SETTING & EXCEPTION & LOGGING

Programming Setting

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 (txt, YAML, ConfigParser)

Command Line Argument

```python
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 없을까?

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

```python
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="Give 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
```

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

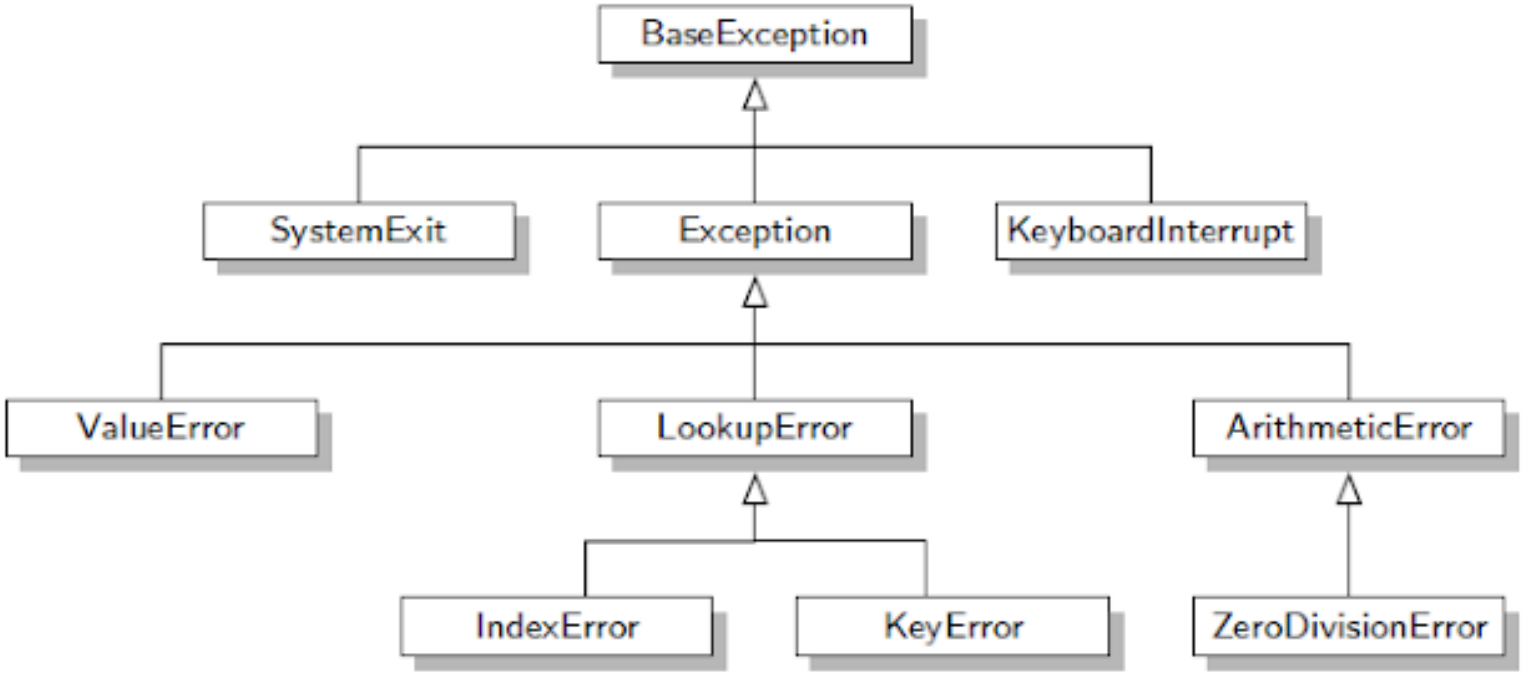
예외 이름 (Exception Class)	설명	발생 가능 예시
IndexError	List의 Index 범위를 넘어감	list[101]
NameError	존재하지 않는 변수를 호출	not_exist + 1
ZeroDivisionError	0으로 숫자를 나눔	10 / 0
ValueError	변환할 수 없는 문자열/숫자를 변환	float("abc")
FileNotFoundError	존재하지 않는 파일 호출	open("not_exist.txt","r")

- 이 외에도 매우 많은 내장 예외 존재

https://docs.python.org/ko/3/library/exceptions.html

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 **참**이 아닐 시 **AssertError** 발생
  - **assert** <조건>
  - **assert** <조건>,<에러 메세지>
- 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
DEBUG	• 상세한 정보, 보통 문제를 진단할 때만 사용	• 변수 A에 값 대입 • 함수 F 호출
INFO	• 프로그램 정상 작동 중에 발생하는 이벤트 보고 • 상태 모니터링이나 결함 조사	• 서버 시작 • 사용자 User가 서버 접속
WARNING	• 예상치 못한 일이 발생하거나 가까운 미래에 발생할 문제에 대한 경고 • 대처할 수 있는 상황이지만 이벤트 주목 필요	• 문자열 입력 대신 숫자 입력 → 문자로 변환 뒤 진행 • 인자로 들어온 리스트 길이가 안 맞음 → 적당히 잘라서 사용
ERROR	• 오류가 발생하였으나 프로그램은 동작 가능 • 프로그램 일부 기능을 수행하지 못함	• 파일을 읽으려니 파일이 없음 → 사용자에게 알림 • 외부 서버와 연결이 불가능 → 사용자에게 대체 서버 요청
CRITICAL	• 심각한 오류 발생 • 프로그램 자체가 계속 실행되지 않을 수 있음	• 중요 파일이 없음 • 사용자가 강제 종료

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

```python
import logging

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

logging.debug("이 메세지는 기록이 안됨")
logging.info("이 메세지는 기록이 됨")
logging.error("이 메세지 역시 기록이 됨")
```

```
INFO:root:이 메세지는 기록이 됨
ERROR:root:이 메세지 역시 기록이 됨
```

test.log

Logger Management

새로운 Logger 생성

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

```
import logging

logging.basicConfig(                    # Root 로깅 설정
    filename='test.log',                # 기록할 파일
)

# 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