Vol.2022-SE-212 No.12
2022/12/10

Pycodestyleによる警告とバグ修正の関係に関する定量分析

高橋 亮至[1,a)]　阿萬 裕久[2,b)]　川原　稔[2,c)]

概要：ソースコードの品質管理を行う上でコーディング規約への準拠は重要な観点の 1 つである．コーディング規約に合わせてプログラミングを行うことにより，可読性の高いソースコードを作り出せるだけでなく，バグ混入のリスクを低減できるという報告もある．静的解析ツールを利用することで，ソースコードがコーディング規約に準拠しているかどうか，換言すれば規約に違反する箇所がないかどうかを自動的に確認することができる．しかしながら，多数の規約違反（警告）が検出されたり，指摘内容が些細なものであったりすることも珍しくなく，開発者たちが静的解析ツールを積極的に活用するという状況には至っていない．そこで本稿では，ツールが出力する警告の優先順位付けに向けて，オープンソース開発プロジェクトを対象とした定量データの収集と分析を行っている．具体的には Python プログラムを対象としたコーディング規約確認ツール Pycodestyle に着目し，45 個のオープンソースソフトウェアについて，それらのコミットごとでの警告を調べたり，バグ修正の前後での警告の増減について調べたりしている．その結果，1 行の長さやインデントの付け方に関する警告が多く出力される傾向にあることが確認されている．さらには，非推奨な書き方や可読性を低下させる恐れのある書き方がなされている場合にはバグ修正も比較的起こりやすい傾向にあることも示されている．

A Quantitative Analysis of the Relationship Between Pycodestyle Warnings and Bug Fixes

1. はじめに

近年，データサイエンスに対する注目の高まりに相まって，Python が広く使われるプログラミング言語となってきている [1]．Python では人工知能や機械学習，データ処理に関するライブラリが充実しており，それらを活用することで複雑・高度な処理を比較的簡単に記述できるという利点がある．それゆえ，Python を主要な開発言語としたアプリケーションやサービスも増加傾向にある．

一方，Python といえども他のプログラミング言語と同様に人間（技術者）がプログラミングを行ったりコードレビューを行ったりすることに変わりはない．即ち，コーディングにおけるバグの混入やレビューでのバグの見落としが起こってしまうリスクは常にあるといえる．そのようなリスクを軽減する 1 つの有効な手段として，コーディング規約への準拠が挙げられる [2]．

コーディング規約に従ってプログラミングを行うことにより，コードの書き方に統一性が生まれ，非推奨な書き方は排除される．それによって直ちにバグのリスクを回避できるわけではないが，コードの書き方に対する各開発者の意識を高めることにつながり，バグ混入に対する 1 つの防止策としては有効である [3]．実際，Google や Microsoft での開発でもコーディング規約は活用されている [4], [5]．

一般にコーディング規約への準拠を確認するには静的解析ツールの使用が有効である．つまり，ソースコードを解析し，所定のルールに合致する部分を自動的に検出するツールの利用である．ここでいう所定のルールとは，コーディング規約で定められた書き方から逸脱したものや経験的に非推奨とされる書き方に該当するもの等が挙げられ

[1] 愛媛大学大学院理工学研究科
Graduate School of Science and Engineering, Ehime University, Matsuyama, Ehime 790–8577, Japan
[2] 愛媛大学総合情報メディアセンター
Center for Information Technology, Ehime University, Matsuyama, Ehime 790–8577, Japan
[a)] i846011m@mails.cc.ehime-u.ac.jp
[b)] aman@ehime-u.ac.jp
[c)] kawahara@ehime-u.ac.jp

る．しかしながら，現実にはそういったツールは積極的には活用されていない [6], [7]．その主な要因としては，ツールによる警告の多くが開発者にとっては有益でないことが指摘されている．実際には 1 つのソースファイルに対していくつも警告が出され，しかもそれらの多くは些細な指摘となっていることも珍しくない．

それゆえ，警告に対して適切な優先順位付けを行い，注目に値する警告のみを提示するよう工夫していく必要がある．そこで本稿ではそういった優先順位付けの実現に向けて，Python 用の代表的な静的解析ツール Pycodestyle[8] に着目し，オープンソースソフトウェアを対象とした解析結果を収集して定量的な分析を行う．

以下，2 章で Python の静的解析ツールについて概説し，本稿での研究動機について述べる．3 章では複数のオープンソース開発プロジェクトに対して行ったデータ分析の内容とその結果について報告する．最後に，4 章で本稿のまとめと今後の課題について述べる．

2. 静的解析ツールによる検査

2.1 Python を対象としたツール

ここでは Python プログラムを対象とした静的解析ツールについて概説する．静的解析とはプログラムを実行せずにソースコードを解析する技術である．これにより，エラーやコードスメル及びスタイルの不一致などを自動的に見つけ出すことができる．

Python 向けの代表的な静的解析ツールとして Pycodestyle[8] がある．Python コミュニティで策定されている Python Enhancement Proposal (PEP)[9] では Python の拡張に関するさまざまな提言がまとめられており，その中の “PEP8” においてコーディング規約が公開されている．Pycodestyle はこの PEP8 に従って Python プログラムの解析を行うツールとなっている．

簡単な例として**図 1** に示す Python プログラム[*1]を対象とした解析結果を紹介する．このプログラムを Pycodestyle で解析すると**図 2** の結果が得られる．図 2 の解析結果から，図 1 のプログラムでは 2 つの警告が報告されている：

- 13 行目：
  “E402” という種類の警告が発せられている．この警告は “import がファイルの先頭に書かれていない（先頭以外の場所に書かれている）” ことを指摘している．PEP8 では import 文は常にファイルの先頭部分（ただし，モジュールに関するコメントよりも後）に書くべきと定められている．
  図 1 の 13 行目では他のモジュールとは離れたこの位置で “youtube_dl” というモジュールが追加で読み込

[*1] オープンソースソフトウェア youtube-dl に含まれる “devscripts/make_supportedsites.py” というソースファイル（ハッシュ値 416c7fcb）である．

```
1  #!/usr/bin/env python
2  from __future__ import unicode_literals
3
4  import io
5  import optparse
6  import os
7  import sys
8
9
10 # Import youtube_dl
11 ROOT_DIR = os.path.join(os.path.dirname(__file__),
       '..')
12 sys.path.append(ROOT_DIR)
13 import youtube_dl
14
15
16 def main():
17     parser = optparse.OptionParser(usage='%prog
           OUTFILE.md')
18     options, args = parser.parse_args()
19     if len(args) != 1:
20         parser.error('Expected an output filename')
21
22     outfile, = args
23
24     def gen_ies_md(ies):
25         for ie in ies:
26             ie_md = '**{}**'.format(ie.IE_NAME)
27             ie_desc = getattr(ie, 'IE_DESC', None)
28             if ie_desc is False:
29                 continue
30             if ie_desc is not None:
31                 ie_md += ': {}'.format(ie.IE_DESC)
32             if not ie.working():
33                 ie_md += ' (Currently broken)'
34             yield ie_md
35
36     ies = sorted(youtube_dl.gen_extractors(), key=
           lambda i: i.IE_NAME.lower())
37     out = '# Supported sites\n' + ''.join(
38           ' - ' + md + '\n'
39           for md in gen_ies_md(ies))
40
41     with io.open(outfile, 'w', encoding='utf-8') as
           outf:
42         outf.write(out)
43
44 if __name__ == '__main__':
45     main()
```

図 1 解析対象プログラム例
Fig. 1 An example of a program to be analyzed.

```
$ pycodestyle xxx.py
make_supportedsites.py:13:1:  E402 module level
import not at top of file
make_supportedsites.py:44:1:  E305 expected 2 blank
lines after class or function definition, found 1
```

図 2 Pycodestyle による解析結果
Fig. 2 An analysis result by Pycodestyle.

まれており，Pycodestyle はこの点を指摘している．モジュールの読み込みがプログラムの途中で行われてしまうことでプログラムの見通しが悪くなったり，読み込み位置の違いによる副作用が生じてしまう恐れがあり，非推奨な書き方になっているものと推察される．

- 44 行目：
  “E305” という種類の警告となっている．これは “関数の終わりの後に 2 つの空白行が必要である” という指摘である．
  PEP8 ではトップレベルの関数やクラス定義は 2 つの空白行で区切ることを推奨している．Python の文法としては空白行は 1 行だけであっても区切りとして有効であるが，あえて 2 行連続して空けることでより明示的に区切りを意識させる狙いがあると思われる．

このソースファイルではその後に 2 回の修正が行われており，E402 についてはそのままであったが，E305 については改善されていた．以上のような警告が Pycodestyle では全部で 84 種類サポートされている [10].

2.2 研究の動機付け

Popić ら [11] は，コーディング規約を遵守することがソフトウェア開発にどのような影響を与えるのかを調査し，その利点について報告している．彼らは Python を開発言語とした以下の 2 つのプロジェクトについて，Pycodestyle を開発に採り入れた場合の効果について実験を行っている．なお，いずれのプロジェクトにおいても，Python でプログラムを実装するたびに Pycodestyle による解析を行い，警告が出るたびに実装を見直すよう求めている：

- プロジェクト 1：IDE（統合開発環境）を使用せずにソフトウェア開発を 100 週間に渡って実施した．前半の 50 週間は 2 人の開発者が担当し，後半の 50 週間では開発者を 1 人増やして 3 人で開発を行った．
- プロジェクト 2：IDE として Eclipse を使用し，2 人の開発者でもって 50 週間かけて開発を行った．なお，IDE には Pycodestyle による自動解析が組み込まれており，コーディングの際にリアルタイムに警告が出るようになっていた．

これらのプロジェクトに対する実験結果から，開発者がコーディング規約に合わせることでバグ修正が必要とされる頻度が下がり，それゆえ新たなバグの発生確率が低くなっていることが報告されている．さらに，実装の途中段階でも Pycodestyle を使用することで，最終的な警告の数を軽減できるということも報告されている．

Popić らの研究では Pycodestyle を使用することの有用性は示されているが，どの種の警告が注目に値するのかといった優先順位付けについては十分に検討されていない．そこで，本稿ではプログラムの修正（リポジトリへのコミット）が行われるたびに Pycodestyle による検査を行い，警告件数の動向を種類ごとに分けて定量的に分析していくことにする．

3. データ分析

本章では本研究で行ったデータ収集と分析の結果について報告する．本分析では Python を開発言語としたオープンソース開発プロジェクトを対象とし，各ソースファイルをコミットごとに Pycodestyle で検査し，そこで出力される警告の動向を調べた．対象プロジェクトとしては GitHub で公開されているプロジェクトを REST API を使って調べ，Stars の上位 100 件に該当するものを使用した．

3.1 データ収集の手順

警告データを以下の手順（1）〜（5）で収集した．なお，GitHub に対する操作には PyGithub[12] を，Git リポジトリに対する操作は GitPython[13] をそれぞれ活用した．

(1) リポジトリの取得
対象プロジェクト 1 つずつについて，その Git リポジトリのクローンをローカル環境に作成した．

(2) コミット履歴を収集
各プロジェクトで開発・保守されている各 Python ファイルについて，それぞれのコミット履歴を追跡して収集した．その際にはファイルの名前変更（リネーム）についても可能な限り追跡した．なお，本分析では Pycodestyle による警告の動向を調査することが目的であるため，コメントまたは空白類のみを変更するようなコミットについては “変更なし” と見なすことにした．あわせて，サンプルプログラムやテストプログラムといった主要な開発対象（プロダクト）以外のプログラムは追跡の対象外とした．

(3) バグ修正コミットの特定
各プロジェクトについて GitHub 上の “Issues” からそこでバグに関係すると思われる issue を抽出した．そして，そのような issue 番号がログに登場するコミットをバグ修正コミットして特定した．なお，本分析では解析の都合上，close された issue のみを対象とした．また，GitHub 上に “Issues” が登場しない（別の課題管理システムを使用している可能性がある）プロジェクトについては解析の対象外とした．

(4) 各コミットの各ソースファイルを Pycodestyle で検査
（2）で取得した各 Python ファイルのコミット履歴情報に従って，各ファイルの各コミットにおける内容（リビジョン）をリポジトリから取得し，それぞれを Pycodestyle によって検査して出力される警告情報を記録した．
表 1 に収集データの例を示す．ここでは “`xxx.py`” の 1 回目のコミット（`138b3ef6`）後の内容（新規作成された内容）に対して警告 E303 が 8 回，E501 が 3 回

表 1 収集データの例
Table 1 An example of collected data.

No.	ファイル名	ハッシュ	出力された警告とその数	バグ修正
1	xxx.py	138b3ef6	E303 (8), E501 (3)	
2	xxx.py	70d89fba	E265 (1), E303 (9), E501 (1)	✓
3	xxx.py	1eabeaf5	E265 (1), E303 (11), E501 (1)	
⋮	⋮	⋮	⋮	⋮
1	yyy.py	7b50af2e	E501 (1), E731 (1)	
2	yyy.py	c6398fd3	E731 (1)	✓
⋮	⋮	⋮	⋮	⋮

出力されたことを表している．続く 2 回目のコミット（70d89fba）はバグ修正コミットとなっており，新たに E265 という警告が 1 回出されるようになり，E303 も 9 回に増えていたことを意味する．一方，E501 は 3 回から 1 回へ減少していたことを表している．

(5) 警告情報の集計
各ソースファイルの各コミットに対する検査結果（警告情報）を整理し，コミット前後での警告数の変化を警告の種類ごとに集計した．
例えば，表 1 に示す“xxx.py”のコミット履歴のうち，最初の 3 回（No.1–3）のみに着目したとすると以下の動向が観測されることになる：

- E265： 2 回目のコミットで増加，3 回目は変化なし
- E303： 2 回目のコミットで増加，3 回目のコミットでも増加
- E501： 2 回目のコミットで減少，3 回目は変化なし

3.2 分析手順

本研究では“バグ修正コミット”に注目し，その際にどういった警告が出ていたのか，さらにはそのコミットを通じて警告数はどのように変化したのかという観点からデータ分析を行う．

前節で説明したデータ収集によって，各ソースファイルの各修正コミットにおける各警告の情報が得られている．ここでは便宜上，1 つのソースファイルに対する 1 回のコミットを“1 つのファイル修正イベント”として数えることにする．例えば，あるコミットで 3 つのソースファイルが同時に修正されていた場合，以下ではそれらを別々に扱い，“3 つのファイル修正イベント”と見なす．

ファイル修正イベントを“バグ修正の場合のみ”に限定した時，各種の警告について，修正イベントは次の 4 種類に分類される（**表 2**）：

(1) バグ修正前に当該警告が出ており，なおかつバグ修正を経て警告数が減少している．便宜上，これを“タイプ D (Decrese)”と呼ぶ．
(2) （1）とは逆に，バグ修正を経て警告数が増加している．これを“タイプ I (Increase)”と呼ぶ．
(3) バグ修正前に当該警告が出ており，なおかつバグ修正を経ても警告に変化がない．便宜上，これを“タイプ C (Constant)”と呼ぶ．
(4) バグ修正の前後で当該警告は観測されていない．これを“タイプ N (None)”と呼ぶ．

タイプ D に該当する修正イベントが多いような警告は，バグ修正を経て減少することが多いということを意味するため，直接的あるいは間接的にバグの潜在と関係していることが疑われる．つまり，より注視すべき警告であると考えられる．逆にタイプ I に該当する修正イベントが多いような警告は，バグとの関連性は低いものと思われる．そこでこれらについてパレート分析を行い，どういった警告がこれらに該当するのかを見ていくことにする．

また，タイプ C 及び N については，単に関係しない部分が修正されていたり，始めから警告が出ていなかったりするため，その件数だけでバグとの関連性を論じるのは容易なことではないと考える．

次に，“警告が出ていた”ことと“バグ修正が行われた”ことの関連性を定量的に分析するため，アソシエーション分析も実施する．具体的には，“警告 XXX が出ている $\Longrightarrow$ バグ修正が行われる”（XXX $\Longrightarrow$ BugFix）というアソシエーションルールを考え，このルールの支持度と信頼度を算出して議論する．いま，ある警告 “XXX” に着目したとき，その警告が出ていてバグ修正が行われる修正イベントの数を m_1, バグ修正は行われない修正イベントの数を m_2 とする．さらに，修正イベントの総数を M とする．このとき，支持度と信頼度はそれぞれ以下のように算出される：

- 支持度（support）

$$supp(\mathrm{XXX} \Longrightarrow \mathrm{BugFix}) = \frac{m_1}{M} \ . \qquad (1)$$

- 信頼度（confidence）

$$conf(\mathrm{XXX} \Longrightarrow \mathrm{BugFix}) = \frac{m_1}{m_1 + m_2} \ . \qquad (2)$$

表 2 バグ修正前後での警告ごとの修正イベントの分類
Table 2 A category of modification events through bug fix commits.

タイプ	内容
D	警告数が減少した
I	警告数が増加した
C	警告数が変化しなかった（ただし，警告数 > 0）
N	警告数が変化しなかった（ただし，警告数 $= 0$）

支持度はそのルールに合致する修正イベント，即ち，“当該警告が出ていて，なおかつバグ修正が行われる”という修正イベントの出現率に相当する．この値が高いほど，その警告はバグ修正との関係性が強いと考えられる．そして，信頼度は“当該警告が出ている”という条件の下での“バグ修正”の起こりやすさを表しており，条件付き確率に相当する．この値が高いほど，その警告が出ていることがバグ修正につながりやすいことを意味している．

3.3 結果

3.1 節で説明した手順に従ってデータ収集を行った結果，**表 3** に示す 45 個のプロジェクトについて警告データを収集できた．残りの 55 個のプロジェクトのうち，36 個についてはバグに関連すると思われる issue 番号が見当たらなかった，あるいは GitHub 上で “Issues” が使われていなかったという理由から対象外とせざるを得なかった．また，19 個については，文字コードの都合で Pycodestyle を適切に動作させることができなかったため対象外とした．

データ収集の結果，139,569 個のファイル修正イベントについて警告データを収集できた．なお，このうちの 6,319 個（約 4.5%）がバグ修正を目的とした修正イベントであり，残りの 133,250 個（約 95.5%）が非バグ修正イベントであった（**表 4**）．

1 つの修正イベント（1 つのファイルの 1 回のコミット）において観測された警告の平均種類数は約 3.9 であり，平均警告数は約 57.8 であった．つまり，1 つのファイルで平均しておおよそ 4 種類の警告が出力され，それによって 58 箇所ほどが指摘を受けていたという計算になる．

各警告の出現回数（指摘件数）の一部を **表 5** に示す．同表から分かるように，E501, E111 及び W191 という警告が特に多く出やすいということが分かった．実際，これらの 3 種類だけで総警告数の約 70% を占めていた．一方，Pycodestyle でサポートされている 84 種類の警告のうち，**表 6** に示す 25 種類の警告は今回の調査では一切観測されなかった．

次に，バグ修正イベントにおける警告数の変化（増減）に着目した分析結果を示す．前述したように，バグ修正を経て警告数が減少する “タイプ D” の警告は，直接的または間接的にバグと関係していた可能性が疑われる．つまり，

表 3 解析対象の全プロジェクト
Table 3 All projects analyzed

youtube-dl	aiohttp
requests	aws-cli
pandas	tensor2tensor
wtfpython	DeDRM_tools
airflow	wagtail
YouCompleteMe	Zappa
django-rest-framework	newspaper
data-science-ipython-notebooks	walle-web
python-fire	ranger
algorithms	mailinabox
glances	awx
requests-html	discord.py
vnpy	mycli
wttr.in	routersploit
luigi	python-mini-projects
pyspider	fashion-mnist
avatarify-python	pydantic
EasyOCR	dgl
modern-cpp-features	visdom
wechat_jump_game	autojump
fabric	d2l-en
ddia	twint
macOS-Security-and-Privacy-Guide	

表 4 収集できたファイル修正イベントの数と割合
Table 4 Collected file modification evets.

バグ修正	6,319	(4.5%)
非バグ修正	133,250	(95.5%)
合計	139,569	

表 5 警告の出現回数（一部）
Table 5 Appearnce counts of warnings.

警告	出現回数	(%)
E501	3,419,876	(40.6%)
E111	1,395,326	(16.6%)
W191	1,037,877	(12.3%)
E251	380,126	(4.5%)
E302	242,514	(2.9%)
E231	161,972	(1.9%)
E201	154,916	(1.8%)
E202	154,442	(1.8%)
E114	149,030	(1.8%)
...	...	
E113	469	(0.006%)
E112	275	(0.003%)
E901	156	(0.002%)
W602	91	(0.001%)
E224	44	(0.0005%)
E273	32	(0.0004%)
W604	1	(0.000012%)
合計	8,424,623	

表 6 登場しなかった警告
Table 6 Warnings that did not appear.

E112	E121	E123	E126	E133	E223
E224	E226	E227	E241	E242	E273
E274	E704	E742	E743	E901	E902
W292	W503	W504	W505	W601	W603
W604					

より注視すべき警告であると思われる．これらについて整理したパレート図を **図 3** に示す．

図 3 から分かるように，E501 が特に多く，これだけで全体の約 42% を占めていた．これを含めて **表 7** に示す 11 種類の警告でもってタイプ D の約 80% が構成されていた．つまり，これら 11 種類が主要なタイプ D 警告であるといえる．

一方，バグ修正イベントを経て件数が増加していた “タイプ I” の警告についてもパレート分析を行った結果を**図 4** 及び **表 8** に示す．タイプ I 警告は，バグ修正が行われた際に増加していることから，バグ修正との関連性は低いと思われる．分析の結果，表 8 に示す 7 種類の警告がこれに該当していた．しかしながら，これら 7 種はいずれもタイプ D 警告としても登場していた．このことから，これらは一般に出現頻度が高く，それゆえバグ修正と関係している場合としていない場合の両方で多く観測されたのではないかと思われる．

続いて，アソシエーション分析の結果を示す．各警告の出現を前件部，バグ修正を後件部としたルールを考え，それらの支持度及び信頼度を算出した．それぞれの上位 10 件を**表 9** 及び**表 10** に示す．

全体的に支持度は低い値となっているが，もともとバグ修正イベントは全体の約 4.5% しかなく，なおかつすべてのソースファイルで警告が出されるわけではないため，結果的に高いものでも 0.6% 程度にしかならなかった．

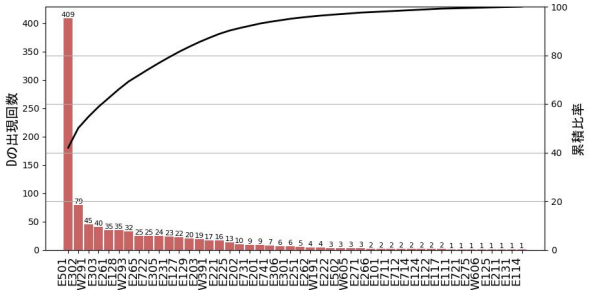

図 3 タイプ D の警告に関するパレート図
Fig. 3 Pareto chart of type-D warnings.

表 7 主要なタイプ D 警告
Table 7 Major type-D warnings.

警告	累積率（%）
E501	42
E302	50
W291	55
E303	59
E261	62
E128	66
W293	69
E265	72
E722	74
E305	77
E231	79

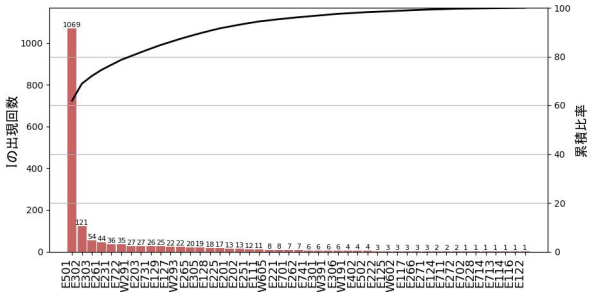

図 4 タイプ I の警告に関するパレート図
Fig. 4 Pareto chart of type-I warnings.

表 8 主要なタイプ I 警告
Table 8 Major type-I warnings.

警告	累積率（%）
E501	62
E302	69
E303	72
E261	75
E231	77
E722	79
W291	80

表 9 支持度（上位 10 件）
Table 9 Support values (top 10).

順位	警告	支持度
1	E501	0.006218
2	E731	0.002106
3	E741	0.001698
4	W605	0.000548
5	E722	0.000516
6	W391	0.000323
7	W293	0.000143
8	W291	0.000117
9	E305	0.000108
10	E302	0.000091

表 10 信頼度（上位 10 件）
Table 10 Confidence values (top10).

順位	警告	信頼度
1	W602	0.1153
2	E741	0.0718
3	E731	0.0576
4	W605	0.0326
5	W391	0.0280
6	E501	0.0264
7	E722	0.0129
8	E113	0.0126
9	E711	0.0110
10	W293	0.0084

また，信頼度も全体的に低い値が多いが，バグ修正イベントの占める割合が約 4.5% であることを考慮すると，信頼度がこれを超える 3 種類の警告（W602, E741, E731）については注視する価値があるのではないかと考えられる．

3.4 考察

本研究では，Pycodestyle を使って指摘される警告（コーディング規約違反）に着目し，実際のオープンソースソフトウェア開発におけるその変化動向を調査することで，どういった警告がバグ修正の前では出ていたのか，さらにはそのコミットを通じて警告数はどのように変化したのかという観点からデータ分析を行った．

Python を開発言語とした 45 個のオープンソース開発プロジェクトから 139,569 件のファイル修正イベント（1 つのファイルの 1 回のコミットを 1 イベントとして計上）を収集し，それぞれについて Pycodestyle によるコード検査を行ったところ，全部で 59 種類，8,424,623 件の警告を観測できた．1 回のファイル修正イベントでは，平均で約 3.9 種類の警告が出され，指摘箇所は平均で約 57.8 個であった．これはつまり，仮に開発者が Pycodestyle をプログラミングに活用していたとして，Python ファイル 1 個あたりで 58 個程度の警告が出されていることになる．このことから，開発者がすべての警告に注意を払って修正することは必ずしも容易な作業ではなく，多数の指摘の中に重要な指摘が埋もれてしまう（見逃されてしまう）可能性は低くないと考える．それゆえ，本研究の研究動機でもある “優先順位付け” の重要性を再確認できたといえる．

データ収集の結果，バグ修正の有無に関わらず E501, E111 及び W191 という種類の警告が大半（約 70%）を占めていることも分かった．これらの警告の内容はそれぞれ次の通りである：

- E501：
  1 行が長すぎる（79 文字を超える）．
  PEP8 では 1 行の長さを最大 79 文字までに制限することを推奨している．
- E111：
  インデントの幅（文字数）が 4 の倍数になっていない．
  PEP8 では 1 つのインデントとして空白を 4 つ使うことを推奨している．
- W191：
  インデントにタブ文字が含まれる．
  PEP8 では空白を使ってインデントすることを推奨しており，タブと空白を混ぜることを禁止している[*2]．

E501 に関しては，出力する文字列が長くなったり正規表現パターンが長くなったりして，やむなく 79 文字を超えてしまうプログラムも珍しくないと考える．実際 PEP8 でも 79 文字を最大長と推奨してはいるが，この制限はチームで合意がとれるようならば緩和してもよいとも書かれている．それゆえ，ソフトウェアによっては多数の E501 警告が観測されたものと思われる．

E111 と W191 はいずれもインデントに関する警告であるが，これらは使用するエディタの影響が大きいものと思われる．プロジェクトに参加するすべての開発者が同じエディタを使用しているとは限らず，エディタの違いによってインデントの幅が違っていたり，タブ文字を使ったりしている場合もあるのではないかと推察される．そのため，これらの警告が多数観測されたものと考える．

次に，パレート分析の結果から，表 7 に示す 11 種類が主要なタイプ D 警告，表 8 に示す 7 種類が主要なタイプ I 警告であることが分かった．しかしながら，7 つ全ての主要タイプ I 警告は主要タイプ D 警告としても登場していた．前述したように，これらはいずれも出現頻度の高い警告であったため，タイプ D とタイプ I のいずれの側面も持ち合わせていたものと推察される．

最後に，アソシエーション分析の結果に注目すると，バグ修正イベントの少なさ（約 4.5%）ゆえに支持度は全体的に低い傾向にあったが，一部のルールについては相対的に高い信頼度を有していた．具体的には，W602，E741 及び E731 という 3 種類の警告に関しては，これらの警告が出ていることを前提とすることでバグ修正の起こりやすさが（もともとの割合である）4.5% を超える結果となった．以下，これらについて考察する．

- W602：
  例外の作り方が非推奨である（将来的にサポートされない可能性がある）．
  “`raise` 例外クラス，メッセージ” という書き方がされているが，これは “`raise` 例外クラス（メッセージ）” に改めるべきであるという指摘である．
  これが直ちにバグにつながるかどうかは断言できないが，少なくとも記述を改めるべき箇所を適切に指摘しており，無視すべきではないと考えられる．
- E741：
  ‘`l`’,‘`O`’,‘`I`’という名前の変数がある．
  PEP8 ではこれらを変数名として使用することを禁止している．実際，`l`（小文字のエル），`I`（大文字のアイ），並びに `1`（数字のイチ）は互いに似ているため，誤って記述しても気が付かない恐れがある．`O`（大文字のオー）と `0`（数字のゼロ）についても同様である．例えば，glances プロジェクトの “`glances_systemd.py`” では **図 5** に示す変数名の修正がバグ修正イベントで行われていた．変数名そのものがバグを誘発していたとは断言できないが，少なくとも可読性の観点からは修正する意義はあったものと思われる．

[*2] 実際にプログラミングを行う際には “タブキー” を使ってインデントを付けることが多いと思われるが，それによってエディタで作られるコードでは空白 4 つになっていることが多い．

(a) 修正前　(b) 修正後

図 5 E741 に該当する規約違反の修正例

Fig. 5 An instance of code change corresponding to E741.

- E731：
  ラムダ式を識別子（変数）に直接結びつけている．
  ラムダ式を変数に代入するように書くと，代入先の変数名が関数名（関数の内容はラムダ式に従う）として使えるようになるが，コードの解読や後の解析を難しくしてしまう恐れがあり，def を使って定義することが推奨されている．
  例えば，**図 6** のプログラムを考えると，f と g はどちらも引数で与えられた値を 2 倍する関数として動作するようになっている．これらをそれぞれ別の変数 h へ代入して文字列表現を出力させると，前者は単にラムダ式として表示されるだけであるが，後者はそれが関数 g であることを示すようになる．
  この場合も必ずしもバグにつながるかどうかは断言できないが，def 宣言した場合に比べて可読性の点では懸念があり，バグの温床となりかねないと考えられる．

4. おわりに

本稿では Python の主要なコーディング規約 PEP8 に注目し，それへの準拠を確認するための静的解析ツール Pycodestyle を用いて実際のオープンソース開発プロジェクトにおけるコーディング規約違反（警告）の変化動向を定量的に調査した．45 個のプロジェクトに対するデータ収集と分析を通じて以下の傾向を確認できた：

- 1 行の長さやインデントの付け方に関する警告（E501, E111 及び W191）が多く出力される傾向にある．
- 非推奨な書き方や可読性を低下させる恐れのある書き方がなされている（警告 W602，E741 及び E731）場合にはバグ修正も比較的起こりやすい傾向にある．

今後，これらのデータを活用して，コーディング規約違反の優先順位付けについて検討を行っていく予定である．なお，今回のアソシエーション分析では，前件部として 1 つの警告のみに着目したが，複数の警告の組合せについても分析する価値があるのではないかと考えている．また，一般のアプリケーション開発のみならず，プログラミング教育への応用についても検討していきたい．

```
1  f = lambda x: 2*x
2  def g(x): return 2*x
3
4  h = f
5  print('h =',h)
6     # 出力：h = <function <lambda> at 0x104ecc040>
7
8  h = g
9  print('h =',h)
10    # 出力：h = <function g at 0x104f6a430>
```

図 6 ラムダ式と def をそれぞれ変数へ代入した例

Fig. 6 An example of lambda expression and def statement assignments to a variable.

謝辞 本研究の一部は JSPS 科研費 課題番号 20H04184, 21K11831，21K11833 の助成を受けたものです．

参考文献

[1] TIOBE Software BV: TIOBE Index, https://www.tiobe.com/tiobe-index/ (2022).
[2] Wang, Y., Zheng, B., Huang, H. et al.: Complying with Coding Standards or Retaining Programming Style: A Quality Outlook at Source Code Level, *J. Softw. Eng. Appl.*, Vol. 1, No. 1, pp. 88–91 (2008).
[3] Li, X. and Prasad, C.: Effectively teaching coding standards in programming, *Proc. 6th Conf. Inf. Tech. Edu.*, pp. 239–244 (2005).
[4] Google: Google Style Guides, https://google.github.io/styleguide/ (2020).
[5] Microsoft: Framework Design Guidelines, https://learn.microsoft.com/en-us/dotnet/standard/design-guidelines/.
[6] Johnson, B., Song, Y., Murphy-Hill, E. and Bowdidge, R.: Why Don't Software Developers Use Static Analysis Tools to Find Bugs?, *Proc. 2013 Int'l Conf. Softw. Eng.*, pp. 672–681 (2013).
[7] Muske, T. B., Baid, A. and Sanas, T.: Review efforts reduction by partitioning of static analysis warnings, *Proc. IEEE 13th Int'l Working Conf. Source Code Analysis & Manipulation*, pp. 106–115 (2013).
[8] Rocholl, J. C., Xicluna, F. and Lee, I.: pycodestyle's documentation, https://pycodestyle.pycqa.org/en/latest/ (2006).
[9] Python developers: PEP 0 — Index of Python Enhancement Proposals (PEPs), https://peps.python.org/pep-0000/ (2000).
[10] Rocholl, J. C., Xicluna, F. and Lee, I.: pycodestyle 2.7.0 documentation, https://pep8.readthedocs.io/en/latest/intro.html (2006).
[11] Popić, S., Velikić, G., Jaroslav, H., Spasić, Z. and Vulić, M.: The Benefits of the Coding Standards Enforcement and it's Influence on the Developers' Coding Behaviour: A Case Study on Two Small Projects, *Proc. 26th Telecommunications Forum*, pp. 420–425 (2018).
[12] Jacques, V.: PyGithub 1.57 documentation, https://pygithub.readthedocs.io/en/latest/introduction.html (2022).
[13] Trier, M. and Thiel, S.: GitPython 3.1.29 documentation, https://gitpython.readthedocs.io/en/stable/ (2008).