

Style Guide for Python Code

`mailofnsh@gmail.com`

Python Style code의 이해

- Pythonic style code 지켜야 하는 이유
 - Pythonic coding: Pythonic 사고체계에 맞춰진 문법양식
 - Pythonic 하지 못한 사고로 만든 코드는 쓰레기다
 - 기본철학 '인간의 시간이 컴퓨터의 시간보다 더 중요하다'으로 작성하는 Coding를 최대한 줄이면서 목표를 달성할 수 있는 문법 체계
 - Standard rule: Opensource부터 Third-party Library, Standard Library에 이르기 까지 Python style code로 작성, 가독성 개선 (의도파악)
 - Time saving: Pythonic programming으로 코드 간결, Code 시간 절약

Python Style code의 이해

- Python Enhance Proposal 형식

- Standard track: 개선 또는 새로운 특징을 기술. 후속 PEP가 미래버전의 표준라이브러리를 지원 하기 전에, Python버전에 대한 표준라이브러리 이외에 지원될 상호운용표준 기술
- Informational: 설계 이슈, Community정보, Guideline 제공. 사용자와 구현자는 이 문서는 무시해도 괜찮음
- Process: 업무절차, 업무절차의 변경 기술함. 구현도 제안 될 수는 있지만, Codebase는 포함하지 않음.



Standard
track

Informational

Process

Python Style code의 이해

- PEP8 - Style guide for Python code
 - 표준문서 Python Enhancement Proposal 8(PEP 8)은 Python Style code를 정의

Python >>> Python Developer's Guide >>> PEP Index >>> PEP 8 -- Style Guide for Python Code	
<h2>PEP 8 -- Style Guide for Python Code</h2>	
PEP:	8
Title:	Style Guide for Python Code
Author:	Guido van Rossum < guido@python.org >, Barry Warsaw < barry@python.org >, Nick Coghlan < ncoghlan@gmail.com >

Coding conventions for the Python code comprising the standard library in the main Python distribution.

Python Style code의 이해

- Python style code의 개념
 - Python style 다운 코드를 작성하는 기법
 - 특별한 문법이 아니라, Python이 기본적으로 제공하는 문법들을 활용하여 Coding하는 것이 바로 Python Style code다.

여러 단어를 붙이는 경우 Pythonic style code

```
>>> colors = ['red', 'blue', 'green', 'yellow']
```

```
>>> result = ''
```

```
>>> for s in colors:
```

```
    result += s
```

```
    print(result)
```

```
redbluegreenyellow
```

```
>>> result = ''.join(colors)
```

```
    print(result)
```

```
redbluegreenyellow
```

Python Style code의 이해

1. Code layout: Code 작성규칙을 정의

1. 들여쓰기는 공백 4개로 한다.
2. 소괄호, 중괄호, 대괄호 사이에 추가 공백을 입력금지
3. 한 줄의 최대 글자는 79자로 한다.
4. File encoding은 UTF-8 또는 ASCII
5. One Line One Import
6. Standard, 3rd Party, Local libraries 순서로 import

2. Naming Convention: 명명규칙

3. Object oriented programming: 객체지향 코드 작성규칙 정의
4. List Comprehension, Map-Reduce...: Python 특화 함수사용법 정의
5. Recommendations: 기타 추천 사항

Python Style code의 이해

- string library에서 표준규칙 확인하기

6.1. `string` — Common string operations

Source code: [Lib/string.py](#)

See also: [Text Sequence Type — str](#)

[String Methods](#)

6.1.1. String constants

The constants defined in this module are:

`string.ascii_letters`

The concatenation of the `ascii_lowercase` and `ascii_uppercase` constants described below. This value is not locale-dependent.

`string.ascii_lowercase`

The lowercase letters `'abcdefghijklmnopqrstuvwxyz'`. This value is not locale-dependent and will not change.

`string.ascii_uppercase`

The uppercase letters `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. This value is not locale-dependent and will not change.

Python Style code 이해

- PEP8 규칙을 지킨 부분, 그렇지 않은 부분을 찾아보세요
 - 무조건 지켜야 하는 법이 아니라, 도움이 되는 핵심을 인지 해야함

```
class string (metaclass=_TemplateMetaclass):
    """A string class for supporting $-substitutions."""
    delimiter = '$'
    # r'[a-z]' matches to non-ASCII letters when used with IGNORECASE,
    # but without ASCII flag. We can't add re.ASCII to flags because of
    # backward compatibility. So we use local -i flag and [a-zA-Z] pattern.
    # See https://bugs.python.org/issue31672
    idpattern = r'(?-i:[_a-zA-Z][_a-zA-Z0-9]*)'
    flags = _re.IGNORECASE

    def __init__(self, template):
        self.template = template

    # Search for $$, $identifier, ${identifier}, and any bare $'s

    def _invalid(self, mo):
        i = mo.start('invalid')
```


도전 하세요!

- 다음 bad_code.py코드를 표준 style로 바꿔보세요!

```
def sum(left,right):  
    return(left+right,left-right)  
  
val1=5  
val2= 8  
print (sum(val1,val2))
```

설치 가이드

```
pip install jupyter_contrib_nbextensions  
autopep8 pylint  
jupyter contrib nbextension install  
jupyter notebook
```

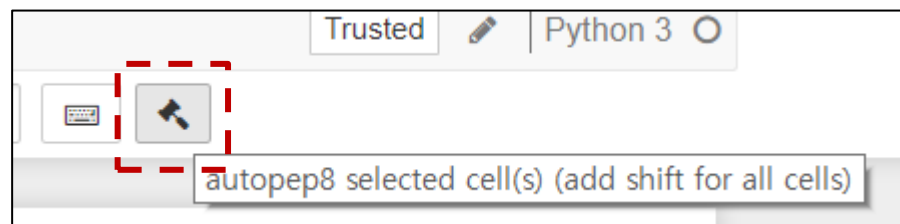
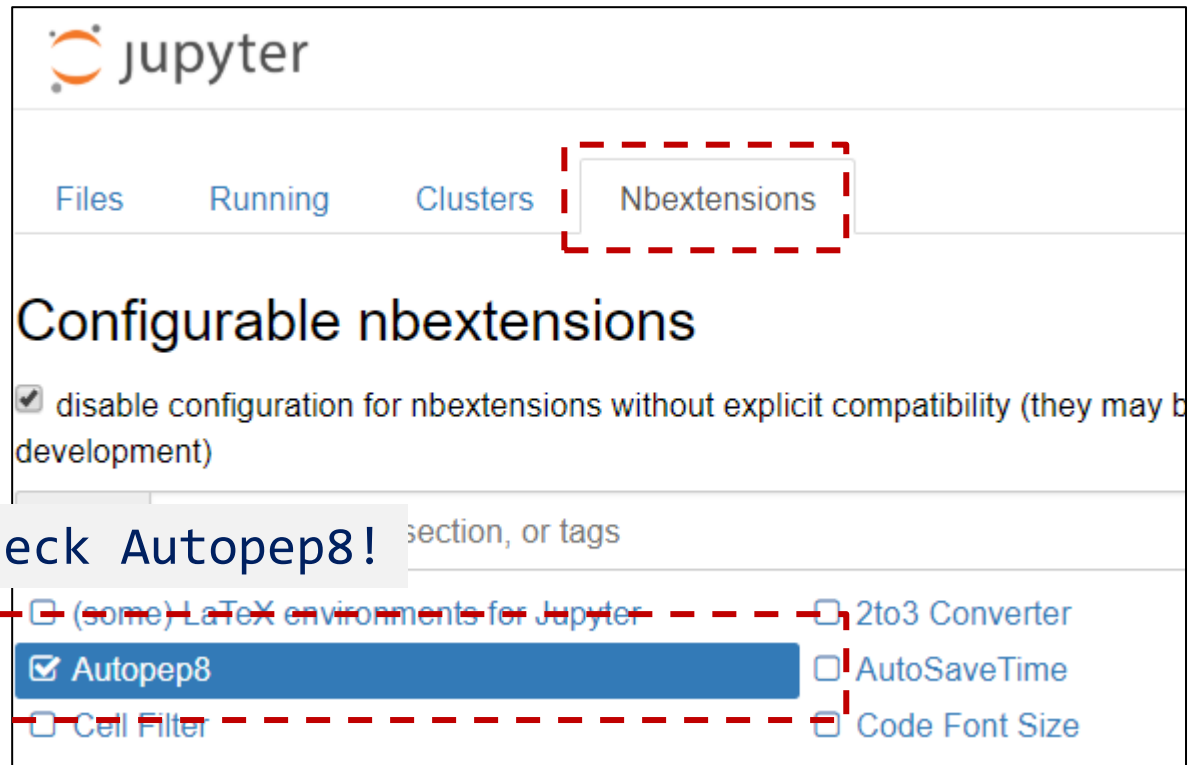
```
conda create -n d_study python=3.6  
conda install -n d_study -c conda-forge autopep8  
pylint jupyter_contrib_nbextensions  
conda activate d_study  
jupyter notebook
```

Autopep8

- autopep8 automatically formats Python code to conform to the PEP 8 style guide. It uses the py-code style utility to determine what parts of the code needs to be formatted. autopep8 can fix most of the formatting issues that can be reported by py-code style.

```
autopep8 <filename>  
--aggressive: 엄격한 규칙 적용  
--in-place: <script-file> 바로 작성  
  
autopep8 --aggressive --aggressive bad_code.py
```

Autopep8 2 – 설정



pylint

- Pylint is a tool that checks for errors in Python code, tries to enforce a coding standard and looks for code smells. It can also look for certain type errors, it can recommend suggestions about how particular blocks can be refactored and can offer you details about the code's complexity.

```
pylint <filename>
-j : 프로세스 개수
--list-msgs : 상세 메시지

pylint bad_code.py
pylint -j4 bad_code.py
pylint -j4 --list-msgs bad_code.py
```

Bad config file found, using default configuration

***** Module test

C: 2, 0: Exactly one space required after comma

```
def sum(left,right):  
    ^ (bad-whitespace)
```

W: 3, 0: Bad indentation. Found 2 spaces, expected 4 (bad-indentation)

C: 3, 0: Exactly one space required after comma

```
    return(left+right,left-right)  
        ^ (bad-whitespace)
```

C: 5, 0: Exactly one space required around assignment

```
val=5  
    ^ (bad-whitespace)
```

C: 6, 0: Exactly one space required before assignment

```
val2= 8  
    ^ (bad-whitespace)
```

C: 7, 0: Bad space allowed before bracket

```
print (sum(val1,val2))  
    ^ (bad-whitespace)
```

C: 7, 0: Exactly one space required after comma

```
print (sum(val1,val2))  
        ^ (bad-whitespace)
```

C: 1, 0: Missing module docstring (missing-docstring)

W: 2, 0: Redefining built-in 'sum' (redefined-builtin)

C: 2, 0: Missing function docstring (missing-docstring)

C: 5, 0: Constant name "val" doesn't conform to UPPER_CASE naming style (invalid-name)

C: 6, 0: Constant name "val2" doesn't conform to UPPER_CASE naming style (invalid-name)

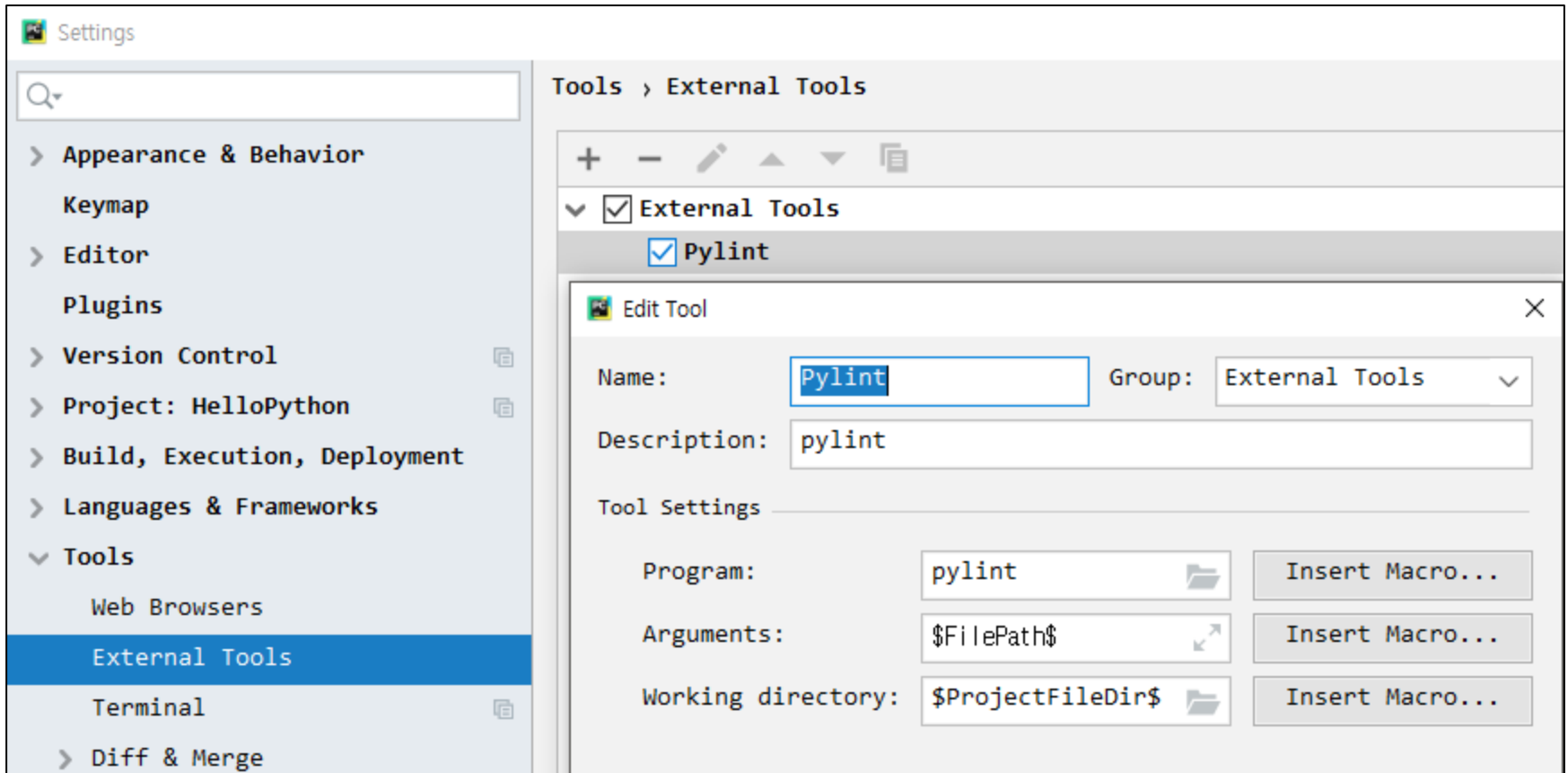
E: 7,11: Undefined variable 'val1' (undefined-variable)

Your code has been rated at -24.00/10 (previous run: -12.00/10, -12.00)

[R]efactor : 모범 사례 위반
[C]onvention : 코딩 표준 위반
[W]arning : 스타일, 사소한 문제
[E]rror : 심각한 프로그래밍 문제
[F]atal : 실행을 멈추는 오류

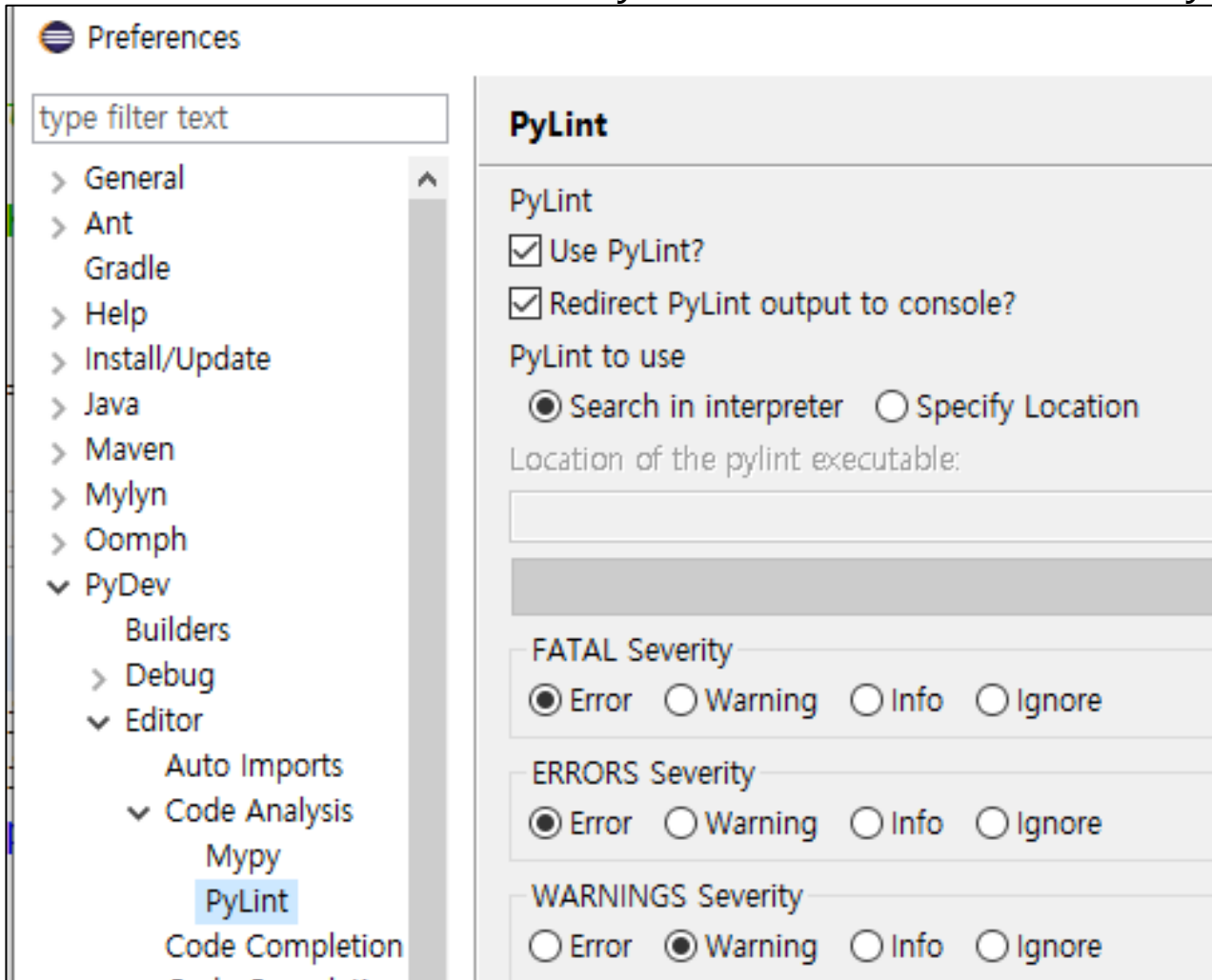
Pycharm for PyLint

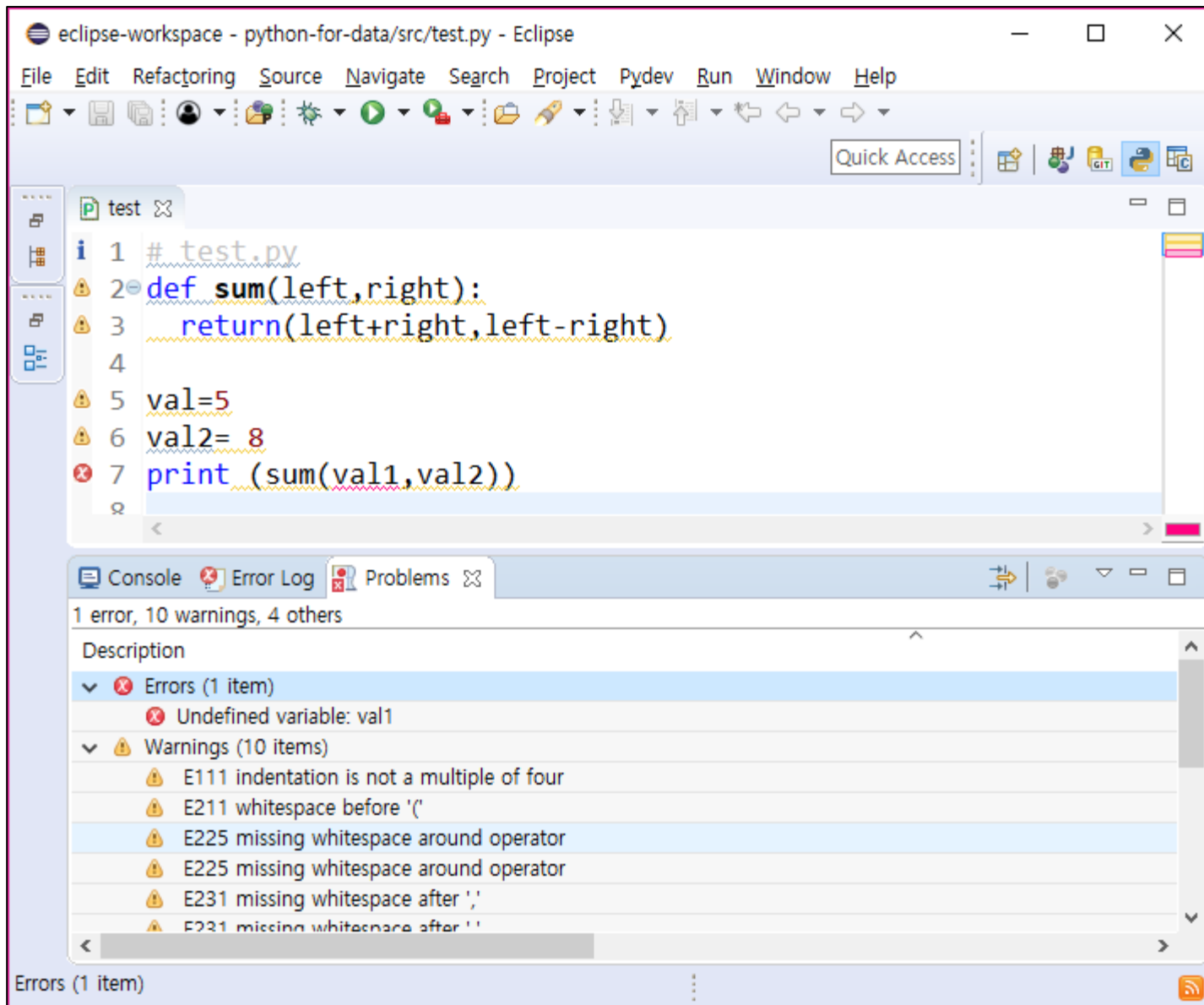
- File > Settings > Tools > External Tools > pylint
- pylint, \$FilePath\$, \$ProjectFileDir\$



Eclipse for PyLint

- Window > Preferences > PyDev > Editor > Code Analysis > PyLint





PEP8의 예외 사항

- 개발그룹과 Library 관례가 먼저다
 1. PEP 8은 반드시 지키지 않아도 괜찮습니다
 2. Legacy code의 변경 량이 현저히 많은 경우
 3. ~~PEP 8 이전에 개발된 code인 경우~~
 4. 가독성이 떨어지는 경우
 5. 오래 전 Python과 호환성을 유지할 필요가 있을 때

Code layout

Part1

mailofnsh@gmail.com

Code Layout

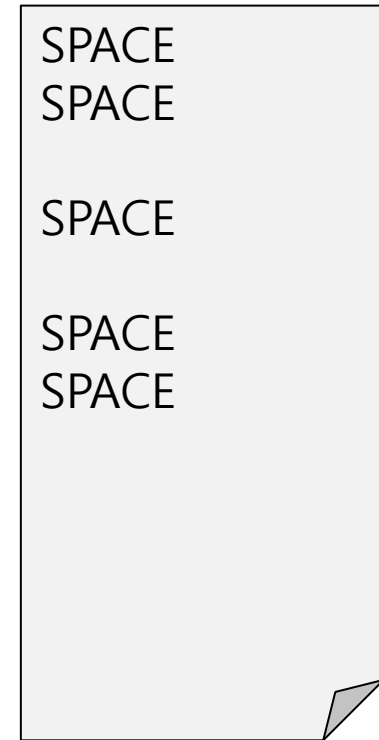
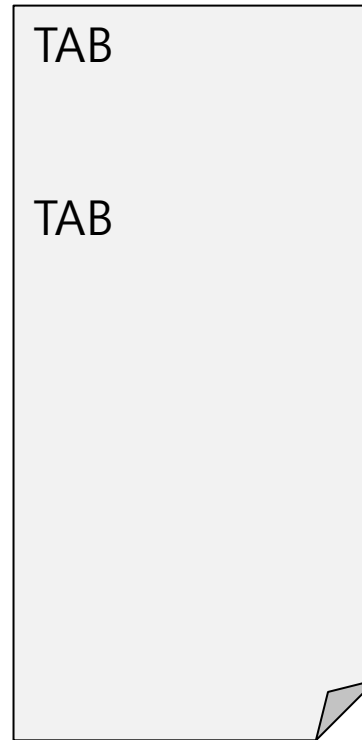
1. Indentation
2. Blank Line
3. Braces
4. Functions declaration, call
5. Operators
6. Maximum Characters in a line
7. Empty line
8. Encoding

Indent – Tab and, Blank

- 모듈 내부에서 Indent는 하나 만 사용

1. Tab ? Blank?

2. Space?



Blank Line – Relationship

- 동일 그룹은 한 줄로 분리, 그렇지 않으면 두 줄로 분리

```
def subtract(left, right):  
    print (left - right)
```

•

```
def plus(left, right):  
    print (left + right)
```

•

•

```
def multiply(left, right):  
    print (left * right)
```

```
def divide(left, right):  
    print (left / right)
```

```
class Bread:  
    pass
```

•

```
class Soup:  
    pass
```

•

•

```
class Bus:  
    pass
```

```
class Car:  
    pass
```

Blank Line – Group

- 서로 다른 종류(Class, Function) 간 빈 두 줄로 분리

```
def subtract(left, right):  
    print (left - right)
```

•

```
def plus(left, right):  
    print (left + right)
```

•

```
def multiply(left, right):  
    print (left * right)
```

•

•

```
class bread:  
    pass
```

```
class Calcuator:  
    pass
```

•

•

```
def subtract(left, right):  
    print (left - right)
```

•

```
def plus(left, right):  
    print (left + right)
```

•

•

```
subtract(10, 3)
```

Indent – Continuous Braces

- 시작 괄호와 요소를 붙임 또는, 개행해서 분리
- 중간 요소는 Indent 1개 더 넣거나, 처음 요소의 Indent에 맞춤
- 종료 괄호는 개행 후 Indent 1개 더 넣거나, 마지막 요소에 붙임

```
my_list = [  
    1, 2, 3,  
    4, 5, 6  
]  
  
r = func(1, 2,  
    3, 4  
)
```

```
my_list = [  
    1, 2, 3,  
    4, 5, 6  
]  
  
r = func(1, 2,  
    3, 4  
)
```

```
my_list = [  
    1, 2, 3,  
    4, 5, 6]  
  
r = func(1, 2,  
    3, 4)
```


Indent – Function Parameter

- 함수 선언에서 길어진 Parameter의 개행 후 indent
 - Indent 1개 추가: 코드와 동일한 Depth
 - Indent 2개 추가: Code indent보다 한 단계 더 Indent 추가로 코드 분리
 - First parameter indent

```
def long_args_func(var_one, var_two,  
    var_three):  
    print(var_one)
```

```
def long_args2_func(var_one, var_two  
    ,var_three):  
    print(var_one)
```

```
def long_args3_func (var_one, var_two,  
    var_three):  
    print(var_one)
```

Indent – Function Argument

- 함수 호출 시 일반적으로 길어지는 Argument Indent
 1. Code indent보다 한 단계 더 함수 선언 Indent 추가
 2. Code indent보다 두 단계 더 함수 선언 Indent 추가
 3. 함수의 첫 Argument 위치에 이어지는 Intent 유지

```
long_args_func(1, 2,  
█ 3, 4)
```

```
long_args_func(1, 2,  
████ 3, 4)
```

```
long_args_func(1, 2,  
████████ 3, 4)
```

Indent – Operator and, Operand

- 좌측은 가독성을 낮춤
 - 연산자 레이아웃 흩어짐
 - Operator의 인접 Right operand가 줄 분리
 - Right operand시점에 Left operand를 확인하러 상단으로 줄 이동

```
income = 1 + \  
         2 + \  
         (3 - 4) - \  
         5 - \  
         6
```

```
income = 1 \  
         + 2 \  
         + (3 - 4) \  
         - 5 \  
         - 6
```

Indent – If

- 여러 줄에 걸쳐 작성가능한 if
 1. Code indent와 일치시킴, 추가 indent는 제거
 2. 조건문과 Code사이에 주석을 추가하여 구문 분리
 3. 연속되는 조건문에 indent 추가

```
if (this_is_one_thing
    and that_is_another_thing):
    long_args_func(1, 2, 3, 4)
```

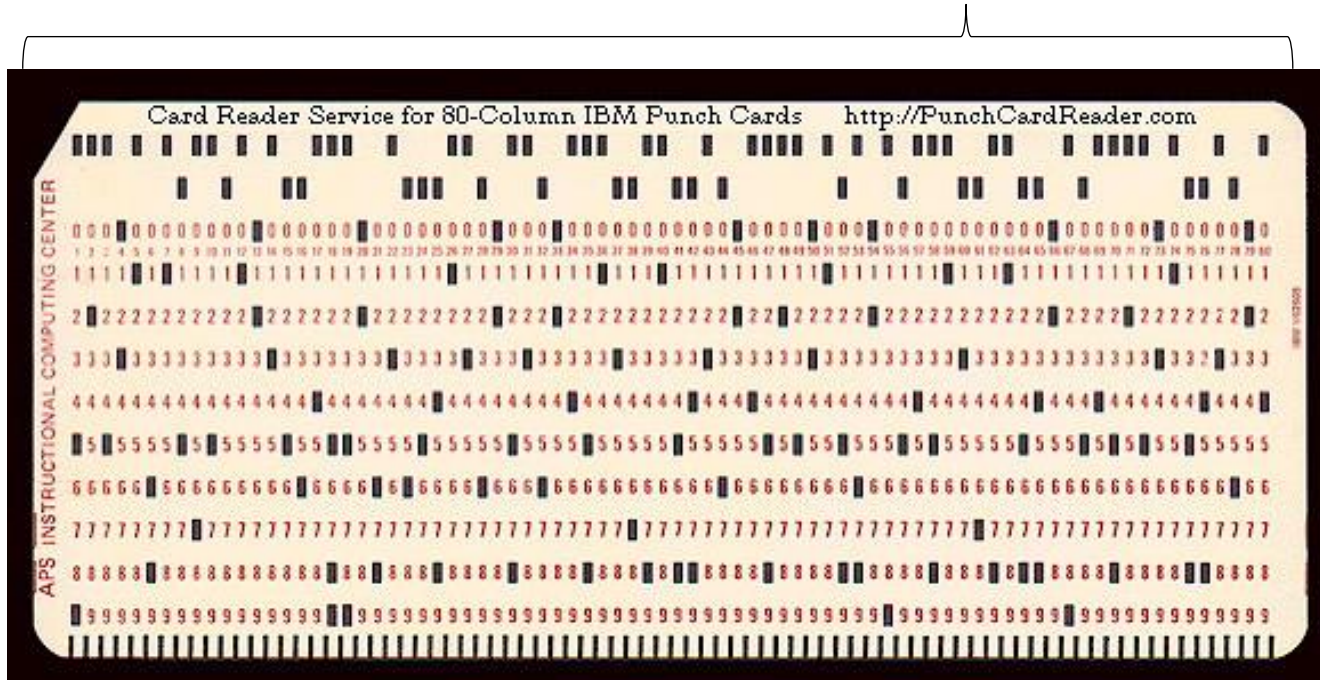
```
if (this_is_one_thing
    and that_is_another_thing):
    # Since both conditions are true
    long_args_func(1, 2, 3, 4)
```

```
if (this_is_one_thing
    and that_is_another_thing):
    long_args_func(1, 2, 3, 4)
```

Max number of chars

- Punched card는 한 줄에 최대 80 문자까지 코드를 작성할 수 있었습니다

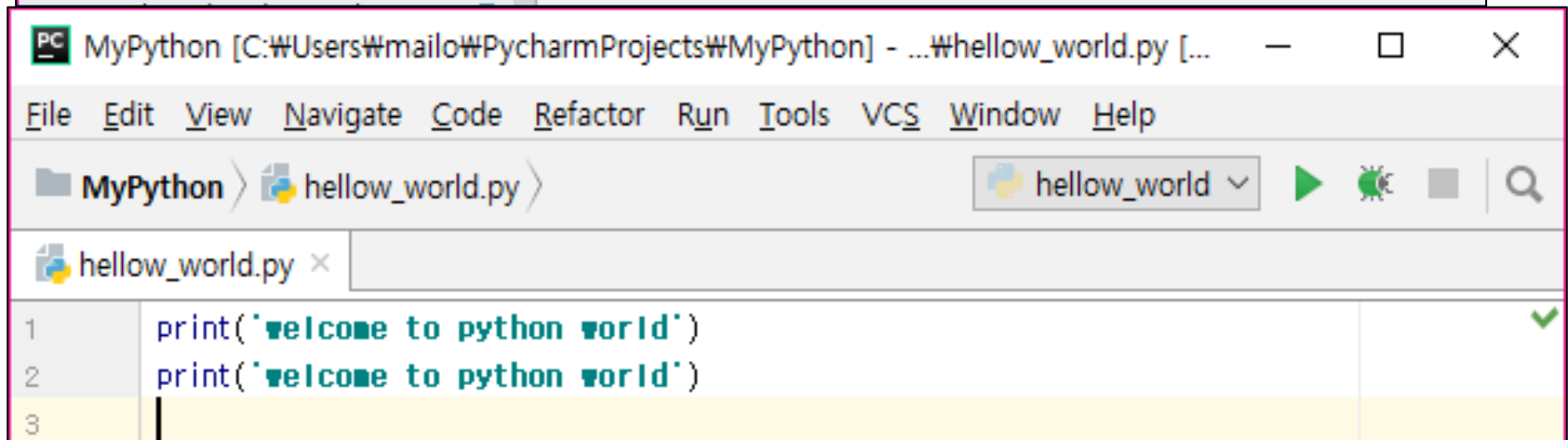
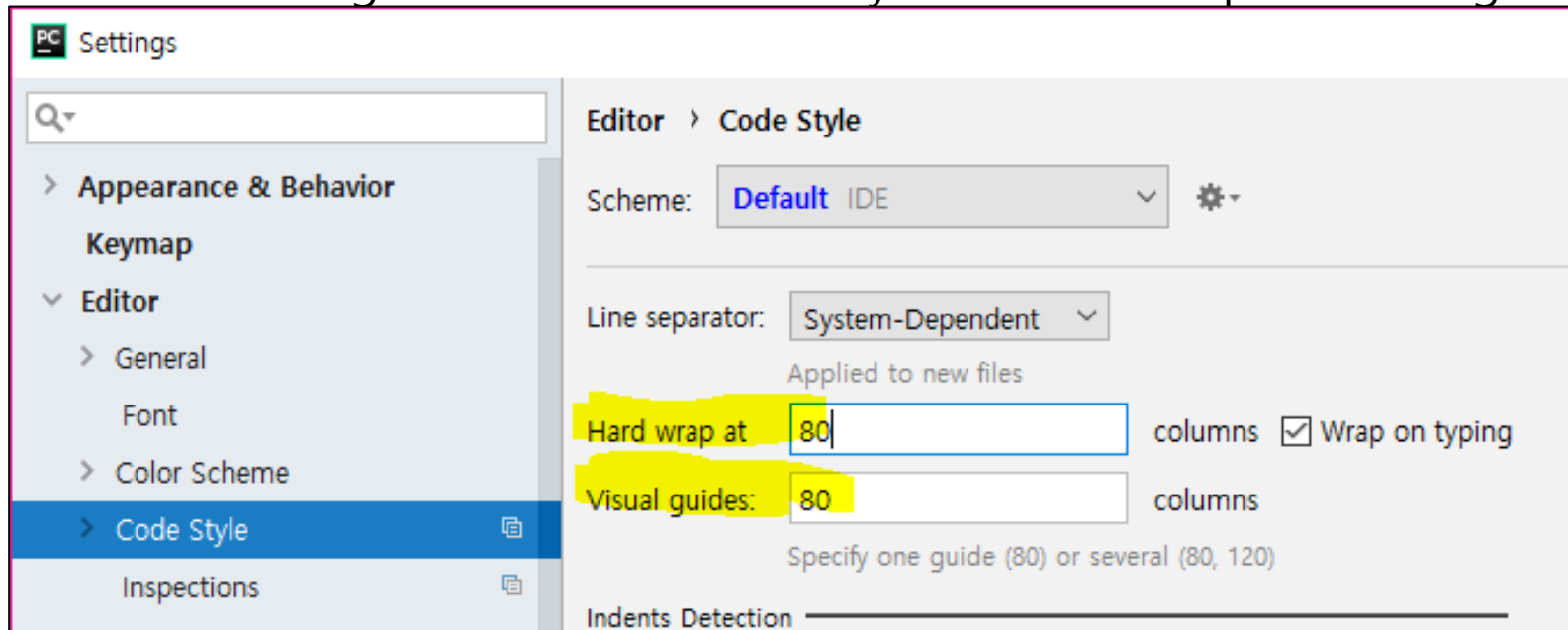
80 chars



IBM's 1928 80 column punched card format

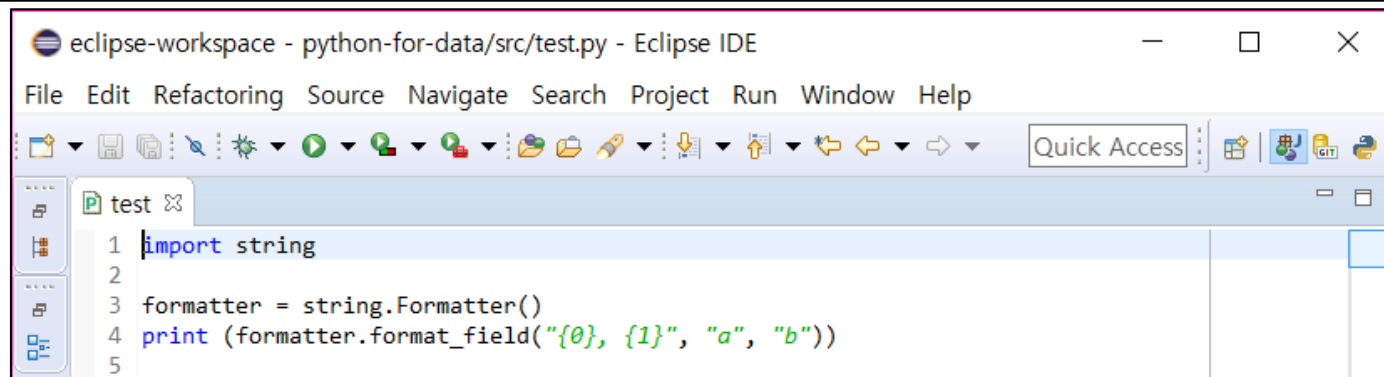
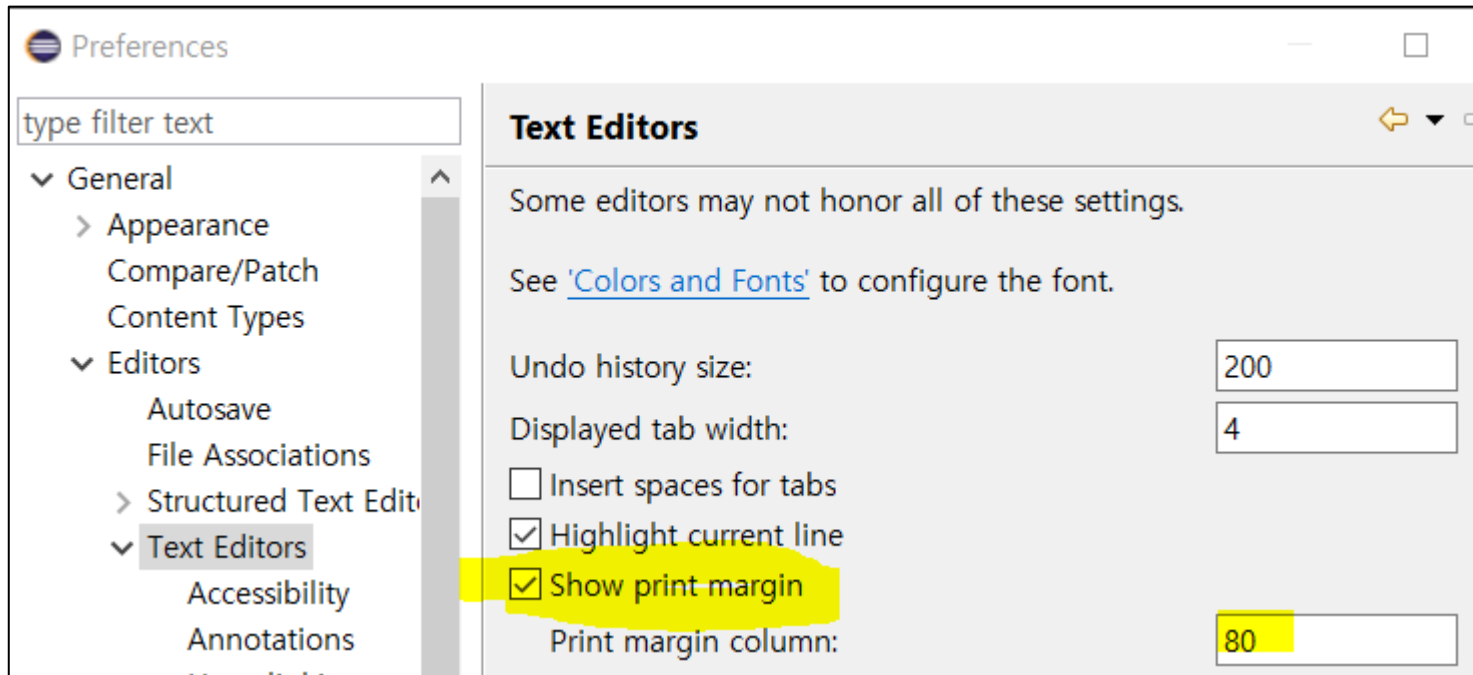
Max number of chars – Pycharm

- File -> Settings -> Editor -> Code Style -> Hard Wrap or Visual guides



Max number of chars – Eclipse

- General -> Editors -> Text Editors -> Show Print Margin



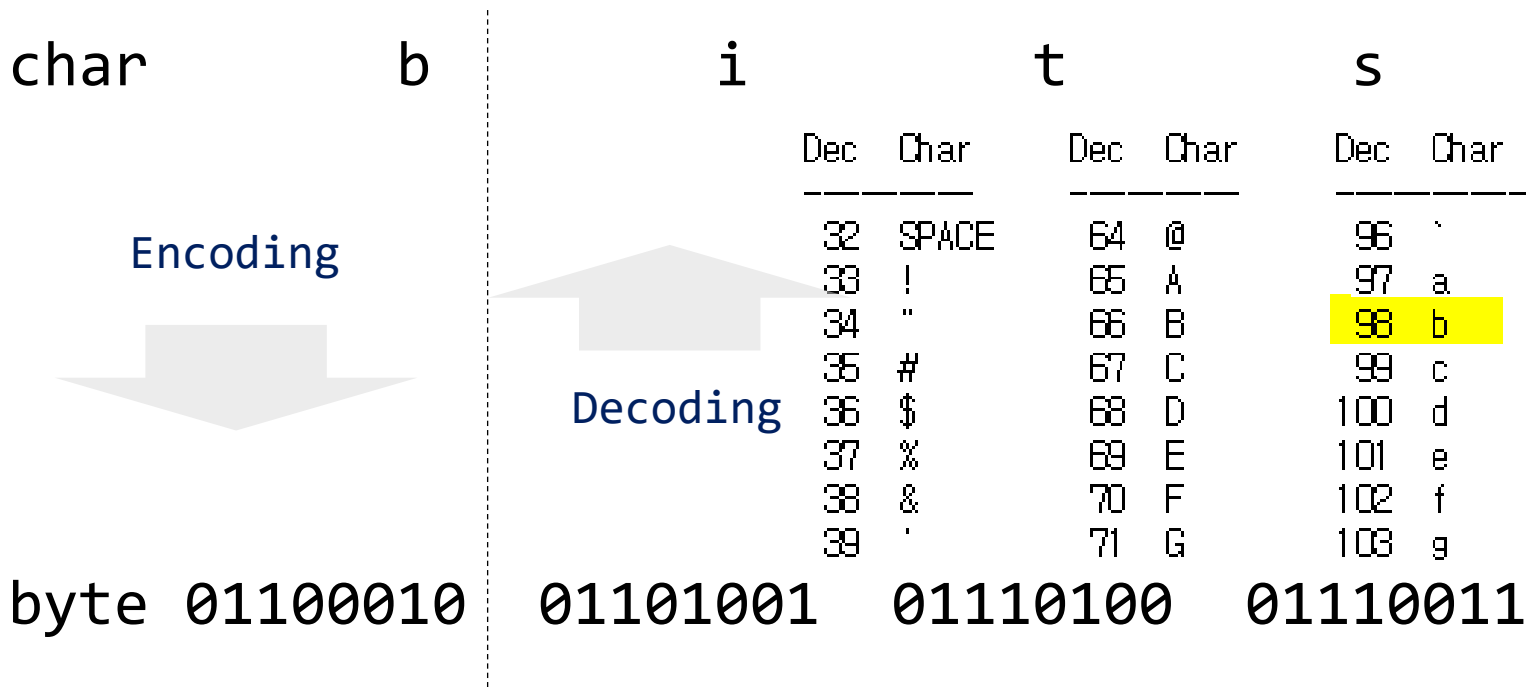
Max number of chars – Backslashes

- Backslashes는 적절하게 문장을 분리 사용
 - 여러 문장을 가진 명령문의 경우 여러 줄로 나누어 암묵적으로 연속되도록 할 수 없지만, backslashes로 다음 줄에 이어지게 연속 시킬 수 있음

```
with open(r'C:\WindowsUpdate.log', 'r') as f1, \  
      open(r'C:\setupact.log', 'r') as f2:  
    pass
```


ASCII Code – Table

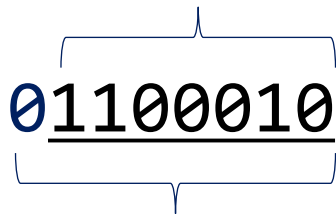
- ASCII code는 총 7Bit로 미리 약속된 문자 1개를 표현
 - 문자 하나(char)를 컴퓨터에 표현하기 위해 가장 효율적인 공간에 맞추어 진 데이터 형인 1Byte(8Bit) 사용
 - Encoding: 문자를 Byte로 변환하는 과정
 - Decoding: Byte의 값을 문자로 변환하는 과정



ASCII Code, UTF-8

- 전세계 모든 언어를 표현하기 위해서 ASCII code 부족
 - 다양한 국가와 그 세부 지역에 따라 수많은 언어가 존재
 - 각 언어 내부엔 다양한 문자도 존재합니다.
- UTF-8
 - bit 개수에 따라 표현 문자를 정의한 ISO표준의 집합
 - UTF-8: 2,097,151 characters
 - 더 많은 bit를 이용해서 ISO표준을 사용하면 그 만큼 문자의 크기와 File size 커짐

7bit 문자표준 → ISO 646 (ASCII code)


01100010

8bit 문자표준 → ISO 8859

... → ...

문자의 표현 – UTF-8

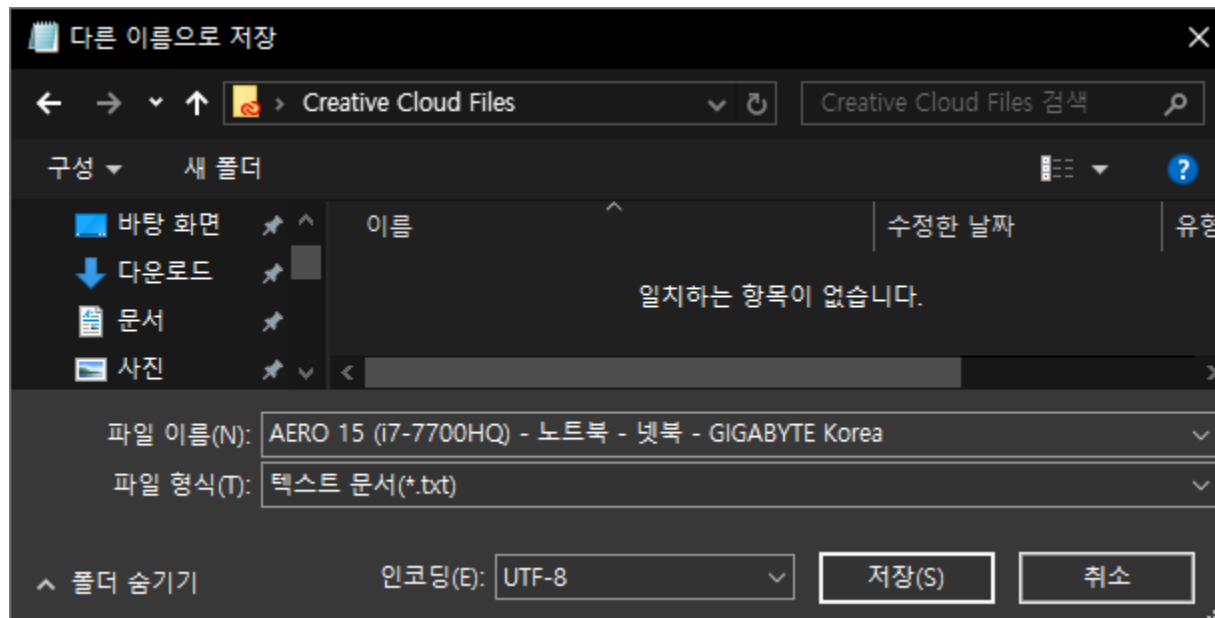
Bits	Range	Byte	Byte	Byte	Byte
7	0-127				0xxxxxxx (7)
11	128-2047			110xxxxx (5)	10xxxxxx (6)
16	2,048-65,535		1110xxxx (4)	10xxxxxx (6)	10xxxxxx (6)
21	65,536-2,097,15	11110xxx (3)	10xxxxxx (6)	10xxxxxx (6)	10xxxxxx (6)

Encoding setting (PEP 263)

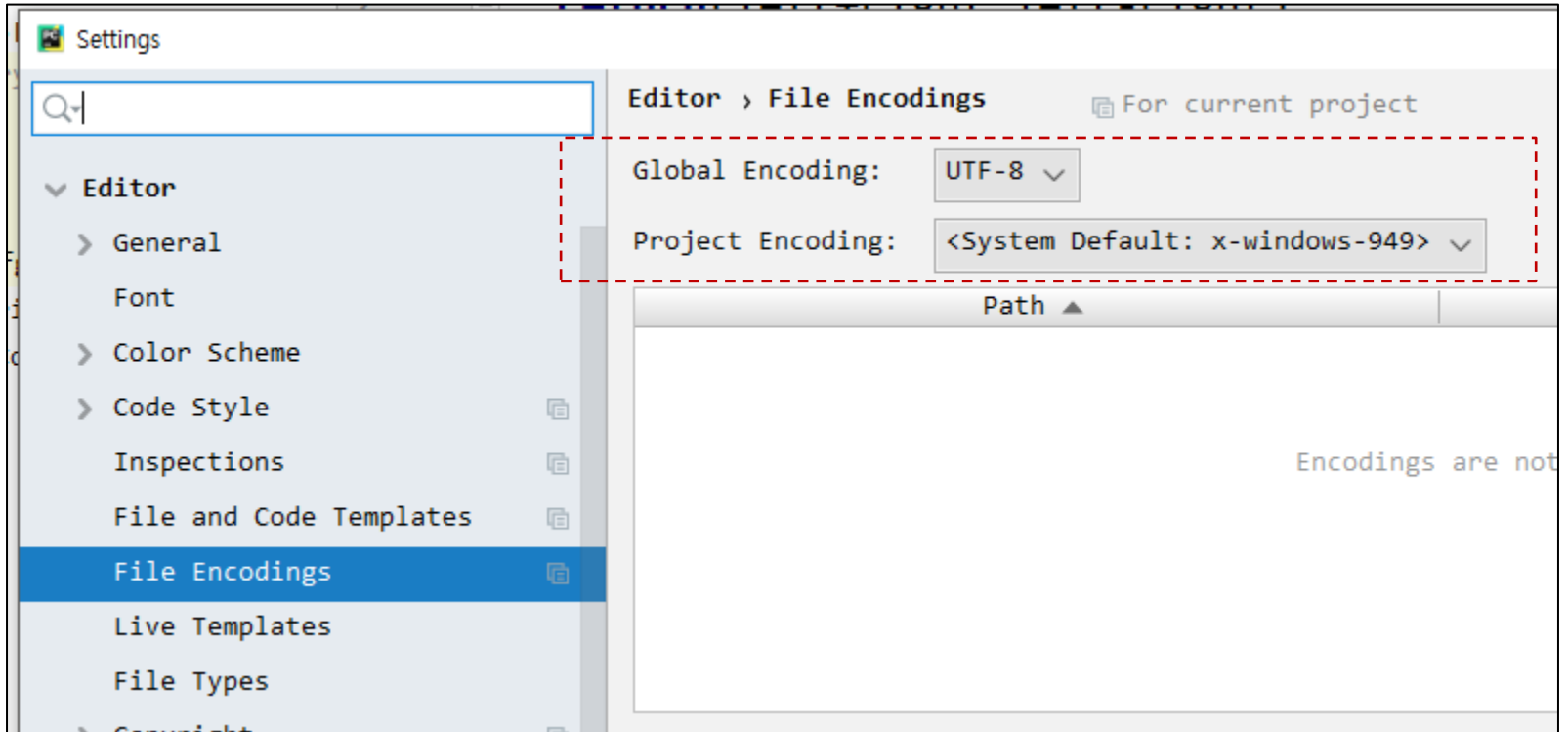
- Python Encoding 셋업 2단계
 1. 주석으로 coding header에 encoding type을 기술
 2. File 저장 시 Encoding Type을 UTF-8

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

print(5)
```



Encoding setting



정리

- Indentation
- Blank Line
- Braces
- Functions declaration, call
- Operators
- Maximum
- Empty line
- Encoding

Code layout

Part2

mailofnsh@gmail.com

String Quotes

- String을 만드는 quote는 '와 "모두 사용 가능
 - 그 기능도 동일해서 다양한 곳에 많은 개발자들이 간혹 섞어서 사용함
 - 한 모듈 내에는 한 개만 일관되게 사용하거나,
기존 코드에서 사용한 quote 사용, 코드 일관성 유지

```
str1 = 'Your'
str2 = "payment"
str3 = 'method'
strs = [ 'will', 'be',
        "automatically",
        "charged",
        "$4.00"
      ]
```


Assignment

- 할당 연산자를 기준으로 Left 와 Right Operand 정렬 금지
 - 할당 연산자를 기준으로 단 1칸의 공백만 허용해서 작성
 - 함수 Default parameter의 할당 연산자는 예외입니다. 공백이 있어서는 안됨

```
x = 1
y = 2
long_variable = 3
```

```
x=1
y=2
long_variable = 3
```

```
def complex(real, imag=0.0):
    return magic(r=real, i=imag)

def complex(real, imag=0.0):
    return magic(r=real, i=imag)
```

Binary Operators

- 이항연산자의 연산자를 기준으로 단 1칸의 공백만 허용
 - 표현식에서 우선순위가 높은 연산자는 양쪽 공백을 제거해서 강조

```
i = i+1  
i = i + 1
```

```
submitted +=1  
submitted += 1
```

```
hypot = x * x + y * y  
hypot = x*x + y*y
```

```
c = (a + b) * (a - b)  
c = (a+b) * (a-b)
```

Right single space

- Right-side operand one space
 - for Comma, Colon, Semicolon
 - 만일 오른쪽 괄호가 위치하면 공백 없음
- Return type annotation("->") 양 쪽 공백
 - 함수 호출구문에서 함수명과 괄호 사이 공백 금지

```
def munge(input_:str)->None:  
def munge(input_: str)->None:  
  
munge(1)  
munge(1)
```

```
FILES = 'setup.cfg',  
FILES = ('setup.cfg',)  
  
foo = (0, )  
foo = (0,)
```

```
if x == 4: print(x, y); x, y = y, x  
  
if x == 4: print(x, y); x, y = y, x
```

Braces – Indexing, Slice

- Indexing
 - "[" 사이 index에 공백금지
 - "]" 양끝에 공백 금지
- Slicing 연산자 ':'
 - 사이에 공백금지

```
spam( ham[1], {eggs: 2})  
spam(ham[1], {eggs: 2})
```

```
dct['key'] = lst[index]  
dct['key'] = lst[index]
```

```
ham[1:9] ham[1:9]  
ham[1:9]
```

```
ham[1:9:3]  
ham[1:9:]  
ham[1:9:3] ham[:9:3] ham[1::3]
```

Braces – Complex Slicing

- Slicing 내부에 함수호출구문과 표현식은 되도록 사용금지
 - 스레드가 임계 영역(Critical Section)에서 동시 접근하면 의도치 않은 결과 발생
- Slicing 내부에 함수호출 및 중첩표현식은 양쪽에 공백 추가
 - 단, 대괄호('[' 또는 ']') 내부에 양끝 앞뒤로 공백 없음

```
ham[low+off:up+off]  
ham[:upper]  
ham[low::up]
```

```
ham[low+off:up+off]  
ham[:upper]  
ham[low::up], ham[low:up]  
ham[low:up:], ham[low::step]
```

```
ham[down_fn(x):up_fn(x):step_fn(x)]  
ham[::step_fn(x)]
```

Compound statements

- 한 줄에 두 개 이상 명령문 배치
 - 복합명령구문은 Colon ":" 과 Semicolon ";"
- 각 기호의 바로 인접한 오른쪽에 공백 1개 추가

```
if foo == 'blah': do_blah_thing(); something();
```

```
if foo == 'blah': do_blah_thing()  
else: do_blah_thing()
```

```
try: something()  
finally: cleanup()
```

```
for x in [0, 1, 2]: total += x  
while total < 10: total = delay()
```

```
do_blah_thing(); something(); do_three(0, 1,  
                                     2, 3)
```

Import – Single unit

- 한 줄 import구문은 한 개 module만

```
[Working Directory]
/ my_module.py
def print_helloworld():
    print("Called at def print_helloworld()")
```

```
# -*- coding: utf-8 -*-
import sys, os, my_module
import sys
import os
from my_module\
    import print_helloworld

print_helloworld()
print(sys.platform)
print(os.cpu_count())
```

Import – Ordering

- Module import 순서
 1. Standard libraries
 2. Third party lib
 3. Private lib

```
import sys
from os import cpu_count

import numpy as np

import my_module

a = np.array([1, 2, 3])
print (a)
```


Import – Code layout

- 첫 번째 줄 Encoding type 선언구문 위치
 - 두 번째 줄은 빈 공백
 - 세 번째 줄은 Document string이 위치
 - 끝나면 바로 밑에 빈 공백을 추가
- 그 다음 Double Underscore(__) 구역
 - "__" 시작하는 module의 import, 변수 및 함수 선언, Class를 선언
 - 모두 완료하면 맨 밑에 빈 공백을 추가
- 그 다음 구역은 일반 코드 위치

```
# -*- coding: utf-8 -*-  
  
1  
"""This is the example module.  This module does stuff.  
1  
"""  
1  
from __future__ import barr  
  
__all__ = ['a', 'b', 'c']  
__version__ = '0.1'  
1  
import os
```

정리

- String Quotes
- Assignment
- Binary Operators
- Right single space
- Braces – Indexing, Slice
- Braces – Complex Slicing
- Compound statements
- Import

Code layout

Part3

mailofnsh@gmail.com

목차

- Inline comments
- Block comments
- Documentation strings (docstrings)

Guidelines

1. 항상 최신을 유지
2. 완전한 문장으로 작성한다
3. 영어는 항상 문장의 첫 번째 단어는 대문자
4. 내용이 짧을 경우 마지막에 오는 마침표는 생략 가능
5. 각 문 단은 마침표로 끝남
6. 되도록 영어로 작성

Lib\os.py

Inline Comments

- 명령문과 같은 라인에 배치하는 주석
 - '#'와 명령문에 최소 2칸 공백으로 분리
 - '#' 이후 1칸 공백을 추가한 다음 내용 기술

```
for iint in lst:  
    iint = iint * 31 + 1 # calculate a hash  
print(list) # print hash value  
  
x = x + 1 # Increment * Compensate for boRder
```

Block comments

- 명령문과 함께 자주 사용
 - 명령문과 동일 위치에 배치
 - 각 라인은 "#" 시작, "#" 이후 1칸 공백을 추가 후 기술.
 - 마침표 "." 이후 2칸 공백 추가
- 단락 사이에는 한 개 #만 추가 후 단락 분리

```
# We may not have read permission for top, in which case we can't
# get a list of the files the directory contains. os.walk
# always suppressed the exception then, rather than blow up for a
# minor reason when (say) a thousand readable directories are still
# left to visit. That logic is copied here.
try:
    # Note that scandir is global in this module due
    # to earlier import-*.
    scandir_it = scandir(top)
except OSError as error:
    if onerror is not None:
        onerror(error)
    return
```

Documentation strings 1

- Docstring
 - `"""` 기호로 시작과 종료하는 주석
 - 공개 모듈, 공개함수, 공개Class라면 docstrings사용
 - 비공개 시 불필요

```
def capwords(s, sep=None):  
    """capwords(s [,sep]) -> string  
  
    Split the argument into words using split,...  
    word using capitalize, and join the capital...  
    join. If the optional second argument sep ...  
    runs of whitespace characters are replaced ...  
    and leading and trailing whitespace are re...  
    sep is used to split and join the words.  
    """  
    return (sep or ' ').join(x.capitalize() for x in s.split(sep))
```


Documentation strings 2

- Header docstrings
 - 모듈의 시작부터 위치, 모듈 설명
 - 별도의 규칙을 갖고 있으며 PEP257 준수

```
r"""OS routines for NT or Posix depending on what system we're on.  
1.
```

```
This exports:
```

- all functions from posix or nt, e.g. unlink, stat, etc.
- os.path is either posixpath or ntpath
- os.name is either 'posix' or 'nt'

```
...
```

```
Programs that import and use 'os' stand a better chance of being  
portable between different platforms. Of course, they must then  
only use functions that are defined by all platforms (e.g., unlink  
and opendir), and leave all pathname manipulation to os.path  
(e.g., split and join).
```

```
"""
```

```
1.
```

```
#'
```

```
import abc
```

정리

- Inline Comments
- Block comments
- Documentation Strings