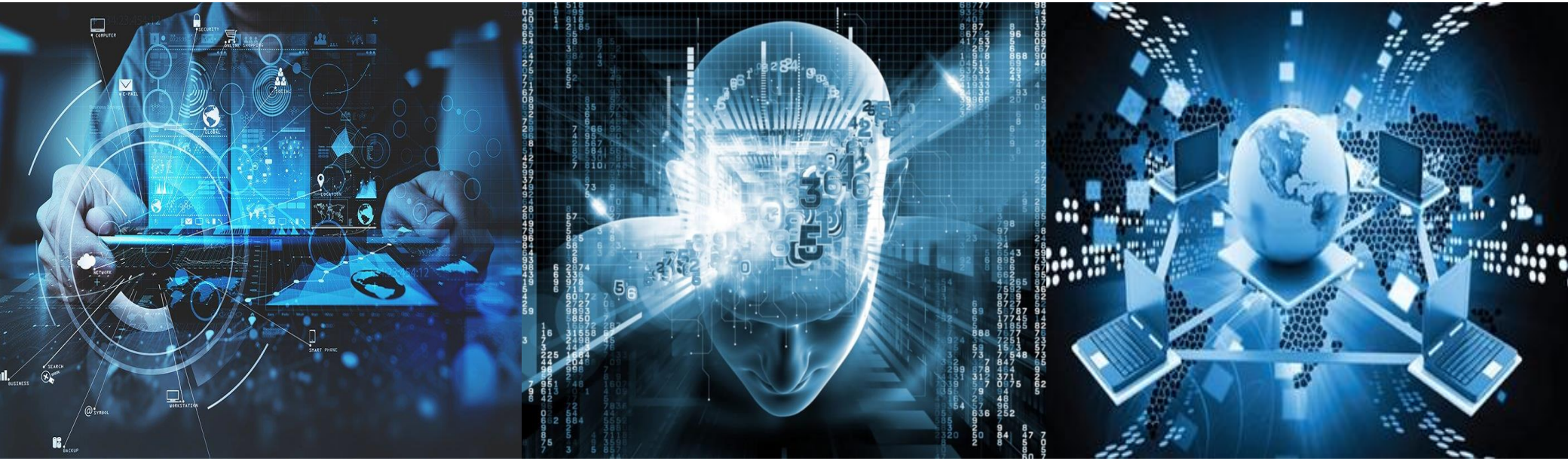
한양대학교 HANYANG UNIVERSITY

비파이 크래프트 Find Information to design valuable society



Data Analytics Based Python

SECT12. 입/출력 그리고 로깅

Innovation Growth Intensive Training
Kim Jin Soo

학습내용

- 포맷을 위한 문자열행 메소드. format( )
- 다양한 포맷에 대한 예제
- 로깅(Logging)의 이해
- 로깅 라이브러리 사용법
- 로깅 제대로 사용하기
- 설정 파일을 활용한 로깅

포맷을 위한 문자열행 메소드. format( )

❖ 표준 출력문의 형태는 다양하다.

- 출력되는 내용들은 소스 코드를 작성하는 개발자나 사용자에게 친숙하면서도 보기가 좋아야 한다.
- 특히, 소스 코드에서 변형한 변수 값을 확인하기 위한 출력문은 무척 흔하게 쓰는 출력문
- 보통, 소스 코드에 문제가 생겼을 때 본인이 작성한 소스 코드의 변수 값들을 차례차례 출력하는 디버깅(Debugging)은 개발자의 필수작업

❖ format( ) 메소드

- 문자열형의 메소드(내장함수)
- 중괄호 기호({ }) 에 함수의 인자값이 들어간다.
- 중괄호 기호({ })의 순서와 format( )의 인자값의 순서에 맞게 값이 대입되어서 출력
- 인덱스 순서를 변경하여 인자 값에 들어온 변수들의 출력 순서가 변경
- 정수형 인덱스 뿐만이 아니라 변수명을 활용하여 표현도 가능

다양한 포맷에 대한 예제

❖ format( ) 인자 값들의 값 출력 포맷을 제어하는 방법

- 문자열이나 튜플, 리스트 형 데이터들을 자동으로 언패킹 하여 값을 출력할 수 있다.
- format( )의 인자값으로 대입할 때 별표기호(*)를 데이터 형 앞에 붙여주면 된다.
- 인자 값의 색인과 중괄호 기호([ ])를 활용하여 튜플이나 리스트, 사전형의 항목을 추출할 수 있다.
- format( )에는 좌측, 우측, 중앙 정렬을 위한 기능도 제공한다.
- 콜론기호(:)는 글자 포맷을 명시하겠다는 의미

숫자 출력 포맷

❖ 숫자 출력 포맷 타입 설명

타입	설명
b	2진수
c	문자
d	10진수
o	8진수 (접두어 : 0o)
x	16진수, 9보다 큰 수에 대해서는 소문자 알파벳 사용 (접두어: 0x)
X	16진수, 9보다 큰 수에 대해서는 대문자 알파벳 사용 (접두어: 0X)
n	숫자. 기본적으로는 'd'와 같음. 하지만, 각 나라에 맞는 숫자 구분자를 집어 넣기 위히서 locale 사용
None	아무 것도 넣지 않은 경우. 'd'와 같음.

문자열 포맷

❖ 파이썬 문자열 포맷 문법 공식 매뉴얼

6.1.3. Format String Syntax

The `str.format()` method and the `Formatter` class share the same syntax for format strings (although in the case of `Formatter`, subclasses can define their own format string syntax).

Format strings contain "replacement fields" surrounded by curly braces `{}`. Anything that is not contained in braces is considered literal text, which is copied unchanged to the output. If you need to include a brace character in the literal text, it can be escaped by doubling: `{{` and `}}`.

The grammar for a replacement field is as follows:

```
replacement_field ::=  "{" [field_name] ["!" conversion] [":" format_spec] "}"
field_name        ::=  arg_name ("." attribute_name | "[" element_index "]")*
arg_name          ::=  [identifier | integer]
attribute_name    ::=  identifier
element_index     ::=  integer | index_string
index_string      ::=  <any source character except "]"> +
conversion        ::=  "r" | "s" | "a"
format_spec       ::=  <described in the next section>
```

In less formal terms, the replacement field can start with a *field_name* that specifies the object whose value is to be formatted and inserted into the output instead of the replacement field. The *field_name* is optionally followed by a *conversion* field, which is preceded by an exclamation point `'!'`, and a *format_spec*, which is preceded by a colon `':'`. These specify a non-default format for the replacement value.

See also the Format Specification Mini-Language section.

The *field_name* itself begins with an *arg_name* that is either a number or a keyword. If it's a number, it refers to a positional argument, and if it's a keyword, it refers to a named keyword argument. If the numerical arg_names in a format string are 0, 1, 2, ... in sequence, they can all be omitted (not just some) and the numbers 0, 1, 2, ... will be automatically inserted in that order. Because *arg_name* is not quote-delimited, it is not possible to specify arbitrary dictionary keys (e.g., the strings `'10'` or `':-]'`) within a format string. The *arg_name* can be followed by any number of index or attribute expressions. An expression of the form `'.name'` selects the named attribute using `getattr()`, while an expression of the form `'[index]'` does an index lookup using `__getitem__()`.

Changed in version 3.1: The positional argument specifiers can be omitted, so `'{} {}'` is equivalent to `'{0} {1}'`.

Some simple format string examples:

```
"First, thou shalt count to {0}" # References first positional argument
"Bring me a {}"                  # Implicitly references the first positional argument
"From {} to {}"                  # Same as "From {0} to {1}"
"My quest is {name}"             # References keyword argument 'name'
"Weight in tons {0.weight}"      # 'weight' attribute of first positional arg
"Units destroyed: {players[0]}"  # First element of keyword argument 'players'.
```

< https://docs.python.org/3.4/library/string.html#format-string-syntax >

로깅(Logging)의 이해

❖ 로깅(Logging)이란?

- 소프트웨어가 실행될 때 일어나는 이벤트들을 추적하기 위한 용도로 소스코드에 집어 넣는 출력문을 작성하는 행위
- 이러한 출력문을 로그라고 부른다.
- 로그파일은 문제가 발생하였을 때 원인 분석을 하기 위한 중요한 데이터로 활용
- 그렇기 때문에 여러 사람과 함께 개발하는 프로젝트에서 아주 중요

중요도에 따른 5가지 로그레벨

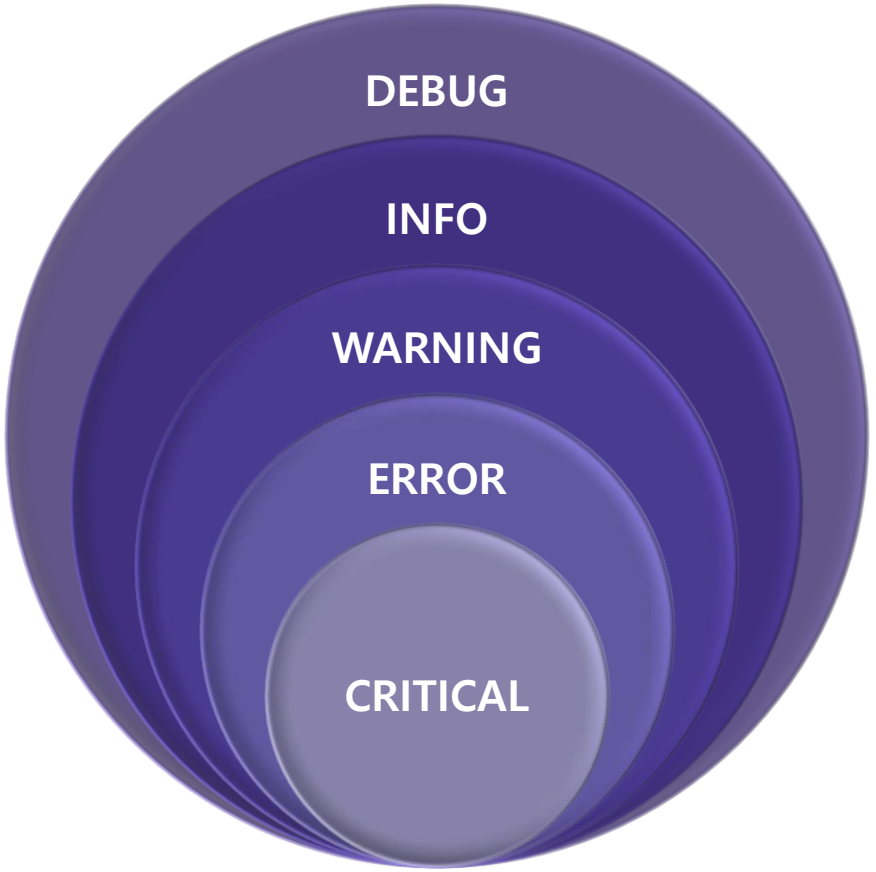
❖ 로그레벨 및 상세 설명

타입	설명
DEBUG	상세한 정보, 통상적으로 문제를 분석할 때에만 필요한 로그
INFO	소스 코드가 기대대로 동작하는지 확인하기 위한 로그
WARING	예상치 못한 상황이 벌어질 조짐이 보이거나, 곧 일어날 문제를 미리 알려주기 위한 로그. SW는 잘 동작하고 있는 상황임. Eg. 디스크 공간 부족
ERROR	Warring 보다 더 심각한 문제로 인하여 소프트웨어의 일부 기능이 제대로 동작하지 않을 때 필요한 로그
CRITICAL	가장 심각한 상황으로 인해 소프트웨어 자체가 장애로 인해 더 이상 실행할 수 없을 때 필요한 로그

로그 레벨의 관계

❖ 로그 레벨의 관계 도식화

- DEBUG < INFO < WARNING < ERROR < CRITICAL



로깅 라이브러리 사용법

❖ 레벨에 따른 로깅을 어떤식으로 하는지 알아보자

- 파이썬의 기본 라이브러리는 로깅을 하기 위한 모듈인 logging 제공

```
Python 3.4.4 Shell
File  Edit  Shell  Debug  Options  Window  Help
>>> import logging
>>>
>>> logging.warning('조심하세요')     # WARNING 메시지 출력
WARNING:root:조심하세요
>>>
>>> logging.info('정보 확인하세요!')  # INFO 메시지 출력
>>>
>>>
                                                    Ln: 113  Col: 4
```

- 파이썬의 기본 로거인 root의 로그 레벨은 WARNING이다.
  - Root 로거의 로그 레벨을 변경한 뒤 다시 INFO 레벨의 메시지 출력

```
Python 3.4.4 Shell
File  Edit  Shell  Debug  Options  Window  Help
>>>
>>>
>>> # 로그 레벨 변경, WARNING -> DEBUG
>>> logging.root.setLevel(logging.DEBUG)
>>> logging.info('정보 확인하세요!')
INFO:root:정보 확인하세요!
>>>
>>>
                                                    Ln: 121  Col: 4
```

로깅 제대로 사용하기

❖ 로깅 모듈의 객체들의 역할

- 로거, Logger
  - 소스 코드에서 바로 호출하여 사용할 수 있는 인터페이스 제공
  - 일반적으로 파이썬 모듈 단위로 로거를 생성
  - 최상단에 root 로거 밑으로 계층 구조로 부모 자식 관계를 형성
- 핸들러, Handlers
  - 로거에 의해 생성된 로그 레코드(LogRecord)를 적절한 곳에 출력
  - 로그는 콘솔이나 파일, 데이터베이스 등에 저장할 수 있다.
  - 로거는 여러 개의 핸들러를 보유할 수 있고, 핸들러는 반드시 특정 로거 객체에 귀속된다
- 포매터, Formatters
  - 로그의 출력 포맷을 결정한다.

로거 사용 예제

❖ 로그 사용 예제 및 실행 결과

logger_example.py - C:/Users/user/Dropbox/P10_Python/project/dk_idle/logger_example.py ...

File Edit Format Run Options Window Help

```
### IDLE 에서 <Ctrl> + <N> 키를 눌러서 새창에서 작업하세요!! ###

# 로깅 모듈 탑재
import logging

# 콘솔 출력용 핸들러 생성
handler = logging.StreamHandler()

# 포매터 생성
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)-8s - %(message)s')

# 핸들러에 포매터 설정
handler.setFormatter(formatter)

# 로거 생성 및 로그 레벨 설정
logger = logging.getLogger(__name__)            # 로거 생성
logger.setLevel(logging.INFO)                   # 로그 레벨설정 : INFO
logger.addHandler(handler)                      # 해당 로거에 핸들러 추가

# 로그 메시지 출력
logger.debug('이 메시지는 개발자만 이해해요.')          # DEBUG 로그 출력
logger.info('생각대로 동작 하고 있어요.')               # INFO 로그 출력
logger.warn('곧 문제가 생길 가능성이 높습니다.')        # WARNING 로그 출력
logger.error('문제가 생겼어요.기능이 동작 안해요.')     # ERROR 로그 출력
logger.critical('시스템이 다운됩니다!!!!')             # CRITICAL 로그 출력
```

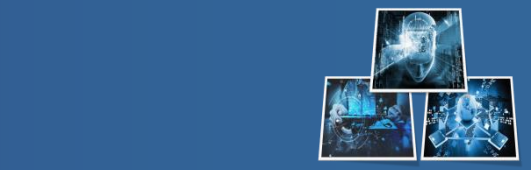
Python 3.4.4 Shell

File Edit Shell Debug Options Window Help

```
>>>
>>>
>>>
 RESTART: C:/Users/user/Dropbox/P10_Python/project/dk_idle/logger_example.py
2016-06-28 14:25:12,086 - __main__ - INFO     - 생각대로 동작 하고 있어요.
2016-06-28 14:25:12,096 - __main__ - WARNING  - 곧 문제가 생길 가능성이 높습니다.
2016-06-28 14:25:12,099 - __main__ - ERROR    - 문제가 생겼어요.기능이 동작 안해요.
2016-06-28 14:25:12,104 - __main__ - CRITICAL - 시스템이 다운됩니다!!!!
>>>
```

Ln: 144 Col: 4

설정 파일을 활용한 로깅



- 전체 키 값 설정, 로거, 핸들러, 포매터의 설정 파일 이름 정의
- root 로거 설정
- 자체 로거 설정
- 핸들러 설정
- 포매터 설정

Wrap-up

- 프로그램 상에 표준 출력문의 값들을 다양한 포맷으로 출력하는 방법에 대해서 알아보았다.
- 단순히 표준 출력문을 사용하는 것보다 더 나아가 로깅의 개념이 무엇인지 살펴보았다.
- 파이썬에서 제공하는 로깅 모듈의 사용법을 알아보았다.
- 설정 파일을 사용함으로써, 소스 코드의 변경 없이 다양한 로그 레벨의 로깅을 할 수 있는 기본적인 방법을 살펴보았다.
- 해당 로그를 파일로 남기는 방법도 살펴보았다.
- 소스코드 작성시 소스코드의 추적을 쉽게 하기 위해 어떤식으로 로그를 남기고, 적절한 로그레벨을 설정하는 방법을 훈련하도록 하자.