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

IIC3745 — Testing
2023 - 1

Tarea 2

Fecha de entrega: Martes 4 de Abril 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. Varias de las practicas de testing vistas en las clases teóricas se pueden realizar de dos formas: automática y manual. 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.

Considere la lista de reglas de warnings mostrados en la Tabla 1.

Nombre y descripción	**Ejemplo**

`LongVariableName` - Se agregará un warning por cada variable que tenga un nombre que exceda de los 15 caracteres. Se debe considerar dos casos: variables temporales y variables de instancia. Cada warning debe ser generado en base al siguiente formato:
 	
`Warning("VariableLongName", <codeline-variable>, "variable " + <var-name> + " has a long name")`
	`variableswithlongnames = []`
	`variableswithshortnames = []`
	
	`self.variableswithlongnames = []`
	`self.variableswithshortnames = []`
	`...`

`UnusedArgument` - Se agregará un warning por cada argumento en la definición de un método que no es usado dentro del mismo método. En el ejemplo, el argumento `x` no está siendo usado. Cada warning debe ser generado en base al siguiente formato: `Warning("UnusedArgument", <codeline-functionDef>, "argument " + <unusedArg-name> + " is not used")`	`def example(x, y, z):` `  return y + z`

`SuperInitNotCalled` - Se agregará un warning cada vez que en el método `__init__` de una subclase no se llame al método `__init__` de su super clase. En el ejemplo, la clase `Circle` es subclase de `Shape` y en el método `__init__` de `Circle` no se llama al método `__init__` de `Shape`. Cada warning debe ser generado en base al siguiente formato:
 	
`Warning("SuperInitNotCalled", <codeline-subclassDef>, "subclass " + <subclass-name> + " does not call to super().__init__()")`
	`class Shape:`
	`  def __init__(self, name):`
	`  self.name = name`
	`  print("Creating" + self.name)`
	
	`class Circle(Shape):`
	`  def __init__(self):`
	`  print("Creating circle")`

Tabla 1: Descripción de las tres reglas que reportan warnings para el Ejercicio 1.

También considere la lista de transformaciones mostradas en la Tabla 2.

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

Nombre	Antes	Después
`MinusEquals`	`x = x - y`	`x -= y`

`SimplifiedIf`	`def example1(x, y):`
	`  return True if <expression> else False`
	
	`def example2(x, y):`
	`  return False if <expression> else True`
		`def example1(x, y):`
		`  return <expression>`
		
		`def example2(x, y):`
		`  return not <expression>`

Tabla 2: Descripción de las dos transformaciones para el Ejercicio 2.

Tarea

Desarrolle los siguientes ejercicios:

- ***Ejercicio 1*** – Implemente las clases: `LongVariableNameRule`, `UnusedArgumentRule` y `SuperInitNotCalledRule` que heredan de la clase `Rule` y están descritas en la tabla 1. Además, implemente el método `analyze` para cada una de ellas y el código que crea necesario para su correcto funcionamiento. También, verifique que su implementación funcione como se espera:
  - **(a)** Verifique que los test del archivo `test_tarea.py` pasen. Tenga en cuenta que el código no debe estar hardcodeado. Note que asignará un puntaje entre 0 y 2, en base al número de tests que pasen (2 puntos).
  - **(b)** Implemente un caso de prueba en el archivo `test_adicionales.py` por cada regla implementada (siga el template del archivo). Los casos de prueba deben demostrar que cada regla funciona para algún otro caso que no fue considerado en el archivo `test_tarea.py`. Adicionalmente, codifique los archivos: `longVariableName.py`, `unusedArgument.py` y `superInitNotCalled.py` en el folder `extra-test-code`. Estos archivos deben tener el código a ser analizado por el respectivo caso de prueba adicional propuesto. Note que los test adicionales se ejecutarán, y se asignará un puntaje entre 0 y 2 en base a los casos considerados y al número de tests que pasen (2 puntos).
- ***Ejercicio 2*** – Implemente las clases: `MinusEqualsRewriterCommand` y `SimplifiedIfRewriterCommand` que heredan de la clase `RewriterCommand` y corresponden a los transformadores de código mostrados en la tabla 2. Además, implemente el método `apply` para cada una de las clases y el código que crea necesario para su correcto funcionamiento. También, verifique que su implementación funcione como se espera:
  - **(a)** Verifique que los tests del archivo `test_tarea.py` pasen. Tenga en cuenta que el código no debe estar hardcodeado. Note que asignará un puntaje entre 0 y 1, en base al número de tests que pasen (1 punto).
  - **(b)** Implemente un caso de prueba en el archivo `test_adicionales.py` por cada transformador implementado (siga el template del archivo). Los casos de prueba deben demostrar que cada transformador funciona para algún otro caso que no fue considerado en el archivo `test_tarea.py`. Adicionalmente, codifique los archivos Python `minusEquals.py` y `simplifiedIf.py` en el folder `extra-test-code`, estos archivos deben tener el código a ser analizado por el respectivo caso de prueba adicional propuesto. También codifique los archivos `expectedMinusEquals.py` y `expectedSimplifiedIf.py` en el folder `extra-test-code`, estos archivos deben tener el código que representa la respuesta esperada luego de aplicar los respectivos transformadores. Note que los test adicionales se ejecutarán, y se asignará un puntaje entre 0 y 1 en base a los casos considerados y al número de tests que pasen (1 punto).

idea vale: imports al inicio, variables declarativas (test más dificil).

- ***Ejercicio 3 (Bonus)***– Proponga e implemente una regla o transformación que sea útil para ayudar a los profesores de Introducción a la Programación. Se publicarán los videos en canvas y un formulario para la votación de las mejores propuestas. El primer lugar tendrá +0.2 décimas y el segundo lugar +0.1 décimas. Las décimas irán a la evaluación donde cada estudiante quiera mejorar su nota (interrogación, tarea o proyecto). Importante esta actividad opcional **no puede ser entregada con atraso** ya que las votaciones comienzan el día siguiente a la fecha de entrega de la tarea. La duración del vídeo **no debe exceder 1 minuto y debe ser entendible sin tener que ralentizarlo** o puede ser descalificado.

Nota: Verifique que los casos de prueba de los archivos `test_tarea.py` y los tests propuestos por ustedes `test_adicionales.py` deben pasar. Tenga en cuenta que al menos deben existir 5 casos de prueba definidos en `test_adicionales.py` (uno por cada regla/transformador) y que se debe seguir la estructura definida en el archivo (llenar los comentarios con la información solicitada, fijarse que las clases implementadas tengan el nombre deseado, etc).

Entregables

- Como se menciono anteriormente, por cada regla/transformacion del ejercicio 1 y 2, usted debe diseñar y codificar un caso de prueba adicional con su respectivo código a ser analizado (como se vio en clase) para probar que las reglas y transformaciones solicitadas funcionan como se espera.
  - Los códigos a ser analizados deben ser diferentes.
  - Los casos de prueba adicionales deben ser diferentes a los entregados en `test_tarea.py`.
  - Deben incluir un comentario por cada caso de prueba adicional (como se observa en `test_tarea.py`). Este comentario debe contener (i) el nombre del test, (ii) el nombre del archivo que contiene el código a ser analizado, (iii) la descripción del caso nuevo que se considero y (iv) el resultado deseado. En cuanto a los casos de prueba adicional, usted puede contemplar casos en los que el código a ser analizado es más complejo (e.g., caso considerando muchos if else anidados) o casos en los cuales una regla o transformador no aplica.
- Debe subir el código de su tarea.
- Debe entregar un archivo README que contiene los nombres de quienes realizaron la tarea y su aporte (Declaración de tarea).
- Los que realicen la actividad 3 deben subir la url a un vídeo explicando solo la regla o transformación desarrollada en dicha actividad. El vídeo debe tener una duración que no supere **un minuto** y debe ser entendible sin tener que ralentizar el vídeo.

Archivos iniciales

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

- *input code (folder)* – contiene el código utilizado en clases para probar las reglas de detección y transformación.
- *rewriter (folder)* – contiene las reglas de transformación implementadas en clase.
- *rules (folders)* – contiene las reglas de detección de code-smells implementadas en clase.
- *transformed-code (folder)* – contiene el resultado de aplicar las reglas de transformación sobre el código de entrada (input code).
- *test-code (folder)* – contiene el código utilizado para probar si las reglas de detección y transformación de la tarea funcionan como se espera.
- *expected-code (folder)* – contiene los archivos Python con el código deseado después de aplicar transformadores (para testear transformadores hechos en clase y los de la tarea).
- *extra-test-code (folder)* – es un folder vacio donde usted debe agregar los archivos Python que contienen el código a ser analizado para validar las reglas/transformadores implementados segun sus casos de prueba propuestos.
- *analyze.py* – ejecuta todas las reglas de detección de code-smells sobre el código de entrada (input code).
- *transform.py* – ejecuta todas las reglas de detección de transformación sobre el código de entrada (input code).
- *test_clases.py* – contiene el código de prueba desarrollado en clase.
- *test_tarea.py* – contiene el código de prueba para el ejercicio 1 y 2.
- *test_adicionales.py* – contiene la estructura para que agreguen un caso de prueba extra por cada regla/-transformador segun la tarea.

Para la tarea usted debe modificar o agregar código a los archivos 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* con copia a *afernandb@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 algun 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 3.9 o superior**.
- 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. 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 el código que utilizaron para su tarea junto con los ejemplos que utilizaron para probar su funcionamiento. La entrega se realizara 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.

Vídeos: Si corresponde, deben subir los vídeos a YouTube e incluir la url en un archivo .txt que entregaran en el buzón junto con el código. Importante que los vídeos no se excedan de la duración máxima establecida. El vídeo debe estar en configuración **"No Listado", o sea, que solo se puedan acceder desde un url**[2].
Atraso: Se efectuara un descuento por entregar tareas atrasadas. Se descontara 0.5 si la tarea se entrega con menos de una hora de retraso. El puntaje final en caso de atraso seria calculado mediante la siguiente formula:

$$PuntajeFinal = PuntajeObtenido - (0,5 + 0,05 \cdot k)$$

, donde k es el numero de horas de retraso menos uno.

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

[2]Con el fin de poder resguardar la integridad académica

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! :)