УДК 004

Использование модуля ast для структурного анализа классов в исходном коде Python

Вихляев Дмитрий Романович
Приамурский государственный университет имени Шолом-Алейхема
Студент

Аннотация
Данная статья содержит описание модуля ast и его основных методов парсинга исходного кода python. Для реализации используется модуль ast входящий в интерпретатор Python. В результате исследования будет создана структурная диаграмма классов, сгенерированная из проекта с файлами исходного кода.
Ключевые слова: Python, парсинг, абстрактно синтаксическое дерево.

Using the ast module for structural analysis of classes in Python source code

Vikhlyaev Dmitry Romanovich
Sholom-Aleichem Priamursky State University
Student

Abstract
This article contains a description of the ast module and its main methods of parsing python source code. The ast module included in the Python interpreter is used for implementation. As a result of the research, a structural class diagram generated from the project with source code files will be created.
Keywords: Python, parsing, abstract syntax tree.

1 Введение

1.1 Актуальность

Модуль ast реализует абстрактное синтаксическое дерево языка программирования Python. Данный модуль входит в стандартную библиотеку, что значит, устанавливается вместе с интерпретатором Python. Сам абстрактный синтаксис языка может изменяться от версии к версии. Поэтому этот инструмент может помочь разработчику понять, как выглядит текущая грамматика. Модуль лучше всего подходит для проведения парсинга исходного кода, так как поставляется разработчиками языка. Модуль позволяет изменять узлы и создавать новые узлы, что полезно для метапрограммирования и написания инструментов, таких как линтеры и трансформеры кода. После модификации дерева синтаксиса, можно сгенерировать изменённый исходный код с помощью модуля ast.

1.2Обзор исследований

Н.А.Субботин реализовала преобразователь python-кода в ast, совместимый с PT.PM [1]. В.Р.Столяров исследовал построение ast встроенными средствами языка программирования python [2]. Д.А.Орлов создал алгоритм поиска идиом в исходных текстах программ, использующий подсчет поддеревьев [3]. С.А.Голубев, И.Н.Муренин, Е.С.Новикова провели анализ подходов к формированию атрибутов для анализа вредоносного кода на основе его графического представления [4]. И.С.Скрыпников разработал статический анализатор кода на основе взаимодействия интервального анализа и анализа указателей [5]. В.О.Афанасьев, А.Е.Бородин, К.И.Вихлянцев, А.А.Белеванцев провели статический анализ на основе обобщённого абстрактного синтаксического дерева [6]. Е.В.Коренькова, Ю.Д.Кореньков обосновали подход к инстанцированию высокоуровневых синтаксических моделей предметноориентированных языков на основе деревьев разбора [7]. П.Д.Волхонцева, О.П.Серебренников, М.А.Ядерцова изучили структуру и алгоритмы абстрактного синтаксического дерева для статического анализа кода [8].

1.3Цель исследования

Цель исследования – описать способ парсинга классов, его атрибутов и методов в python программе с помощью модуля ast.

2 Материалы и методы

Для описания процессов работы применяются готовые примеры кода. Создание диаграмм осуществляется через установленный модуль graphviz.

3 Результаты и обсуждения

Функция ast.parse() является одной из ключевых функций модуля ast в Python, которая позволяет преобразовать строку с исходным кодом Python в абстрактное синтаксическое дерево (AST). Это дерево представляет структуру кода в виде объектов Python, которые могут быть проанализированы и модифицированы программно. Функция ast.parse() принимает исходный код в виде строки и возвращает объект дерева синтаксиса. Метод принимает три возможных параметра. Обязательный «source» строка с исходным кодом Python, который нужно проанализировать. Дополнительный параметр «filename» имя файла, из которого был получен исходный код. Это имя используется в диагностических сообщениях. По умолчанию значение "<unknown>". Третий параметр «mode» режим парсинга. Может быть "exec", "eval" или "single". Режим "exec" (по умолчанию) используется для парсинга модуля или скрипта. Режим "eval" используется для парсинга выражения, а режим "single" используется для парсинга единственной инструкции.

Функция ast.dump() используется для получения строкового представления абстрактного синтаксического дерева (AST). Это полезно для отладки и анализа структуры дерева, поскольку позволяет визуально

оценить, как Python интерпретирует и разбирает исходный код. node: корневой узел AST (обычно объект, возвращаемый функцией ast.parse()).

Имеет параметры логического типа «annotate_fields» и «include_attributes» если True, добавляет аннотации полей в выходное представление и включает атрибуты, такие как номера строк и смещения в выходное представление соответственно. Параметр «indent» форматирует вывод с указанным количеством пробелов для каждого уровня вложенности. По умолчанию None (однострочный вывод).

Пример простого использования приведён на рисунке 1. Строка исходного кода описывает создание класса с одной функцией инициализации. Результат парсинга представлен на рисунке 2.

```
import ast

code="""
class My:
    def __init__(self):
        print('g')

"""
print(ast.dump(ast.parse(code),indent=4))
```

Рис. 1. Простой пример парсинга

```
Module(
    body=[
        ClassDef(
            name='My',
            bases=[],
            keywords=[],
            body=[
                FunctionDef(
                    name='__init__',
                    args=arguments(
                        posonlyargs=[],
                        args=[
                            arg(arg='self')],
                        kwonlyargs=[],
                        kw_defaults=[],
                        defaults=[]),
                    body=[
                        Expr(
                            value=Call(
                                func=Name(id='print', ctx=Load()),
                                args=[
                                    Constant(value='g')],
                                keywords=[]))],
                    decorator_list=[])],
            decorator_list=[])],
    type_ignores=[])
```

Рис. 2. Представление структуры AST

Класс ast.NodeVisitor в модуле ast используется для обхода абстрактного синтаксического дерева (AST). Он предоставляет удобный способ обработки узлов дерева, позволяя определять методы для обработки различных типов узлов. Это особенно полезно для анализа и трансформации кода. Чтобы использовать ast.NodeVisitor, необходимо создать подкласс и

переопределить методы visit_<NodeType> для узлов, которые необходимо обрабатывать. Метод generic_visit вызывается для всех узлов, для которых не определены конкретные методы (рис.3).

```
class ClassVisitor(ast.NodeVisitor):
    def __init__(self):
        self.classes = []
```

Рис. 3. Класс наследник ast.NodeVisitor

Для того чтобы найти все классы и их функции в дереве AST, можно расширить подход с использованием ast.NodeVisitor. В этом примере создаётся метод, который будет находить все классы их методы и атрибуты.

```
def visit_ClassDef(self, node):
    class_info = {
        "name": node.name,
        "methods": [],
        "attributes": []
    }
    for item in node.body:
        if isinstance(item, ast.FunctionDef):
            class_info["methods"].append(item.name)
            # Рекурсивно обходим узлы внутри функции-метода
            for subitem in item.body:
                if isinstance(subitem, (ast.Assign, ast.AnnAssign)):
                    assign = self.find_attributes(subitem)
                    if(assign is not None):
                        class_info["attributes"].append(assign)
        elif isinstance(item, (ast.Assign, ast.AnnAssign)):
            assign = self.find_attributes(item)
            if(assign is not None):
                class_info["attributes"].append(assign)

    self.classes.append(class_info)
    self.generic_visit(node)
```

Рис. 4. Метод, реагирующий на найденный класс

Метод find_attributes() в классе принимает узел AST в качестве аргумента и проверяет, является ли этот узел присваиванием (ast.Assign). Если узел является присваиванием, метод извлекает список целей (targets) из этого узла. Если узел не является присваиванием, метод предполагает, что это одиночное присваивание и извлекает одну цель (item.target). Затем метод проходит по всем целям и проверяет, является ли каждая из них атрибутом (ast.Attribute). Если цель является атрибутом, метод возвращает имя атрибута (target.attr). Если ни одна из целей не является атрибутом, метод возвращает None. (рис.5).

```
def find_attributes(self,item):
    if isinstance(item, ast.Assign):
        targets = item.targets
    else:
        targets = [item.target]

    for target in targets:
        if isinstance(target, ast.Attribute):
            return target.attr
    return None
```

Рис. 5. Метод поиска атрибутов

Специально для задачи написан модуль, который из указанной директории собирает все файлы с расширением py и читает исходный код каждого. Строка сначала отправляется на обработку ast, а затем в созданный класс (рис.6).

```
def parse_project(direcory):

    visitor = ClassVisitor()

    files = read_pyfile.get_python_files(direcory)
    for file in files:
        code=read_pyfile.read_python_file(file)
        tree = ast.parse(code)
        visitor.visit(tree)
    return visitor.classes
```

Рис. 6. Пример использования созданного класса

Результатом является массив классов. Для удобного представления результата, также написан метод генерации диаграмм с сохранением в изображение диаграммы (рис.7).

```
from graphviz import Digraph

def create_class_diagram(classes):
    dot = Digraph(comment='Class Diagram')

    for cls in classes:
        label = '{{{}|{}|{}}}'.format(
            cls["name"],
            '\\l'.join(cls["attributes"]) + '\\l',
            '\\l'.join(cls["methods"]) + '\\l'
        )
        dot.node(cls['name'], label=label, shape='record')

        dot.render('class_diagram', format='png', cleanup=True)
```

Рис. 7. Генерация диаграммы классов с помощью graphviz

На рисунке 8 представлены примеры исходных кодов четырёх классов, находящихся в указанной директории. Используя парсер классов и вспомогательные методы, в папке с исходным кодом появится изображение в формате png со структурным описанием классов. Результат представлен на рисунке 9.

```
class BankAccount:
    def __init__(self, account_number, owner, balance=0):
        self.account_number = account_number
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
        print(f"Deposited {amount}. New balance: {self.balance}")

    def withdraw(self, amount):
        if 0 < amount <= self.balance:
            self.balance -= amount
        print(f"Withdrew {amount}. New balance: {self.balance}")

    def get_balance(self):
        return self.balance

    def get_account_info(self):
        return f"Account Number: {self.account_number}, Owner: {self.owner}
```

```
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages
        self.current_page = 1

    def read_page(self):
        if self.current_page < self.pages:
            self.current_page += 1
        print(f"Reading page {self.current_page} of {self.title}")

    def bookmark_page(self, page):
        if 1 <= page <= self.pages:
            self.current_page = page
        print(f"Bookmarked page {self.current_page} of {self.title}")

    def get_book_info(self):
        return f"Title: {self.title}, Author: {self.author}, Pages: {self.pages}

    def get_current_page(self):
        return self.current_page
```

```python
class Car:
    def __init__(self, make, model, year, fuel_level=100):
        self.make = make
        self.model = model
        self.year = year
        self.fuel_level = fuel_level

    def drive(self, distance):
        fuel_needed = distance / 10
        if self.fuel_level >= fuel_needed:
            self.fuel_level -= fuel_needed
            print(f"Drove {distance} km. Fuel level: {self.fuel_level}")
        else:
            print("Not enough fuel to drive")

    def refuel(self, amount):
        self.fuel_level += amount
        print(f"Refueled {amount} liters. Fuel level: {self.fuel_level}")

    def get_car_info(self):
        return f"{self.year} {self.make} {self.model}"

    def get_fuel_level(self):
        return self.fuel_level
```

```python
class Student:
    def __init__(self, name, student_id, grades=None):
        self.name = name
        self.student_id = student_id
        self.grades = grades if grades is not None else []

    def add_grade(self, grade):
        self.grades.append(grade)
        print(f"Added grade: {grade}. Grades: {self.grades}")

    def get_average_grade(self):
        if self.grades:
            average = sum(self.grades) / len(self.grades)
            return average
        return 0

    def get_student_info(self):
        return f"Name: {self.name}, Student ID: {self.student_id}"

    def get_grades(self):
        return self.grades
```

Рис. 8. Реализация классов

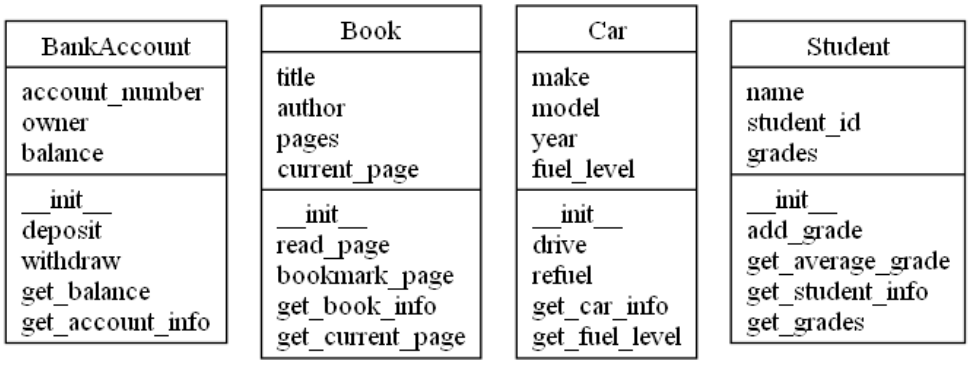

Рис. 9. Диаграмма структуры классов

В результате данной статьи, был описан способ проведения парсинга исходного кода Python для структурного анализа классов с помощью стандартного модуля ast. Представлены используемые методы и примеры работы функций.

Библиографический список

1. Субботин Н.А. Преобразователь python-кода в ast, совместимый с PT.PM // Решетневские чтения. 2018. Т. 2. С. 350-351.
2. Столяров В.Р. Исследование построения ast встроенными средствами языка программирования python // В сборнике: Молодой исследователь: вызовы и перспективы. Сборник статей по материалам CCLXIII международной научно-практической конференции. Москва, 2022. С. 316-319.
3. Орлов Д.А. Алгоритм поиска идиом в исходных текстах программ, использующий подсчет поддеревьев // Программные продукты и системы. 2022. № 1. С. 065-074.
4. Голубев С.А., Муренин И.Н., Новикова Е.С. Анализ подходов к формированию атрибутов для анализа вредоносного кода на основе его графического представления // В сборнике: Информационная безопасность регионов России (ИБРР-2021). Материалы XII Санкт-Петербургской межрегиональной конференции. Санкт-Петербург, 2021. С. 75-76.
5. Скрыпников И.С. Статический анализатор кода на основе взаимодействия

интервального анализа и анализа указателей // Молодой ученый. 2016. № 23 (127). С. 19-24.
6. Афанасьев В.О., Бородин А.Е., Вихлянцев К.И., Белеванцев А.А. Статический анализ на основе обобщённого абстрактного синтаксического дерева // Труды Института системного программирования РАН. 2023. Т. 35. № 6. С. 103-120.
7. Коренькова Е.В., Кореньков Ю.Д. Подход к инстанцированию высокоуровневых синтаксических моделей предметноориентированных языков на основе деревьев разбора // Научно-технический вестник Поволжья. 2024. № 4. С. 259-264.
8. Волхонцева П.Д., Серебренников О.П., Ядерцова М.А. Абстрактные синтаксические деревья для статического анализа кода // Вопросы устойчивого развития общества. 2022. № 5. С. 914-922.