Занятие 28

SQLAlchemy

Давайте пройдем тест
https://kursy.guru/test/python/

Задача 27-1

Введите число n от 1 до 18.

Напечатайте квадратную матрицу, имитирующую Дартс.

Например для n = 5.

1 1 1 1 1

1 2 2 2 1

1 2 3 2 1

1 2 2 2 1

1 1 1 1 1

Задача 27-2

Необходимо реализовать класс Item, описывающий предмет, конструктор которого принимает три аргумента:

name — название предмета

price — цена предмета в рублях

quantity — количество предметов

При обращении к атрибуту name экземпляра класса Item будет возвращаться его название с заглавной буквы, а при обращении к атрибуту total — произведение цены предмета на его количество.

Решение задачи 27-2

```
class Item:
    def __init__(self, name, price, quantity):
        self.name = name
        self.price = price
        self.quantity = quantity

    def __getattribute__(self, attrname):
        if attrname == "name":
            return object.__getattribute__(self, attrname).title()       # Почему нельзя self.name.title()?
        else:
            return object.__getattribute__(self, attrname)

    def __getattr__(self, attrname):
        if attrname == "total":
            return self.price * self.quantity
        else:
            raise AttributeError
 @property
    def color(self):
        return "Red"
```

Задача 27-3

Дан список.

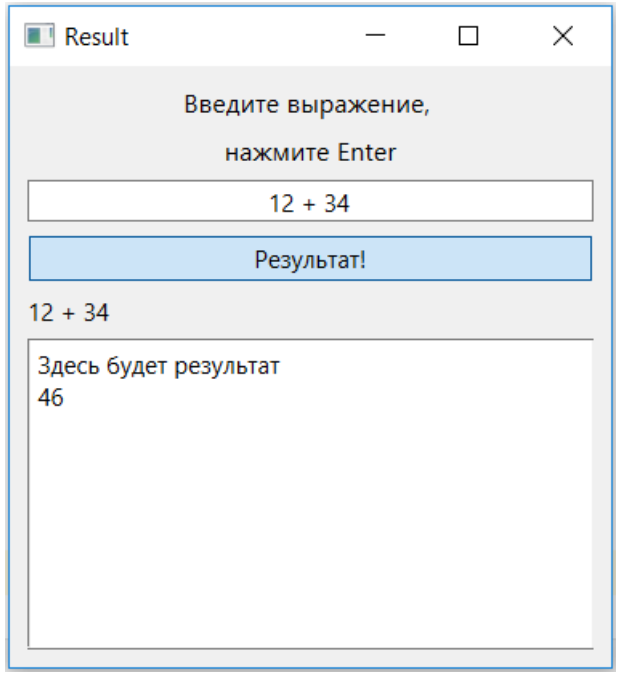
Посчитайте сколько в нем элементов, включая вложенные списки.

Например:

```
[]                  -->  0
[1, 2, 3]           -->  3
["x", "y", ["z"]]   -->  4
[1, 2, [3, 4, [5]]] -->  7
```

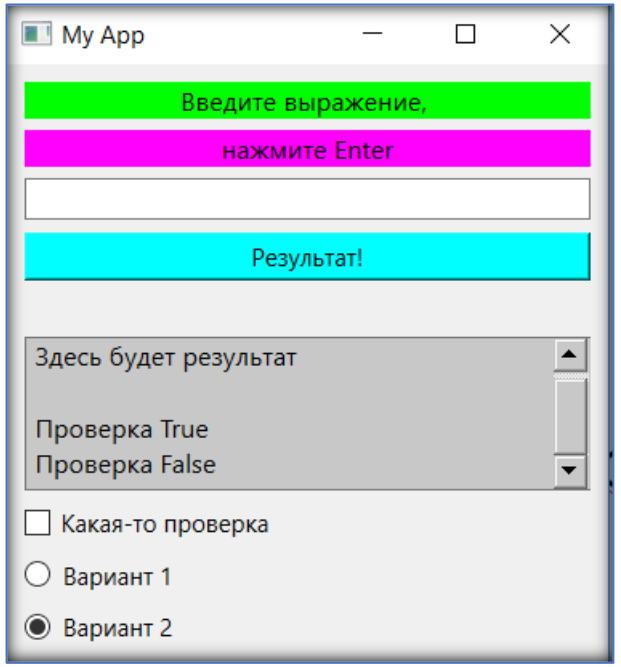
Вывод результатов в TextEdit

```
self.qte = QTextEdit()  # Предварительно надо
#                         import QTextEdit

self.qte.append("Здесь будет результат")
...
widgets = [widget0, widget1, self.widget2, button,
self.label_result, self.qte]
...
def the_button_was_clicked(self):
        print("Clicked!")
        self.label_result.setText(self.text)

        try:
            res = str(eval(self.text))
            self.qte.append(res)
        except:
            pass

        if self.tf:
            self.setWindowTitle('Result')
            self.tf = False
...
```



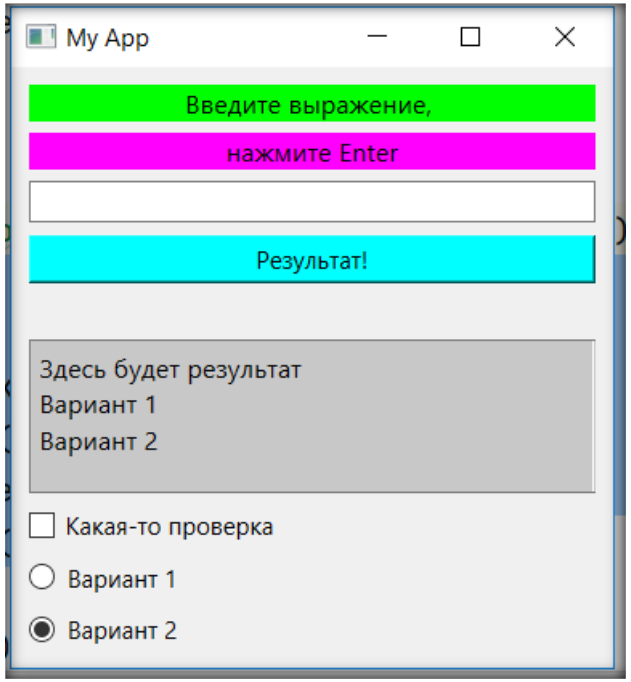
CheckBox

```
self.cb = QCheckBox('Какая-то проверка', self)
self.cb.stateChanged.connect(self.checking)

def checking(self):
    self.qte.append("Проверка " + str(self.cb.isChecked()))
```



RadioButton

```
self.rb_1 = QRadioButton('Вариант 1', self)
self.rb_1.clicked.connect(self.radio)

self.rb_2 = QRadioButton('Вариант 2', self)
self.rb_2.clicked.connect(self.radio)

def radio(self):
        if self.rb_1.isChecked():
            self.qte.append('Вариант 1')
        elif self.rb_2.isChecked():
            self.qte.append('Вариант 2')
```



Задание

Добавьте в программу виджеты QEditText, CheckButton, RadioButton и методы работы с ними, используя метод append, isChecked и т.д.

Функции для работы с атрибутами

__getattribute__(self, name) — вызывается при обращении к любому атрибуту

__getattr__(self, name) — вызывается при обращении к несуществующему атрибуту

Разница между ними в том, что метод __getattribute__() вызывается первым и вызывается всегда, а метод __getattr__() вызывается только в том случае, если атрибута, к которому происходит обращение, не существует.

Если атрибут существует, метод __getattribute__() возвращает его значение, в противном случае вызывается метод __getattr__().

__setattr__(self, name, value) — вызывается при установке атрибута или изменении его значения

__delattr__(self, name) — вызывается при удалении любого атрибута

hasattr()

```
class Person:
    age = 38
    name = "Ivan"
person = Person()
print("Возраст:", hasattr(person, "age"))
print("Зарплата:", hasattr(person, "salary"))
# Результат
Возраст: True
Зарплата: False
```

Задание

Используя функцию type(Имя класса, Родители, Словарь атрибутов),

создайте класс Point с двумя переменными x, y.

Используя список [(0,0), (1, 1), (2,2), (1,2), (0,2), (2,1), (2,0)], создайте экземпляры этого класса.

Напечатайте список линий = пар точек, расстояние между которыми больше 1

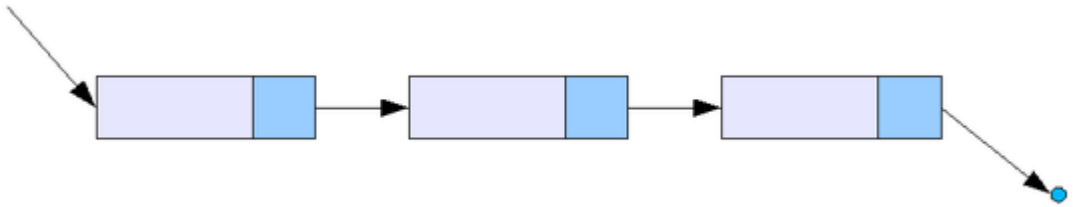
Завершите программу

```
def __init__(self, x, y):
    self.x = x
    self.y = y

Point = type('Point', (), {'__init__':__init__}) # можно так добавить метод

def __str__(self):
    return str((self.x, self.y))
Point.__str__ = __str__                  # а можно и так добавить метод

lst = [(0,0), (1,1), (2,2), …]
plist = []
for i in lst:
    plist.append(Point(i[0], i[1]))   # Создаем список из экземпляров класса Point
for i in plist:
    print(i)
```

#Создайте функцию вычисления расстояния между точками distance(self, other), добавьте ее в класс, напечатайте те пары точек, расстояние между которыми больше 2.

Динамические структуры данных - Связные списки

```
class Node:
    def __init__(self, value):
        self.value = value              # Значение узла
        self.next_node = None           # Ссылка на следующий узел
a = Node(1)
b = Node(22)
a.next_node = b
c = Node(333)
b.next_node = c
d = Node(4444)
c.next_node = d


x = a
while x.next_node != None:
    print(x.value)
    x = x.next_node
```



Связные списки – типичные функции

1. Добавить узел x в конец связного списка, головой которого является a
2. Сцепить два списка
3. Добавить узел в голову списка (поменять голову)
4. Сосчитать количество элементов связного списка.
5. Поменять местами два элемента.

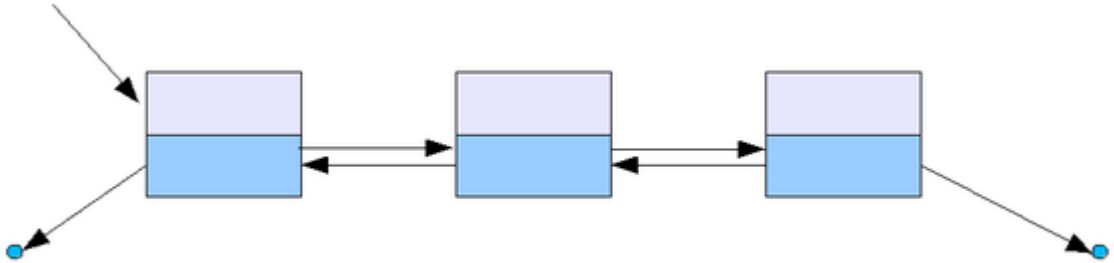
Давайте решим эти задачи.

```
class Node:
    def __init__(self, value):
        self.value = value                  # Значение узла
        self.next_node = None # Ссылка на следующий узел
```

a = Node(1) # голова списка, а где конец, мы и не знаем

x = Node(0) – это узел, который надо добавить или голова второго списка.

Двусвязные списки



```
class Node:
    def __init__(self, value):
        self.value = value                  # Значение узла
        self.next_node = None               # Ссылка на следующий узел
        self.prev_node = None               # Ссылка на предыдущий узел
```

Паттерны проектирования

Шаблоны проектирования

- Шаблоны проектирования – это стандартные(обобщенные) решения для определенных задач.

- Предложены хорошими специалистами
- Проверены временем
- Повсеместно используемые

- В области ПО использование шаблонов проектирования было предложено и развито Gamma, Helms, Johnson и Vlissades – в книге «Design Patterns: Elements of Reusable Object-Oriented Software» 1995

Классификация паттернов GoF

- Порождающие (Creational)
- Структурные (Structural)
- Поведенческие (Behavioral)

1. Порождающие шаблоны	2. Структурные шаблоны	3. Шаблоны поведения
1. Abstract Factory	1. Adapter	1. Chain of Responsibility
2. Builder	2. Bridge	2. Command
3. Factory Method	3. Composite	3. Interpreter
4. Prototype	4. Decorator	4. Iterator
5. Singleton	5. Façade	5. Mediator
	6. Flyweight	6. Momento
	7. Proxy	7. Observer
		8. State
		9. Strategy
		10. Template Method
		11. Visitor

Порождающие шаблоны **(Creational)** — шаблоны проектирования, которые абстрагируют процесс инстанцирования.
Они позволяют сделать систему независимой от способа создания, композиции и представления объектов.
Шаблон, порождающий классы, использует наследование, чтобы изменять инстанцируемый класс, а шаблон, порождающий объекты, делегирует инстанцирование другому объекту.

Абстрактная фабрика	Abstract factory	Класс, который представляет собой интерфейс для создания компонентов системы.
Строитель	Builder	Класс, который представляет собой интерфейс для создания сложного объекта.
Фабричный метод	Factory method	Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать.
Прототип	Prototype	Определяет интерфейс создания объекта через клонирование другого объекта вместо создания через конструктор.
Одиночка	Singleton	Класс, который может иметь только один экземпляр.

Структурные шаблоны (Structural) определяют различные сложные структуры, которые изменяют интерфейс уже существующих объектов или его реализацию, позволяя облегчить разработку и оптимизировать программу.

Адаптер	Adapter / Wrapper	Объект, обеспечивающий взаимодействие двух других объектов, один из которых использует, а другой предоставляет несовместимый с первым интерфейс.
Мост	Bridge	Структура, позволяющая изменять интерфейс обращения и интерфейс реализации класса независимо.
Компоновщик	Composite	Объект, который объединяет в себе объекты, подобные ему самому.
Декоратор или Wrapper/Обёртка	Decorator	Класс, расширяющий функциональность другого класса без использования наследования.
Фасад	Facade	Объект, который абстрагирует работу с несколькими классами, объединяя их в единое целое.
Приспособленец	Flyweight	Это объект, представляющий себя как уникальный экземпляр в разных местах программы, но фактически не являющийся таковым.
Заместитель	Proxy	Объект, который является посредником между двумя другими объектами, и который реализует/ограничивает доступ к объекту, к которому обращаются через него.

Поведенческие шаблоны (Behavioral) определяют взаимодействие между объектами, увеличивая таким образом его гибкость.

Цепочка обязанностей	Chain of responsibility	Предназначен для организации в системе уровней ответственности.
Команда, Action, Transaction	Command	Представляет действие. Объект команды заключает в себе само действие и его параметры.
Интерпретатор	Interpreter	Решает часто встречающуюся, но подверженную изменениям, задачу.
Итератор, Cursor	Iterator	Представляет собой объект, позволяющий получить последовательный доступ к элементам объекта-агрегата без использования описаний каждого из объектов, входящих в состав агрегации.
Посредник	Mediator	Обеспечивает взаимодействие множества объектов, формируя при этом слабую связанность и избавляя объекты от необходимости явно ссылаться друг на друга.
Хранитель	Memento	Позволяет не нарушая инкапсуляцию зафиксировать и сохранить внутренние состояния объекта так, чтобы позднее восстановить его в этих состояниях.
Наблюдатель или Издатель-подписчик	Observer	Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.
Состояние	State	Используется в тех случаях, когда во время выполнения программы объект должен менять своё поведение в зависимости от своего состояния.
Стратегия	Strategy	Предназначен для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости.
Шаблонный метод	Template method	Определяет основу алгоритма и позволяет наследникам переопределять некоторые шаги алгоритма, не изменяя его структуру в целом.
Посетитель	Visitor	Описывает операцию, которая выполняется над объектами других классов. При изменении класса Visitor нет необходимости изменять обслуживаемые классы.

Как выбрать паттерн проектирования

- ✔ Подумайте, как паттерны решают проблемы проектирования
- ✔ Пролистайте разделы каталога, описывающие назначение паттернов
- ✔ Изучите взаимосвязи паттернов
- ✔ Проанализируйте паттерны со сходными целями
- ✔ Разберитесь в причинах, вызывающих перепроектирование
- ✔ Посмотрите, что в вашем дизайне должно быть изменяющимся

Singleton

Синглтон (одиночка) – это паттерн проектирования, цель которого ограничить возможность создания объектов данного класса одним экземпляром.

Он обеспечивает глобальность до одного экземпляра и глобальный доступ к созданному объекту.

Примеры использования

Класс в вашей программе имеет только один экземпляр, доступный всем клиентам.

Например, один объект базы данных, разделяемый различными частями программы.

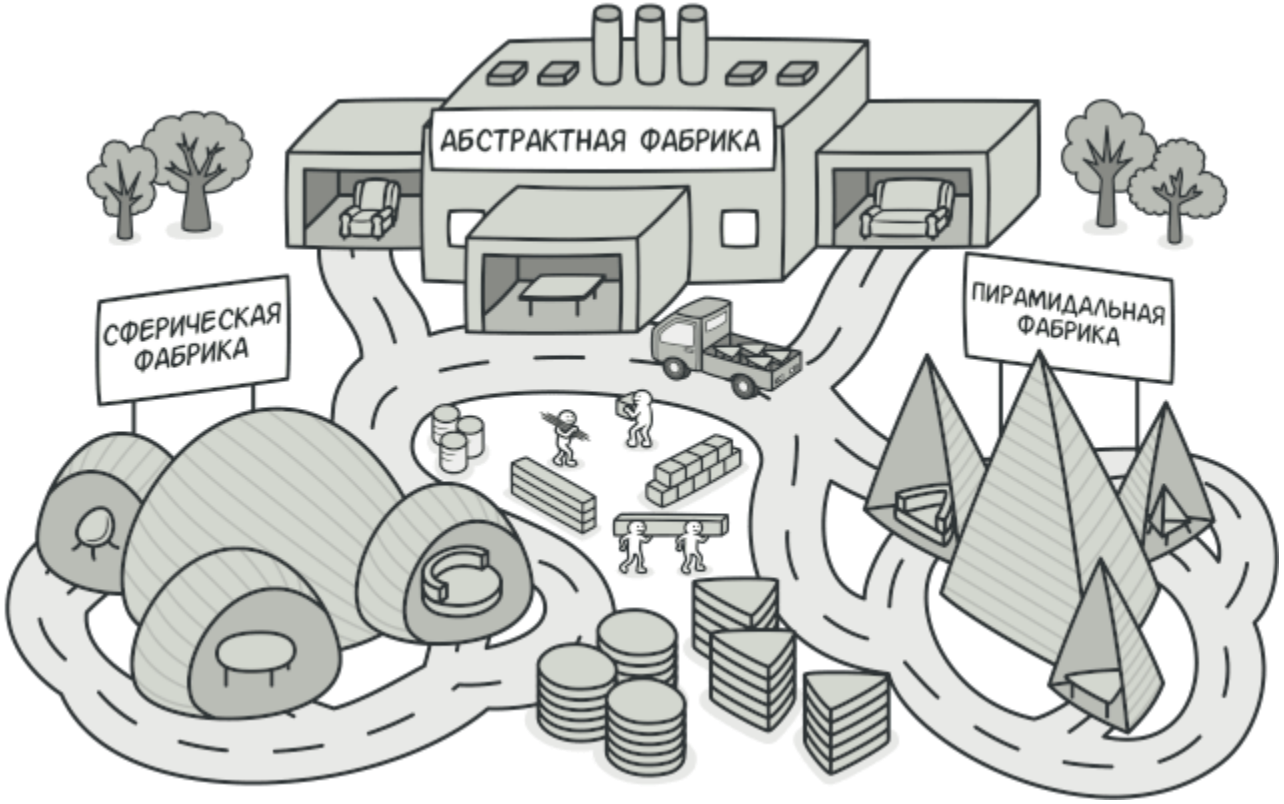
В случае если вам необходим более строгий контроль над глобальными переменными.

Простой способ

```
class Singleton:
    def __new__(cls):
        if not hasattr(cls, 'instance'):
            cls.instance = super(Singleton, cls).__new__(cls)
        return cls.instance


s1 = Singleton()
s2 = Singleton()
print(s1 is s2) # True
s1.x = 123
print(s2.x)
# Выполните этот код, убедитесь, что это один объект
```

Абстрактная фабрика

Абстрактная фабрика — это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.



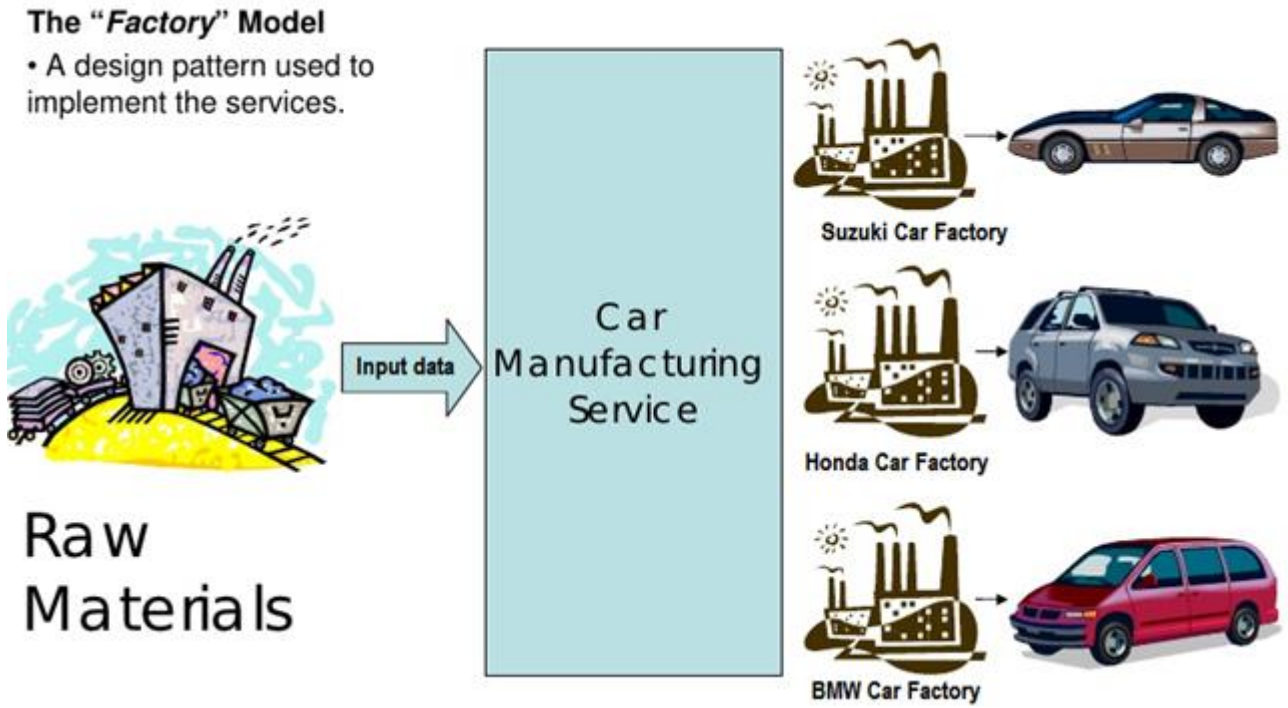
Фабричный (Factory) метод

Назначение: определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать. Фабричный метод позволяет классу делегировать инстанцирование подклассам.

Когда следует использовать фабричный метод
Когда заранее неизвестно, объекты каких типов необходимо создавать;

Когда система должна быть независимой от процесса создания новых объектов и расширяемой: в нее можно легко вводить новые классы, объекты которых система должна создавать;

Когда создание новых объектов необходимо делегировать из базового класса классам наследникам;



Фасад

Представьте, что у вас есть система со значительным количеством объектов.

Каждый объект предлагает богатый набор методов API.

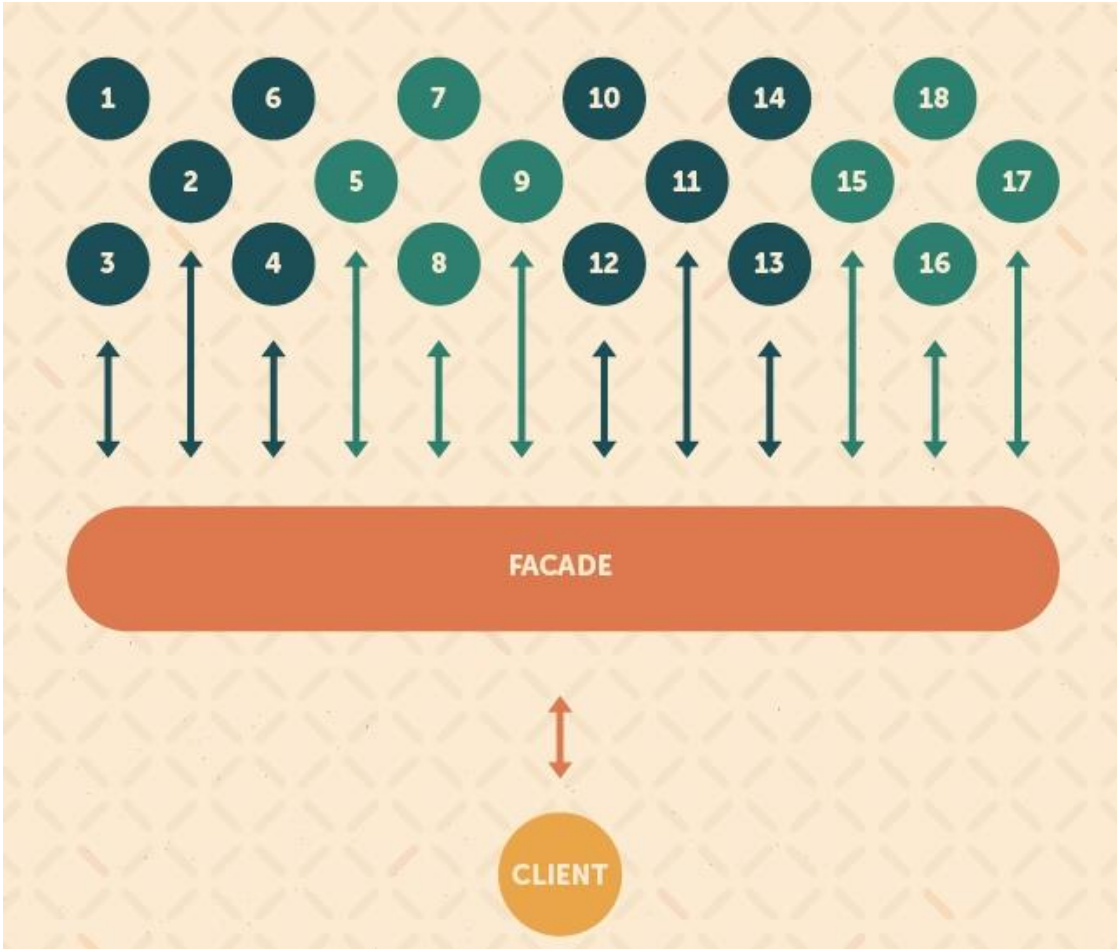
Возможности этой системы велики, но ее интерфейс слишком сложный.

Для удобства можно добавить новый объект, представляющий хорошо продуманные комбинации методов. Это и есть Фасад.

Python предлагает очень элегантную реализацию шаблона.

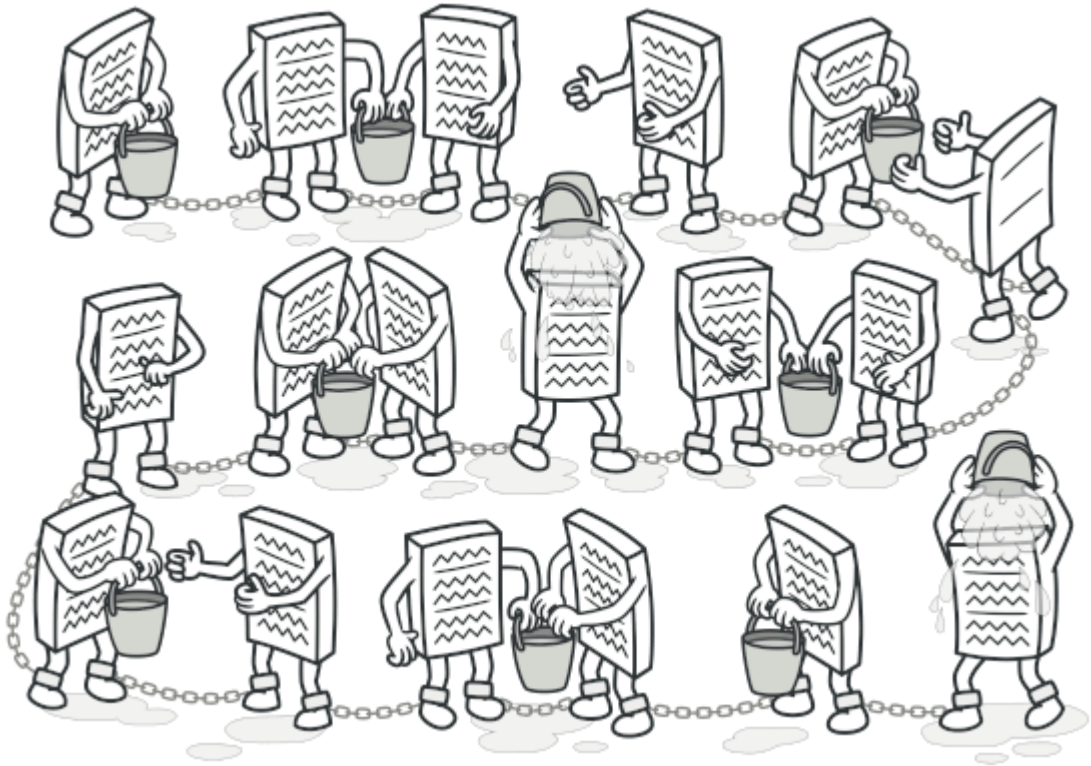
```
class Car(object):
    def __init__(self):
        self._tyres = [Tyre('front_left'),
                       Tyre('front_right'),
                       Tyre('rear_left'),
                       Tyre('rear_right'), ]
        self._tank = Tank(70)

    def tyres_pressure(self):
        return [tyre.pressure for tyre in self._tyres]

    def fuel_level(self):
        return self._tank.level
```

Без всяких трюков и фокусов класс Car стал Фасадом.



Chain of Responsibility

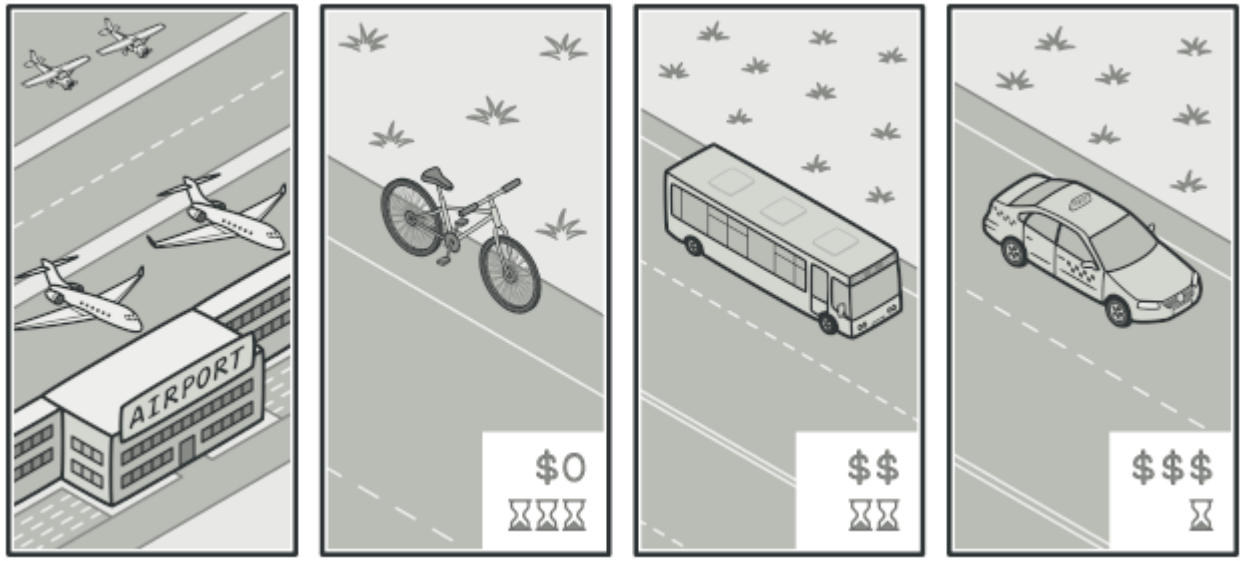
Цепочка обязанностей — это поведенческий паттерн проектирования, который позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.



Стратегия

Стратегия – это поведенческий паттерн проектирования, который определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.

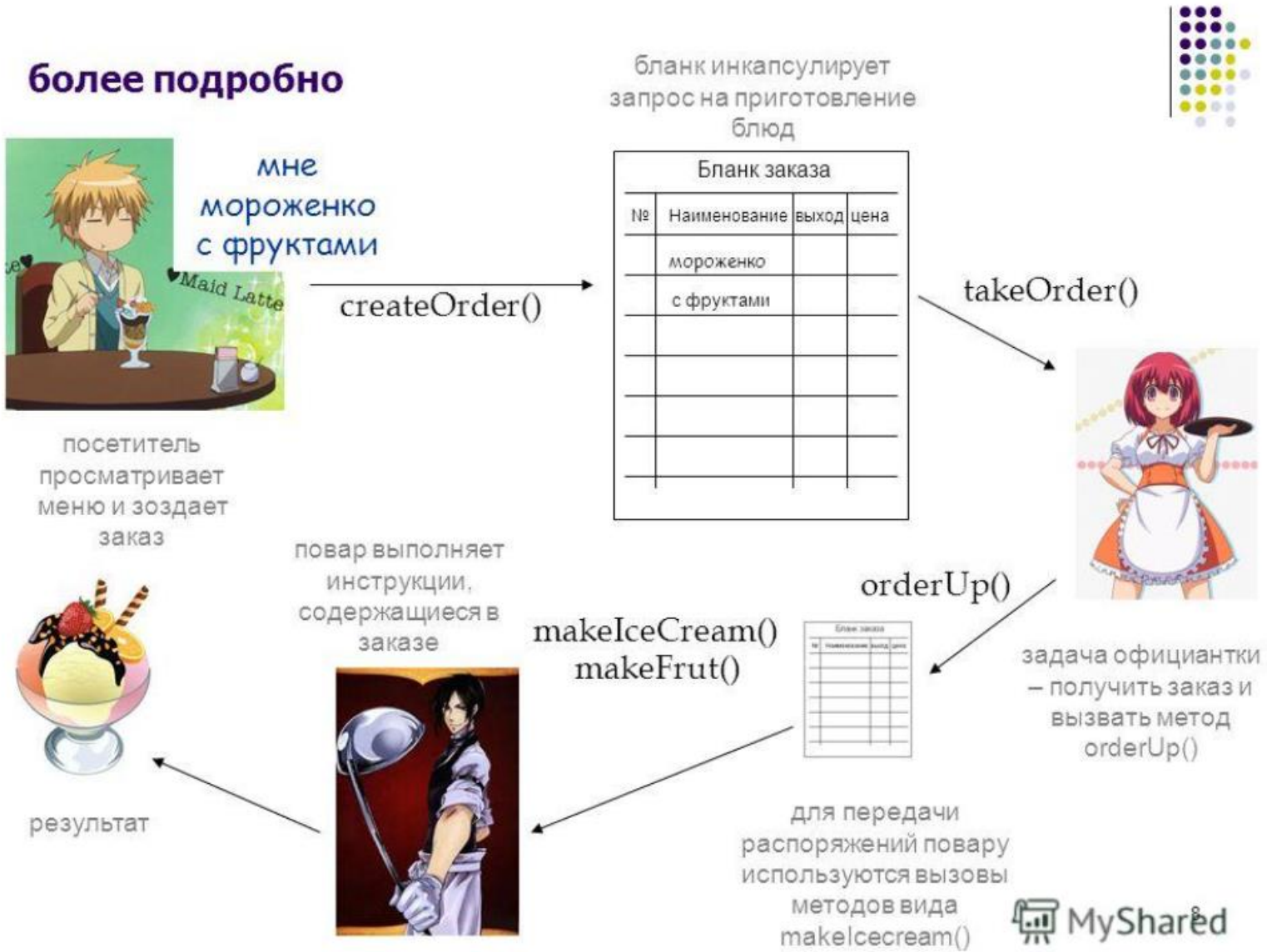
Вам нужно добраться до аэропорта. Можно доехать на автобусе, такси или велосипеде. Здесь вид транспорта является стратегией. Вы выбираете конкретную стратегию в зависимости от контекста – наличия денег или времени до отлёта.



Команда


Взаимодействие объектов, на примере кафе
Maid Latte
Maid Latte
Бланк заказа
№ Наименование выход цена
1
Посетитель передает официантке свой заказ
2
Официантка получает заказ, кладет его на стойку и говорит «У нас заказ!»
3
повар готовит блюда, входящие в заказ
MyShared

Команда


более подробно
мне мороженко с фруктами
Maid Latte
createOrder()
посетитель просматривает меню и зоздает заказ
бланк инкапсулирует запрос на приготовление блюд
Бланк заказа
№ Наименование выход цена
мороженко
с фруктами
takeOrder()
orderUp()
задача официантки – получить заказ и вызвать метод orderUp()
повар выполняет инструкции, содержащиеся в заказе
makeIceCream()
makeFrut()
результат
для передачи распоряжений повару используются вызовы методов вида makeIcecream()
MyShared

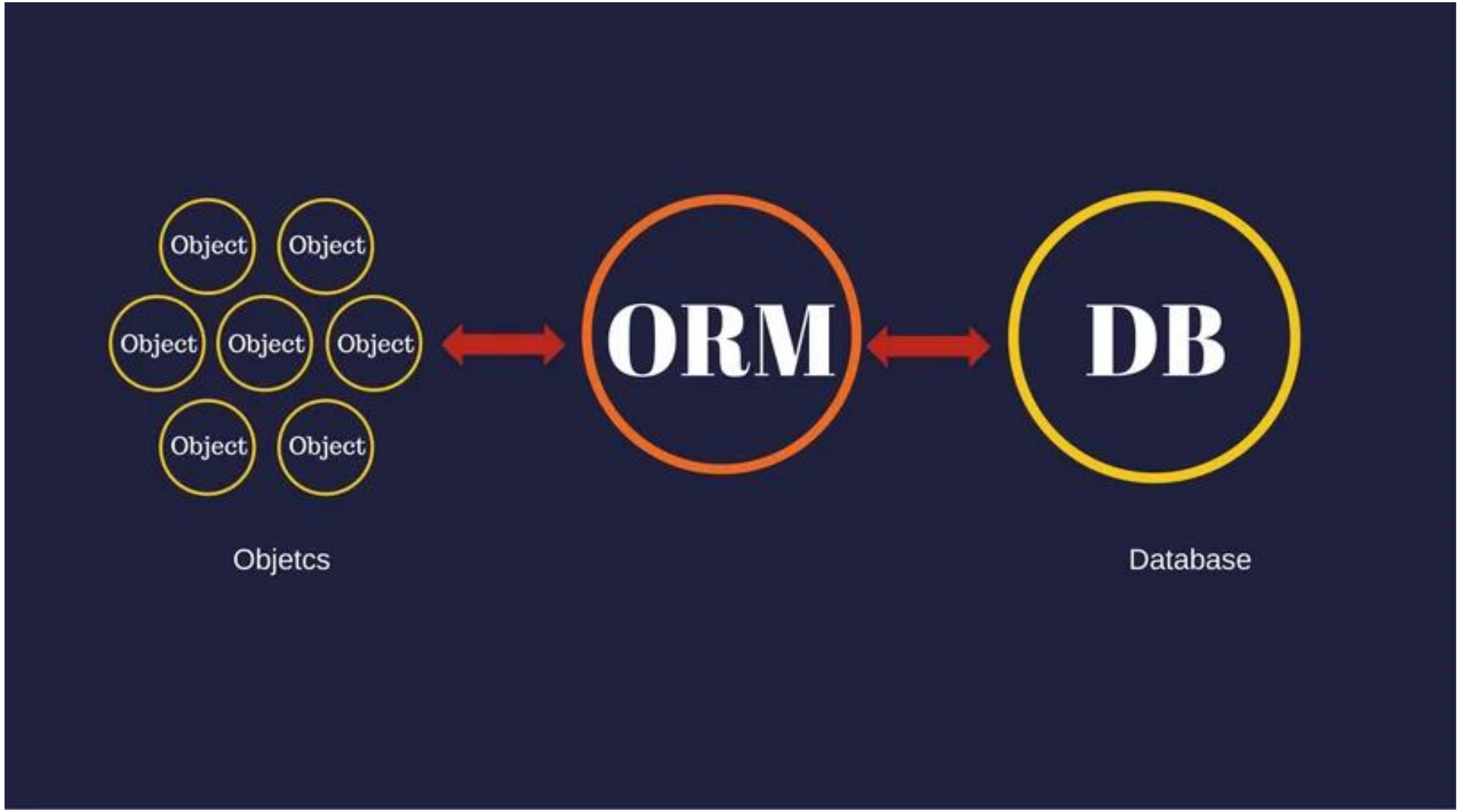
Команда


От кафе к паттерну Команда
Клиент отвечает за создание объекта команды, содержащего набор операций с получателем
Команда содержит единственный метод execute(), в котором инкапсулированы операции с получателем
create Command Object()
createCommandObject()
execute()
клиент
команда
Клиент вызывает метод setCommant() Инициатора, передавая ему объект команды. Инициатор хранит его до момента использования
запрос на выполнение комвнды
setCommand()
setCommand()
инициатор
инициатор вызывает метод execute()
execute()
action1()
action2()
...
execute()
получатель
команда
action1()
action2()
активизируются операции с получателем
MyShared

ORM SQLAlchemy

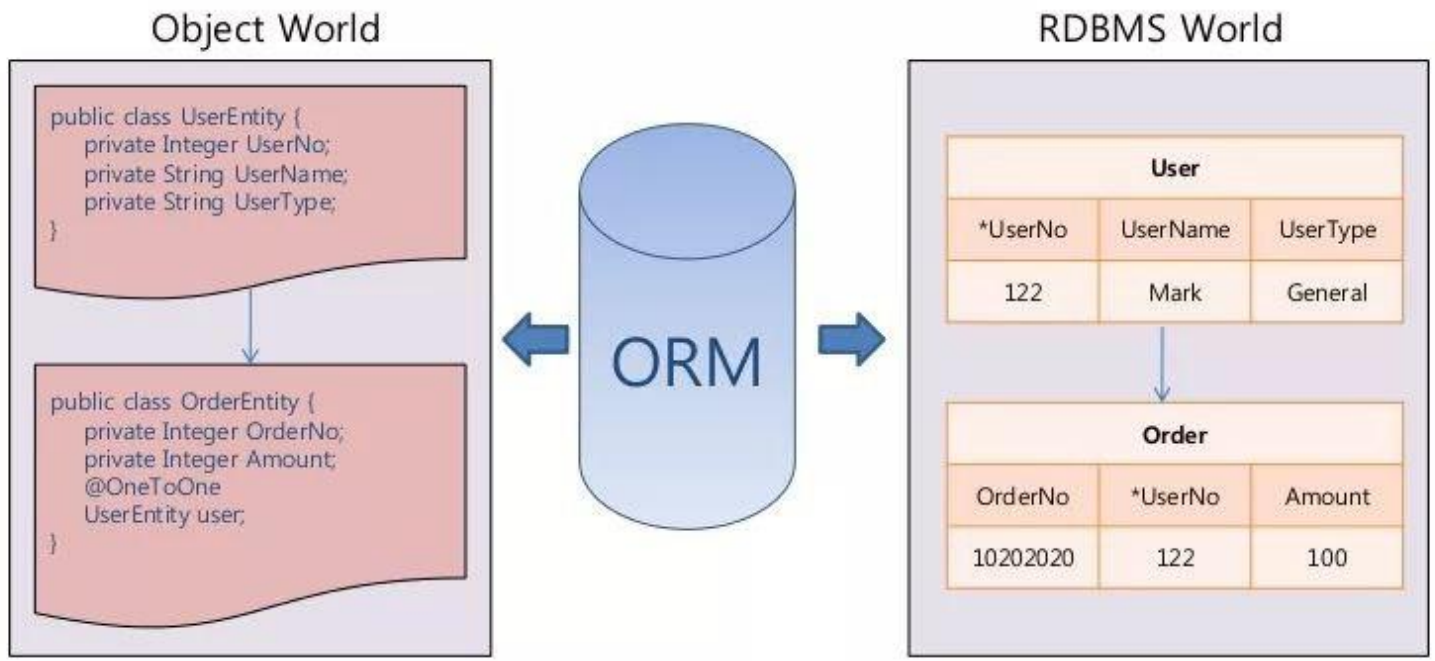
Определение

Объектно-реляционное отображение (Object Relational Mapping) — это метод, который позволяет вам запрашивать и манипулировать данными из базы данных, используя объектно-ориентированную парадигму.


Object
Object
Object
Object
Object
Object
Object
ORM
DB
Objetcs
Database

Object Relational Mapping

-Bridge between relational database and object world



SELECT

```
cur = con.cursor()
cur.execute("SELECT admission, name, age, course, department from STUDENT")

rows = cur.fetchall() # возвращает список всех строк
for row in rows:
    print("ADMISSION =", row[0])
    print("NAME =", row[1])
    print("AGE =", row[2])
    print("COURSE =", row[3])
    print("DEPARTMENT =", row[4], "\n")
con.close()
```

Достоинства ORM

Безопасность запросов

Представление параметров типами данных основного языка (без преобразования в типы БД)

Прозрачное кеширование данных и возможность выполнения отложенных запросов

Переносимость (использование разных СУБД (без дополнительных правок в коде)

Избавление программиста от необходимости вникать в детали реализации той или иной СУБД и синтаксиса соответствующего диалекта языка запросов.

Недостатки ORM

Абстрагирование от языка SQL. Во многих случаях выборка данных перестает быть интуитивно понятной и очевидной.

Дополнительные накладные расходы на конвертацию запросов и создание внутренних объектов самого ORM.

Возможна потеря производительности

Философия SQLAlchemy

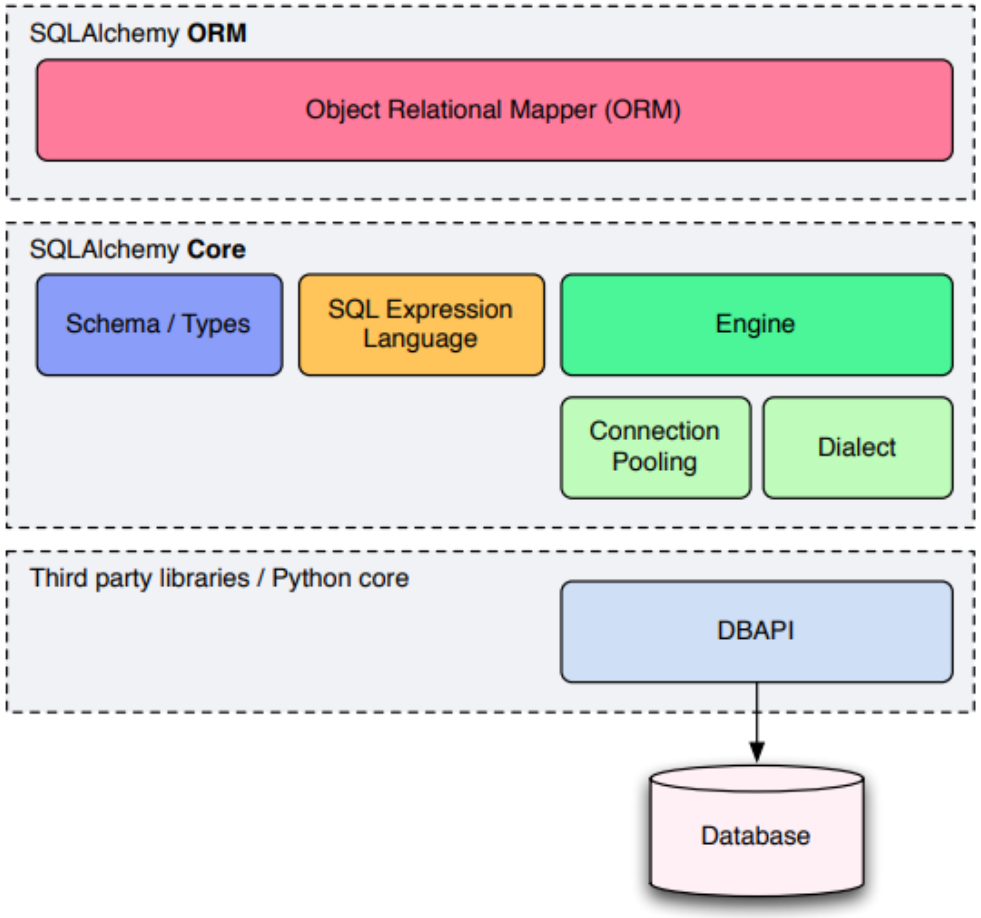
Привести использование различных баз данных и адаптеров к максимально согласованному интерфейсу

Никогда не "скрывайте" базу данных или ее концепции разработчики должны знать / продолжать думать на языке SQL.

Обеспечить автоматизацию рутинных операций CRUD

Разрешить выразить синтаксис DB/SQL в декларативном шаблоне.

Архитектура SQLAlchemy



SQAlchemy состоит из двух компонентов ORM и CORE

Компонента SQLAlchemy - Core

SQLAlchemy Core - это абстракция над традиционным SQL. Он предоставляет SQL Expression Language, позволяющий генерировать SQL-инструкции с помощью конструкций Python.

- **Engine** - механизм, который обеспечивает подключение к конкретному серверу базы данных.
- **Dialect** - интерпретирует разные диалекты SQL и команды базы данных в синтаксис конкретного DBAPI и серверной части базы данных.
- **Connection Pool** - хранит коллекцию подключений к БД для быстрого повторного использования
- **SQL Expression Language** - позволяет писать SQL запрос с помощью выражений Python
- **Schema/Types** - использует объекты Python для представления таблиц, столбцов и типов данных.

SQLAlchemy - ORM

Позволяет создавать объекты Python, которые могут быть сопоставлены с таблицами реляционной базы данных

Предоставляет систему запросов, которая загружает объекты и атрибуты с использованием SQL, сгенерированного на основе сопоставлений.

Выстроена поверх Core - использует Core для создания SQL и обращений с базой данных

Представляет несколько более объектно-ориентированную перспективу,в отличие от перспективы, ориентированной на схему

Установить SQLAlchemy

cmd: pip install SQLAlchemy

PyCharm: File / Settings / Project ___ / Python Interpretator / + / SQLAlchemy / Install Package

Создание базы

```
from sqlalchemy import create_engine, MetaData, Table, Integer, String,  Column, DateTime, ForeignKey, Numeric, SmallInteger
from sqlalchemy.orm import relationship, declarative_base
from sqlalchemy_utils import create_database
engine = create_engine("postgresql+psycopg2://postgres:Ваш_пароль@localhost/sqlalchemy_tuts5")
create_database(engine.url)


Base = declarative_base()
class Customer(Base):       # Класс, который соответствует таблице
    __tablename__ = 'customers'
    id = Column(Integer(), primary_key=True)
    first_name = Column(String(100), nullable=False)
    last_name = Column(String(100), nullable=False)
    username = Column(String(50), nullable=False)
    email = Column(String(200), nullable=False)
Base.metadata.create_all(engine)
```

Добавляем книгу

```
from sqlalchemy.orm import Session, sessionmaker
from sqlalchemy import create_engine, MetaData, Table, Integer, String,   Column, DateTime, ForeignKey,
Numeric, SmallInteger
from sqlalchemy.orm import relationship, declarative_base

engine = create_engine("postgresql+psycopg2://postgres:############@localhost/postgres")
session = Session(bind=engine)

Base = declarative_base()
class Book(Base):      # класс, который соответствует таблице
    __tablename__ = 'book'
    book_id = Column(Integer(), primary_key=True)
    title = Column(String(100), nullable=False)
    author_id = Column(Integer(), nullable=False)
    price = Column(Integer(), nullable=False)
    amount = Column(Integer(), nullable=False)

c1 = Book( book_id = 124,  title = 'Title',   author_id = 1,  price = 123,  amount = 45 )

session.add(c1)
session.commit()
```

Запросы

```
from sqlalchemy.orm import Session, sessionmaker
from sqlalchemy import create_engine, MetaData, Table, Integer, String,   Column, DateTime, ForeignKey, Numeric, SmallInteger
from sqlalchemy.orm import relationship, declarative_base
engine = create_engine("postgresql+psycopg2://postgres:############@localhost/postgres")
session = Session(bind=engine)
Base = declarative_base()
class Book(Base):     # класс, который соответствует таблице
    __tablename__ = 'book'
    book_id = Column(Integer(), primary_key=True)
    title = Column(String(100), nullable=False)
    author_id = Column(Integer(), nullable=False)
    price = Column(Integer(), nullable=False)
    amount = Column(Integer(), nullable=False)
q = session.query(Book)                 # Все книги
for c in q:
    print(c.book_id, c.title, c.author_id, c.amount, c.price)
print('Количество книг', session.query(Book).count())

#  i = session.query(Book).get(123)             # Запросить одну книгу с book_id = 123 (устаревший метод)
i = session.get(Book, 123)                      # Запросить одну книгу с book_id = 123
i.title = "New_Title"                   # Изменение значения
session.add(i)

q = session.query(Book)
for c in q:
    print(c.book_id, c.title, c.author_id, c.amount, c.price)
session.commit()
```

Методы

Метод	Описание
all()	Возвращает результат запроса (объект Query) в виде списка
count()	Возвращает общее количество записей в запросе
first()	Возвращает первый результат из запроса или None, если записей нет
scalar()	Возвращает первую колонку первой записи или None, если результат пустой. Если записей несколько, то бросает исключение MultipleResultsFound
one()	Возвращает одну запись. Если их несколько, бросает исключение MutlipleResultsFound. Если данных нет, бросает NoResultFound
get(pk)	Возвращает объект по первичному ключу (pk) или None, если объект не был найден
add()	Добавление записи
commit()	Сохранение данных

Методы

```
q = session.query(Book)
for c in q:     print(c.book_id, c.title, c.author_id, c.amount, c.price)


print('Количество книг', session.query(Book).count())
for c in session.query(Book).filter_by(author_id=1):     print(c.title)


print(session.query(Book).filter_by(author_id=2).first().title)


for i in session.query(Book.price).order_by(Book.price).limit(10).all(): print(int(i[0]))


for i in session.query(Book.title, Book.price,
Book.amount).order_by(Book.price).limit(10).all():
    print(f"{i[0]:20} {int(i[1]):5} {i[2]:5}")
```

Задание

Добавьте в таблицу новую запись.

Потом загрузите все данных из таблицы и убедитесь, что новая запись добавилась.

Задача 28-1

Дан список чисел a. Назовем пару (a[i], a[j]) инверсией, если i < j, а a[i] > a[j]. Напишите функцию, которая возвращает количество инверсий в списке.

Например:

[1,2,3,4,5] -> 0

[5,4,3,2,1] -> 10

Задача 28-2

Дана квадратная матрица чисел, некоторые клеточки содержат отрицательные числа, туда нельзя заходить.

Ваша задача построить оптимальный путь из произвольной точки в произвольную точку.

Оптимальность пути определяется суммой клеточек от начальной точки до конечной включительно.

Вы можете двигаться вправо, влево, вверх, вниз, не вылезая за границы матрицы, не заходя на клеточки с отрицательными числами.

Например:

```
matrix =    [[1,  2,  3],
             [ 4, -1,  6],
             [ 7,  8,  9]]
```

Оптимальным маршрутом из точки (0,0) в точку (2,2) будет путь: (0,0),(0,1),(0,2),(1,2),(2,2) «длиной» 21.

Задача 28-3

Напишите функцию, которая рассчитывает наименьшее число перестановок при перемещении Ханойской башни (n дисков насаженных на одном стержне). Требуется переместить эти диски на соседний стержень. Разрешается использовать третий стержень. Диски можно класть только на диски большего диаметра.
Для n = 1, число перестановок равно 1.
Для n = 2, число перестановок равно 3.