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

IIC3745 — Testing
2024 - 1

Tarea 1

Fecha de entrega: Martes 9 de Abril del 2024 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 manualmente y automáticamente. 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 consta en extender los tipos de alertas y transformaciones soportadas por la librería desarrollada en clase.

Alertas

A continuación se detallan las alertas a implementar:

Invalid Name (10 pts) – Se agregará una alerta por cada nombre de clase, función o método que no siga las convenciones de nombre de Python[2]. La alerta a crear tendrá el respectivo formato, dependiendo del tipo de elemento que tenga nombre invalido:

```
Warning('InvalidName', <line_number>, 'invalid class name ' + <class_name>)
Warning('InvalidName', <line_number>, 'invalid function name ' + <function_name>)
Warning('InvalidName', <line_number>, 'invalid method name ' + <method_name>)
```

El número de línea debe ser la línea donde esta definida la clase, la función o el método en cuestión. Por ejemplo, considere el siguiente código en Python, el mismo debería lanzar una alerta de `InvalidName` para clase en la línea 1 y para metodo en la línea 2.

```
class cat:
    def Meow(self, number_of_meow):
        print("Meow" * number_of_meow)
        return number_of_meow
```

Por lo tanto, se tendrían las siguientes alertas:

```
Warning('InvalidName', 1, 'invalid class name cat')
Warning('InvalidName', 2, 'invalid method name Meow')
```

Simplifiable If (10 pts) – Se agregará una alerta por cada if que pueda ser simplificado por su condición o la negación de la condición. En especifico, se tomará en cuenta cuando un if ternario o if solo retorne True o False. También se agregará una alerta cuando un if statement tiene como "body" u "orelse" una asignación a la misma variable con los valores True o False. La alerta creada debe seguir el siguiente formato:

```
Warning('SimplifiableIf', <line_number>,  'if statement can be replaced with a bool(
    test)')
```

Por ejemplo, considere el siguiente código en Python, el mismo debería lanzar una alerta de `SimplifiableIf` en la línea 2 debido a que el if statement puede ser reemplazado por $res = not(x \leq 1000)$. Además, se levantará una alerta de `SimplifiableIf` en la línea 8 debido a que el if ternario puede ser reemplazado por $return(y == 0)$.

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

[2] En Python, la convención de nombres para las clases es satisfecha por la expresión regular: `[A-Z_][a-zA-Z0-9]+$` Mientras que para los nombres de funciones/métodos se tiene la siguiente expresión regular: `[a-z_][a-z0-9_]{2,30}$`

```python
def greaterThan1000(x):
    if (x <= 1000):
        res = False
    else:
        res = True

def foo(y):
    return True if y == 0 else False
```

Unused Argument (15 pts) – Se agregará una alerta por cada argumento en la definición de un método o función que no es usado dentro del mismo método o función. Cada warning debe ser generado en base al siguiente formato::

```
Warning('UnusedArgument', <line_number>, <argument_name> + ' argument has not been used!')
```

Para ilustrar, en el código de ejemplo, el argumento `x` no está siendo usado. Por lo que una alerta `UnusedArgument` debe ser creada, en la línea 1.

```python
def example(x, y, z):
    return y + z
```

Uninitialized Attribute (15 pts) – Se agregarán alertas en caso de que algun atributo de una clase no sea inicializado debidamente. Considere que un atributo fue inicializado debidamente si es que lo fue en el método `__init__`. La alerta debe tener el siguiente formato:

```
Warning('UninitializedAttribute', <line_number>, <attribute_name> + ' attribute was not initialized!')
```

Por ejemplo, en el siguiente código, se tendrá una alerta de tipo `UninitializedAttribute` en la línea 3, debido a que `firstName` y `lastName` no fueron inicializados en el método `__init__`.

```python
class Person:
    def fullName(self):
        return self.firstName + self.lastName
```

Transformaciones

A continuación se detallan las transformaciones a implementar:

Operator Equals (15 pts) – Este transformador busca todas las asignaciones que se pueden simplificar usando: `+=`, `-=`, `*=`, `/=`, `%=`, `**=` y `//=`. Por ejemplo si se tiene el código:

```
a = a + 7
b = b - c
d = d * a
e = e / 2
f = f % 5
x = x ** y
z = z // 3
```

La transformación es la siguiente:

```
a += 7
b -= c
d *= a
e /= 2
f %= 5
x **= y
z //= 3
```

Simplified True Rewriter (15 pts)– Este transformador busca todos if ternarios que puedan ser simplificados usando la condición o la negación de la condición. Para ilustrar, considere el siguiente código:

```
def example1(x, y):
    return True if x > y else False
def example2(z):
    return False if z else True
```

La transformación respectiva seria:

```
def example1(x, y):
    return x > y
def example2(z):
    return not z
```

Tarea

Desarrolle los siguientes ejercicios:

- ***Ejercicio 1*** – Implemente las clases visitor y rule para cada Alerta:
  - `InvalidNameVisitor` y `InvalidNameRule` en `rules\invalid_name.py`.
  - `SimplifiableIfVisitor` y `SimplifiableIfRule` en `rules\simplifiable_if.py`.
  - `UnusedArgumentVisitor` y `UnusedArgumentRule` en `rules\unused_argument.py`.
  - `UninitializedAttributeVisitor` y `UninitializedAttributeRule` en `rules\uninitialized_attribute.py`.

  Todas las alertas anteriores se encuentran definidas en la sección de Alertas. Puede crear clases o métodos adicionales que considere necesarias dadas las clases pedidas. Se le entregarán 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:
  - `OperatorEqualsCommand` en `transformers\operator_equals_rewriter.py`.
  - `SimplifiedIfCommand` en `transformers\simplified_if_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:

- El puntaje a cada regla o transformación será asignado basado en la cantidad de tests que pasen exitosamente. Se consideraran tanto los tests entregados en los archivos iniciales, como los tests secretos para verificar más casos.
- 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 más tests proporcionados en el enunciado de forma intencionada. Las soluciones deben apuntar a resolver el problema para cualquier código que se le envié de input.

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.

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 *nicolas.olmos@uc.cl* explicando en detalle lo ocurrido. Posterior a eso se revisara en detalle la contribución del equipo 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 recibirá la 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. Si el grupo decide activarlo en una tarea podrán realizar la entrega dos días después de la fecha original de entrega. Para ocuparlo solo deben realizar una entrega pasado la fecha señalada en el enunciado. Realizar la entrega atrasada sin contar con un cupón para usar 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! :)