中国农业大学

编译原理实验

（2023-2024 学年春季学期）

实验名称： Python 编程语言设计

任课教师： 王耀君

班　　级： 计算机 211

学　　号： 2020301010225

姓　　名： 夏婉可

时　　间： 2024/07

目录

1：编译原理角度切入的 Python 概述

Python 是一种高级编程语言，该语言的主要特点是在设计阶段强调代码的可读性和简洁性，即使用显式语法结构和缩进来定义代码块。Python 支持多种编程范型，包括结构化、过程式、反射式、面向对象和函数式编程。Python 广泛用于各种应用领域，如 Web 开发、数据科学、人工智能、自动化脚本等。

Python 虽然是一种解释型语言，但在其执行过程仍然涉及编译原理中的多个关键阶段，包括词法分析、语法分析、语义分析、中间代码生成、代码优化、目标代码生成等主要阶段。这些阶段帮助将高级 Python 代码转换为可执行形式，使得 Python 程序能够在不同平台上运行。

1.1：Python 的词法分析

词法分析是编译器的第一个阶段，其任务是将源代码转换成一系列记号。这些记号是源代码的基本组成部分，如关键字、标识符、运算符和标点符号。

输入	源代码文本（字符串序列）
输出	记号序列，每个记号包含类型和文本值，例如标识符、数字、操作符等

表 1-1：词法分析的输入和输出内容

Python 使用了一个词法分析器来扫描源代码并生成记号序列。例如，下面的 Python 源代码经过词法分析器，将转换成 3 个部分，分别是标识符（x）、等号（=）、数字（42）。

Python 示例
`x = 42`

表 1-2：Python 的词法分析示例

1.2：Python 的语法分析

语法分析是将记号序列转换成语法树（通常为抽象语法树，缩写为 AST）。语法分析器检查记号序列是否符合语言的语法规则，并构建一个树状结构来表示代

码的层次结构。

输入	记号序列
输出	抽象语法树（AST）

表 1-3：语法分析的输入和输出内容

Python 的语法分析器根据 Python 语法规则生成 AST。例如，上述经过词法分析得到的记号序列，将对应一棵以赋值操作 Assign 为根节点的语法树，其左子树为变量 Name，右子树为常量 Constant。

Python 示例

```
Assign
  ├── Name (id='x')
  └── Constant (value=42)
```

表 1-4：Python 的语法分析示例

1.3：Python 的语义分析

语义分析检查 AST 是否符合语言的语义规则，这包括类型检查、变量作用域检查等。语义分析确保代码在语法上正确的同时，其含义也是正确的。

输入	抽象语法树（AST）
输出	经过语义检查的 AST（包含类型信息和其他注释）

表 1-5：语义分析的输入和输出内容

Python 的语义分析器的任务包括确保变量在使用前已定义，检查类型兼容性等。例如，下面将整型变量和字符串进行相加的代码，在语义分析阶段会发现其加法操作是不合法的。

Python 示例

```python
x = 42
y = x + "hello"
```

表 1-4：Python 的语义分析示例

1.4：Python 的中间代码生成

经过语义分析的 AST，通常会被转换成一种中间表示（缩写为 IR），中间代

码与目标机无关，通常可以采用三地址码表示。因此，中间代码更接近机器语言，但仍保持一定的抽象层次。

输入	经过语义检查的 AST
输出	中间代码

表 1-5：中间代码生成的输入和输出内容

1.5：Python 的代码优化

代码优化阶段对中间代码进行各种优化，以提高代码运行效率。代码优化的任务主要包括消除冗余代码、循环优化等。

输入	中间代码
输出	优化后的中间代码

表 1-6：代码优化的输入和输出内容

1.6：Python 的目标代码生成

目标代码生成阶段将优化后的中间代码转换成具体机器的可执行代码，例如机器码。最终阶段将生成的目标代码和库代码链接在一起，形成最终的可执行文件。

输入	优化后的中间代码
输出	目标代码

表 1-7：目标代码生成的输入和输出内容

1.7：Python 的编译器实现

CPython 是 Python 的主要实现，它将 Python 代码编译成字节码，然后由 Python 虚拟机解释执行。其核心要点主要有以下 3 个部分：

（1） 词法分析和语法分析：将 Python 源代码转换为 AST。

（2） 字节码生成：将 AST 转换为字节码。

（3） 字节码解释：Python 虚拟机逐行解释执行字节码。

2：基于 Python 的模块改进

本次实验的任务主要是通过设计和实现一系列扩展模块，改进和增强 Python 语言的功能。这些改进的模块涵盖了从词法和语法分析到高级数学运算、错误处理、数据结构操作等多个方面，进而为用户提供更全面的编程工具和更好的编程体验。

本次实验主要专注于改进现有 Python 代码中的三个主要模块：**Lexer（词法分析器）、SyntaxAnalyzer（语法分析器）、IOExtension（代码优化——I/O 格式化输入输出）**。

2.1：主程序逻辑解释

在主程序 test.py 文件中，主要包括导入模块、类定义、主程序流程、异常处理、退出机制等多个部分。

主程序展示了如何使用 Python 中的多种模块和功能，通过交互式控制台界面提供了一个综合性的编程环境和工具集合。

2.1.1：导入模块

在导入模块部分，主程序导入了一系列标准库和自定义模块，例如 re、ast、autopep8、textwrap、numpy、inspect、threading、time、datetime 等第三方依赖包，用于支持不同的功能和操作。以下将具体阐述上述第三方依赖包的功能和示例代码。

（1）re 依赖包的作用主要是：正则表达式匹配和搜索、字符串替换和分割、提取字符串中的特定模式。

Python 示例

```python
import re

pattern = r'\b\d+\b'
text = "There are 24 hours in a day."
```

```
matches = re.findall(pattern, text)
print(matches)  # 输出: ['24']
```

表 2-1：re 依赖包的 Python 示例

（2）ast 依赖包的作用主要是：解析 Python 源码为 AST、分析和修改 AST、将 AST 转换回源代码。

Python 示例

```
import ast

code = "x = 42"
tree = ast.parse(code)

print(ast.dump(tree))  # 输出代码的 AST 表示
```

表 2-2：ast 依赖包的 Python 示例

（3）autopep8 依赖包的作用主要是：自动修复 PEP 8 风格问题、提高代码可读性和一致性。

Python 示例

```
import autopep8

code = "def foo():\n    print( 'Hello, world!' )\n"
formatted_code = autopep8.fix_code(code)

print(formatted_code)
```

表 2-3：autopep8 依赖包的 Python 示例

（4）textwrap 依赖包的作用主要是：包装和填充文本、控制文本的缩进和对齐。

Python 示例

```
import textwrap

text = "This is a very long sentence that needs to be wrapped."
wrapped_text = textwrap.fill(text, width=20)

print(wrapped_text)
```

表 2-4：textwrap 依赖包的 Python 示例

（5）numpy 依赖包的作用主要是：高效的数组操作和运算、数据处理和分析、数学函数。

Python 示例

```python
import numpy as np

array = np.array([1, 2, 3, 4])
print(array * 2)  # 输出: [2 4 6 8]
```

表 2-5：numpy 依赖包的 Python 示例

（6）inspect 依赖包的作用主要是：检查对象的类型和属性、获取源代码和函数名以及参数信息、分析调用堆栈。

Python 示例

```python
import inspect

def example_function():
    pass

print(inspect.getsource(example_function))  # 输出函数的源代码
```

表 2-6：inspect 依赖包的 Python 示例

（7）threading 依赖包的作用主要是：创建和管理线程、线程同步和通信、并发编程。

Python 示例

```python
import threading

def print_numbers():
    for i in range(5):
        print(i)

thread = threading.Thread(target=print_numbers)
thread.start()
thread.join()  # 等待线程完成
```

表 2-7：threading 依赖包的 Python 示例

2.1.2：类定义

本实验根据词法分析、语法分析、代码优化等改进方向，制定了 13 个类模块，各个类模块负责编译过程中的不同功能。每个类模块的概要解释如下：

（1） Lexer 类：词法分析器，用于对输入的代码进行高亮显示。

（2） SyntaxAnalyzer 类：语法分析器，用于解析和分析输入代码的语法结构。

（3） IOExtension 类：输入输出扩展模块，包含格式化输入和格式化输出的方法。

（4） DataStructureExtension 类：数据结构扩展模块，用于处理整数数组、浮点数数组、字符串列表和字典结构。

（5） ErrorHandling 类：错误处理模块，包含处理除零错误、索引错误、键错误和类型错误的方法。

（6） AdvancedFeatures 类：高级功能扩展模块，提供数组运算、统计和幂运算等功能。

（7） FunctionSignature 类：函数签名模块，用于获取函数的签名信息。

（8） CodeFormatter 类：代码格式化模块，用于自动格式化输入的代码。

（9） FileOperations 类：文件操作模块，包含读取和写入文件的方法。

（10）MathOperations 类：数学操作模块，提供阶乘、最大公约数（GCD）、最小公倍数（LCM）等数学计算。

（11）DateTimeOperations 类：日期时间操作模块，支持当前日期时间获取、日期格式化、日期差计算和日期增减。

（12）Encryption 类：加密解密模块，提供文本加密和解密功能。

（13）Logging 类：日志记录模块，用于记录操作的日志信息。

2.1.3：主程序流程

主程序进入一个无限循环，每次循环都提示用户输入代码进行处理，当用户输入 0 的时候，则退出程序。

每次循环内，程序依次执行以下步骤：

- 代码高亮显示（Lexer 模块）；
- 语法分析和格式化（SyntaxAnalyzer 类模块）；
- 格式化输入和输出（IOExtension 类模块）；
- 数据结构扩展（DataStructureExtension 类模块）；

- 错误处理（ErrorHandling 类模块）；
- 高级功能（AdvancedFeatures 类模块）；
- 函数签名获取（FunctionSignature 类模块）；
- 代码格式化（CodeFormatter 类模块）；
- 文件操作（FileOperations 类模块）；
- 数学操作 （MathOperations 类模块）；
- 日期时间操作（DateTimeOperations 类模块）；
- 加密解密（Encryption 类模块）；
- 日志记录（Logging 类模块）；

2.1.4：异常处理

主程序使用 try 和 except 关键字来捕获并处理可能发生的异常，以确保程序可以继续运行或者提供友好的错误提示。

2.1.5：退出机制

当用户输入 0 的时候，可以退出主程序循环，程序结束执行。

2.2：具体改进的类模块

2.2.1：Lexer 类

Lexer 类的代码内容，如表 2-8 所示。Lexer 类中具备初始化函数（init）、高亮方法函数（highlight）。

初始化函数（init）的实现流程如下：

- 设置关键字列表。
- 设置数据类型列表。
- 设置操作符号列表。
- 设置括号列表。

- 设置高亮配置。

高亮方法函数（highlight）的实现流程如下：

- 获取输入代码并进行词法分析，将代码分割成记号流序列。
- 初始化高亮字符串。
- 遍历每个记号，进行分类高亮：
  - 如果记号是关键字，应用关键字高亮。
  - 如果记号是数据类型，应用数据类型高亮。
  - 如果记号是操作符号，应用操作符号高亮。
  - 如果记号是括号，应用括号高亮。
  - 其他记号保持原样。
- 返回高亮后的代码字符串。

```
# 模块一：词法分析器
class Lexer:
    # 初始化
    def __init__(self):
        # 关键字
        self.keywords = [
            # 条件判断
            'if', 'else',
            # 循环
            'for', 'while',
            # 函数和类
            'return', 'def', 'class',
            # 引入依赖
            'import', 'from', 'as',
            # 其他关键字
            'try', 'except', 'finally'
        ]

        # 数据类型
        self.datatypes = [
            # 单变量
            'int', 'float', 'str', 'bool',
            # 多变量集合
            'list', 'dict', 'tuple', 'set',
            # 空
            'None'
        ]
```

```
        # 操作符号
        self.operators = [
            # 四则运算
            '+', '-', '*', '/', '=',
            # 大小判断
            '==', '!=', '<', '>', '<=', '>=',
            # 四则运算简写
            '+=', '-=', '*=', '/=',
            # 与或非
            'and', 'or', 'not'
        ]

        # 括号
        self.brackets = [
            # 小括号
            '(', ')',
            # 中括号
            '[', ']',
            # 大括号
            '{', '}'
        ]

        # 高亮设置
        self.highlights = {
            # 关键字高亮
            'keyword': '\033[95m',
            # 数据类型高亮
            'datatype': '\033[94m',
            # 操作符号高亮
            'operator': '\033[93m',
            # 括号高亮
            'bracket': '\033[92m',
            # 结束高亮
            'end': '\033[0m'
        }

    # 高亮设置
    def highlight(self, code):
        # 获取记号流序列
        tokens = re.split(r'(\W)', code)

        # 高亮字符串初始化
```

```
        highlighted_code = ''

        # 遍历记号流
        for token in tokens:
            # 关键字定位
            if token in self.keywords:
                highlighted_code += self.highlights['keyword'] + token
+ self.highlights['end']
            # 数据类型定位
            elif token in self.datatypes:
                highlighted_code += self.highlights['datatype'] +
token + self.highlights['end']
            # 操作符号定位
            elif token in self.operators:
                highlighted_code += self.highlights['operator'] +
token + self.highlights['end']
            # 括号定位
            elif token in self.brackets:
                highlighted_code += self.highlights['bracket'] + token
+ self.highlights['end']
            # 其他非必要高亮的字符
            else:
                highlighted_code += token

        return highlighted_code
```

表 2-8：Lexer 类的代码

2.2.2：SyntaxAnalyzer 类

SyntaxAnalyzer 类的代码内容，如表 2-9 所示。SyntaxAnalyzer 类中具备初始化函数（init）、获取函数名函数（get_function_signatures）、格式化函数（format_code）、获取抽象语法树函数（get_ast）。

初始化函数（init）的实现流程如下：

- 接收代码并去除公共前导空格和首尾空格。
- 解析代码生成抽象语法树（AST）。

获取函数名函数（get_function_signatures）的实现流程如下：

- 遍历语法树，筛选出函数定义节点。
- 提取每个函数的名称和参数列表，生成函数签名字符串。

- 返回函数签名列表。

格式化函数（format_code）的实现流程如下：

- 使用 autopep8 格式化代码。
- 返回格式化后的代码。

获取抽象语法树函数（get_ast）的实现流程如下：

- 以缩进格式返回抽象语法树的字符串表示。

```
# 模块二：语法分析器
class SyntaxAnalyzer:
    # 初始化，接收记号流 code
    def __init__(self, code):
        # 去除代码的公共前导空格并去除首尾空格
        self.code = textwrap.dedent(code).strip()
        # 解析代码生成抽象语法树
        self.tree = ast.parse(self.code)

    # 获取函数签名
    def get_function_signatures(self):
        # 遍历语法树中的所有节点，筛选出函数定义节点
        functions = [node for node in ast.walk(self.tree) if
isinstance(node, ast.FunctionDef)]

        # 初始化函数签名
        signatures = []

        # 提取每个函数的名称和参数列表，形成函数签名字符串
        for func in functions:
            params = [arg.arg for arg in func.args.args]
            signatures.append(f"Function '{func.name}({',
'.join(params)})'")

        # 返回函数签名
        return signatures

    # 格式化代码
    def format_code(self):
        # autopep8 格式化代码
        formatted_code = autopep8.fix_code(self.code)
        return formatted_code

    # 获取抽象语法树
```

```
    def get_ast(self):
        # 以缩进格式返回抽象语法树的字符串表示
        return ast.dump(self.tree, indent=4)
```

表 2-9：SyntaxAnalyzer 类的代码

2.2.3：IOExtension 类

IOExtension 类的代码内容，如表 2-10 所示。IOExtension 类中具备格式化输入函数（formatted_input）、格式化输出函数（formatted_output）。

- 格式化输入函数（formatted_input）的实现流程如下：显示提示信息，并获取用户输入。
- 尝试将输入转换为指定的数据类型。
- 如果转换成功，返回转换后的值。
- 如果转换失败，提示用户并递归调用自身重新获取输入。

格式化输出函数（formatted_output）的实现流程如下：

- 根据数据类型进行不同的格式化输出。
- 如果数据是浮点数，按两位小数格式输出。
- 如果数据是列表，按列表格式输出。
- 如果数据是字典，按字典格式输出。
- 其他类型的数据，按默认格式输出。

```
# 模块三：I/O 扩展模块
class IOExtension:
    # 静态方法下的格式化输入
    @staticmethod
    def formatted_input(prompt, dtype=str):
        # 显示提示信息，并获取用户输入
        value = input(prompt)
        # 尝试将输入转换为指定的数据类型
        try:
            return dtype(value)
        # 转换失败
        except ValueError:
            # 提示用户
            print(f"Invalid {dtype.__name__}, try again.")
            # 递归调用自身重新获取输入
```

```
            return IOExtension.formatted_input(prompt, dtype)

    # 静态方法下的格式化输出
    @staticmethod
    def formatted_output(data):
        # 数据是浮点数，按两位小数格式输出
        if isinstance(data, float):
            print(f"Formatted Output: {data:.2f}")
        # 数据是列表，按列表格式输出
        elif isinstance(data, list):
            print("List Output: ", data)
        # 数据是字典，按字典格式输出
        elif isinstance(data, dict):
            print("Dict Output: ", data)
        # 其他类型的数据，按默认格式输出
        else:
            print(f"Output: {data}")
```

表 2-10：IOExtension 类的代码

2.2.4：DataStructureExtension 类

DataStructureExtension 类的代码内容，如表 2-11 所示。DataStructureExtension 类中具备初始化函数（init）、添加整数函数（add_int）、添加浮点数函数（add_float）、添加字符串函数（add_str）、添加键值对到字典函数（add_dict）、获取数据结构函数（get_int_array、get_float_array、get_str_array 和 get_dict）。

初始化函数（init）的实现流程如下：

- 初始化整数数组、浮点数数组、字符串列表和字典。

添加整数函数（add_int）的实现流程如下：

- 检查输入是否为整数类型。
- 如果是，添加到整数数组。
- 如果不是，抛出类型错误。

添加浮点数函数（add_float）的实现流程如下：

- 检查输入是否为浮点数类型。
- 如果是，添加到浮点数数组。

- 如果不是，抛出类型错误。

添加字符串函数（add_str）的实现流程如下：

- 检查输入是否为字符串类型。
- 如果是，添加到字符串列表。
- 如果不是，抛出类型错误。

添加键值对到字典函数（add_dict）的实现流程如下：

- 直接将键值对添加到字典。

获取数据结构函数（get_int_array、get_float_array、get_str_array 和 get_dict）的实现流程如下：

- 获取整数数组。
- 获取浮点数数组。
- 获取字符串列表。
- 获取字典结构。

```
# 模块四：数据结构扩展模块
class DataStructureExtension:
    # 初始化
    def __init__(self):
        # 整数数组
        self.int_array = []
        # 浮点数数组
        self.float_array = []
        # 字符串列表
        self.str_list = []
        # 字典
        self.dict_structure = {}

    # 添加整数
    def add_int(self, value):
        # 检查是否为整数类型
        if isinstance(value, int):
            self.int_array.append(value)
        else:
            raise TypeError("Only integers are allowed in int_array")

    # 添加浮点数
    def add_float(self, value):
        # 检查是否为浮点数类型
```

```
        if isinstance(value, float):
            self.float_array.append(value)
        else:
            raise TypeError("Only floats are allowed in float_array")

    # 添加字符串
    def add_str(self, value):
        # 检查是否为字符串类型
        if isinstance(value, str):
            self.str_list.append(value)
        else:
            raise TypeError("Only strings are allowed in str_list")

    # 添加键值对到字典
    def add_dict(self, key, value):
        self.dict_structure[key] = value

    # 获取整数数组
    def get_int_array(self):
        return self.int_array

    # 获取浮点数数组
    def get_float_array(self):
        return self.float_array

    # 获取字符串列表
    def get_str_list(self):
        return self.str_list

    # 获取字典结构
    def get_dict(self):
        return self.dict_structure
```

表 2-11：DataStructureExtension 类的代码

2.2.5：ErrorHandling 类

ErrorHandling 类的代码内容，如表 2-12 所示。ErrorHandling 类中具备处理错误并打印错误信息函数（handle_error）、除法操作函数（divide）、索引错误处理函数（index_error）、键错误处理函数（key_error）、类型错误处理函数（type_error）。

错误信息函数（handle_error）的实现流程如下：

- 打印错误类型和错误信息。

除法操作函数（divide）的实现流程如下：

- 尝试进行除法操作。
- 捕获并处理除零错误、类型错误及其他异常。

索引错误处理函数（index_error）的实现流程如下：

- 尝试获取列表中的元素。
- 捕获并处理索引超出范围错误及其他异常。

键错误处理函数（key_error）的实现流程如下：

- 尝试获取字典中的值。
- 捕获并处理键不存在错误及其他异常。

类型错误处理函数（type_error）的实现流程如下：

- 尝试执行操作。
- 捕获并处理类型错误及其他异常。

```
# 模块五：错误处理模块
class ErrorHandling:
    # 静态方法下的处理错误并打印错误信息
    @staticmethod
    def handle_error(err_type, message):
        print(f"{err_type}: {message}")

    # 除法，带有错误处理
    def divide(self, a, b):
        try:
            return a / b
        except ZeroDivisionError:
            self.handle_error("ZeroDivisionError", "Cannot divide by
zero")
        except TypeError:
            self.handle_error("TypeError", "Unsupported operand
type(s)")
        except Exception as e:
            self.handle_error(type(e).__name__, str(e))

    # 索引错误处理
    def index_error(self, lst, index):
```

```
        # 尝试获取列表中的元素
        try:
            return lst[index]
        # 处理索引超出范围错误
        except IndexError:
            self.handle_error("IndexError", "List index out of range")
        # 处理其他异常
        except Exception as e:
            self.handle_error(type(e).__name__, str(e))

    # 键错误处理
    def key_error(self, dct, key):
        # 尝试获取字典中的值
        try:
            return dct[key]
        # 处理键不存在错误
        except KeyError:
            self.handle_error("KeyError", "Key not found in
dictionary")
        # 处理其他异常
        except Exception as e:
            self.handle_error(type(e).__name__, str(e))

    # 类型错误处理
    def type_error(self, operation):
        # 尝试执行操作
        try:
            eval(operation)
        # 处理类型错误
        except TypeError:
            self.handle_error("TypeError", "Unsupported operand
type(s)")
        # 处理其他异常
        except Exception as e:
            self.handle_error(type(e).__name__, str(e))
```

表 2-12：ErrorHandling 类的代码

2.2.6：AdvancedFeatures 类

AdvancedFeatures 类的代码内容，如表 2-13 所示。AdvancedFeatures 类中具备初始化函数（init）、数组操作函数（array_operations）、统计信息函数

（statistics）、幂运算函数（power）。

初始化函数（init）的实现流程如下：

- 接受一个数组并将其转换为 NumPy 数组。

数组操作函数（array_operations）的实现流程如下：

- 其他数组强制转换为 NumPy 数组。
- 根据指定的操作（加、减、乘、除）对两个数组进行相应运算。
- 如果操作无效，抛出值错误。

统计信息函数（statistics）的实现流程如下：

- 返回数组的统计信息，包括平均值、最大值、最小值、总和、标准差和方差。

幂运算函数（power）的实现流程如下：

- 对数组中的每个元素进行幂运算。

```
# 模块六：高级功能扩展模块
class AdvancedFeatures:
    # 初始化
    def __init__(self, array):
        # 接受一个数组并将其转换为 numpy 数组
        self.array = np.array(array)

    # 数组操作
    def array_operations(self, other_array, operation):
        # 其他数组强制转换为 numpy 数组
        other_array = np.array(other_array)

        # 数组加法
        if operation == 'add':
            return self.array + other_array
        # 数组减法
        elif operation == 'subtract':
            return self.array - other_array
        # 数组乘法
        elif operation == 'multiply':
            return self.array * other_array
        # 数组除法
        elif operation == 'divide':
            return self.array / other_array
        # 无效操作
```

```
        else:
            raise ValueError("Invalid operation")

    # 返回数组的统计信息
    def statistics(self):
        return {
            # 平均值
            'mean': np.mean(self.array),
            # 最大值
            'max': np.max(self.array),
            # 最小值
            'min': np.min(self.array),
            # 总和
            'sum': np.sum(self.array),
            # 标准差
            'std': np.std(self.array),
            # 方差
            'var': np.var(self.array)
        }

    # 幂运算
    def power(self, exponent):
        return np.power(self.array, exponent)
```

表 2-13：AdvancedFeatures 类的代码

2.2.7：FunctionSignature 类

FunctionSignature 类的代码内容，如表 2-14 所示。FunctionSignature 类中具备获取函数名函数（get_signature）。

获取函数名函数（get_signature）的实现流程如下：

- 调用 inspect.signature 获取函数的签名。
- 返回函数名和签名字符串。

```
# 模块七：提示函数参数
class FunctionSignature:
    # 静态方法下的获取函数签名
    @staticmethod
    def get_signature(func):
        # 获取函数的签名
        signature = inspect.signature(func)
```

```
        # 返回函数名和签名字符串
        return f"{func.__name__}{signature}"
```

表 2-14：FunctionSignature 类的代码

2.2.8：CodeFormatter 类

CodeFormatter 类的代码内容，如表 2-15 所示。CodeFormatter 类中具备格式化代码函数（format_code）。

格式化代码函数（format_code）的实现流程如下：

- 使用 autopep8.fix_code 对代码进行格式化。
- 返回格式化后的代码。

```
# 模块八：代码格式化
class CodeFormatter:
    # 静态方法下的格式化代码
    @staticmethod
    def format_code(code):
        # 使用 autopep8 对代码进行格式化
        return autopep8.fix_code(code)
```

表 2-15：CodeFormatter 类的代码

2.2.9：FileOperations 类

FileOperations 类的代码内容，如表 2-16 所示。FileOperations 类中具备读取文件内容函数（read_file）、内容写入文件函数（write_to_file）。

读取文件内容函数（read_file）的实现流程如下：

- 尝试以读取模式打开文件。
- 读取文件内容并返回。
- 捕获并处理文件未找到错误。
- 捕获并处理其他异常。

内容写入文件函数（write_to_file）的实现流程如下：

- 尝试以写入模式打开文件。
- 将内容写入文件。
- 打印成功写入的信息。

- 捕获并处理写入文件时的异常。

```
# 模块九：文件操作模块
class FileOperations:
    # 静态方法下的读取文件内容
    @staticmethod
    def read_file(filename):
        # 尝试以读取模式打开文件
        try:
            with open(filename, 'r') as file:
                # 读取文件内容
                content = file.read()
            return content
        # 处理文件未找到错误
        except FileNotFoundError:
            print(f"File '{filename}' not found.")
            return None
        # 处理其他异常
        except Exception as e:
            print(f"Error reading file '{filename}': {str(e)}")
            return None

    # 静态方法下的将内容写入文件
    @staticmethod
    def write_to_file(filename, content):
        # 尝试以写入模式打开文件
        try:
            with open(filename, 'w') as file:
                # 将内容写入文件
                file.write(content)
            print(f"Content successfully written to '{filename}'.")
        # 处理写入文件时的异常
        except Exception as e:
            print(f"Error writing to file '{filename}': {str(e)}")
```

表 2-16：FileOperations 类的代码

2.2.10：MathOperations 类

MathOperations 类的代码内容，如表 2-17 所示。MathOperations 类中具备计算阶乘函数（factorial）、计算最大公约数函数（gcd）、计算最小公倍数函数（lcm）。

计算阶乘函数（factorial）的实现流程如下：

- 尝试直接计算阶乘。
- 捕获并处理值错误。
- 捕获并处理其他异常。

计算最大公约数函数（gcd）的实现流程如下：

- 尝试直接计算最大公约数。
- 捕获并处理异常。

计算最小公倍数函数（lcm）的实现流程如下：

- 尝试使用最大公约数计算最小公倍数。
- 捕获并处理异常。

```
# 模块十：数学操作模块
class MathOperations:
    # 静态方法下的计算阶乘
    @staticmethod
    def factorial(n):
        # 尝试直接计算阶乘
        try:
            return math.factorial(n)
        # 处理值错误（例如传入负数）
        except ValueError as ve:
            print(f"ValueError: {str(ve)}")
            return None
        # 处理其他异常
        except Exception as e:
            print(f"Error calculating factorial: {str(e)}")
            return None

    # 静态方法下的计算最大公约数
    @staticmethod
    def gcd(a, b):
        # 尝试直接计算公约数
        try:
            return math.gcd(a, b)
        # 处理异常
        except Exception as e:
            print(f"Error calculating GCD: {str(e)}")
            return None
```

```
    # 静态方法下的计算最小公倍数
    @staticmethod
    def lcm(a, b):
        # 尝试使用最大公约数计算最小公倍数
        try:
            return abs(a * b) // math.gcd(a, b)
        # 处理异常
        except Exception as e:
            print(f"Error calculating LCM: {str(e)}")
            return None
```

表 2-17：MathOperations 类的代码

2.2.11：DateTimeOperations 类

DateTimeOperations 类的代码内容，如表 2-18 所示。DateTimeOperations 类中具备获取当前日期和时间函数（current_datetime）、格式化给定的日期时间对象函数（format_datetime）、计算两个日期之间的天数函数（days_between_dates）、在给定日期上增加指定天数函数（add_days）。

获取当前日期和时间函数（current_datetime）的实现流程如下：

- 时间格式化为字符串并返回。

格式化给定的日期时间对象函数（format_datetime）的实现流程如下：

- 将日期时间对象格式化为字符串并返回。

计算两个日期之间的天数函数（days_between_dates）的实现流程如下：

- 尝试将字符串格式的日期转换为日期对象。
- 计算日期差并返回天数。
- 捕获并处理日期格式错误。
- 捕获并处理其他异常。

在给定日期上增加指定天数函数（add_days）的实现流程如下：

- 尝试将字符串格式的日期转换为日期对象。
- 增加指定天数并格式化为字符串返回。
- 捕获并处理日期格式错误。
- 捕获并处理其他异常。

```
# 模块十一：日期时间操作模块
```

```
class DateTimeOperations:
    # 静态方法下的获取当前日期和时间
    @staticmethod
    def current_datetime():
        # 时间格式化为字符串
        return datetime.now().strftime("%Y-%m-%d %H:%M:%S")

    # 静态方法下的格式化给定的日期时间对象
    @staticmethod
    def format_datetime(dt):
        # 时间格式化为字符串
        return dt.strftime("%Y-%m-%d %H:%M:%S")

    # 静态方法下的计算两个日期之间的天数
    @staticmethod
    def days_between_dates(date1, date2):
        # 尝试直接计算日期之差
        try:
            # 将字符串格式的日期转换为日期对象
            date1 = datetime.strptime(date1, "%Y-%m-%d")
            date2 = datetime.strptime(date2, "%Y-%m-%d")
            # 计算日期差
            delta = date2 - date1
            return delta.days
        # 处理日期格式错误
        except ValueError:
            print("Invalid date format. Please use YYYY-MM-DD.")
            return None
        # 处理其他异常
        except Exception as e:
            print(f"Error calculating days between dates: {str(e)}")
        return None

    # 静态方法下的在给定日期上增加指定天数
    @staticmethod
    def add_days(date_str, days_to_add):
        # 尝试直接处理
        try:
            # 将字符串格式的日期转换为日期对象
            date = datetime.strptime(date_str, "%Y-%m-%d")
            # 增加指定天数
            new_date = date + timedelta(days=days_to_add)
            # 将新的日期格式化为字符串
```

```
            return new_date.strftime("%Y-%m-%d")
        # 处理日期格式错误
        except ValueError:
            print("Invalid date format. Please use YYYY-MM-DD.")
            return None
        # 处理其他异常
        except Exception as e:
            print(f"Error adding days to date: {str(e)}")
            return None
```

表 2-18：DateTimeOperations 类的代码

2.2.12：Encryption 类

Encryption 类的代码内容，如表 2-19 所示。Encryption 类中具备加密文本函数（encrypt）、解密文本函数（decrypt）。

加密文本函数（encrypt）的实现流程如下：

- 初始化存储加密后的字符。
- 遍历每个字符，对其 ASCII 码加上密钥值。
- 将加密后的字符存储在列表中。
- 将列表中的字符连接成字符串并返回。

解密文本函数（decrypt）的实现流程如下：

- 初始化存储解密后的字符。
- 遍历每个字符，对其 ASCII 码减去密钥值。
- 将解密后的字符存储在列表中。
- 将列表中的字符连接成字符串并返回。

```
# 模块十二：加密解密模块
class Encryption:
    @staticmethod
    # 静态方法下的加密文本
    def encrypt(text, key):
        # 初始化存储加密后的字符
        encrypted_text = []

        # 对每个字符的 ASCII 码加上密钥值
        for char in text:
            encrypted_text.append(chr(ord(char) + key))
```

```
        # 将列表中的字符连接成字符串并返回
        return ''.join(encrypted_text)

    # 静态方法下的解密文本
    @staticmethod
    def decrypt(encrypted_text, key):
        # 初始化存储解密后的字符
        decrypted_text = []

        # 对每个字符的 ASCII 码减去密钥值
        for char in encrypted_text:
            decrypted_text.append(chr(ord(char) - key))

        # 将列表中的字符连接成字符串并返回
        return ''.join(decrypted_text)
```

表 2-19：Encryption 类的代码

2.2.13：Logging 类

Logging 类的代码内容，如表 2-20 所示。Logging 类中具备记录日志信息函数（log）。

记录日志信息函数（log）的实现流程如下：

- 获取当前时间戳，并格式化为字符串。
- 打印带有时间戳的日志信息。

```
# 模块十三：日志记录模块
class Logging:
    # 静态方法下的记录日志信息
    @staticmethod
    def log(message):
        # 获取当前时间戳，并格式化为字符串
        timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        # 打印带有时间戳的日志信息
        print(f"[{timestamp}] {message}")
```

表 2-20：Logging 类的代码

3：类模块测试

3.1：词法分析

```
35  # 示例用法
36  lexer = Lexer()
37  code = "def func(x): if x > 0: return int(x)"
38  highlighted_code = lexer.highlight(code)
39  print(highlighted_code)
40
```

图 3-1：词法分析的测试代码

```
PS C:\Users\86158\Desktop\compile> python exp1.py
def func(x): if x > 0: return int(x)
```

图 3-2：词法分析的测试结果

在输出结果中，def、if、return 被高亮为紫色（代表该字符流是关键字），int 被高亮为蓝色（代表该字符流是数据类型），> 和 = 被高亮为黄色（代表该字符流是操作符号），(和) 被高亮为绿色（代表该字符流是括号）。

3.2：语法分析

```
24  # 示例用法
25  code = """
26  def my_function(a, b):
27      return a + b
28  """
29
30  analyzer = SyntaxAnalyzer(code)
31  signatures = analyzer.get_function_signatures()
32  formatted_code = analyzer.format_code()
33  print("Function Signatures:", signatures)
34  print("Formatted Code:\n", formatted_code)
35
```

图 3-3：语法分析的测试代码

```
PS C:\Users\86158\Desktop\compile> python exp2.py
Function Signatures: ["Function 'my_function(a, b)'"]
Formatted Code:
 def my_function(a, b):
    return a + b
```

图 3-4：语法分析的测试结果

输出结果显示了函数签名 my_function(a, b) 和格式化后的代码。上述结果展示了解析 Python 代码，提取代码中的函数签名，并格式化代码以符合 PEP 8 规范的过程。

3.3：语义分析

```
18  # 示例用法
19  age = IOExtension.formatted_input("Enter your age: ", int)
20  IOExtension.formatted_output(age)
21
```

图 3-5：语义分析的测试代码

```
PS C:\Users\86158\Desktop\compile> python exp3.py
Enter your age: abc
Invalid int, try again.
Enter your age: 10.5
Invalid int, try again.
Enter your age: [1,2,3]
Invalid int, try again.
Enter your age: 25
Output: 25
```

图 3-6：语义分析的测试结果

上述输出结果表明，类模块可以有效处理用户输入的错误类型，并提示用户重新输入，直到输入正确的数据类型为止。同时，类模块可以根据数据类型提供不同的输出格式。通过不断提示用户并正确处理错误输入，程序提升了用户体验，确保最终获得的是有效数据。

3.4：数组添加和查询

```
24  # 示例用法
25  ds = DataStructureExtension()
26  ds.add_int(5)
27  ds.add_float(3.14)
28  print("Int Array:", ds.get_int_array())
29  print("Float Array:", ds.get_float_array())
30
```

图 3-7：数组添加和查询的测试代码

```
PS C:\Users\86158\Desktop\compile> python exp4.py
Int Array: [5]
Float Array: [3.14]
```

图 3-8：数组添加和查询的测试结果

上述输出结果展示了类模块在管理和操作整数数组和浮点数数组时的基本功能和输出结果。通过内部函数 add 进行数组元素的增加，同时根据内部函数 get 获取不同属性（整型数据、浮点型数据）的元素。

3.5：出错处理

```
16  # 示例用法
17  eh = ErrorHandling()
18  print(eh.divide(10, 0))
19  print(eh.divide(10, "a"))
20
```

图 3-9：出错处理的测试代码

```
PS C:\Users\86158\Desktop\compile> python exp5.py
ZeroDivisionError: Cannot divide by zero
None
TypeError: Unsupported operand type(s)
None
```

图 3-10：出错处理的测试结果

上述输出结果展示了类模块在处理除法操作时的鲁棒性和错误处理能力。当除数为零时捕获，输出相应错误信息。当操作数类型不支持时捕获，输出相应错误信息。同时，当发生异常时，divide 方法返回 None。

3.6：数组内部计算

```
28  # 示例用法
29  af = AdvancedFeatures([1, 2, 3, 4, 5])
30  print("Array Operations (Add):", af.array_operations([5, 4, 3, 2, 1], 'add'))
31  print("Statistics:", af.statistics())
32
```

图 3-11：数组内部计算的测试代码

```
PS C:\Users\86158\Desktop\compile> python exp6.py
Array Operations (Add): [6 6 6 6 6]
Statistics: {'mean': 3.0, 'max': 5, 'min': 1, 'sum': 15}
```

图 3-12：数组内部计算的测试结果

上述输出结果展示了类模块在处理数组运算和统计计算时的功能和输出结果。

4：实际问题案例对比

4.1：案例 1——任务管理清单

4.1.1：自编写 Python 编译下的代码及其运行结果

```
# 主程序
def main():
    # 初始化数据结构扩展模块
    ds = DataStructureExtension()

    # 初始化错误处理模块
    eh = ErrorHandling()

    # 添加任务
    while True:
        task = input("请输入任务（或输入 exit 退出）: ")
        if task.lower() == 'exit':
            break
        ds.add_dict(task, False)  # False 表示任务未完成

    # 标记任务为完成
    while True:
```

```
        task = input("请输入要标记为完成的任务（或输入 exit 退出）: ")
        if task.lower() == 'exit':
            break

        try:
            if task in ds.get_dict():
                ds.add_dict(task, True)  # True 表示任务已完成
            else:
                print("任务不存在，请重新输入。")
        except KeyError:
            eh.handle_error("KeyError", "任务不存在")

    # 显示所有任务及其状态
    for task, completed in ds.get_dict().items():
        status = "完成" if completed else "未完成"
        IOExtension.formatted_output(f"任务: {task} 状态: {status}")

    # 处理可能的输入错误
    try:
        test_task = IOExtension.formatted_input("请输入测试任务: ",
str)
        IOExtension.formatted_output(test_task)
    except ValueError:
        eh.handle_error("ValueError", "输入的任务无效")

# 主函数
if __name__ == "__main__":
    main()
```

表 4-1：案例 1 的自编写 Python 编译下的代码

```
PS C:\Users\86158\Desktop\compile> python eg1.py
请输入任务（或输入exit退出）: task1
请输入任务（或输入exit退出）: task2
请输入任务（或输入exit退出）: task3
请输入任务（或输入exit退出）: exit
请输入要标记为完成的任务（或输入exit退出）: task1
请输入要标记为完成的任务（或输入exit退出）: exit
Output: 任务: task1 状态: 完成
Output: 任务: task2 状态: 未完成
Output: 任务: task3 状态: 未完成
请输入测试任务: task3
Output: task3
```

图 4-1：案例 1 的自编写 Python 编译下的代码运行结果

4.1.2：原生 Python 编译下的代码及其运行结果

```
# 任务管理系统
class TaskManager:
    def __init__(self):
        self.tasks = {}

    def add_task(self, task):
        if task in self.tasks:
            print(f"任务 '{task}' 已经存在.")
        else:
            self.tasks[task] = False  # False 表示任务未完成

    def complete_task(self, task):
        if task in self.tasks:
            self.tasks[task] = True  # True 表示任务已完成
        else:
            print(f"任务 '{task}' 不存在.")

    def show_tasks(self):
        for task, completed in self.tasks.items():
            status = "完成" if completed else "未完成"
            print(f"任务: {task} 状态: {status}")

    def handle_error(self, err_type, message):
        print(f"{err_type}: {message}")

# 主程序
def main():
    # 初始化任务管理系统
    task_manager = TaskManager()

    # 添加任务
    while True:
        task = input("请输入任务（或输入 exit 退出）: ")
        if task.lower() == 'exit':
            break
        try:
            task_manager.add_task(task)
        except Exception as e:
```

```
            task_manager.handle_error(type(e).__name__, str(e))

    # 标记任务为完成
    while True:
        task = input("请输入要标记为完成的任务（或输入 exit 退出）: ")
        if task.lower() == 'exit':
            break
        try:
            task_manager.complete_task(task)
        except Exception as e:
            task_manager.handle_error(type(e).__name__, str(e))

    # 显示所有任务及其状态
    task_manager.show_tasks()

if __name__ == "__main__":
    main()
```

表 4-2：案例 1 的原生 Python 编译下的代码

```
PS C:\Users\86158\Desktop\compile> python eg1-origin.py
请输入任务（或输入exit退出）: task1
请输入任务（或输入exit退出）: task2
请输入任务（或输入exit退出）: task3
请输入任务（或输入exit退出）: exit
请输入要标记为完成的任务（或输入exit退出）: task1
请输入要标记为完成的任务（或输入exit退出）: exit
任务: task1 状态: 完成
任务: task2 状态: 未完成
任务: task3 状态: 未完成
```

图 4-2：案例 1 的原生 Python 编译下的代码运行结果

4.2：案例 2——鸡兔同笼问题

4.2.1：自编写 Python 编译下的代码及其运行结果

```
from test2 import Lexer, SyntaxAnalyzer, IOExtension,
DataStructureExtension, ErrorHandling, AdvancedFeatures,
FunctionSignature, CodeFormatter, FileOperations, MathOperations,
DateTimeOperations, Encryption, Logging

# 主程序
```

```
def main():
    # 初始化错误处理模块
    eh = ErrorHandling()

    while True:
        try:
            # 输入总头数
            heads = IOExtension.formatted_input("请输入总头数（或输入-1
退出）: ", int)
            if heads == -1:
                break

            # 输入总脚数
            legs = IOExtension.formatted_input("请输入总脚数: ", int)

            # 计算兔子的数量
            rabbits = (legs - 2 * heads) / 2
            # 计算鸡的数量
            chickens = heads - rabbits

            # 检查计算结果是否为非负整数
            if rabbits < 0 or chickens < 0 or not rabbits.is_integer()
or not chickens.is_integer():
                eh.handle_error("ValueError", "输入的头数和脚数不可能对应
实际的鸡和兔子数量")
            else:
                IOExtension.formatted_output(f"鸡的数量:
{int(chickens)}")
                IOExtension.formatted_output(f"兔子的数量:
{int(rabbits)}")

        except ValueError:
            eh.handle_error("ValueError", "输入的值无效，请输入整数")
        except KeyboardInterrupt:
            print("\n 操作中断，程序退出。")
            break
        except Exception as e:
            eh.handle_error(type(e).__name__, str(e))

if __name__ == "__main__":
    main()
```

表 4-3：案例 2 的自编写 Python 编译下的代码

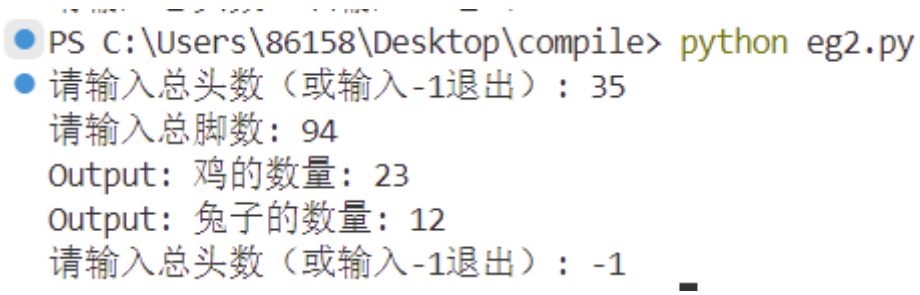

图 4-3：案例 2 的自编写 Python 编译下的代码运行结果

4.2.2：原生 Python 编译下的代码及其运行结果

```
def solve_chicken_rabbit(total_heads, total_legs):
    # Chicken has 2 legs, rabbit has 4 legs
    for chickens in range(total_heads + 1):
        rabbits = total_heads - chickens
        if (chickens * 2 + rabbits * 4) == total_legs:
            return chickens, rabbits
    return None, None

total_heads = 35
total_legs = 94
chickens, rabbits = solve_chicken_rabbit(total_heads, total_legs)

if chickens is not None and rabbits is not None:
    print(f"Number of chickens: {chickens}, Number of rabbits:
{rabbits}")
else:
    print("No solution found")
```

表 4-4：案例 2 的原生 Python 编译下的代码

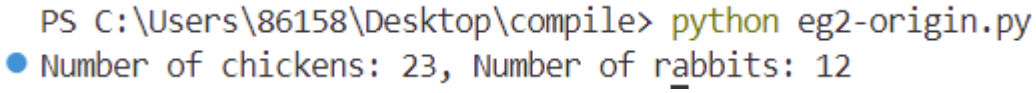

图 4-4：案例 2 的原生 Python 编译下的代码运行结果

4.3：效果对比

（1）代码结构与组织。

在原生 Python 代码中，所有功能（添加任务、标记任务、显示任务状态、错误处理）都集中在一个类中，代码较为集中和简洁。

在使用模块代码中，功能分散在不同的模块中，如 DataStructureExtension、ErrorHandling 和 IOExtension，代码更为模块化，职责单一。

（2）可重用性。

在原生 Python 代码中，特定类中的逻辑是独立实现的，复用性较低，需要修改代码才能用于其他场景。

在使用模块代码中，不同功能分散在独立的模块中，各模块可以复用，如 ErrorHandling 和 IOExtension 模块，可以在其他程序中直接使用，提高代码的复用性。

（3）代码可读性。

在原生 Python 代码中，代码结构简单，所有逻辑集中在一个类中，适合小规模的应用。

在使用模块代码中，代码较为分散，各模块之间的调用关系需要理解模块的功能和接口，适合大型项目和复杂的应用。

（4）格式化输入输出。

在原生 Python 代码中，直接使用 print 和 input 进行输入输出。

在使用模块代码中，使用 IOExtension 模块进行格式化输入输出，代码更加规范和统一。

（5）数据处理。

在原生 Python 代码中，使用字典存储任务及其状态，数据结构简单直观。

在使用模块代码中，使用 DataStructureExtension 模块存储任务及其状态，增加了一层抽象，但功能更为强大和灵活。

（6）出错处理。

在原生 Python 代码中，错误处理集中在 handle_error 方法中，根据异常类型显示错误信息。

在使用模块代码中，使用 ErrorHandling 模块处理错误，错误处理逻辑统一，便于管理。

5：主程序输出案例

5.1：案例 1——函数签名

函数签名获取了函数的名称和参数列表，如 Function 'func_name(param1, param2=default_value)'。这种格式清晰地显示了函数的名称和每个参数的名称，对于理解函数功能和正确使用至关重要。函数签名能够处理多种参数类型和情况，包括位置参数、默认值、可变位置参数 args 和可变关键字参数 kwargs。这种灵活性使得签名可以适应不同函数定义的情况，提供了广泛的支持。在实验中，使用 eval 函数获取函数对象时，可能会遇到函数不存在或输入错误的情况。通过适当的异常处理，如捕获 NameError，能够有效地提示用户输入正确的函数名或处理不存在的函数情况。

```
PS C:\Users\86158\Desktop\compile> python test2.py
请输入代码进行处理（输入0退出）: math.factorial
Highlighted Code:
 math.factorial
Function Signatures: []
Formatted Code:
 math.factorial

Abstract Syntax Tree (AST):
 Module(
    body=[
        Expr(
            value=Attribute(
                value=Name(id='math', ctx=Load()),
                attr='factorial',
                ctx=Load()))],
    type_ignores=[])
请输入提示信息:
```

图 5-1：案例 1 的输出结果

5.2：案例 2——高级功能扩展

用户可以输入两个数组，选择加法、减法、乘法、除法等操作。这些操作通过 array_operations 方法实现，利用 NumPy 库进行高效处理。用户可以选择计算数组的统计信息，如均值、最大值、最小值、总和、标准差和方差。这些统计量通过 statistics 方法返回，帮助用户快速了解和分析数据集的特征。用户可以对数组中的每个元素进行幂运算。这通过 power 方法实现，允许用户灵活地对数组进行指数计算。

```
PS C:\Users\86158\Desktop\compile> python test2.py
请输入代码进行处理（输入0退出）: 1
Highlighted Code:
 1
Function Signatures: []
Formatted Code:
 1

Abstract Syntax Tree (AST):
 Module(
    body=[
        Expr(
            value=Constant(value=1))],
    type_ignores=[])
请输入提示信息:
请输入数据类型（int, float, str, list, dict）:
int
Output: int
请输入数据类型和值（如: int 5, float 3.14, str 'hello', dict key:value）或输入exit退出: int 4.5
发生异常: invalid literal for int() with base 10: '4.5', 请检查并重试。
请输入代码进行处理（输入0退出）: 0
```

图 5-2：案例 2 的输出结果

5.3：案例 3——异常报错处理

每种错误类型都有相应的处理方法，例如除零错误会输出“Cannot divide by zero”，索引错误会输出“List index out of range”，以及键错误会输出“Key not found in dictionary”。在主程序的执行过程中，用户可以通过选择不同的错误类型来模拟可能发生的异常情况。例如，在进行除零操作时，程序能够捕获并处理 ZeroDivisionError；在访问列表时，程序能够捕获并处理 IndexError；在访问字典时，程序能够捕获并处理 KeyError；而在其他不可预见的异常情况下，程序也能够通过通用异常处理输出详细的错误信息。

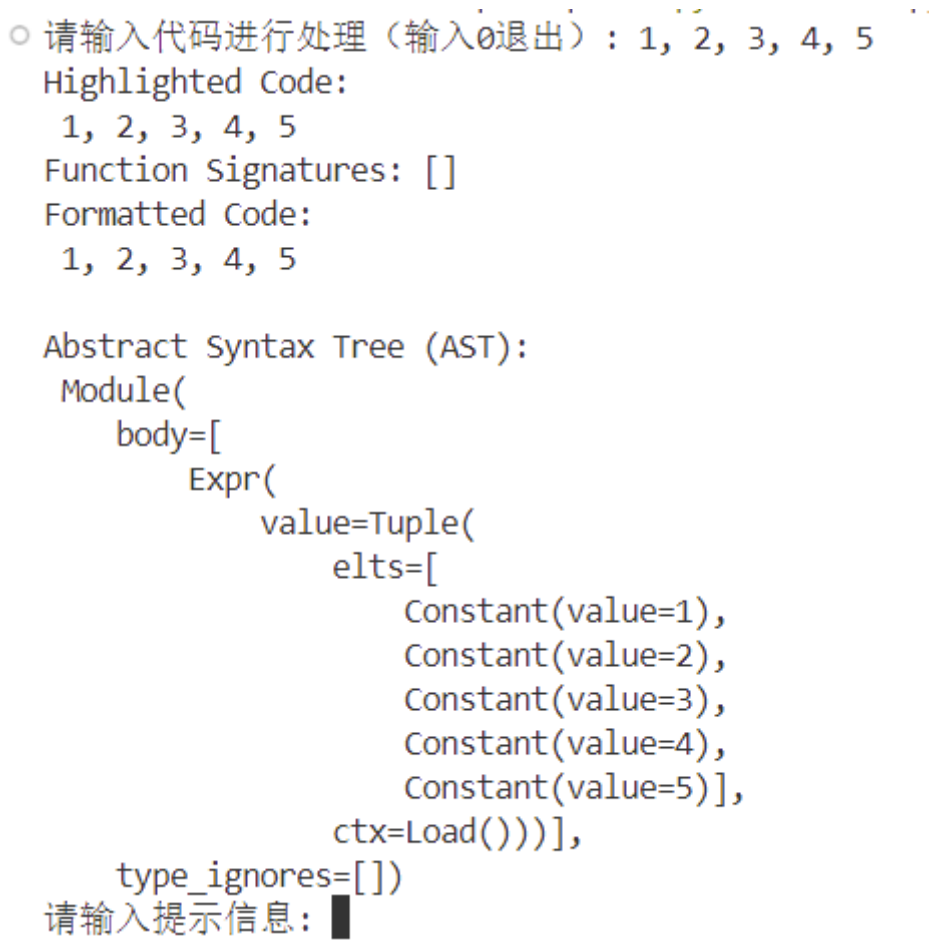

图 5-3：案例 3 的输出结果

5.4：案例 4——语法树生成

在主程序的执行过程中，用户输入的代码经过词法分析器高亮显示后，进入语法分析器进行解析。语法分析器通过构建语法树，能够深入理解代码的结构，并提取出其中的函数签名信息。这些信息在展示给用户的同时，也为程序后续的操作提供了重要的数据支持。

```
PS C:\Users\86158\Desktop\compile> python test2.py
请输入代码进行处理（输入0退出）: for i in range(5): print(i)
Highlighted Code:
 for i in range(5): print(i)
Function Signatures: []
Formatted Code:
 for i in range(5):
    print(i)

Abstract Syntax Tree (AST):
 Module(
    body=[
        For(
            target=Name(id='i', ctx=Store()),
            iter=Call(
                func=Name(id='range', ctx=Load()),
                args=[
                    Constant(value=5)],
                keywords=[]),
            body=[
                Expr(
                    value=Call(
                        func=Name(id='print', ctx=Load()),
                        args=[
                            Name(id='i', ctx=Load())],
                        keywords=[]))],
            orelse=[])],
    type_ignores=[])
请输入提示信息:
```

图 5-4：案例 4 的输出结果

5.5：案例 5——I/O 格式化分析

通过自动化的代码格式化工具，程序能够根据语法树规范化和调整代码的缩进、空格和行距，确保代码的风格符合编程标准，提高代码的可读性和可维护性。

```
PS C:\Users\86158\Desktop\compile> python test2.py
请输入代码进行处理（输入0退出）: 1
Highlighted Code:
 1
Function Signatures: []
Formatted Code:
 1

Abstract Syntax Tree (AST):
 Module(
    body=[
        Expr(
            value=Constant(value=1))],
    type_ignores=[])
请输入提示信息:
请输入数据类型（int, float, str, list, dict）: int
5
Output: 5
请输入数据类型和值（如：int 5, float 3.14, str 'hello', dict key:value）或输入exit退出:
```

图 5-5：案例 5 的输出结果

5.6：案例 6——高亮

使用词法分析器，程序首先对用户输入的代码进行高亮处理，突出显示关键字、运算符、数据类型和括号等不同的语法成分。这种高亮显示不仅美化了代码展示，还帮助用户快速理解代码结构。

```
PS C:\Users\86158\Desktop\compile> python test2.py
请输入代码进行处理（输入0退出）: if n == 0 i = 2;
Highlighted Code:
 if n == 0 i = 2;
发生异常: invalid syntax (<unknown>, line 1)，请检查并重试。
请输入代码进行处理（输入0退出）:
```

图 5-6：案例 6 的输出结果

6：实验总结

本实验开发了一个多功能的命令行程序，通过改进 Python 语言，能够对用户输入的代码进行高亮显示、语法分析、错误处理、数据结构操作等多个功能的扩展。通过词法分析器实现了代码的分词和高亮显示，使得代码结构更加清晰可见。利用语法分析器解析用户输入的代码并构建语法树，实现了函数签名提取、代码格式化和抽象语法树的展示，从而增强了程序对代码结构的理解能力。数据

结构扩展模块允许动态添加和查询整数数组、浮点数数组、字符串列表和字典，通过语法分析器实现了实时更新和状态展示。错误处理模块设计了多个异常处理函数，包括除零错误、索引错误、键错误和类型错误，提升了程序的健壮性和稳定性。高级功能拓展模块引入了数学运算、日期时间操作、加密解密等功能，增强了程序的实用性和功能性，使其能够满足更广泛的应用场景需求。

参考资料

1：维基百科（https://zh.wikipedia.org/wiki/Python）。

2：Python Lex-Yacc（https://github.com/dabeaz/ply）。