|  |
| --- |
| Trivial-Purfuit  Version not provided  Code analysis |

|  |
| --- |
| **By: Sam Pagano**  **2020-08-08** |

# Content

[Content 1](#_Toc7772423)

[Introduction 2](#_Toc7772424)

[Configuration 2](#_Toc7772425)

[Synthesis 2](#_Toc7772426)

[Metrics 2](#_Toc7772427)

[Volume 3](#_Toc7772428)

[Issues count by severity and type 3](#_Toc7772429)

[Charts 3](#_Toc7772430)

[Issues 3](#_Toc7772431)

# Introduction

This document contains results of the code analysis of Trivial-Purfuit.

# Configuration

* Quality Profiles
  + Names: Sonar way [CSS]; Sonar way [JavaScript]; Sonar way [Python]; Sonar way [TypeScript]; Sonar way [HTML];
  + Files: AXPOxSLkoGHM501Hzcj6.json; AXPOxSTHoGHM501Hzcop.json; AXPOxSa3oGHM501Hzcwe.json; AXPOxS-ooGHM501HzdSl.json; AXPOxSr9oGHM501HzdHh.json;
* Quality Gate
  + Name: Sonar way
  + File: Sonar way.xml

# Synthesis

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Quality Gate | Reliability | Security | Maintainability | Coverage | Duplication |
| OK | **B** | **E** | **A** | **0.0 %** | **0.0 %** |

# Metrics

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Cyclomatic  Complexity | Cognitive  Complexity | Lines of code per file | Comment  density (%) | Coverage | Duplication (%) |
| Min | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| Max | 121.0 | 115.0 | 545.0 | 100.0 | XX-MAXCOVERAGE-XX | 0.0 |

# Volume

|  |  |
| --- | --- |
| Language | Number |
| CSS | 245 |
| Python | 746 |
| HTML | 121 |
| Total | 1112 |

# Issues count by severity and type

|  |  |  |
| --- | --- | --- |
| Type | Severity | Number |
| VULNERABILITY | BLOCKER | 1 |
| VULNERABILITY | CRITICAL | 0 |
| VULNERABILITY | MAJOR | 0 |
| VULNERABILITY | MINOR | 0 |
| VULNERABILITY | INFO | 0 |
| BUG | BLOCKER | 0 |
| BUG | CRITICAL | 0 |
| BUG | MAJOR | 0 |
| BUG | MINOR | 3 |
| BUG | INFO | 0 |
| CODE\_SMELL | BLOCKER | 0 |
| CODE\_SMELL | CRITICAL | 9 |
| CODE\_SMELL | MAJOR | 14 |
| CODE\_SMELL | MINOR | 19 |
| CODE\_SMELL | INFO | 0 |
| SECURITY\_HOTSPOT | BLOCKER | 0 |
| SECURITY\_HOTSPOT | CRITICAL | 0 |
| SECURITY\_HOTSPOT | MAJOR | 0 |
| SECURITY\_HOTSPOT | MINOR | 0 |
| SECURITY\_HOTSPOT | INFO | 0 |

# Charts

# Issues

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Description | Type | Severity | Number |
| "<strong>" and "<em>" tags should be used |  | BUG | MINOR | 3 |
| Functions and methods should not be empty | There are several reasons for a function or a method not to have a body: It is an unintentional omission, and should be fixed to prevent an unexpected behavior in production. It is not yet, or never will be, supported. In this case an exception should be thrown. The method is an intentionally-blank override. In this case a nested comment should explain the reason for the blank override. Noncompliant Code Example def myfunc1(foo="Noncompliant"): pass class MyClass: def mymethod1(self, foo="Noncompliant"): pass Compliant Solution def myfunc1(): pass # comment explaining why this function is empty def myfunc2(): raise NotImplementedError() def myfunc3(): """ Docstring explaining why this function is empty. """ class MyClass: def mymethod1(self): pass # comment explaining why this function is empty def mymethod2(self): raise NotImplementedError() def mymethod3(self): """ Docstring explaining why this method is empty. Note that this is not recommended for classes which are meant to be subclassed. """ Exceptions No issue will be raised when the empty method is abstract and meant to be overriden in a subclass, i.e. it is decorated with abc.abstractmethod, abc.abstractstaticmethod, abc.abstractclassmethod or abc.abstractproperty. Note however that these methods should normally have a docstring explaining how subclasses should implement these methods. import abc class MyAbstractClass(abc.ABC): @abc.abstractproperty def myproperty(self): pass @abc.abstractclassmethod def myclassmethod(cls): pass @abc.abstractmethod def mymethod(self): pass @abc.abstractstaticmethod def mystaticmethod(): pass | CODE\_SMELL | CRITICAL | 4 |
| String literals should not be duplicated | Duplicated string literals make the process of refactoring error-prone, since you must be sure to update all occurrences. On the other hand, constants can be referenced from many places, but only need to be updated in a single place. Noncompliant Code Example With the default threshold of 3: def run(): prepare("this is a duplicate") # Noncompliant - "this is a duplicate" is duplicated 3 times execute("this is a duplicate") release("this is a duplicate") Compliant Solution ACTION\_1 = "action1" def run(): prepare(ACTION\_1) execute(ACTION\_1) release(ACTION\_1) Exceptions No issue will be raised on: duplicated string in decorators strings with less than 5 characters strings with only letters, numbers and underscores @app.route("/api/users/", methods=['GET', 'POST', 'PUT']) def users(): pass @app.route("/api/projects/", methods=['GET', 'POST', 'PUT']) # Compliant def projects(): pass | CODE\_SMELL | CRITICAL | 3 |
| Cognitive Complexity of functions should not be too high | Cognitive Complexity is a measure of how hard the control flow of a function is to understand. Functions with high Cognitive Complexity will be difficult to maintain. See Cognitive Complexity | CODE\_SMELL | CRITICAL | 2 |
| "aria-label" or "aria-labelledby" attributes should be used to differentiate similar elements |  | CODE\_SMELL | MAJOR | 6 |
| Collapsible "if" statements should be merged | Merging collapsible if statements increases the code's readability. Noncompliant Code Example if condition1: if condition2: # ... Compliant Solution if condition1 and condition2: # ... | CODE\_SMELL | MAJOR | 1 |
| "Exception" and "BaseException" should not be raised | Raising instances of Exception and BaseException will have a negative impact on any code trying to catch these exceptions. First, the only way to handle differently multiple Exceptions is to check their message, which is error-prone and difficult to maintain. What's more, it becomes difficult to catch only your exception. The best practice is to catch only exceptions which require a specific handling. When you raise Exception or BaseException in a function the caller will have to add an except Exception or except BaseException and re-raise all exceptions which were unintentionally caught. This can create tricky bugs when the caller forgets to re-raise exceptions such as SystemExit and the software cannot be stopped. It is recommended to either: raise a more specific Built-in exception when one matches. For example TypeError should be raised when the type of a parameter is not the one expected. create a custom exception class deriving from Exception or one of its subclasses. A common practice for libraries is to have one custom root exception class from which every other custom exception class inherits. It enables other projects using this library to catch all errors coming from the library with a single "except" statement This rule raises an issue when Exception or BaseException are raised. Noncompliant Code Example def process1(): raise BaseException("Wrong user input for field X") # Noncompliant def process2(): raise BaseException("Wrong configuration") # Noncompliant def process3(param): if not isinstance(param, int): raise Exception("param should be an integer") # Noncompliant def caller(): try: process1() process2() process3() except BaseException as e: if e.args[0] == "Wrong user input for field X": # process error pass elif e.args[0] == "Wrong configuration": # process error pass else: # re-raise other exceptions raise Compliant Solution class MyProjectError(Exception): """Exception class from which every exception in this library will derive. It enables other projects using this library to catch all errors coming from the library with a single "except" statement """ pass class BadUserInputError(MyProjectError): """A specific error""" pass class ConfigurationError(MyProjectError): """A specific error""" pass def process1(): raise BadUserInputError("Wrong user input for field X") def process2(): raise ConfigurationError("Wrong configuration") def process3(param): if not isinstance(param, int): raise TypeError("param should be an integer") def caller(): try: process1() process2() process3() except BadUserInputError as e: # process error pass except ConfigurationError as e: # process error pass See PEP 352 Required Superclass for Exceptions Python Documentation - Built-in exceptions MITRE, CWE-397 - Declaration of Throws for Generic Exception CERT, ERR07-J. - Do not throw RuntimeException, Exception, or Throwable See MITRE, CWE-397 - Declaration of Throws for Generic Exception CERT, ERR07-J. - Do not throw RuntimeException, Exception, or Throwable | CODE\_SMELL | MAJOR | 5 |
| Sections of code should not be commented out | Programmers should not comment out code as it bloats programs and reduces readability. Unused code should be deleted and can be retrieved from source control history if required. | CODE\_SMELL | MAJOR | 1 |
| A field should not duplicate the name of its containing class | It's confusing to have a class member with the same name (case differences aside) as its enclosing class. This is particularly so when you consider the common practice of naming a class instance for the class itself. Best practice dictates that any field or member with the same name as the enclosing class be renamed to be more descriptive of the particular aspect of the class it represents or holds. Noncompliant Code Example class Foo: foo = '' def getFoo(self): ... foo = Foo() foo.getFoo() # what does this return? Compliant Solution class Foo: name = '' def getName(self): ... foo = Foo() foo.getName() | CODE\_SMELL | MAJOR | 1 |
| Class names should comply with a naming convention | Shared coding conventions allow teams to collaborate effectively. This rule allows to check that all class names match a provided regular expression. Noncompliant Code Example With default provided regular expression ^[A-Z][a-zA-Z0-9]\*$: class myClass: ... Compliant Solution class MyClass: ... | CODE\_SMELL | MINOR | 2 |
| Local variable and function parameter names should comply with a naming convention | Shared naming conventions allow teams to collaborate effectively. This rule raises an issue when a local variable or function parameter name does not match the provided regular expression. Exceptions Loop counters are ignored by this rule. for i in range(limit): # Compliant print(i) | CODE\_SMELL | MINOR | 8 |
| Unused local variables should be removed | If a local variable is declared but not used, it is dead code and should be removed. Doing so will improve maintainability because developers will not wonder what the variable is used for. Noncompliant Code Example def hello(name): message = "Hello " + name # Noncompliant print(name) for i in range(10): foo() Compliant Solution def hello(name): message = "Hello " + name print(message) for \_ in range(10): foo() Exceptions \_ as well as tuples will not raise an issue for this rule. The following examples are compliant: for \_ in range(10): do\_something() username, login, password = auth do\_something\_else(username, login) | CODE\_SMELL | MINOR | 4 |
| Jump statements should not be redundant | Jump statements, such as return, break and continue let you change the default flow of program execution, but jump statements that direct the control flow to the original direction are just a waste of keystrokes. Noncompliant Code Example def redundant\_jump(x): if x == 1: print(True) return # NonCompliant Compliant Solution def redundant\_jump(x): if x == 1: print(True) Exceptions No issue is raised if the jump statement is the only statement of a statement suite: def my\_function(x): if x &gt; 5: do\_something() elif x == 0: return # ok even it could be changed to "pass" else: do\_something\_else() No issue is raised for return None because this was certainly done on purpose to be explicit that a function is really returning None. | CODE\_SMELL | MINOR | 5 |
| Links with "target=\_blank" should prevent phishing attacks |  | VULNERABILITY | BLOCKER | 1 |