

# PYTHON PROGRAMMING

## LECTURE 12: IO

# Standard Input & Output

```
>>> print("This", "Sentence")
This Sentence

>>> print("This", "Sentence", sep=", ")
This, Sentence
```

```
>>> var = input()
입력을 받습니다.
>>> var
'입력을 받습니다.'
>>> var = input()
12345
>>> var
'12345'
```

## 표준 출력 (stdout)

- 따로 Redirection 없으면 콘솔 출력

## 표준 입력 (stdin)

- 따로 Redirection 없으면 콘솔 입력

# File Open

- 파이썬은 File Descriptor를 열기 위하여 **open** 내장 함수 사용

```
fd = open("<파일이름>", "<접근 모드>", encoding="utf8")    # 파일 열기
                                     # Unix 인코딩 기본 인자는 utf8
fd.close()                                                  # 파일 닫기
```

접근 모드	설명
r	읽기 모드 - 파일을 텍스트 형태로 읽을 때 사용
rb	이진 읽기 모드 - 파일을 바이너리 형태로 읽을 때 사용
w	쓰기 모드 - 파일을 텍스트 형태로 쓸 때 사용
wb	이진 쓰기 모드 - 파일을 바이너리 형태로 쓸 때 사용
a	추가 모드 - 파일의 마지막에 새로운 텍스트를 추가할 때 사용

## File Read

- Read 메소드로 파일 읽기 가능

```
fd = open("text.txt", "r")
contents = fd.read()
fd.close()

print(contents)
```

# 파일 전체 읽기

- File descriptor 닫는 것을 깜빡할 때가 많음
  - Context manager 형태로 사용 → 자동으로 닫아줌
  - **with <ContextManager> as <ReturnValue>** 구문

```
with open("text.txt", "r") as fd:
    contents = fd.read()

print(contents)
```

# Context Manager  
# 파일 전체 읽기

## File Read Lines

- 줄 단위로 잘라서 읽기, `\n`가 사라지는 건 아니다.

```
contents = []  
  
with open("text.txt", "r") as f:  
    for sentence in f:                # readline() 활용도 가능  
        contents.append(sentence)  
  
print(contents)
```

- 전체 읽어 줄단위로 잘라서 반환 → `readlines`
  - String List가 반환

```
with open("text.txt", "r") as f:  
    contents = f.readlines()          # 전체 읽기 후 줄 단위 자름  
  
print(contents)
```

## File Write

- Write 메소드로 파일 쓰기 가능

```
with open("text.txt", "w") as fd:
    for i in range(10):
        fd.write(f"{i + 1}번 째 문장\n")
```

- Writelines 메소드로 여러 줄 작성
  - 줄 바꿈 문자 \n 을 넣어주진 않음

```
with open("text.txt", "w") as fd:          # String Iterable으로 쓰기
    fd.writelines(f"{i + 1}번째 줄입니다\n" for i in range(10))
```

- 추가하기 모드 ("a")로 파일 뒤에 덧붙이기 가능

```
i = 10
with open("text.txt", "a") as fd:          # Append 모드
    fd.write("내용을 추가합니다\n")
    fd.writelines(f"{i + 1}번째 줄입니다\n" for i in range(i, i+10))
```

# Directory

- os 라이브러리로 플랫폼 독립적인 폴더 생성 가능
  - 파이썬에선 Windows, Unix 모두 “/”로 폴더를 나타냄
- path 라이브러리로 경로 관련 연산 가능

```
import os

os.mkdir("test")          # 폴더 하나 만들기, 이미 있으면 에러 발생

if not os.path.isdir("test"):  # 폴더가 있는지 확인
    os.mkdir("test")          # 폴더가 아니거나 없으면 False

# 하위 폴더 한번에 만들기, exist_ok 옵션으로 이미 있으면 무시할지 확인
os.makedirs("test/a/b/c", exist_ok=True)
```

## Listing Directory

- `listdir` 함수로 폴더내 파일/하위 폴더 검색

```
>>> import os  
  
>>> print(*[entry for entry in os.listdir('test')])  
a.txt b.txt
```

- `glob` 라이브러리로 유닉스 스타일 경로명 패턴 확장 적용

```
>>> import glob  
  
>>> print(*[entry for entry in glob.glob('test/*.txt')])  
test/a.txt test/b.txt
```



# Pickle

- 파이썬 객체를 그대로 저장하고 싶다면 → **Pickle**
  - 객체를 직렬화 (serialize)하여 파일로 저장

```
import pickle # pickle 라이브러리 import

seq = [[i * j for j in range(100)] for i in range(100)] # 저장하고 싶은 객체

with open("test.pkl", "wb") as fd:
    pickle.dump(seq, fd) # pickle 저장
del seq # 개체 삭제

with open("test.pkl", "rb") as fd:
    seq = pickle.load(fd) # pickle 불러오기
print(seq[12][9]) # 개체 접근 가능
```

## 장점

- 쓰기 쉽다
- 파이썬 개체를 그대로 저장

## 단점

- 파이썬에서만 읽을 수 있다.
- **보안 문제**가 있다
  - 신뢰할 수 있는 개체만 불러올 것!

# Class Pickling

```
import pickle

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

    def __add__(self, other):
        return MyComplex(
            self.real + other.real,
            self.imaginary + other.imaginary
        )
```

```
my_complex = MyComplex(3, -5)

with open("test.pkl", "wb") as fd:
    pickle.dump(my_complex, fd)
del my_complex

with open("test.pkl", "rb") as fd:
    my_complex = pickle.load(fd)

del MyComplex
with open("test.pkl", "rb") as fd:
    my_complex = pickle.load(fd) # 에러 발생!
```

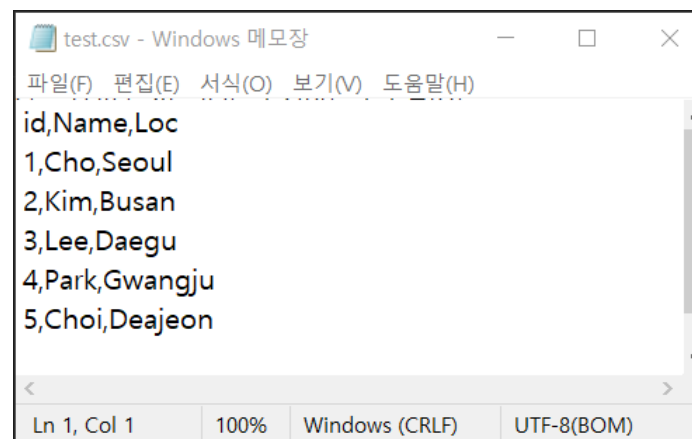
- Class 객체를 직렬하기 위해선 해당 클래스가 직렬화 가능 필요
  - 모든 속성 (attribute)가 직렬화 가능 필요
- 저장된 객체 pickle을 로드하고 싶다면 미리 해당 클래스 선언 필요
  - 해당 클래스 정보가 없다면 역직렬화 불가능

# CSV

## Comma Separate Values

- 표 데이터를 프로그램에 상관없이 쓰기 위한 데이터 형식
  - 필드를 쉼표(,)로 구분한 텍스트 파일
  - 탭(TSV), 공백(SSV) 등으로 구분하기도 함
  - 통칭하여 Character Separated Values (CSV)라 지칭
- Readlines로 읽을 수 있음 ← 구현이 귀찮음

	A	B	C
1	id	Name	Loc
2		1 Cho	Seoul
3		2 Kim	Busan
4		3 Lee	Daegu
5		4 Park	Gwangju
6		5 Choi	Deajeon



```
test.csv - Windows 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
id,Name,Loc
1,Cho,Seoul
2,Kim,Busan
3,Lee,Daegu
4,Park,Gwangju
5,Choi,Deajeon
Ln 1, Col 1 100% Windows (CRLF) UTF-8(BOM)
```

## Reading CSV

- csv 라이브러리로 쉽게 csv 읽기 쓰기 가능

```
import csv

with open('test.csv', 'r') as fd:
    reader = csv.reader(fd,                # File Descriptor, 필수
                        delimiter=',',     # 구분자, 기본: ,
                        quotechar='"',      # 텍스트 감싸기 문자, 기본: "
                        quoting = csv.QUOTE_MINIMAL  # Parsing 방식, 기본: 최소 길이
    )

    for entry in reader:                  # 한 줄 씩 순환
        print(entry)                     # Row를 List형태로 출력
```

## Writing CSV

- csv 라이브러리로 쉽게 csv 읽기 쓰기 가능

```
import csv

with open('test.csv', 'w') as fd:
    writer = csv.writer(fd,                # File Descriptor, 필수
                        delimiter=',',      # 구분자, 기본: ,
                        quotechar='"',      # 텍스트 감싸기 문자, 기본: "
                        quoting = csv.QUOTE_MINIMAL  # Parsing 방식, 기본: 최소 길이
    )

    writer.writerow(['id', 'label'])        # 한 줄 쓰기
    writer.writerows([I, f'label_{i}'] for I in range(10)) # 여러 줄 쓰기
```

# JSON

## JavaScript Object Notation

- 웹 언어인 Javascript 의 데이터 객체 표현 방식
  - 자료 구조 양식을 문자열로 표현
  - 간결하게 표현되어 사람과 컴퓨터 모두 읽기 편한
  - 코드에서 불러오기 쉽고 파일 크기 역시 작은 편
  - 최근 각광 받는 자료구조 형식
- 그럼에도 Parser 직접 작성은 매우 귀찮음

```
{
  "ID": null,
  "name": "Doe",
  "first-name": "John",
  "age": 25,
  "hobbies": [
    "reading",
    "cinema",
    {
      "sports": [
        "volley-ball",
        "snowboard"
      ]
    }
  ],
  "address": {
  }
```

Dictionary와  
비슷해 보인다

# Reading JSON

- json 라이브러리로 읽기 쓰기 가능

```
>>> import json

>>> with open('test.json', 'r') as fd:
...     data = json.load(fd)          # json 읽기
...

>>> print(data['hobbies'])           # Python 객체처럼 읽기
['reading', 'cinema', {'sports': ['volley-ball', 'snowboard']}]

>>> print(data['hobbies'][2]['sports'][0])
volley-ball
```

```
{
  "ID": null,
  "name": "Doe",
  "first-name": "John",
  "age": 25,
  "hobbies": [
    "reading",
    "cinema",
    {
      "sports": [
        "volley-ball",
        "snowboard"
      ]
    }
  ],
  "address": {}
}
```

test.json

# Writing JSON

- json 라이브러리로 읽기 쓰기 가능

```
import json

obj = {
    "ID": None,
    "bool": False,
    "hobbies": {
        "sports": [
            "snowboard",
            "volley-ball"
        ]
    }
}

with open('test.json', 'w') as fd:
    json.dump(obj, fd)    # json 쓰기
```

- 직렬화 가능 개체
  - 원시 타입
    - str
    - int
    - float
    - bool
    - None
  - 자료 구조
    - list
    - Dict
- 이 외에는 Decoder 작성 필요



# XML

## eXtensible Markup Language

- 데이터 구조와 의미를 설명하는 **태그**를 활용한 언어
  - **<태그>**와 **</태그>** 사이에 값이 표시
  - 문자열으로 처리
  - **<태그 속성=값>** 형태로 태그에 속성부여
  - HTML은 웹 페이지 표시를 위한 XML
  - 정규표현식으로 parsing 가능
- **HTML 파일을 읽어 웹 크롤러 제작 가능**

note.xml

```
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

```
<!DOCTYPE html>
<html lang="ko" data-
Edg/92.0.902.62">
  <head>...</head>
  <body>
    <div id="u_skip">
      <a href="#newsstand">...</a>
      <a href="#themecast">...</a>
      <a href="#timesquare">...</a>
      <a href="#shopcast">...</a>
      <a href="#account">...</a>
    </div>
    <div id="wrap">
      <style type="text/css">...</style>
      <div id="NM_TOP_BANNER" data-clk-prefix="top" class="_1hiMwemA" st
      <div id="header" role="banner">...</div>
      <div id="container" role="main">...</div>
      <div id="footer" role="contentinfo">...</div>
    </div>
```

# Beautiful Soup

- 파이썬 기본 **XML Parser**는 다소 불편
  - 일반적으로 XML, HTML 파싱을 위해 외부 라이브러리 사용
  - BeautifulSoup, xmltodict, ...
- **Beautiful Soup**
  - <https://www.crummy.com/software/BeautifulSoup/>
  - 가장 많이 쓰이는 parser 중 하나
  - HTML, XML 등 Markup 언어 Scraping을 위한 도구
  - 속도는 다소 느리나 간편하게 사용

```
(nlp) ➤ napping ➤ ~/test_nlp  
➤ conda install beautifulsoup4
```

# Reading XML

- Beautiful Soup로 XML 읽어 보기

```
from bs4 import BeautifulSoup          # BeautifulSoup 객체 들고오기

with open("test.xml", 'r') as fd:
    soup = BeautifulSoup(
        fd.read(),                      # Parsing할 문자열
        'html.parser'                  # 사용할 parser
    )

to_tag = soup.find(name='to')           # 문서 전체에서 "to" 태그 찾기
print(to_tag.string)                   # "to" 태그 내 문자열 출력

for cite_tag in soup.findAll(name='cite'): # "cite" 태그 모두 찾기
    print(cite_tag.string)

cites_tag = soup.find(name='cites')     # "cites" 태그 찾기
print(cites_tag.attrs)                  # "cites" 태그의 모든 속성
print(cites_tag['attr'])                 # "attr" 속성 값 참조

cites_tag = soup.find(attrs={'attr': 'name'}) # 속성으로 태그 찾기
for cite_tag in cites_tag.find_all(name='cite'): # 태그 내 검색
    print(cite_tag.string)
```

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
  <cites attr=name>
    <cite>Cho</cite>
    <cite>Lee</cite>
    <cite>Park</cite>
  </cites>
</note>
```

test.xml

# PYTHON PROGRAMMING

## LECTURE 13: SETTING & EXCEPTION & LOGGING

### 프로그램의 설정 값을 만들어 주고 싶다면?

- 실행할 때마다 필요한 설정 값
  - 딥러닝 학습 횟수 (Epoch), 학습 계수 (Learning rate)
  - 사용하는 GPU 개수

→ **Command Line Argument (명령행 인자)**로 입력
- 한번 설정하면 수정을 잘 안 하는 설정 값
  - 학습 자료 폴더 위치
  - 웹 서버의 Listening Port

→ **설정 파일에서 불러들이기**

# Command Line Argument

```
import sys  
print(sys.argv)
```

main.py

```
(nlp) napping ~/test_nlp  
python main.py  
['main.py']  
(nlp) napping ~/test_nlp  
python main.py arguments  
['main.py', 'arguments']  
(nlp) napping ~/test_nlp  
python main.py --options 1234  
['main.py', '--options', '1234']
```

- Console 창에서 프로그램 실행 시 프로그램에 넘겨주는 인자 값
- Command-line Interface (CLI)에서 흔히 쓰이는 방식
- 파이썬에선 `sys` 라이브러리의 `argv` 속성으로 접근
  - 공백 기준으로 잘라져 문자열 형태로 입력

→ 좀 더 쉽게 관리할 수 있을까?

# argparser

## argparser 라이브러리를 활용

- 인자 flag를 설정 가능하여 flag 별 입력 가능 (긴 flag, 짧은 flag 활용)
- 기본값 설정 가능
- Help 제공하여 사용자 편의 향상
- Type 설정 가능 (문자열에서 변환)
- 이 외 명령 줄 인자와 관련된 여러 도구 포함

```
import argparse

parser = argparse.ArgumentParser()
# parser.add_argument(<짧은 Flag>, <긴 Flag>)
parser.add_argument('-l', '--left', type=int) # 타입 설정
Parser.add_argument('-r', '--right', type=int)
Parser.add_argument('--operation',
                    dest='op', # 타겟 속성, 기본은 -- 없이
                    help="Set Operation", # 인자 설명
                    default='sum') # 기본 값

args = parser.parse_args()
print(args)

if args.op == 'sum': # 인자 접근
    out = args.left + args.right
elif args.op == 'sub':
    out = args.left - args.right
print(out)
```

```
(nlp) napping ~/test_nlp
> python main.py -l 1 -r 2
Namespace(left=1, right=2, op='sum')
3
(nlp) napping ~/test_nlp
> python main.py --left 3 --right 5 --operation=sub
Namespace(left=3, right=5, op='sub')
-2
(nlp) napping ~/test_nlp
> python main.py --help
usage: main.py [-h] [-l LEFT] [-r RIGHT] [--operation OP]

optional arguments:
  -h, --help            show this help message and exit
  -l LEFT, --left LEFT
  -r RIGHT, --right RIGHT
  --operation OP        Give Operation
(nlp) napping ~/test_nlp
>
```

# Config File

- 프로그램 실행 설정을 file에 저장함
- Section, Key, Value 값의 형태
  - **[Section]** – 설정 범주, **Key: Value** – 키: 값
  - **[DEFAULT]** – 기본 범주
- 이중 Dictionary 형태 – 모든 key, value가 str

```
{  
    "DEFAULT": {                # DEFAULT SECTION  
        "ServerAliveInterval": "45",  
        "Compression": "yes",  
        "CompressionLevel": "9",  
        "ForwardX11": "yes"  
    },  
    "bitbucket.org": {          # SECTION  
        "User": "hg"  
    },  
    "topsecret.server.com": {   # SECTION  
        "Port": "50022",  
        "ForwardX11": "no"  
    }  
}
```

setting.cfg - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

[DEFAULT]  
ServerAliveInterval = 45  
Compression = yes  
CompressionLevel = 9  
ForwardX11 = yes

[bitbucket.org]  
User = hg

[topsecret.server.com]  
Port = 50022  
ForwardX11 = no

Ln 12, Col 16 150% Windows (CRLF) UTF-8



# configparser

## configparser 라이브러리를 활용

- 파이썬 기본 Dictionary처럼 사용
- 기본적으로 모든 건 str 타입으로 처리

```
import configparser

config = configparser.ConfigParser()
config.read('test.cfg')          # 설정 불러오기

print(config.sections())        # 모든 범주 불러오기

port = config["topsecret.server.com"]["port"]
print(type(port), port)         # (str, 50022)
port = config["topsecret.server.com"].getint("port")
print(type(port), port)         # (int, 50022)

for name, section in config.items(): # dict 처럼 접근
    print(name)
    for key, value in section.items(): # dict 처럼 접근
        print(key, value)

with open("test.cfg", "w") as fd: # 설정 저장
    config.write(fd)
```

```
[DEFAULT]
serveraliveinterval = 45
compression = yes
compressionlevel = 9
forwardx11 = yes

[bitbucket.org]
user = hg

[topsecret.server.com]
port = 50022
forwardx11 = no
```

test.cfg

## Exception Handling

### 프로그램 실행 중에는 다양한 예외/에러가 발생

- 예외가 발생할 경우 대응 조치가 필요
  - 불러올 파일이 없는 경우 → 파일이 없음을 사용자에게 알림
  - 서버와 연결이 끊김 → 다른 서버로 Redirection
- 예외가 발생할 수 있는 코드 → (특정 예외 발생시) 대응코드 → 계속 진행

```
try:  
    <예외 발생 가능 코드>  
except <예외 클래스>:  
    <대응 코드>
```

## Exception Handling Example

# 0으로 숫자를 나누었을 때 예외 처리하기

```
for i in range(-5, 5):  
    try:                                # 코드 실행 블록  
        print(10 / i)  
    except ZeroDivisionError:          # 0으로 나누기 에러 발생 시  
        print("Zero Division, skip the number.")
```

## Built-in Exceptions

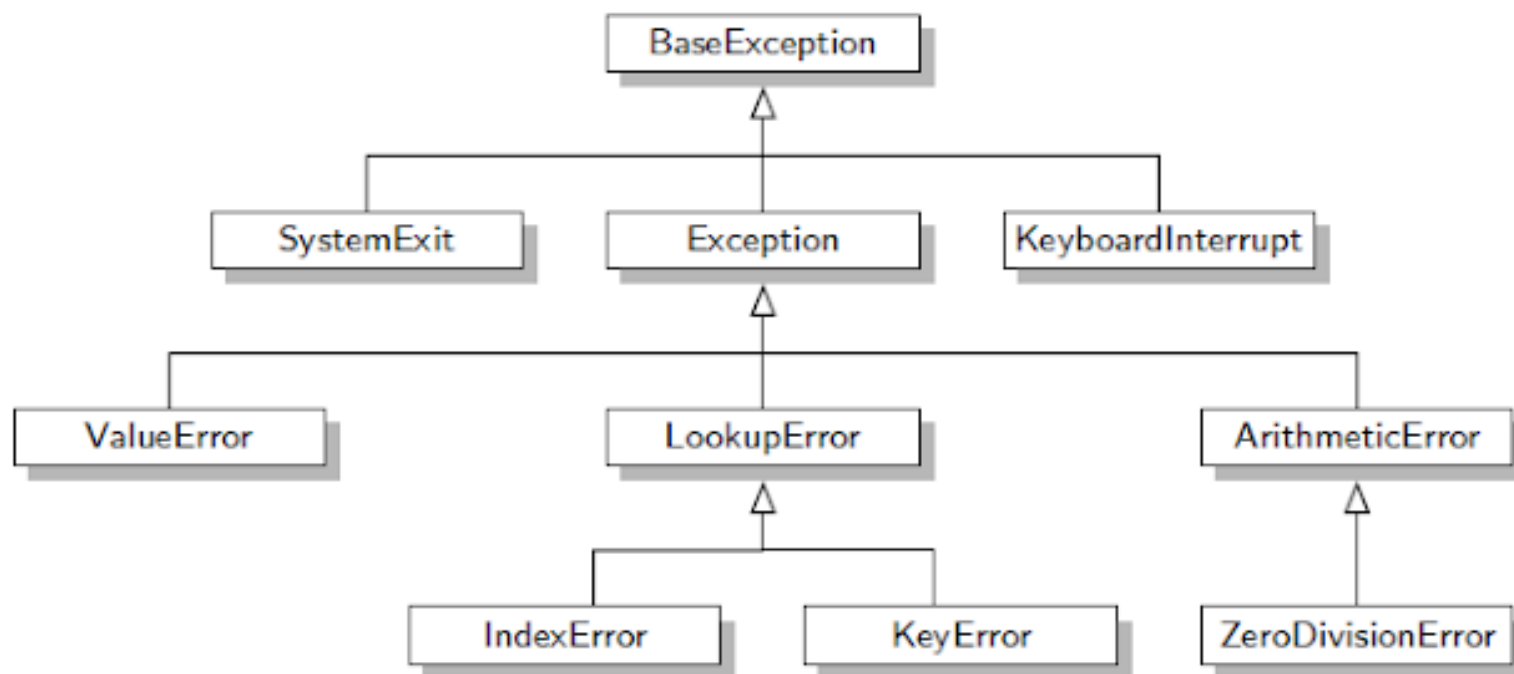
- 내장 예외: 기본 정의된 예외 클래스

예외 이름 (Exception Class)	설명	발생 가능 예시
IndexError	List의 Index 범위를 넘어감	<code>list[101]</code>
NameError	존재하지 않는 변수를 호출	<code>not_exist + 1</code>
ZeroDivisionError	0으로 숫자를 나눔	<code>10 / 0</code>
ValueError	변환할 수 없는 문자열/숫자를 변환	<code>float("abc")</code>
FileNotFoundError	존재하지 않는 파일 호출	<code>open("not_exist.txt", "r")</code>

- 이 외에도 매우 많은 내장 예외 존재

## Exception Class

- 파이썬 예외는 모두 **BaseException** 상속
  - 대부분 try로 최대 Exception단 까지만 잡음
  - Exception Class를 상속하여 새로운 예외 생성 가능



## Raising & Referencing Exceptions

- Raise 구문으로 예외 발생
  - **raise** <예외 객체>
- As 구문으로 잡힌 에러를 참조 가능
  - **except** <예외 클래스> **as** <예외 객체>

```
try:
    while True:
        value = input("A B C 중 하나를 입력하세요: ")

        if len(value) == 1 and value not in "ABC":
            raise ValueError("잘못된 입력입니다. 종료합니다.") # 예외 발생

        print("선택된 옵션:", value)

except ValueError as e: # as 이하 구문으로 예외 객체 들고오기 가능
    print(e)
```

# Assertion

- 조건을 확인하여 참이 아닐 시 **AssertionError** 발생
  - **assert** <조건>
  - **assert** <조건>, <에러 메시지>
- 에러 메시지가 없을 경우 빈 칸으로 처리

```
def add_int(param):  
    assert isinstance(param, int), "int만 됨"           # 조건 확인  
    return param + 1  
  
try:  
    print(add_int(10))  
    print(add_int('str'))  
  
except AssertionError as e:  
    print(e)                                             # "int만 됨" 출력
```

# Post-error Processing

```
try:
    functions()
except SomeError as e:
    print(e, "예외 발생")
print("예외 이후")
```

## 아무 구문 없음

- 일반 진행
- 예외 발생이 없을 경우
  - “예외 이후” 출력
- SomeError 발생
  - “예외 발생” 출력
  - “예외 이후” 출력
- 다른 예외 발생
  - 프로그램 비정상 종료

```
try:
    functions()
except SomeError as e:
    print(e, "예외 발생")
else:
    print("예외 이후")
```

## else 구문

- 예외가 없을 경우 진행
- 예외 발생이 없을 경우
  - “예외 이후” 출력
- SomeError 발생
  - “예외 발생” 출력
- 다른 예외 발생
  - 프로그램 비정상 종료

```
try:
    functions()
except SomeError as e:
    print(e, "예외 발생")
finally:
    print("예외 이후")
```

## finally 구문

- 모든 경우 진행
- 예외 발생이 없을 경우
  - “예외 이후” 출력
- SomeError 발생
  - “예외 발생” 출력
  - “예외 이후” 출력
- 다른 예외 발생
  - “예외 이후” 출력
  - 프로그램 비정상 종료



# Exception Handling Example

```
for i in range(5, -5, -1):
    try:
        value /= I

    except NameError:                # 참조 에러 처리
        print("No value on Value: Set 0")
        value = 10

    except ZeroDivisionError:        # 0 나누기 에러 처리
        print("Zero division: Skip")

    except Exception as e:          # 처리되지 않은 에러 처리
        print(type(e), e)
        raise e                     # 처리되지 않은 에러 재발생

    else:                            # 예외가 발생하지 않은 경우
        print(value)

    finally:                          # 모든 경우 출력
        print("Step")
```

## Logging

- **프로그램이 일어나는 동안 발생했던 정보를 기록**

- 결과 처리, 유저 접근, 예외 발생 ... 등
- 기록된 로그 분석을 통한 디버깅 & 유저 패턴 파악

- **기록 용도에 따른 차이**

- 용도에 따라 출력 형식 및 필터링 필요

- **어떻게 표출 할까?**

- 표준 에러 출력 – 일시적, 기록을 위해선 Redirection 필요, 구조화 필요
- 파일 출력 – 반 영구적, 매번 file description을 열고 닫아야 함

**체계적으로 로깅을 할 수는 없을까?**

# Logging Module

- 파이썬 기본 Logging 모듈
  - 상황에 따라 다른 Level의 로그 출력
  - DEBUG < INFO < WARNING < ERROR < Critical

```
import logging

logging.debug("디버깅")
logging.info("정보 확인")
logging.warning("경고")
logging.error("에러")
logging.critical("치명적 오류")
```

## Logging Level

Level	설명	예시
<b>DEBUG</b>	<ul style="list-style-type: none"> <li>상세한 정보, 보통 문제를 진단할 때만 사용</li> </ul>	<ul style="list-style-type: none"> <li>변수 A에 값 대입</li> <li>함수 F 호출</li> </ul>
<b>INFO</b>	<ul style="list-style-type: none"> <li>프로그램 정상 작동 중에 발생하는 이벤트 보고</li> <li>상태 모니터링이나 결함 조사</li> </ul>	<ul style="list-style-type: none"> <li>서버 시작</li> <li>사용자 User가 서버 접속</li> </ul>
<b>WARNING</b>	<ul style="list-style-type: none"> <li>예상치 못한 일이 발생하거나 가까운 미래에 발생할 문제에 대한 경고</li> <li>대처할 수 있는 상황이지만 이벤트 주목 필요</li> </ul>	<ul style="list-style-type: none"> <li>문자열 입력 대신 숫자 입력 → 숫자로 변환 뒤 진행</li> <li>인자로 들어온 리스트 길이가 안 맞음 → 적당히 잘라서 사용</li> </ul>
<b>ERROR</b>	<ul style="list-style-type: none"> <li>오류가 발생하였으나 프로그램은 동작 가능</li> <li>프로그램 일부 기능을 수행하지 못함</li> </ul>	<ul style="list-style-type: none"> <li>파일을 읽으려니 파일이 없음 → 사용자에게 알림</li> <li>외부 서버와 연결이 불가능 → 사용자에게 대체 서버 요청</li> </ul>
<b>CRITICAL</b>	<ul style="list-style-type: none"> <li>심각한 오류 발생</li> <li>프로그램 자체가 계속 실행되지 않을 수 있음</li> </ul>	<ul style="list-style-type: none"> <li>중요 파일이 없음</li> <li>사용자가 강제 종료</li> </ul>

# Root Logging

## 기본 설정된 로깅 – Root 로깅

- Basic config 로 간단하게 설정 가능
  - 로그를 기록할 파일 이름
  - 로그 레벨을 설정하여 특정 레벨 이상 출력
- 기본설정
  - 표준 에러 출력
  - WARNING 이상 출력

```
import logging

logging.basicConfig(          # 로깅 설정
    filename='test.log',      # 기록할 파일
    level=logging.INFO        # 로그 레벨 설정
)

logging.debug("이 메시지는 기록이 안됨")
logging.info("이 메시지는 기록이 됨")
logging.error("이 메시지 역시 기록이 됨")
```

```
INFO:root:이 메시지는 기록이 됨
ERROR:root:이 메시지 역시 기록이 됨
```

test.log

# Logger Management

## 새로운 Logger 생성

- getLogger로 새로운 이름의 Logger 생성
  - 이름이 같은 Logger가 존재할 경우 해당 객체를 들고 옴
  - 따로 설정이 되어 있지 않을 경우 Root의 설정을 상속함

```
import logging

logging.basicConfig(
    filename='test.log',
)

# logger = logging.getLogger("main") # 새로운 logger 생성
logger = logging.getLogger(__name__) # 일반적으로 모듈 별로 이름을 만든다
logger.setLevel(logging.INFO)        # 새 logger의 레벨 설정

logging.info("Root에 info 기록")      # Root에 리록
logging.warning("Root에 Warning 기록")

logger.info("메인에서 info 기록")     # 새로 만든 logger에 기록
logger.warning("메인에서 Warning 기록")
```

```
WARNING:root:Root에 Warning 기록
INFO:__main__:메인에서 info 기록
WARNING:__main__:메인에서 Warning 기록
```

test.log

# PYTHON PROGRAMMING

## LECTURE 14: WEB

# Web Page

## 인터넷 웹사이트 들은 어떻게 보여지는가?

네이버를 시작페이지로 > | [주니어네이버](#) [해피빈](#)



메일 카페 블로그 지식iN 쇼핑 쇼핑LIVE Pay TV 사전 뉴스 증권 부동산 지도 VIBE 책 웹툰 더보기 >

27.0° 맑음 25.0° / 35.0° 정자동

연합뉴스 > 습도 높아 체감온도 35도 이상...곳곳 돌풍 동반 소나기

TOKYO 2020 · 네이버뉴스 · 연예 스포츠

뉴스스탠드 > 구독한 언론사 · 전체언론사



파이낸셜 뉴스	Korea JoongAng Daily	세계일보	한국경제TV	ZNet Korea	경향신문
시사iN	Insight	조선일보	kbc 광주방송	Daum 디지털데일리	MBC
OBS	N24 뉴스투데이	KBS	연합뉴스	매일노동뉴스	식품저널 foodnews
코메디닷컴	노동법률	텐아시아	연합뉴스	HelloDD	RBS 한국농어촌방송

네이버를 더 안전하고 편리하게 이용하세요

NAVER 로그인

아이디 · 비밀번호찾기

회원가입

증시 | 나스닥 14,794.08 ▲114.58 +0.78%



다크 모드로 보기



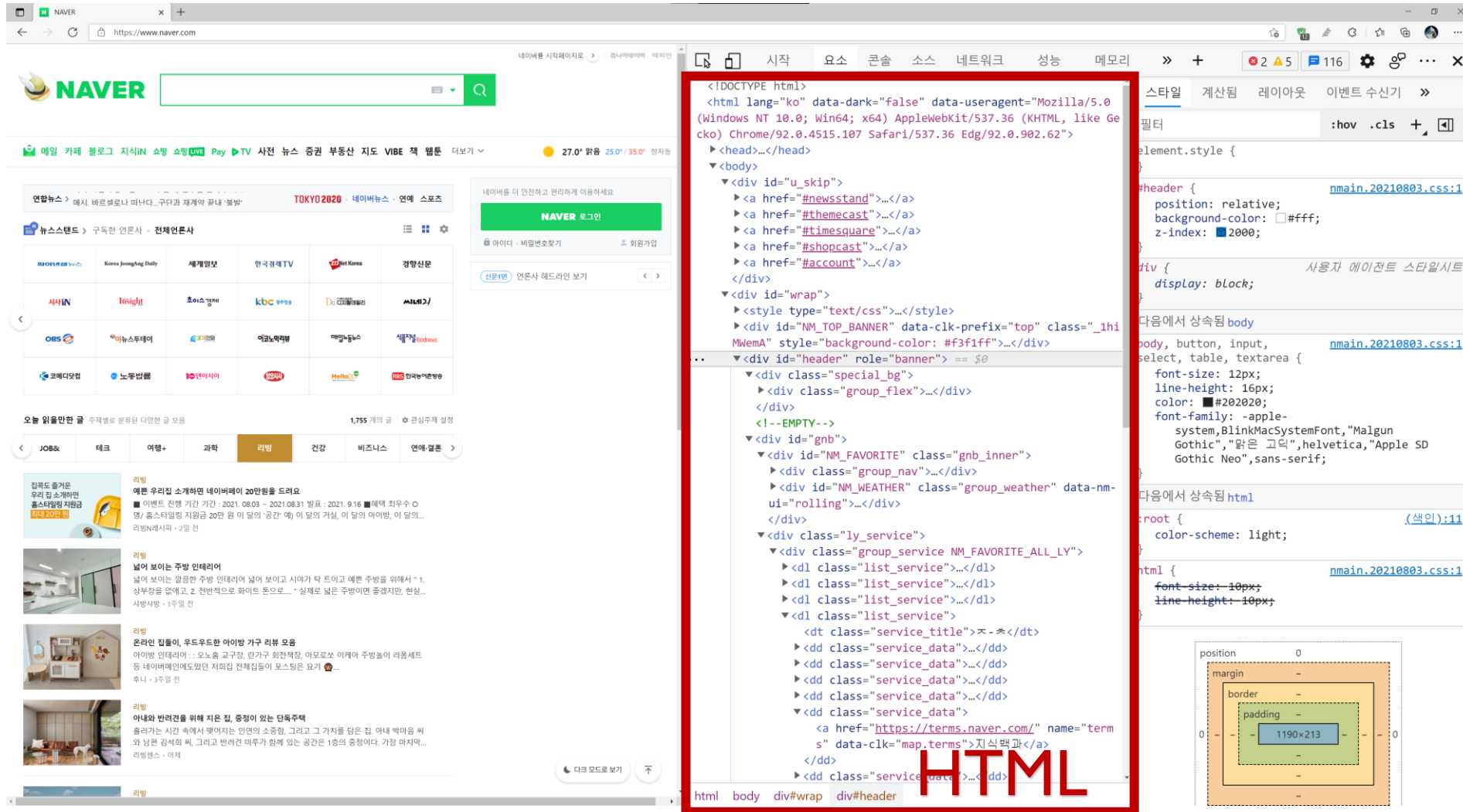


# HTML

- XML형태로 웹 페이지의 구조를 표현
  - BeautifulSoup 등 XML parser로 해석 가능
- 다운받은 HTML 파일을 웹 브라우저가 해석 & 화면 표시

```
<!doctypehtml>
<html>
  <head>
    <title>Hello HTML</title>
  </head>
  <body>
    <p>Hello World!</p>
  </body>
</html>
```

# Development Tool



F12를 눌러  
개발자 모드

# Requests

- 웹페이지를 읽기 위해서 일반적으로 Requests 라이브러리 사용

```
(nlp_test) ➤ napping ➤ ~/test_nlp  
➤ conda install requests
```

```
import requests  
  
URL = 'https://www.naver.com'  
response = requests.get(URL)      # GET으로 접근  
  
print(response.status_code)      # 결과 코드, 200이면 정상이라는 뜻  
print(response.text)             # 응답, 웹서버의 경우 HTML 코드
```

# Crawling

- 오늘자 네이버 스포츠 뉴스 기사 제목 크롤링하기

```
import requests
from bs4 import BeautifulSoup

URL = "https://sports.news.naver.com/index" # 네이버 스포츠 뉴스
response = requests.get(URL)

soup = BeautifulSoup(                                # HTML Parsing
    response.text,
    'html.parser'
)

headline = soup.find(name='ul', attrs={'class': 'today_list'})
for title in headline.find_all(name='strong', attrs={'class': 'title'}):
    print(title.string)
```

**Thank You for Listening!**