Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC3745 — Testing
2023 - 2

Tarea 1

Fecha de entrega: Jueves 7 de Septiembre del 2023 a las 23:59

Información General

La siguiente tarea contempla como objetivo principal: crear reglas de detección de code-smells y reglas de transformación para programas escritos en Python. La revisión de calidad de código fuente se puede realizar tanto de forma manual y automática. El análisis estático nos permite realizar ciertos tipos de pruebas de forma automática.

Objetivos

- Entender en detalle como las herramientas de análisis estático están implementadas.
- Crear reglas para la revisión de código fuente de manera automática y personalizada.
- Crear transformaciones de código fuente, de manera automática, de tal forma que sean útiles para auto-reparar defectos.

Contexto

En clase se vio el módulo ast[1] de Python. Este módulo modela la sintaxis de Python y ayuda a analizar código escrito en Python a través de algoritmos basados en árboles de sintaxis abstracta. Asimismo, se creó una pequeña librería que permite detectar/reportar "warnings" y ejecutar transformaciones de código automáticamente.

La tarea trata de extender los tipos de alertas y transformaciones soportadas por la librería desarrollada en clase. En particular estas alertas y transformaciones tiene el objetivo de evaluar de forma automática pruebas de unidad.

Alertas

A continuación se detallan las alteras a implementar:

True Assertions (10 pts) – Se agregará una alerta por cada expresión `self.asssertTrue(True)` dentro de una prueba de unidad. La alerta a crear debe tener el siguiente formato:

```
Warning('AssertTrueWarning', <line_number>, 'useless assert true detected')
```

El numero de linea debe ser la linea donde el `assertTrue` esta posicionado. Por ejemplo, considere el siguiente test en Python, el mismo debería lanzar una alerta de `AssertTrueWarning` en la linea 3.

```
1 def test_x(self):
2     x = 2
3     self.assertTrue(True)
```

Assertion Less Tests (10 pts) – Se agregará un solo warning por cada test que no tenga ningún assertion. En el ejemplo, el test de la derecha no tiene ningun "assert...", el warning a crear tiene el siguiente formato:

```
Warning('AssertionLessWarning', <line_number>,  'it is an assertion less test')
```

Por ejemplo, considere el siguiente test en Python, el mismo debería lanzar una alerta de `AssertionLessWarning` en la linea 1. El numero de linea debe ser la linea donde el nombre del test fue definido, en el ejemplo, la linea es 1.

```
1 def test_x(self):
2     x = 2
3     y = 3
```

Unused variable (15 pts) – Se debe agregar una alerta por cada variable dentro de un test que sea inicializada, pero no utilizada. La alerta debe tener el siguiente formato:

```
Warning('UnusedVariable', <line_number>,'variable ', <variable_name>, 'has not been
    used')
```

Por ejemplo, en el test abajo la variable "x" y "y" no fueron utilizadas después de ser inicializadas. El numero de linea es la linea donde se definió la variable no utilizada. Por lo que dos alertas `UnusedVariable` deben ser lanzadas, en la linea 2 por la variable x y en la linea 3 por la variable y.

[1] https://docs.python.org/3/library/ast.html

```
def test_x(self):
    x = 2
    y = 2
    z = False
    self.assertTrue(z)
```

Duplicated Setup (15 pts) – Debe asumir que los test solo tendrán test-clases, es decir, que serán clases donde todos su métodos son tests. Se debe lanzar una alerta si todos los test de la clase tiene al menos una linea en común al inicio. La alerta debe tener el siguiente formato:

```
Warning('DuplicatedSetup', <num_dupl_stms>, 'there are <num_dupl_stms> duplicated
    setup statements')
```

Por ejemplo, en la clase a continuación todos los test tiene dos lineas duplicadas al inicio. Por lo que una alerta de tipo `DuplicatedSetup` debe ser lanzada indicando el numero de lineas duplicadas al inicio.

```
class TestX(TestCase):
    def test_x(self):
        x = 2
        y = 2
        self.assertEquals(x,y)
    def test_y(self):
        x = 2
        y = 2
        self.assertEquals(y,x)
```

Transformaciones

A continuación se detallan las transformaciones a implementar:

Assert True Rewriter (10 pts)– Este transformador busca todos las expresiones `self.assertEquals(x,True)` y lo transforma a `self.assertTrue(x)`.

Variable Inlining (20 pts) – Este transformador busca todas las variables que se utilizaron una sola vez después de ser inicializadas, posteriormente elimina estas variables sin modificar la lógica del test. Para este efecto, reemplaza la variable a eliminar por la expresión utilizada para inicializar la misma. A este proceso se le llama inlining. Por ejemplo, considere el siguiente código:

```
def test(self):
    x = 2 + 2
    self.assertEquals(x,2)
```

Después de la transformación debería quedar de la siguiente forma:

```
def test(self):
    self.assertEquals(2 + 2,2)
```

También debe considerar que luego de hacer el inline de una variable, puede que otra variable quede "habilitada" para hacer inline. Por ejemplo, considere el siguiente ejemplo:

```
def test(self):
    x = complex_method()
    y = x + 2
    z = x + y
    self.assertEquals(z,2)
```

Luego de hacer de aplicar la transformación múltiples veces quedaría de la siguiente forma:

```
def test(self):
    x = complex_method()
    self.assertEquals(x + (x + 2), 2)
```

Setup Method Extraction (20 pts) – Este transformador extrae las lineas duplicadas de todos los métodos de una clase de pruebas, y las acomoda en un método llamado setUp. Note que las pruebas de unidad en python, antes de ejecutar cualquier método, ejecutan primero el método setUp. Por lo que esta transformación no alteraria el resultado de ejecutar las pruebas.
Por ejemplo, la siguiente clase:

```
class TestX(TestCase):
    def test_x(self):
        x = 2
        y = 2
        self.assertEquals(x,y)
    def test_y(self):
        x = 2
        y = 2
        self.assertEquals(y,x)
```

Después de aplicar la transformación debería quedar de esta forma:

```
class TestX(TestCase):
    def setUp(self):
        self.x = 2
        self.y = 2
    def test_x(self):
        self.assertEquals(self.x,self.y)
    def test_y(self):
        self.assertEquals(self.y,self.x)
```

Note que las variables del código extraído se volvieron variables de instancia, puede asumir que este transformador siempre tomara de input una clase de prueba y que si existen lineas duplicadas al inicio de todos los métodos, estas siempre pueden ser extraíbles.

Tareas

Desarrolle los siguientes ejercicios:

- ***Ejercicio 1*** – Implemente las clases visitor y rule para cada Alerta:
  - `AssertionTrueVisitor` y `AssertionTrueTestRule` en `rules\assertion_true.py`.
  - `AssertionLessVisitor` y `AssertionLessTestRule` en `rules\assertion_less.py`.
  - `UnusedVariableVisitor` y `UnusedVariableTestRule` en `rules\unused_variable.py`.
  - `DuplicatedSetupVisitor` y `DuplicatedSetupRule` en `rules\duplicate_setup.py`.

  Todas las alertas anteriores se encuentran definidas en la sección de Alertas. Puede crear clases o métodos adicionales para usar en las clases pedidas. Se le entregan algunos tests para cada alerta para que pruebe su implementación sin embargo son libres de crear más tests si lo estiman necesario. Para ejecutar los tests necesita ejecutarlos desde el directorio principal más detalles en los consejos.

- ***Ejercicio 2*** – Implemente las siguientes transformaciones:
  - `AssertTrueCommand` en `transformers\assertion_true_rewriter.py`.
  - `InlineCommand` en `transformers\inline_rewriter.py`.
  - `ExtractSetupCommand` en `transformers\extract_setup_rewriter.py`.

  Cada uno de los comandos anteriores corresponde a las transformaciones definidas en la sección de Transformaciones. Para la implementación de las transformaciones dependiendo del caso puede necesitar implementar algunos `NodeTransformer` y/o `NodeVisitor`. Se le entregan algunos tests para cada transformación para que pruebe su implementación sin embargo son libres de crear más tests si lo estiman necesario. Para ejecutar los tests necesita ejecutarlos desde el directorio principal más detalles en los consejos.

Importante:no usar

- Se espera que ambos ejercicios sean resueltos usando `NodeTransformer` y/o `NodeVisitor` por lo que usar métodos de ast iterativos como `ast.walk` o `ast.iter_child_nodes` para revisar los nodos del árbol sintáctico abstracto implicara un descuento de la mitad del puntaje del item en el cual fue aplicado.
- Cualquier solución hardcodeada no recibirá puntaje. Nos referimos a soluciones que solo funcionen para ciertos casos, por ejemplo, que solo apunten a resolver 2 o mas test proporcionados en el enunciado de forma intencionada. Las soluciones deben apuntar a resolver el problema para cualquier código que se le envié de input. Note que los inputs que utilizaremos para probar tendrán las restricciones que dice el enunciado, por ejemplo, hay alertas que el enunciado menciona explícitamente que se enviara el código de clase.

Consejos

- Recuerden que pueden utilizar print en su código y en los tests para debugear si algo no les esta funcionando. Es muy útil para entender como el visitor visita los nodos incluir un pequeño print en los métodos visit_{NodeName} para entender el orden en que lo hace y en que nodos se detiene.
- Recuerden que los NodeVisitor y NodeTransformer son clases por lo que pueden guardar información agregando atributos a las clases que implemente la cual puede ser usada en sus visitas.
- Para ejecutar todos los tests se debe hacer desde el directorio principal (fuera del directorio core y test) ejecutar en consola `pytest` (requiere instalar la librería del mismo nombre) o ejecutar `python -m unittest`. Para ejecutar un solo archivo de pruebas desde el directorio principal ejecutar `pytest test\<Nombre Archivo>.py` o con `python -m unittest test\<Nombre Archivo>.py`.

- Si tienen dudas del código base les recomendamos jugar un poco con el código agregar prints para que puedan ir viendo lo que va pasando y como se van visitando los nodos. Si a pesar de eso hay algo que no les quede claro pueden recurrir al foro de dudas de la tarea y preguntar.

Entregables

- Debe subir todo el código de su tarea en el buzón en canvas mediante un archivo .zip.
- Junto al código deberán entregar un archivo README que contiene los nombres de quienes realizaron la tarea y su aporte (Declaración de tarea). Puede ser un archivo .md o un archivo de texto.

Archivos iniciales

En canvas se encuentra el código desarrollado en clase y los archivos relacionados a los casos de prueba para la tarea:

- *core\rules (folder)* – contiene el codigo relacionado a la implementación de las alertas.
- *core\transformers (folder)* – contiene el código relacionado a la implementación de las transformaciones.
- *test (folder)* – contiene todos los archivos de prueba para reglas y transformaciones.

Para la tarea usted debe modificar o agregar código a las ubicaciones anteriormente mencionados según corresponda.

Reportar problemas en el equipo

En el caso de que algún integrante no aportara como fue esperado en la tarea, podrán reportarlo enviando un correo con asunto **Problema Equipo {NumeroGrupo} Testing** a *juanandresarriagada@uc.cl* explicando en detalle lo ocurrido. Posterior a eso se revisara el caso en detalle con los involucrados y se analizara si corresponde aplicar algún descuento. Instamos a todas las parejas que mantengan una buena comunicación y sean responsables con el resto de su equipo para evitar problemas de este estilo.
Advertencia: Si algún integrante del grupo no aporta en las tareas puede implicar recibir nota mínima en esa entrega.

Restricciones y alcances

- Su programa debe ser desarrollado en **Python 3.10**.
- No debe modificar los nombres de los archivos y clases entregados ya que de lo contrario los tests de la corrección podrían fallar. Pueden crear nuevas clases o archivos adicionales a los entregados para usar dentro de las clases pedidas.
- Los archivos de código entregados deben terminar con la extensión **.py**.
- En caso de dudas con respecto al enunciado deben realizarlas en un foro relacionado a la tarea que se encontrara disponible en canvas.
- Si no se encuentra especificado en el enunciado, supón que el uso de cualquier librería de Python adicional a las utilizadas en el código base se encuentran prohibidas. Esta permitido ocupar librerías nativas que hemos usado en clases por ejemplo operator, collections, functools, entre otras. En caso de que estimes necesario podrás preguntar en el foro de la tarea por el uso de alguna librería adicional.

Entrega

- **Código:** Deberán entregar todo el código por medio de un buzón de canvas habilitado para esta tarea.
- **Declaración de tarea:** Deberán entregar un archivo README que contiene los nombres de quienes realizaron la tarea y su aporte. La entrega de este archivo se realizara junto con el código por medio de un buzón de canvas habilitado para esta tarea.

Atraso: Cada Grupo cuenta con un cupón de atraso para las Tareas el cual entrega dos días adicionales para entregar la tarea. Para ocuparlo solo deben realizar una entrega pasado la fecha señalada en el enunciado. Realizar la entrega atrasada sin contar con el cupón implicara obtener nota mínima en la entrega.

Integridad académica

Este curso se adscribe al Código de Honor establecido por la Escuela de Ingeniería. Todo trabajo evaluado en este curso debe ser hecho **individualmente** o en **los grupos asignados** según sea definido en la evaluación y **sin apoyo de terceros**. Se espera que los alumnos mantengan altos estándares de honestidad académica, acorde al Código de Honor de la Universidad. Cualquier acto deshonesto o fraude académico está prohibido; los alumnos que incurran en este tipo de acciones se exponen a un Procedimiento Sumario. Es responsabilidad de cada alumno conocer y respetar el documento sobre Integridad Académica publicado por la Dirección de Pregrado de la Escuela de Ingeniería.

¡Éxito! :)