Основы разработки сложных модульных программ

- СУБД
- SQL
- Основные запросы
- ORM
- Другие СУБД

- Модульный подход
- ООП
- Паттерн проектирования
- SOLID

Модульное программирование – это процесс разделения компьютерной программы на отдельные подпрограммы.
Модуль – это отдельный программный компонент. Его часто можно использовать в различных приложениях и функциях с другими компонентами системы.

МОДУЛЬНЫЙ ПОДХОД.ПРИМЕР

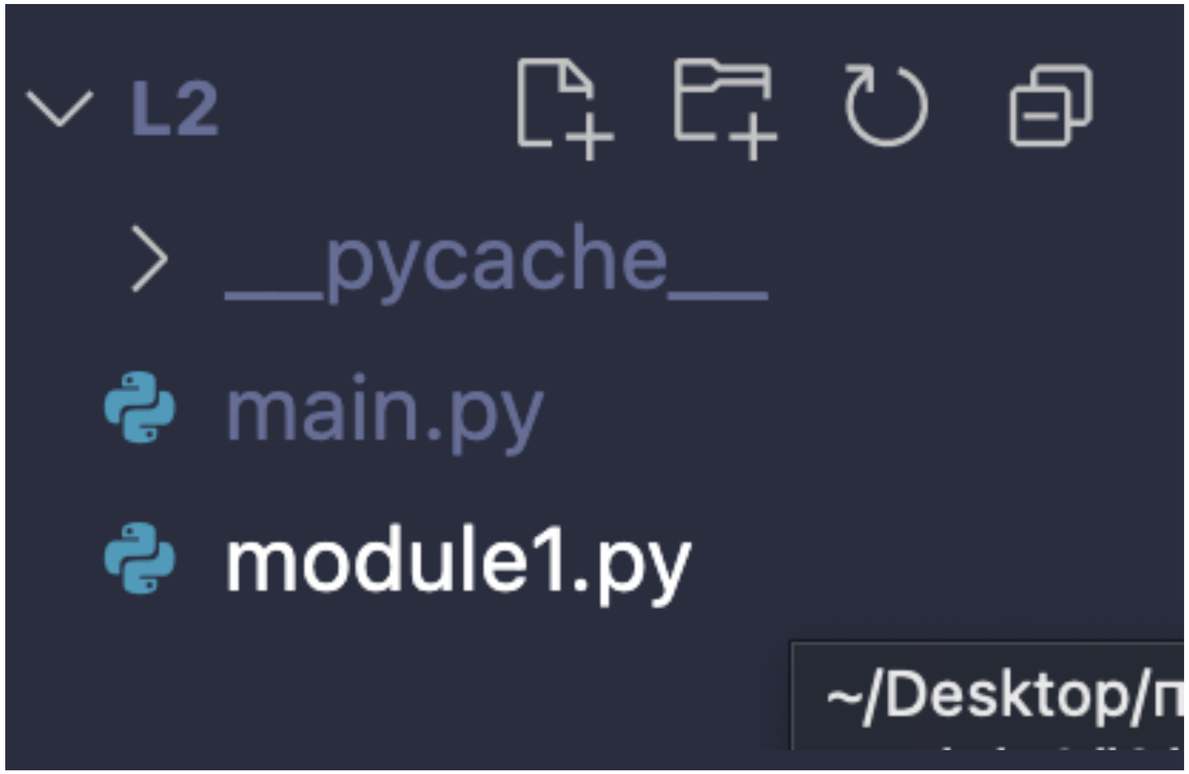
```
import module1

obj=module1.Module1Clas
s()
obj.method1()
```

```
class Module1Class:
    def method1(self):

print('method1')
```

МОДУЛЬНЫЙ ПОДХОД.ПРИМЕР



https://github.com/django/django/tree/main/django

Объектно-ориентированное программирование, ООП — это одна из парадигм разработки, подразумевающая организацию программного кода, ориентируясь на данные и объекты, а не на функции и логические структуры.

ООП. ОСНОВНЫЕ ПРИНЦИПЫ

- Абстракция. Моделирование требуемых атрибутов и взаимодействий сущностей в виде классов для определения абстрактного представления системы.
- Инкапсуляция. Скрытие внутреннего состояния и функций объекта и предоставление доступа только через открытый набор функций.
- Наследование. Возможность создания новых абстракций на основе существующих.
- Полиморфизм. Возможность реализации наследуемых свойств или методов отличающимися способами в рамках множества абстракций.

- Класс
- Объект
- Метод
- Конструктор
- Поле класса
- Статические методы
- Магические методы
- Связи между классами
- Абстракция и наследование *

- Порождающие (фабрика, абстрактная фабрика)
- Структурные (декоратор)
- Поведенческие (наблюдатель)

- S: Single Responsibility Principle (Принцип единственной ответственности).
- O: Open-Closed Principle (Принцип открытости-закрытости).
- L: Liskov Substitution Principle (Принцип подстановки Барбары Лисков).
- I: Interface Segregation Principle (Принцип разделения интерфейса).
- D: Dependency Inversion Principle (Принцип инверсии зависимостей)

```
class Connector:
    def connect(*args):
        return
    def
set_data(*args):
        return
```

```
class Connector:
    def connect(*args):
        return
```

```
class Discount:
    def
get_discount(self,*args):
        if args[0]=='simple':
            return args[1] *
0.2
        if args[0]=='fav':
            return args[1] *
0.4
```

```
class
FavDisCount(Discount):
    def
get_discount(self,
*args):
            return
args[0]*0.4
```

```python
import abc
class Animal(abc.ABC):
    @abc.abstractmethod
    def sound(self):
        pass
class Lion(Animal):
    def sound(self):
        print('rrr')

class Wolf(Animal):
    def sound(self):
        print('auf')

def implement_sound(animal : Animal):
    animal.sound()

implement_sound(Wolf())
```

```python
import abc

class Developer(abc.ABC):
    @abc.abstractmethod
    def develop(self):
        pass

class Manager(abc.ABC):
    @abc.abstractmethod
    def manage(self):
        pass

class AbstractTeamLead(Manager, Developer):

    def manage(self):
        return super().manage()
    def develop(self):
        return super().develop()
```

```python
class TeamLead(AbstractTeamLead):

    def manage(self):
        print('manage')
    def develop(self):
        print('develop')
```

```python
tl = TeamLead()
tl.manage()
tl.develop()
```

```python
import abc

class Developer(abc.ABC):
    @abc.abstractmethod
    def develop(self):
        pass

class Manager(abc.ABC):
    @abc.abstractmethod
    def manage(self):
        pass

class AbstractTeamLead(Manager, Developer):

    def manage(self):
        return super().manage()
    def develop(self):
        return super().develop()
```

```python
class TeamLead(AbstractTeamLead):

    def manage(self):
        print('manage')
    def develop(self):
        print('develop')
```

```python
tl = TeamLead()
tl.manage()
tl.develop()
```

ВЫВОДЫ