 수 있음
- 다형성: 같은 이름의 메소드를 다르게 작성하는 것
  - 각 자식 클래스가 다른 클래스와 차별화 됨
- **Python에서의 상속과 다형성**
  - 다중 상속 지원
  - 죽음의 다이아몬드
    - 메서드 탐색 순서를 따름 (mro)
  - super 내장 함수를 이용하여 상위 클래스 접근 가능



# Inheritance & Polymorphism

```
class Student:
    def __init__(self, name: str, sid: int):
        self.name = name
        self.sid = sid
        self.classes = []

    def __str__(self):
        return self.name + "_" + str(self.sid)

    def take_class(self, class_name: str) -> None:
        self.classes.append(class_name)

class Master(Student):
    # Student 상속
    def __init__(self, name: str, sid: int, professor: str):
        super().__init__(name, sid)
        # 부모 클래스 생성자 접근, 정해진 부르는 타이밍은 없다.
        self.professor = professor

    def __str__(self):
        # __str__ 재정의 → 다형성
        return super().__str__() + "_" + str(self.professor)

master = Master('Hojun Cho', 20194548, 'Jaegul Choo')
print(master)
print(super(Master, master).__str__())
# super로 언제나 원하는 상위 클래스로 변환
```

# Visibility

- 가시성이란 사용자에게 모델의 내부를 감추는 것을 의미
  - 캡슐화, 정보 은닉, 클래스 간 간섭 최소화
  - 최소한의 정보만을 특정 API로 공개
  - C나 Java에선 `private` & `protected`로 구현
- **Python에서의 가시성**
  - 명시적인 `private` & `protected` 범위가 없음 → 모두 `public`
  - `private` 변수/함수 이름 앞에 “`__`”를 붙임 (밑줄 2개)
    - Ex) `self.__name`, `self.__sid`
  - `protected` 변수/함수 이름 앞에 “`_`”를 붙임 (밑줄 1개)
    - Ex) `self._name`, `self._sid`

## Visibility

```
class TestClass(object):
    def __init__(self):
        self.attr = 1           # Public
        self._attr = 2         # Protected
        self.__attr = 3        # Private

instance = TestClass()
print(dir(instance))
```

**['\_TestClass\_attr', '\_attr', 'attr']를 포함**

- “\_”의 경우 변수명 앞에 Class 이름을 넣어 Mangling – 자식과 이름이 안 겹침
- Private와 Protected는 코드 완성 등에서 안 보임
- **그러나 둘 다 public과 기능적 차이는 없다 (밖에서 접근 가능함)**

PYTHON PROGRAMMING

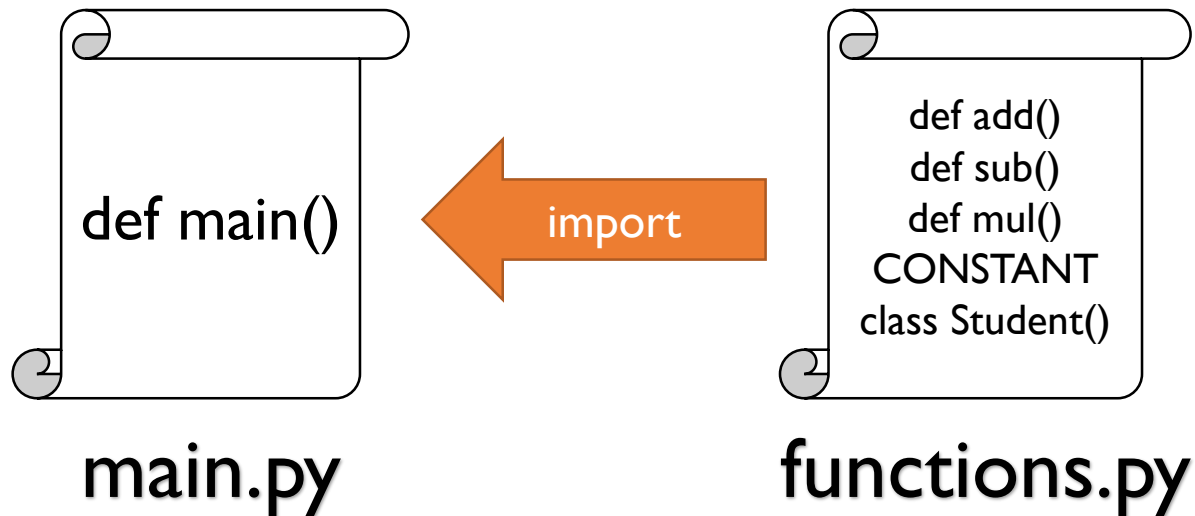
LECTURE 9: MODULE & PACKAGES

goorm

**KAIST AI**  
Graduate School of AI



## Importing from Other Files



**만약 다른 파일의 함수를 가지고 오고 싶다면?**

- 다른 사람이 완성한 함수와 클래스 사용
- 내장 & 외부 라이브러리 사용

**→ 모듈화가 필요**

# Import

./main.py

```
import functions  
print(functions.add(1, 2))
```

./functions.py

```
def add(num1: int, num2: int) -> int:  
    return num1 + num2
```

- 파이썬에선 모듈 == .py 파일
- Import 구문을 사용하여 모듈을 불러옴
  - 해당 파일 최상위에 선언된 모듈의 요소들을 불러오기 가능
  - module.element 식으로 사용
- . 혹은 .. 없이는 **절대 경로 기준** (Python이 실행되는 곳)

# Notion for Importing

main.py

```
import functions  
print(functions.add(1, 2))
```



functions.py

```
def add(num1: int, num2: int) -> int:  
    return num1 + num2  
print("Import 문은 global 코드 전체를 실행한다!")
```

- Import 문은 Import된 .py 파일을 처음부터 끝까지 실행시킨다
- 만약 해당 모듈을 main으로 했을 때 특정 Block을 실행시키고 싶다면?
  - `__name__` 기본 변수는 현재 모듈의 이름을 보여줌
  - Main으로 실행 중에는 "`__main__`"이라는 특수 이름을 가짐

```
def add(num1: int, num2: int) -> int:  
    return num1 + num2  
  
print(__name__)  
if __name__ == "__main__":  
    print("이 코드는 functions 모듈이 메인일 때만 실행")
```

# Import Examples I

## main.py

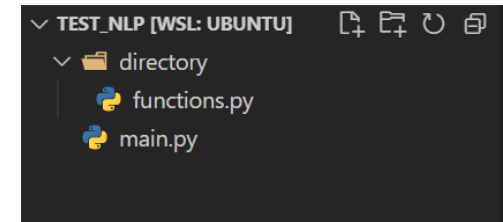
```
import directory.functions          # 폴더내 .py 파일
print(directory.functions.add(1, 2))

import directory.functions as func  # as로 별칭 만들기
Print(func.add(1, 2))

from directory import functions     # from으로 특정 부분 import
print(functions.add(1, 2))

from directory.functions import add # 특정 함수 import
Print(add(1, 2))

From directory.functions import *   # *로 모두 import
print(add(1, CONSTANT))            # 권장하지 않음
```



## directory/functions.py

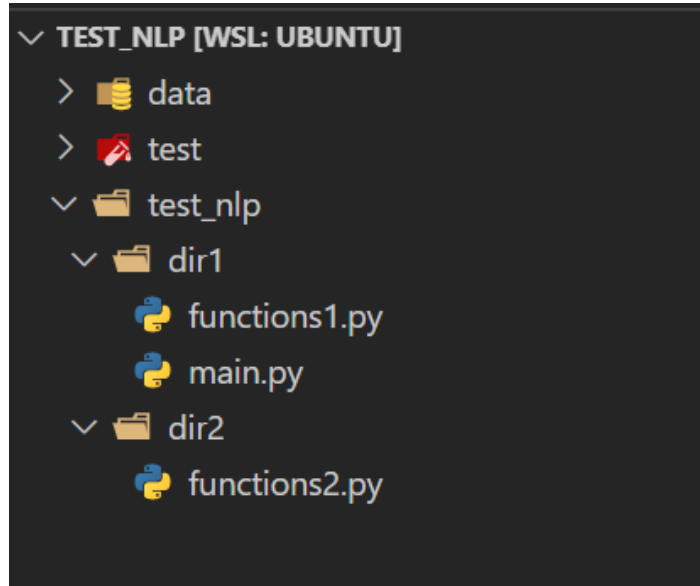
```
CONSTANT = 10

def add(num1: int, num2: int) -> int:
    return num1 + num
```

- **from**: 특정 모듈/폴더내에서 import
- **as**: 별칭 만들기 (alias)



## Import Examples 2



### test\_nlp/dir1/main.py

```
from test_nlp.dir1.functions1 import add # 절대 경로
print(add(1, 2))

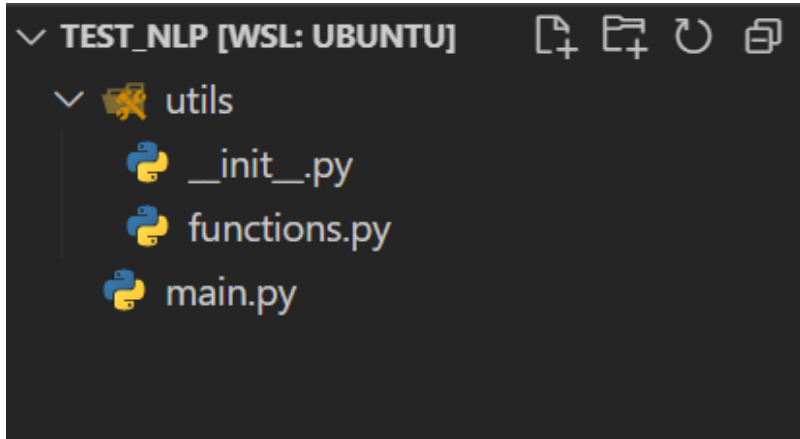
from .functions1 import add # 현재 폴더
Print(add(1, 2))

From ..dir2.functions2 import sub # 부모 폴더
Print(sub(1, 2)) # 모듈 형태로 실행 필요
```

- 최상위에선 상대 경로가 작동되지 않음
  - 최상위를 거치는 경우 포함
  - 일반적으로 프로젝트 이름으로 폴더를 만들어 코드를 넣음
  - 부모 폴더 접근을 위해서는 모듈 형태로 실행 필요

```
(base) ➤ napping ➤ ~/test_nlp ➤
python -m test_nlp.dir1.sub_main
```

## \_\_init\_\_.py File



**\_\_init\_\_.py 파일  
폴더 import시 초기화 가능**

### main.py

```
import utils  
  
print(utils.add(1, utils.CONSTANT))
```

### utils/\_\_init\_\_.py

```
from .functions import add  
  
CONSTANT = 10
```

### utils/functions.py

```
def add(num1, num2):  
    return num1 + num2
```

# The Python Standard Library

```
import random                                # 난수 관련 라이브러리
print (random.randint(0, 100))               # 0에서 100까지 정수 중 하나 반환
Print (random.uniform(0, 1))                 # 0에서 1까지 균등 분포에서 샘플링

Import time                                  # 시간 관련 라이브러리
Start = time.time()                           # 현재 시각 (unix 시간 기준)
time.sleep(1)                                 # 1초 기다리기
print(time.time() - start)

import threading                              # 쓰레드 관련 라이브러리

def print_function():
    print("사실, 파이썬은 GIL 때문에 사실상 싱글 쓰레드입니다.")
    time.sleep(1)

thread = threading.Thread(target=print_function) # 쓰레드 만들기
Thread.start()                                # 쓰레드 시작
Thread.join()                                # 쓰레드 수거
```

파이썬은 강력하고 다양한 표준 라이브러리를 가지고 있음

## External Library & Package Managing

- **파이썬 표준 라이브러리로 해결할 수 없다면....?**
  - 인터넷 상의 오픈 소스 라이브러리 설치 필요
  - 수치 그래프 그리기 → matplotlib
  - 웹 서버 만들기 → flask
  - GPU 연산 사용하기 → cupy
  - 딥러닝 전용 라이브러리 → tensorflow & pytorch
- **웹 서버 프로젝트와 딥러닝 프로젝트가 따로 있다면...?**
  - 한 파이썬 위에 둘 다 설치한다 → 관리가 어려움
  - 각각 다른 환경 & 파이썬에서 돌리고 싶은데...

**→ 패키지 관리자가 필요!**

# Python Package Manager



## ***PIP + Virtual env***

Python 기본 패키지 관리 프로그램



## ***Anaconda3***

<https://www.anaconda.com/products/individual>  
기계학습 및 수치해석 특화 패키지 관리 프로그램  
**상용 프로그램이다**

## Notions for Colab Users



- 코랩은 개별 노트북마다 개별 환경이 설정됨
    - **노트북이 꺼지면 환경이 삭제됨**
  - 대부분의 라이브러리가 사전 설치되어 있음
    - matplotlib, tensorflow, pytorch, scipy, numpy, ...
    - 만약 추가 설치가 필요한 경우 pip을 사용
- 굳이 Anaconda를 설치할 필요가 없음**

# Anaconda & Minconda

- **Anaconda**
  - 기본 설치판
  - 200개 이상의 라이브러리가 사전 설치됨
  - 무거움
  - <https://www.anaconda.com/products/individual>
- **Minconda**
  - 최소 설치판
  - Anaconda의 기본 기능만을 설치
  - 가벼움
  - [Miniconda — Conda documentation](#)

# Creating Virtual Environment

```
(base) napping ~/test_nlp  
conda create -n nlp
```

가상환경      환경 이름  
만들기

---

```
(base) napping ~/test_nlp  
conda activate nlp
```

가상환경 활성화

---

활성화된 가상환경

```
(nlp) napping ~/test_nlp  
|
```



# Managing Virtual Environment

```
(nlp) ➤ napping ➤ ~/test_nlp ➤  
conda deactivate
```

가상환경 나가기

---

```
(base) ➤ napping ➤ ~/test_nlp ➤  
conda install <패키지 이름> -c <설치 채널>
```

패키지 설치

설치 채널

```
(base) ➤ napping ➤ ~/test_nlp ➤  
conda install python=3.9 pytorch -c pytorch
```

- 채널이 명시되어 있지 않을 경우 default 채널 탐색
- 채널은 문서를 참고하거나 Anaconda에 검색

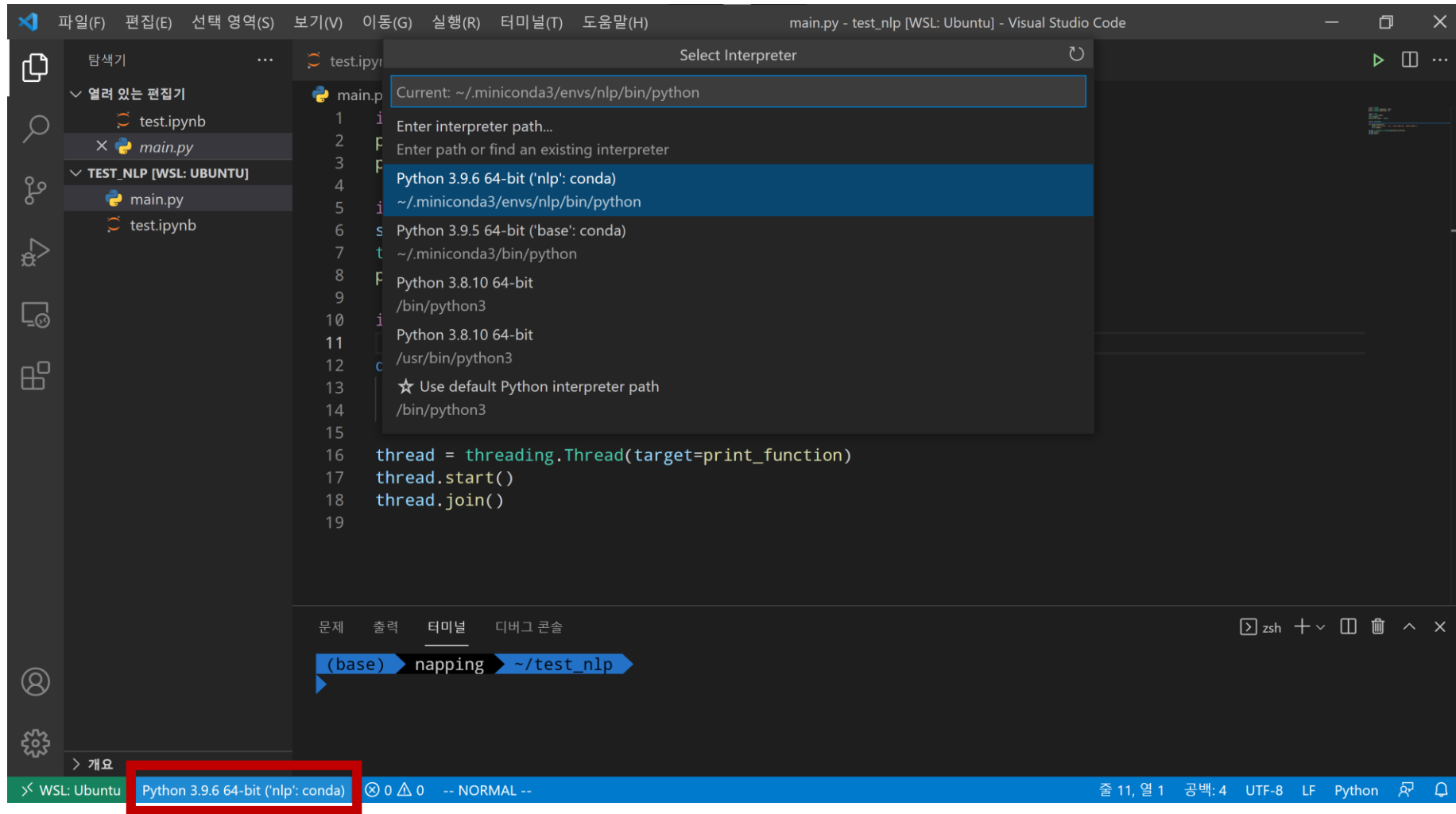
# Managing Virtual Environment

```
(nlp) napping ~/test_nlp
conda list
# packages in environment at /home/napping/.miniconda3/envs/nlp:
#
# Name                        Version                        Build      Channel
_libgcc_mutex                 0.1                            main
_openmp_mutex                 4.5                            1_gnu
blas                           1.0                             mkl
ca-certificates               2021.7.5                       h06a4308_1
certifi                       2021.5.30                       py39h06a4308_0
```

```
(nlp) napping ~/test_nlp
conda list | grep numpy
numpy                        1.20.3                       py39hf144106_0
numpy-base                   1.20.3                       py39h74d4b33_0
```

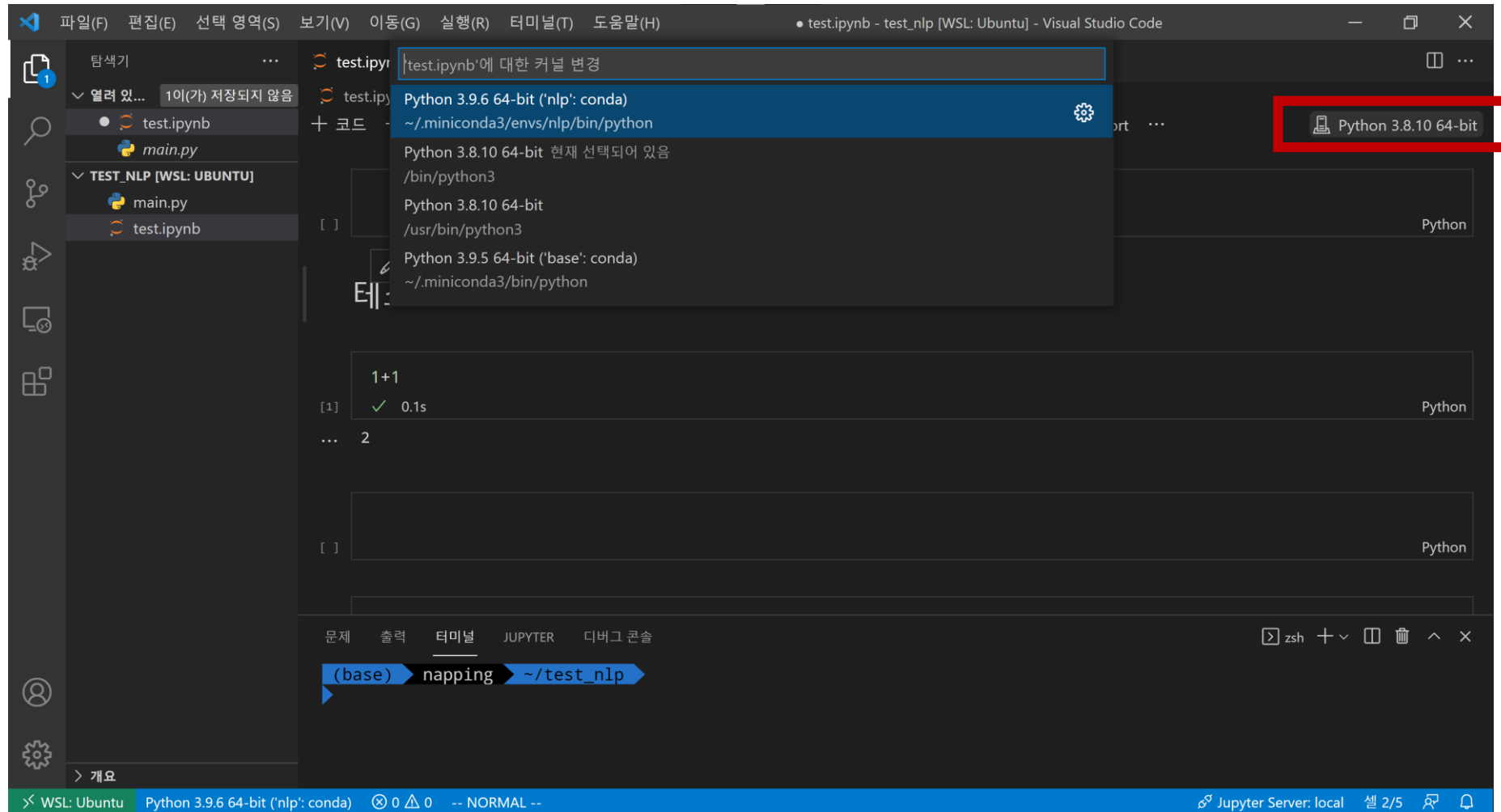
- conda list로 현재 환경에 설치된 패키지 확인
  - Linux grep 명령어랑 결합시 편리

# Conda in VsCode



왼쪽 아래 버튼을 눌러 가상환경 설정가능

# Conda in VsCode with Jupyter

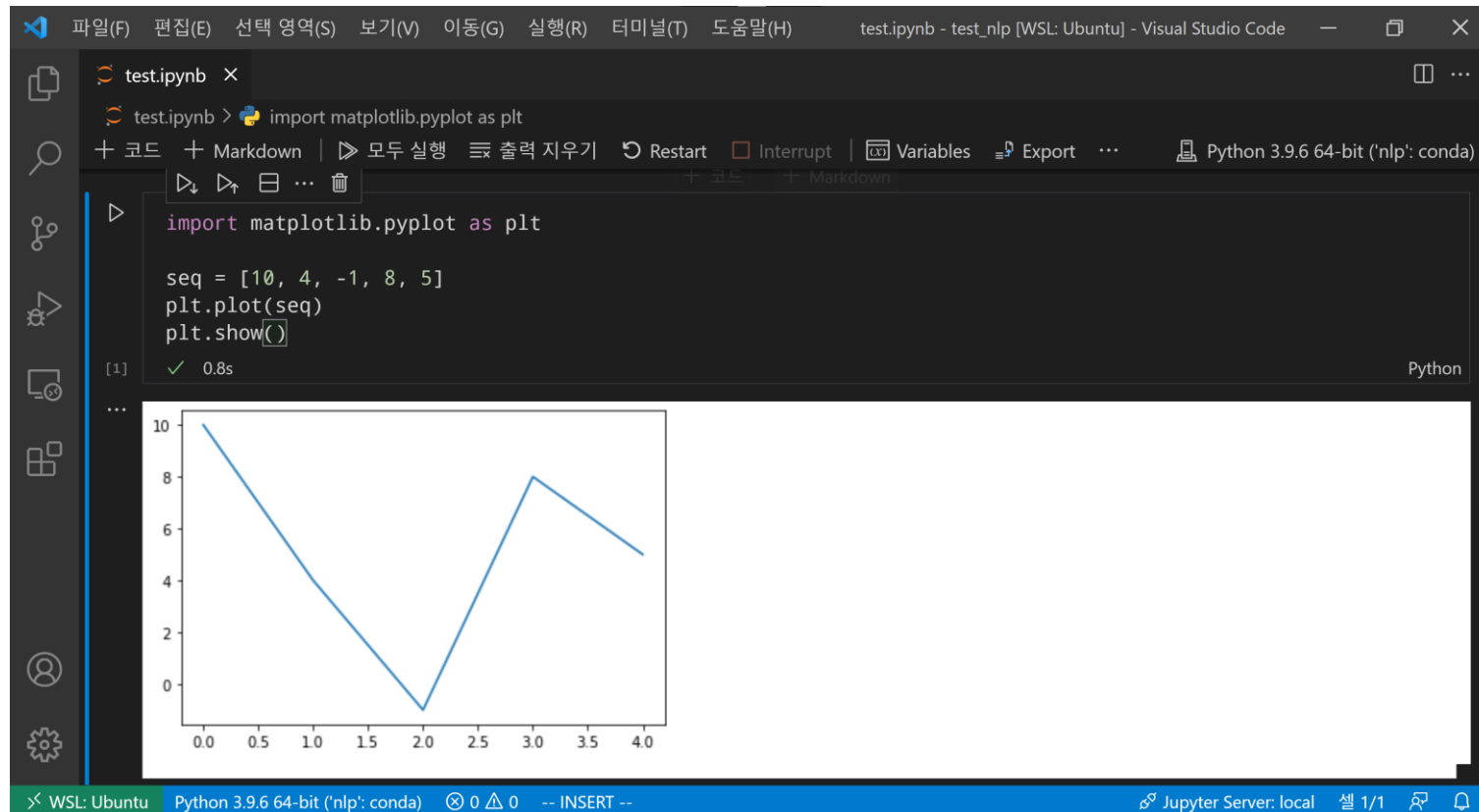


**.ipynb의 경우 쥬피터 서버의 가상환경 설정 필요**

# Using External Libraries: matplotlib

## matplotlib: 그래프 그리기 라이브러리

```
(nlp) ➔ napping ➔ ~/test_nlp ➔  
conda install matplotlib
```



## Using External Libraries: tqdm

### tqdm: 진행 바 (Progress Bar) 만들기

```
(nlp) napping ~/test_nlp  
conda install tqdm
```

```
Python 3.9.6 (default, Jul 30 2021, 16:35:19)  
[GCC 7.5.0] :: Anaconda, Inc. on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> import time  
>>> from tqdm.auto import trange  
>>> for _ in trange(10):  
...     time.sleep(1)  
...  
30%|██████████          | 3/10 [00:03<00:07, 1.00s/it]
```

# PYTHON PROGRAMMING

## LECTURE 10: ADVANCED DATA STRUCTURE

goorm

**KAIST AI**  
Graduate School of AI



## Over the Pre-defined Data Structure

**파이썬에는 기본적으로 몇 가지 자료구조가 사전에 선언**

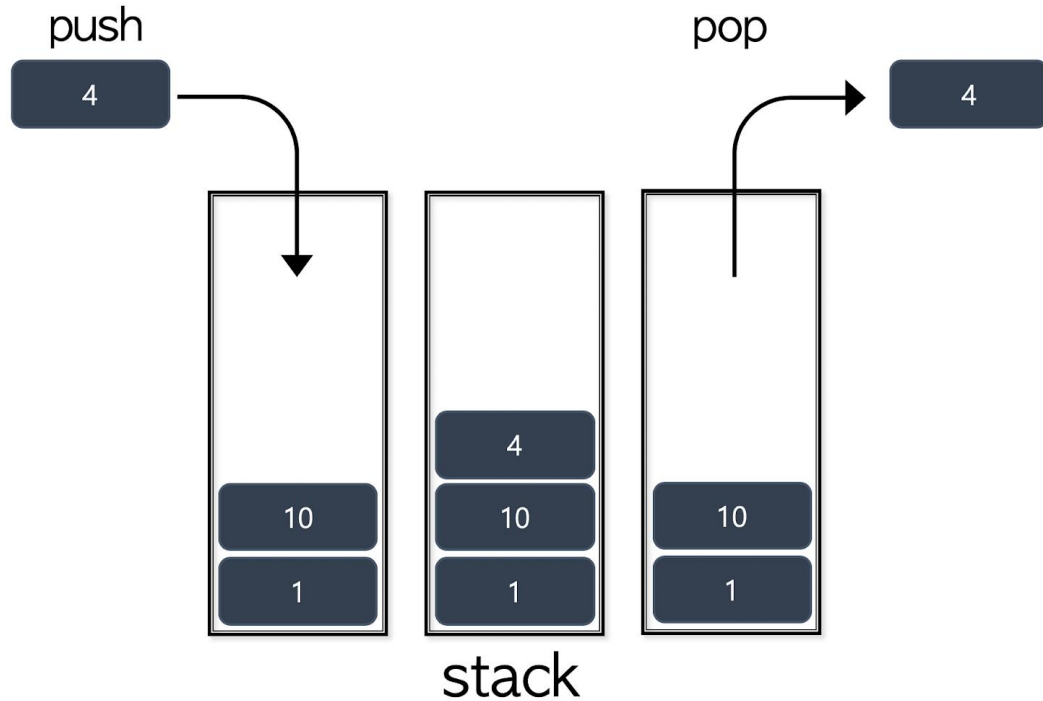
- list, set, dict 등등

**좀 더 강력한 자료구조를 사용하고 싶다면...?**

- 기존 자료구조를 활용
- Python Standard Library로 대부분 해결!
  - Stack & Queue
  - Linked List
  - 기본값이 있는 dictionary
  - 개수를 세는 자료구조
  - ...



# Stack



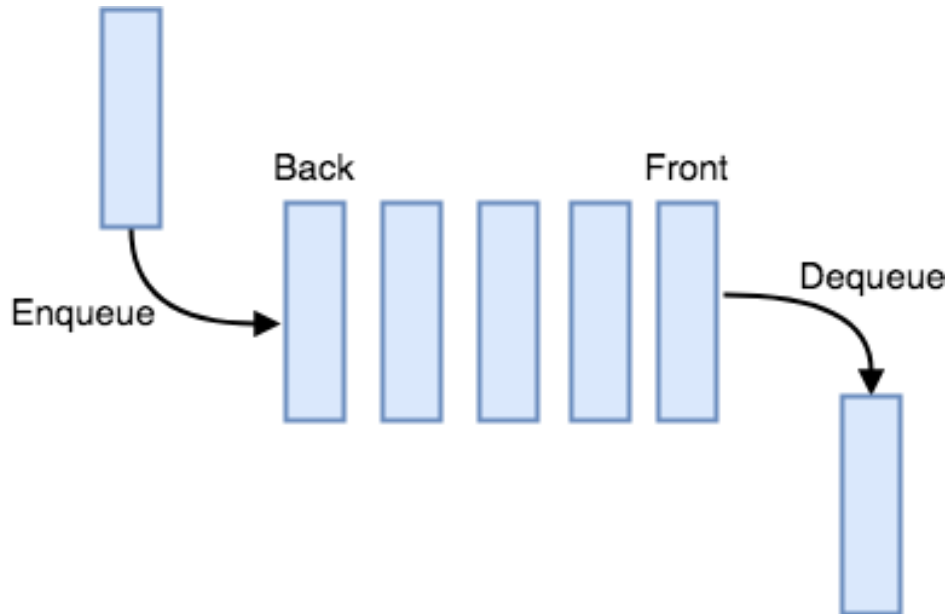
```
>>> a = [1, 10]
>>> a.append(4)
>>> a.append(20)

>>> a.pop()
20
>>> a.pop()
4
```

스택 구조는 기존 List를 활용

- 동적 배열이기 때문에 push와 pop이  $O(1)$

# Queue

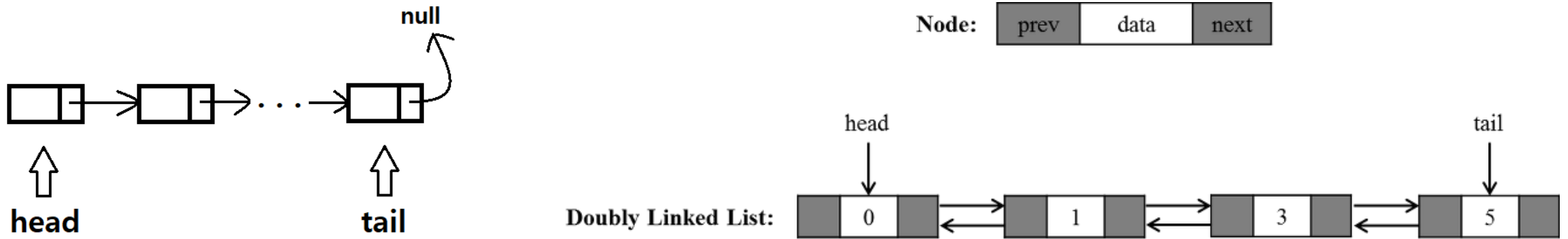


```
>>> a = [1, 10]
>>> a.insert(0, 20)
>>> a.insert(0, 15)
>>> a.pop()
10
>>> a.pop()
1
```

## 큐 구조를 기존 List로 만들 경우

- List는 한 쪽 방향으로 열려 있는 동적 배열  
→ 처음 위치 삽입 혹은 삭제가  $O(N)$  걸림
- 다른 형태의 데이터 구조가 필요

# Linked List



Queue 구조 구현을 위해선 연결 리스트 자료 구조가 필요

- 양쪽으로 자유로운 입출력
- 중간 참조는 오래 걸리나, 큐 구조에선 상관 없음

파이썬 *collection* Library의 *deque*를 사용!

# Deque

```
>>> from collections import deque          # deque import
>>> queue = deque([10, 5, 12])              # deque 생성

>>> queue.appendleft(16)                    # 왼쪽 삽입
>>> queue.pop()                             # 오른쪽 삭제
12

>>> queue.append(20)                        # 오른쪽 삽입
>>> queue.popleft()                         # 왼쪽 삭제
16

>>> queue
deque([10, 5, 20])
>>> deque(reversed(queue))                  # deque 뒤집기 O(N)
deque([20, 5, 10])
```

이중 연결 리스트의 함수를 지원

## Defaultdict

```
>>> d = {"first": 0}

>>> d["second"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'second'

>>> d.get("second", "없어요")
'없어요'
```

파이썬 기본 dictionary는 키가 없을 경우 에러

- 에러 발생을 막기 위해선 get 메소드의 default 값 지정 필요

# Defaultdict

```
text = """
유구한 역사와 전통에 빛나는 우리 대한국민은
3·1운동으로 건립된 대한민국임시정부의 법통과 불의에 항거한 4·19민주이념을 계승하고,
조국의 민주개혁과 평화적 통일의 사명에 입각하여 정의·인도와 동포애로써 민족의 단결을 공고히 하고,
모든 사회적 폐습과 불의를 타파하며,
자유와 조화를 바탕으로 자유민주적 기본질서를 더욱 확고히 하여 정치·경제·사회·문화의 모든 영역에 있어서 각인의 기회를 균등히 하고,
능력을 최고도로 발휘하게 하며, 자유와 권리에 따르는 책임과 의무를 완수하게 하여,
안으로는 국민생활의 균등한 향상을 기하고 밖으로는 항구적인 세계평화와 인류공영에 이바지함으로써
우리들과 우리들의 자손의 안전과 자유와 행복을 영원히 확보할 것을 다짐하면서
1948년 7월 12일에 제정되고 8차에 걸쳐 개정된 헌법을 이제 국회의 의결을 거쳐 국민투표에 의하여 개정한다.
1987년 10월 29일.
"""
```

```
characters = {}

for char in text:
    count = characters.get(char, None)

    if count is None:
        characters[char] = 0

    characters[char] += 1
```



```
from collections import defaultdict

characters = defaultdict(int) # 기본값 int()
# characters = defaultdict(lambda: 0)
for char in text:
    characters[char] += 1
```

Dictionary의 기본 값을 지정 가능

# Counter

## Dictionary 처럼 생성 및 관리

```
>>> c = Counter({"Korean": 2, "English": 3})
>>> c
Counter({'English': 3, 'Korean': 2})

>>> c.keys()                                # 일반적인 dict과 차이 없음
Dict_keys(['Korean', 'English'])
>>> c.values()
Dict_values([2, 3])
>>> c["Korean"]
2

>>> list(c.elements())                      # 모든 요소 반환
['Korean', 'Korean', 'English', 'English', 'English']
```

## kwargs로 생성

```
>>> c = Counter(Korean=2, English=3)
>>> c
Counter({'English': 3, 'Korean': 2})
```

# Counter

## 집합 연산 지원

```
>>> from collections import Counter
>>> a = Counter([1, 1, 2, 2, 2, 3])
>>> b = Counter([2, 3, 3, 4])

>>> a + b                                     # 횟수 더하기
Counter({2: 4, 3: 3, 1: 2, 4: 1})

>>> a & b                                     # 교집합
Counter({2: 1, 3: 1})

>>> a | b                                     # 교집합
Counter({2: 3, 1: 2, 3: 2, 4: 1})

>>> a - b                                     # 차집합
Counter({1: 2, 2: 2})
```



# Named Tuple

```
class Coords3D:
    def __init__(self, x, y, z):
        self._x = x
        self._y = y
        self._z = z

    @property
    def x(self):
        return self._x

    @property
    def y(self):
        return self._y

    @property
    def z(self):
        return self._z
```

## 데이터만을 담기 위한 클래스 사용

- 클래스 선언 및 getter 작성 필요
- 숫자로 Indexing 불가능
  - `__getitem__` 작성 필요

## → 귀찮음

- 간단한 데이터 구조인데 굳이 클래스를..?

# Named Tuple

```
point = (10, 20, 30)

print(point[0]) # 뭐가 x였지...?
print(point[1])
```

- 튜플로 관리할 시 쉽게 자료구조를 만들 수는 있음
  - 그러나 관리하기에는 불편
    - Attribute명으로 접근이 불가능
- 

```
from collections import namedtuple

# 새로운 타입 생성
Coords3D = namedtuple("Coords3D", ['x', 'y', 'z'])

point = Coords3D(10, 20, z=30)
print(point.x)          # Attribute 이름으로 참조
Print(point[1])         # Index로 참조
print(*point)           # Tuple Unpacking

point[1] += 1           # Error 발생
```

## Named Tuple

- 각 튜플 원소에 이름 붙이기 가능
- 클래스가 아니라 튜플이다
  - Unpacking 가능

# Dataclass

```
from dataclasses import dataclass

@dataclass
class Coords3D:
    x: float
    y: float
    z: float = 0

    def norm(self) -> float:
        return (self.x ** 2 + self.y ** 2 + self.z ** 2) ** .5

point = Coords3D(10, 20, z=30)
print(point)
print(point.norm())
```

Pythonic한 데이터 클래스를 위해선 *dataclasses*의 *dataclass* 데코레이터 활용

PYTHON PROGRAMMING

LECTURE 11: STRING

goorm

**KAIST AI**  
Graduate School of AI

 **DAVIAN**

# Recap Python String

## 파이썬 String의 특징

- 원시 자료형이자, 불변 타입이다.
- 큰 따옴표 " 혹은 작은 따옴표 ' 표기된다.
- 따옴표를 3개 연달아 쓰면 여러 줄을 넣을 수 있다.
  - `"""text"""`, `"""여러줄 적기"""`
- Indexing 및 Slicing이 가능하다.
- 덧셈 및 곱셈이 가능하다
- in & not in 연산이 가능하다
  - tuple과 다소 다르게 작동
- Unicode로 처리된다.

```
>>> text = """
... 여러줄 적기
... 가 가능합니다
... """
```

```
>>> text
'\n여러줄 적기\n가 가능합니다\n'
```

# Special Characters

## Escape 문자 \ 를 사용하여 특수 문자 활용

문자	설명
\ [Enter]	다음 줄과 연속임을 표현
\\	\ 문자
\'	' 문자
\"	" 문자
\b	백스페이스
\n	줄 바꾸기
\t	TAB 키
\e	ESC 키

```
>>> text = '\
... 이렇게 적으면 \
... 엔터 없이 \
... 여러 줄을 적어요\
... '
```

```
>>> text
'이렇게 적으면 엔터 없이 여러 줄을 적어요'
```

```
>>> text = 'print할 때랑 \n평가할 때랑 달라요'
```

```
>>> text
'print할 때랑 \n평가할 때랑 달라요'
```

```
>>> print(text)
print할 때랑
평가할 때랑 달라요
```

# Raw String

**r“<TEXT>” 형태로 \ 를 무시하고 문자 그대로 취급 가능**

```
>>> string = "여기서는 역 슬래시는 \n 특별한 의미를 가져요"
>>> string
'여기서는 역 슬래시는 \n 특별한 의미를 가져요'
>>> print(string)
여기서는 역 슬래시는
특별한 의미를 가져요

>>> raw_string = r"여기서는 역 슬래시가 \n 특별한 의미를 가지지 않아요"
>>> raw_string
'여기서는 역 슬래시가 \\n 특별한 의미를 가지지 않아요'
>>> print(raw_string)
여기서는 역 슬래시가 \n 특별한 의미를 가지지 않아요
```

# 역 슬래시가 일반 문자로 변환

```
>>> locate = "C:\\Users\\hojuncho"
>>> print(locate)
C:\Users\hojuncho

>>> locate = r"C:\Users\hojuncho"
>>> print(locate)
C:\Users\hojuncho
```

**주로 정규 표현식에서 사용**

# String Functions

함수 형태	기능
<code>len(string)</code>	문자 개수 반환
<code>string.upper()</code>	대문자로 변환
<code>string.lower()</code>	소문자로 변환
<code>string.capitalize()</code>	문자열 시작 문자를 대문자로 변환
<code>string.title()</code>	단어 시작을 대문자로 변환

```
>>> text = 'this is the Python course'
>>> len(text)
25

>>> text.upper()
'THIS IS THE PYTHON COURSE'
>>> text.lower()
'this is the python course'
>>> text.capitalize()
'This is the python course'
>>> text.title()
'This Is The Python Course'
```



# String Functions

함수 형태	기능
string.strip()	좌우 공백 제거
string.lstrip()	왼쪽 공백 제거
string.rstrip()	오른쪽 공백 제거
string.isdigit()	숫자 형태인지 확인
string.isupper()	대문자로만 이루어져 있는지 확인
string.islower()	소문자로만 이루어져 있는지 확인

```
>>> test = '    공백이 \t 있어요 \t\n '
>>> test.strip()
'공백이 \t 있어요'
>>> test.lstrip()
'공백이 \t 있어요 \t\n '
>>> test.rstrip()
'    공백이 \t 있어요'
```

```
>>> '12345'.isdigit()
True
>>> '1.24e-3'.isdigit()
False
>>> 'Capitalize'.isupper()
False
>>> 'lower_case'.islower()
True
```

# String Pattern Matching

함수 형태	기능
string.count(pattern)	문자열 string 내에 pattern 등장 횟수 반환
string.find(pattern)	문자열 string 내에 pattern 첫 등장 위치 반환 (앞에서 부터)
string.rfind(pattern)	문자열 string 내에 pattern 첫 등장 위치 반환 (뒤에서 부터)
string.startswith(pattern)	문자열 string이 pattern으로 시작하는지 확인
string.endswith(pattern)	문자열 string이 pattern으로 끝나는지 확인

```
>>> text = 'abc_text_abc_ee'
>>> pattern = 'abc'

>>> text.count(pattern)
2
>>> text.find(pattern)
0
>>> text.rfind(pattern)
9
>>> text.startswith(pattern)
True
>>> text.endswith(pattern)
False
```

# String Split & Join

함수 형태	기능
string.split()	공백을 기준으로 문자열 나누기
string.split(pattern)	Pattern을 기준으로 문자열 나누기
string.join(iterable)	String을 중간에 두고 iterable 원소들 합치기

```
>>> text = '한국어 abc 테스트 \n abc 중 \t 입니다'

>>> text.split()                                # 공백으로 나누기
['한국어', 'abc', '테스트', 'abc', '중', '입니다']

>>> text.split('abc')                          # 'abc' 패턴으로 나누기
['한국어 ', ' 테스트 \n ', ' 중 \t 입니다']

>>> ' '.join(text.split())                     # ' '를 중간에 두고 문자열들 합치기
'한국어 abc 테스트 abc 중 입니다'

>>> ', '.join(str(i) for i in range(10))      # ', '를 중간에 두고 문자열들 합치기
'0, 1, 2, 3, 4, 5, 6, 7, 8, 9'
```

# String Formatting

## Print등을 할 때 보기 좋게 값들을 확인하고 싶음

- 일정한 형태로 변수들을 문자열로 출력
- + 를 써서 구현은 할 수 있지만....

```
>>> a, b, c = 10, 1.725, 'sample'
>>> str(a) + ": " + str(b) + " - " + c # 하기도 보기도 불편
'10: 1.725 - sample'

>>> "%d: %f - %s" % (a, b, c)           # %-formatting
'10: 1.725000 - sample'

>>> "{}: {} - {}".format(a, b, c)       # .format 함수
'10: 1.725 - sample'


>>> f"{a}: {b} - {c}"                   # f-string
'10: 1.725 - sample'
```

## %-formatting

### “%datatype” % variables 형태로 표현

- C나 Java에서 주로 쓰이는 방식

"Art: %5d, Price per Unit: %8.2f" % (453, 59.058)



```
>>> "Art: %5d, Price per Unit: %8.2f" % (453, 59.058)
'Art: 453, Price per Unit: 59.06'
```

%datatype	설명
%s	문자열 (str)
%d	정수 (int)
%f	부동소수점 (float)
%o	8진수
%x	16진수

## Padding

### “%[padding]datatype” 형태로 패딩 가능

```
>>> "%d+%d+%d" % (1, 10, 100)
'1+10+100'
>>> "%4d+%4d+%4d" % (1, 10, 100)
'   1+  10+ 100'
>>> "%-4d+% -4d+% -4d" % (1, 10, 100)
'1   +10  +100 '
>>> "%04d+%04d+%04d" % (1, 10, 100)
'0001+0010+0100'
```

### “%[padding].[precision]datatype” 형태로 정확도 지정

```
>>> "%f+%f+%f" % (123.4, 12.34, 1.234)
'123.400000+12.340000+1.234000'
>>> "%.3f+%.3f+%.3f" % (123.4, 12.34, 1.234)
'123.400+12.340+1.234'
>>> "%8.3f+%8.3f+%8.3f" % (123.4, 12.34, 1.234)
' 123.400+ 12.340+ 1.234'
>>> "%08.3f+%08.3f+%08.3f" % (123.4, 12.34, 1.234)
'0123.400+0012.340+0001.234'
```

# String.format Method

## “{0}”.format(variables) 형태로 표현

`"Art: {0:5d}, Price per Unit: {1:8.2f}".format(453, 59.058)`



순서 설정

```
>>> a, b, c = 10, 1.725, 'sample'
>>> "{}: {} - {}".format(a, b, c)
'10: 1.725 - sample'
>>> "{0}: {1} - {2}".format(a, b, c)
'10: 1.725 - sample'
>>> "{0}: {2} - {1}".format(a, b, c)
'10: sample - 1.725'
```

패딩 설정

```
>>> "{0}+{1}+{2}".format(123.4, 12.34, 1.234)
'123.4+12.34+1.234'
>>> "{0:.3f}+{1:.3f}+{2:.3f}".format(123.4, 12.34, 1.234)
'123.400+12.340+1.234'
>>> "{:8.3f}+{:8.3f}+{:8.3f}".format(123.4, 12.34, 1.234)
' 123.400+ 12.340+ 1.234'
```

## Naming

### 순서가 헛갈리지 않게 각 위치에 이름 붙이기

“%([name])format” 형태

```
>>> "%(first)5.2f - %(second)5.2f" % {"first": 10.2, "second": 5.62}
'10.20 - 5.62'
```

“{[name]:format}” 형태

```
>>> "{first:5.2f} - {second:5.2f}".format(first=10.2, second=5.62)
'10.20 - 5.62'

>>> "{first:5.2f} - {second:5.2f}".format(**{"first": 10.2, "second": 5.62})
'10.20 - 5.62'
```



# F-string

## f“{variable}” 형태로 표현

- Python 3.6 이상 부터 지원
- 변수 위치에 원하는 변수를 위치하면 됨

### 변수 삽입

```
>>> a, b, c = 10, 1.725, 'sample'
>>> f"{a}: {b} - {c}"
'10: 1.725 - sample'
>>> f"{a}: {c} - {b}"
'10: sample - 1.725'
```

### 패딩 설정

```
>>> value = 12.34
>>> f"{value*10}+{value}+{value/10}"
'123.4+12.34+1.234'
>>> f"{value*10:.3f}+{value:.3f}+{value/10:.3f}"
'123.400+12.340+1.234'
>>> f"{value*10:8.3f}+{value:8.3f}+{value/10:8.3f}"
' 123.400+ 12.340+ 1.234'
```

## Finding Patterns

야 이거 **#%이름#**거 아니냐?  
**#%이름#**에게 물어봐봐  
**#%이모티콘#**

문자열에서 **#%SOMETHING#**를 찾거나 치환하는 방법은?

- Find 메소드는 정확히 일치하는 문자열만 찾을 수 있음
- 문자열에서 특정한 패턴을 정의하고 찾는 방법이 필요

# Regular Expression

## 정규 표현식

- 특정한 규칙을 가진 문자열의 집합을 표현하는데 사용하는 형식 언어
- 많은 텍스트 편집기와 프로그래밍 언어에서 문자열 검색과 치환에 활용

### 패턴

`\d{3}\-\d{4}\-\d{r}`

`\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}`

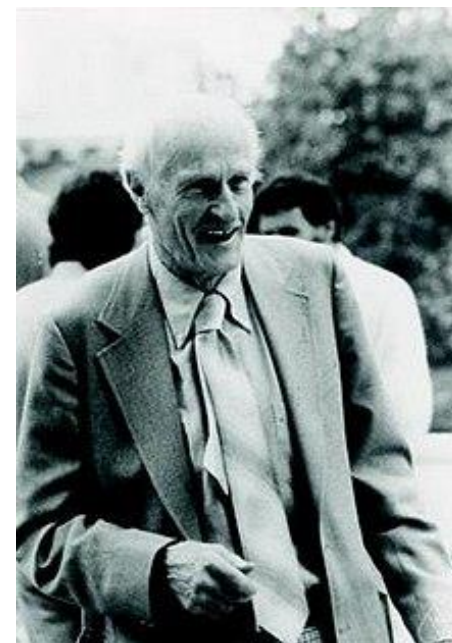
`#%[^#]+#`

### 예시

010-1234-5678

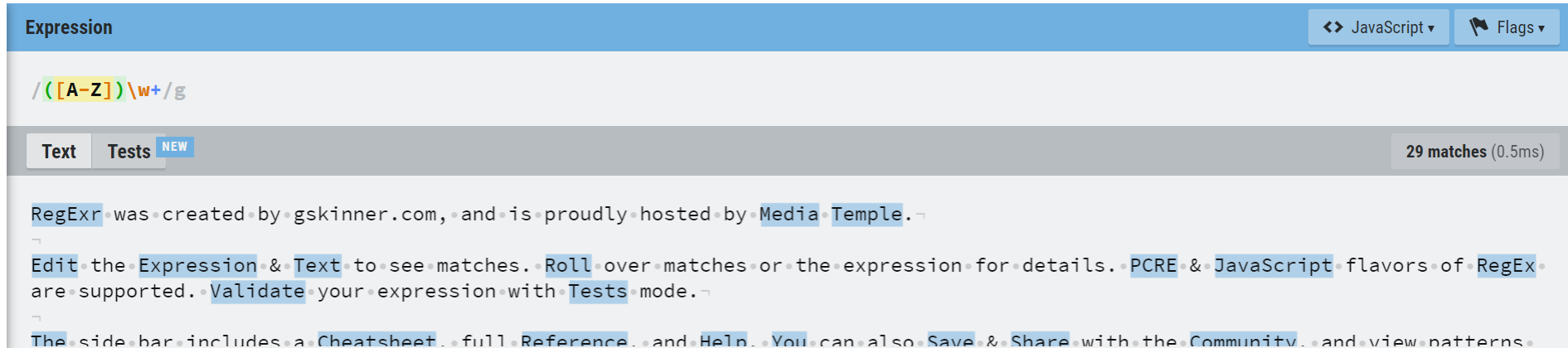
192.168.0.20

#%이모티콘#



*Stephen Cole Kleene*

# Regular Expression



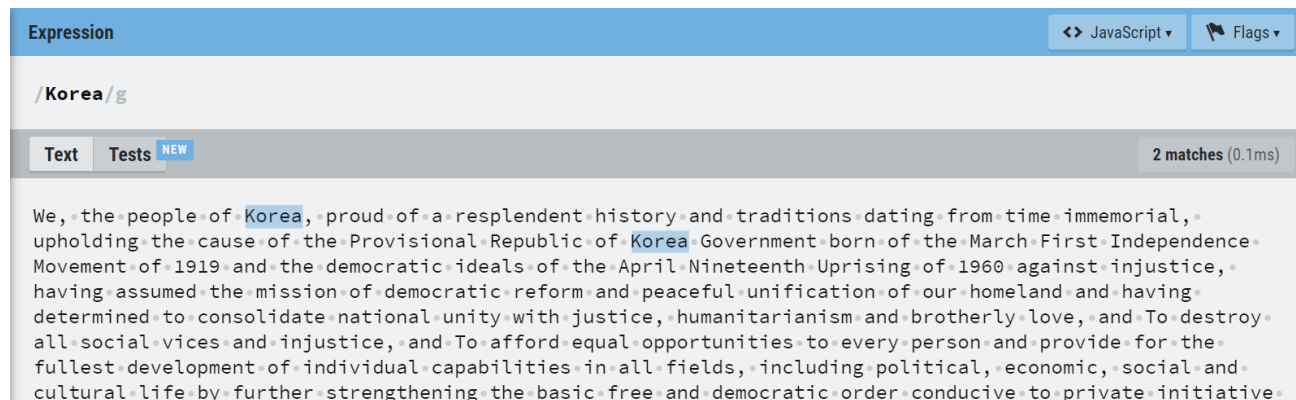
## 정규 표현식 공부하기 → 언어 하나 더 배우기

- 정규식 연습장을 활용해보자 (<http://www.regexr.com/>)
- 문법이 매우 방대
  - ppt 슬라이드 30장 정도 더 필요
  - 그래도 기본적인 몇 가지 요소를 설명하자면...

# Regular Expression

- 일반 텍스트 패턴

- 정확히 일치되는 문자열 찾기
- .find 함수와 동일



- 특수 문자

- 정규식만에서 쓰이는 문자
- 공백, 영단어 등
- 일반 문자와 섞어 씀

특수 글자	설명	Regex
<code>\w</code>	영숫자 + “_”	<code>[A-Za-z0-9_]</code>
<code>\W</code>	(영숫자 + “_”)를 제외한 문자	<code>[^A-Za-z0-9_]</code>
<code>\d</code>	숫자	<code>[0-9]</code>
<code>\D</code>	숫자가 아닌 문자	<code>[^0-9]</code>
<code>\s</code>	공백 문자	<code>[\t\r\n\v\f]</code>
<code>\S</code>	공백이 아닌 문자	<code>[^\t\r\n\v\f]</code>
<code>\b</code>	단어 경계	<code>(?&lt;=\W)(?=\w) (?&lt;=\w)(?=\W)</code>

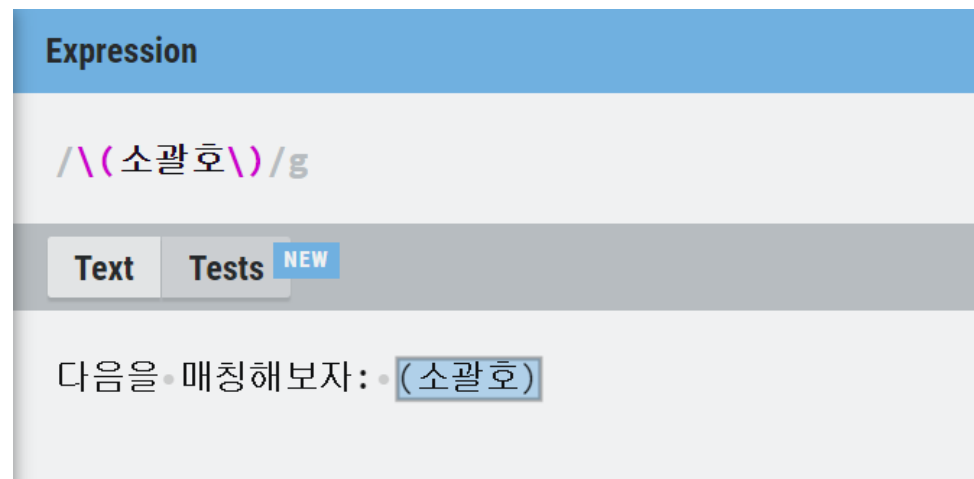
## Meta Character

- 메타 문자: 정규식의 문법적인 요소를 담당하는 문자

. ^ \$ \* + ? { } [ ] \ | ( )

- 이 문자들은 특수한 의미의 문자이므로 사용 불가능

- 만약 문자 그대로 매칭하고 싶다면 Escape 문자 \ 붙여 사용



# Regular Expression: Character Class

## 문자 클래스 [ ]

- [와 ] 사이의 문자들 중 하나와 매칭
- “-”를 사용하여 범위를 지정 가능
  - [a-z] [A-Z0-9] [\d\s]

Expression
/[abcde]/g
Text Tests NEW
My·number·is·010-1234-5678

Expression
/[0-9]/g
Text Tests NEW
My·number·is·010-1234-5678

## 부정 [^ ]

- [^와 ] 사이에 없는 문자를 매칭
- “-”를 사용하여 범위를 지정 가능
  - [^a-z] [^A-Z0-9] [^s]

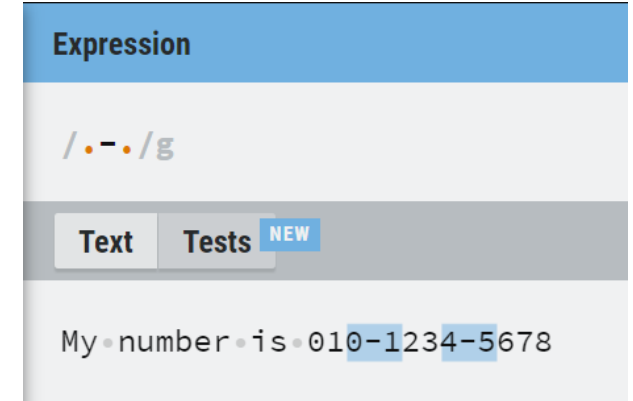
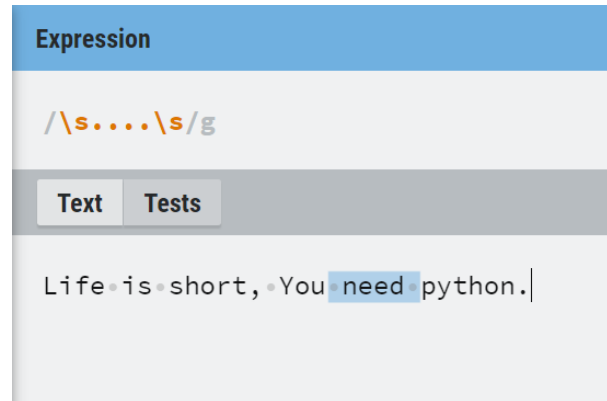
Expression
/[^cho]/g
Text Tests NEW
abcde·fghij·klm·n

Expression
/[^s\d]/g
Text Tests NEW
My·number·is·010-1234-5678

# Regular Expression: Repetition

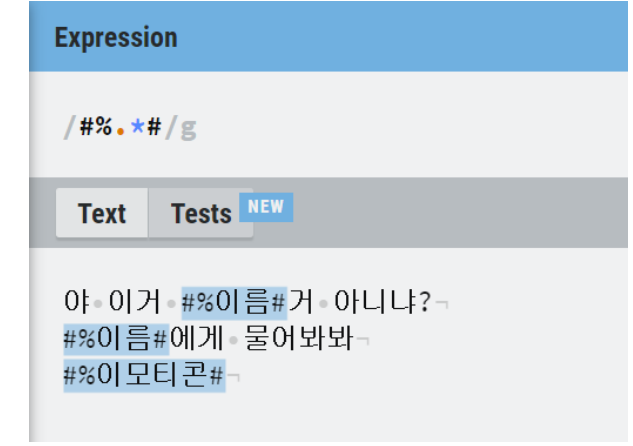
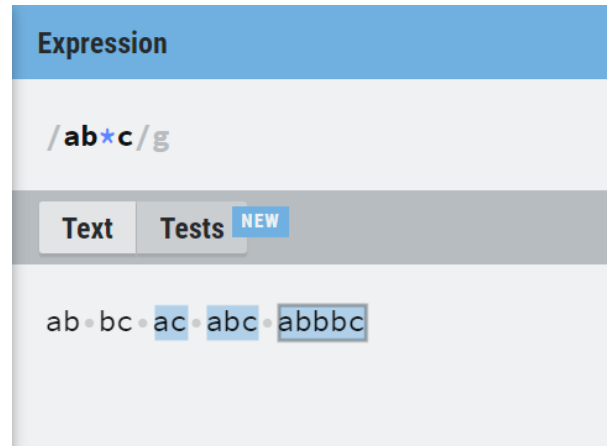
## 문자 .

- 아무 문자나 하나를 매칭
- 줄 바꿈 문자 \n는 제외



## 0회 이상 \*

- 앞의 문자를 0개 이상 포함

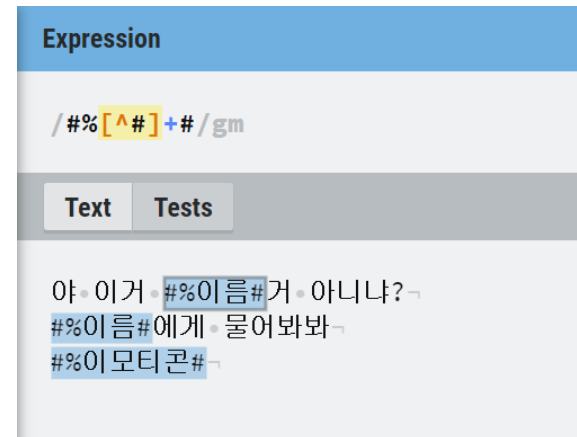
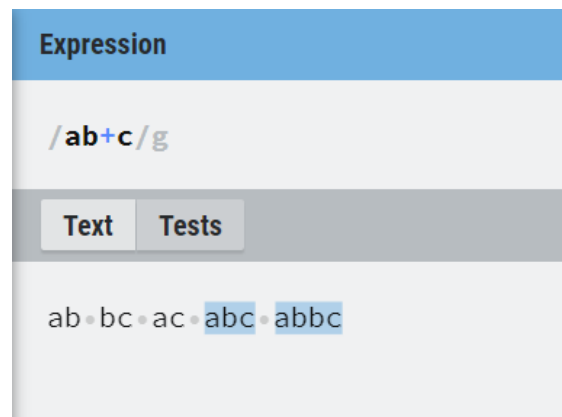




# Regular Expression: Repetition

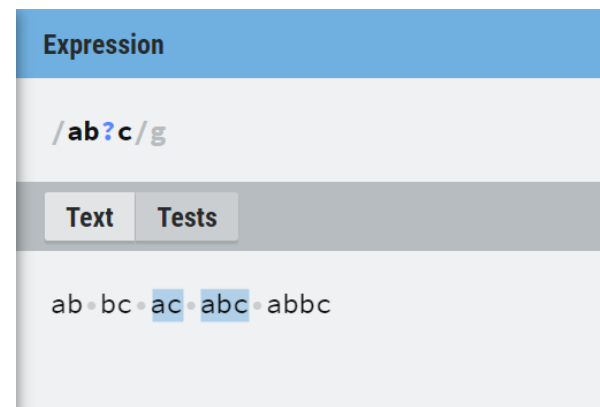
## 1회 이상 +

- 앞의 문자를 1개 이상 포함



## 0 또는 1회 ?

- 앞의 문자가 없을 수도 있음



# Regular Expression: Repetition

## 반복 횟수 지정 {m,n}

- m번 이상 n번 이하 반복
- 숫자가 하나만 주어졌을 경우
  - {m}: m번 반복
  - {m,}: m번 이상 반복
  - {,n}: n번 이하 반복

Expression	
<code>/0\d{1,2}-\d{3,4}-\d{4}/g</code>	
Text	Tests
<pre>010-1234-5678 051-123-4567 042-1234-526 02-444-1235 024-23-4343</pre>	

## Lazy Matching ?

- \* + { } 와 같은 반복 연산은 욕심쟁이
  - 최대한 길게 반복
- 최대한 짧게 매칭하기 위해선 뒤에 ? 삽입
  - \*?, +?, {n,m}?

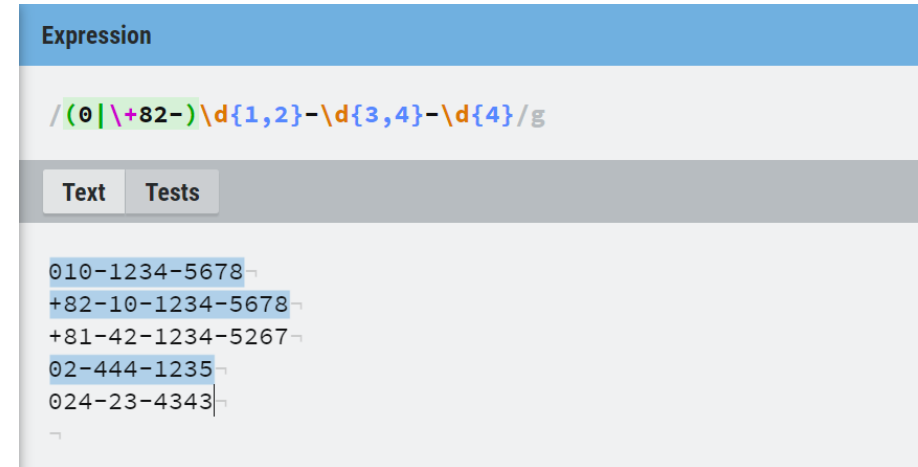
Expression	
<code>/&lt;.+&gt;/gm</code>	
Text	Tests
<pre>This is a &lt;div&gt; simple div &lt;/div&gt; test</pre>	

Expression	
<code>/&lt;.+?&gt;/gm</code>	
Text	Tests
<pre>This is a &lt;div&gt; simple div &lt;/div&gt; test</pre>	

# Regular Expression: Boundary

## 선택 |

- 여러 식 중 하나를 선택



Expression

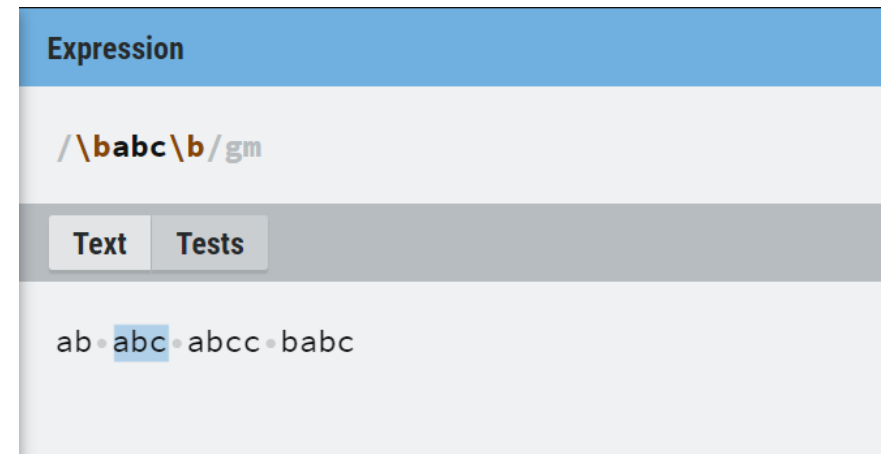
```
/(\d|\+82-)\d{1,2}-\d{3,4}-\d{4}/g
```

Text Tests

```
010-1234-5678
+82-10-1234-5678
+81-42-1234-5267
02-444-1235
024-23-4343
```

## 단어 경계 \b

- 단어의 경계 지점인지 확인



Expression

```
/\babc\b/gm
```

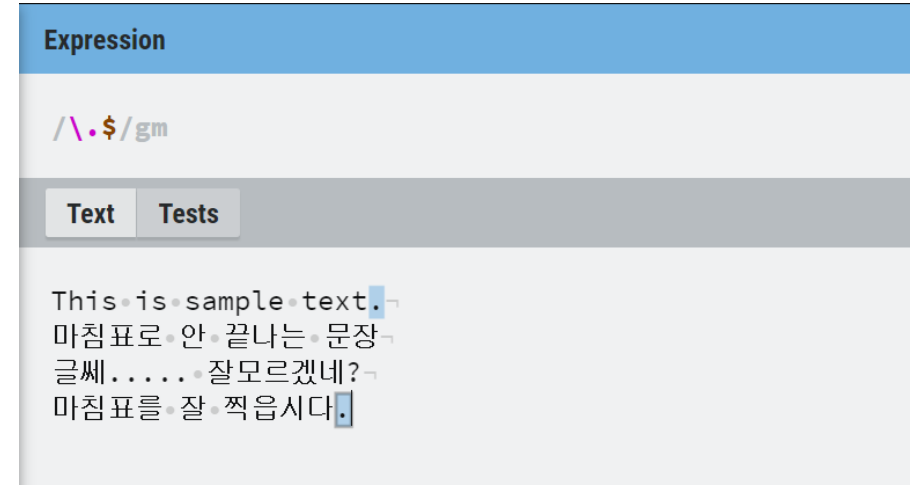
Text Tests

```
ab•abc•abcc•babc
```

# Regular Expression: Boundary

## 줄의 시작 ^

- 줄이나 문자열의 시작점
- Multiline flag 필요



Expression

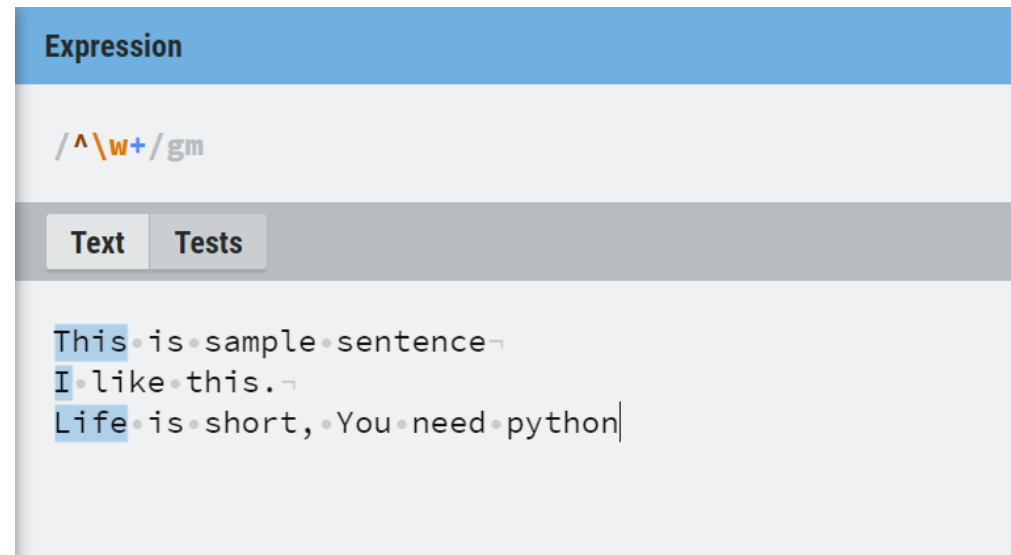
```
/\.$/gm
```

Text Tests

This is sample text.  
마침표로 안 끝나는 문장  
글쎄.....잘 모르겠네?  
마침표를 잘 찍읍시다.

## 줄의 끝 \$

- 줄이나 문자열의 끝
- Multiline flag 필요



Expression

```
/^w+/gm
```

Text Tests

This is sample sentence  
I like this.  
Life is short, You need python

# Regular Expression: Capture

## 그룹 캡처 ()

- 괄호이므로 우선 순위가 있음
- 해당 문자열을 캡처한다.
- 캡처된 텍스트를 불러올 수 있다
  - \1, \2, \3, ..., \[NUMBER]**

```
Expression
/(w{2,}).+\1/gm

Text Tests

tomato
abcde
one-to-one
btomatu
abcdebch
```

## 비 그룹 캡처 (?:)

- 괄호이므로 우선 순위가 있음
- 캡처를 하지는 않는다
- 그냥 괄호다

```
Expression
/(:0|+82-)\d{1,2}-(\d{4})-\1/gm

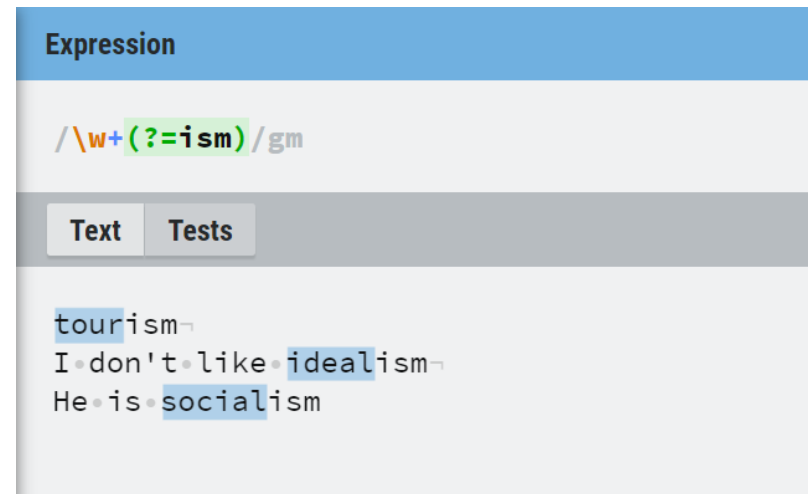
Text Tests

010-1234-1234
010-1234-5678
+82-10-5678-5678
+82-4123-1234
```

# Regular Expression: Condition

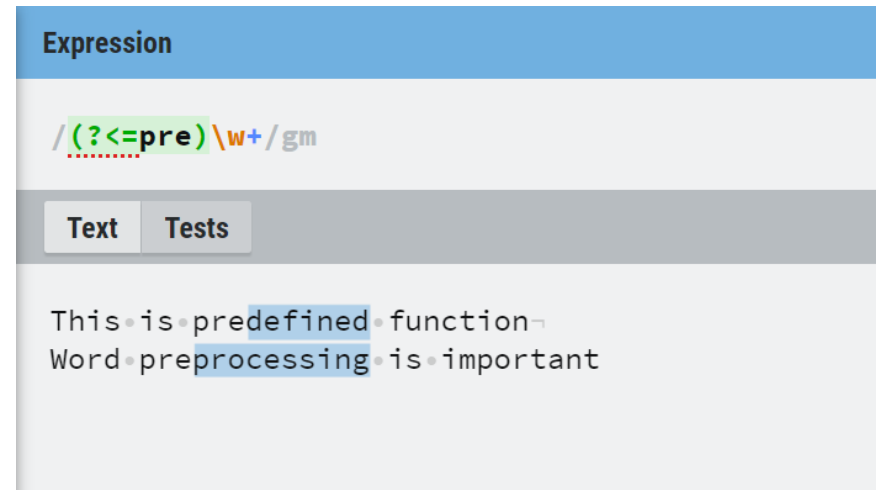
## 뒷 패턴 확인 $D(?=R)$

- R이 바로 뒤에 있는 D를 매칭
- R 부분은 포함되지 않음



## 앞 패턴 확인 $D(?<=R)$

- R이 바로 앞에 있는 D를 매칭
- R 부분은 포함되지 않음



# Regular Expression in Python

그래서 파이썬에서는 어떻게 쓰나요?

```
string = \
"""
010-1234-1234
010-1234-5678
+82-10-5678-5678
+82-4123-1234
"""

pattern = r'^(?:0|\+82-)\d{1,2}-(\d{4})-\d{1}$' # 패턴이 raw string이어야 함
```

→ 파이썬 표준 **re** 패키지를 사용

```
import re

for match in re.finditer(pattern, string, re.MULTILINE):
    print("전체 문자열", match.group(0))
    print(r"\1 문자열", match.group(1))
```

# Regular Expression in Python

## 탐색

```
# 처음 매칭되는 문자열 탐색  
match = re.search(pattern, string, re.MULTILINE)  
print(match.group(0))
```

## 모두 탐색

```
# 모든 매칭되는 문자열 탐색  
for match in re.finditer(pattern, string, re.MULTILINE):  
    print(match.group(0))
```

## 치환

```
# 매칭되는 패턴 치환  
repl = r"치환됨-\1"  
print(re.sub(pattern, repl, string, flags=re.MULTILINE))
```

## 나누기

```
# 매칭되는 패턴을 기준으로 나누기  
splited = re.split(pattern, string, flags=re.MULTILINE)  
print(splited)
```



# Regular Expression Compile

정규 표현식을 평가하는 것은 오래 걸림

```
for string in dataset:  
    match = re.search(pattern, string, re.MULTILINE)  
    print(match.group(0))
```



정규 표현식을 미리 컴파일 → 훨씬 빠름

```
compiled = re.compile(pattern, flags=re.MULTILINE)  
  
for string in dataset:  
    match = compiled.search(string)  
    print(match.group(0))
```