


# Объектно-ориентированное программирование на Python

Осипов Никита Алексеевич



# ЛЕКЦИЯ 1. ОБЗОР ОБЪЕКТНО-ОРИЕНТИРОВАННОЙ ПАРАДИГМЫ И СРЕДСТВ ЯЗЫКА PYTHON ДЛЯ ЕЕ РЕАЛИЗАЦИИ

## *Учебные вопросы:*

1. Основные элементы объектно-ориентированного подхода
2. Обзор особенностей языка Python

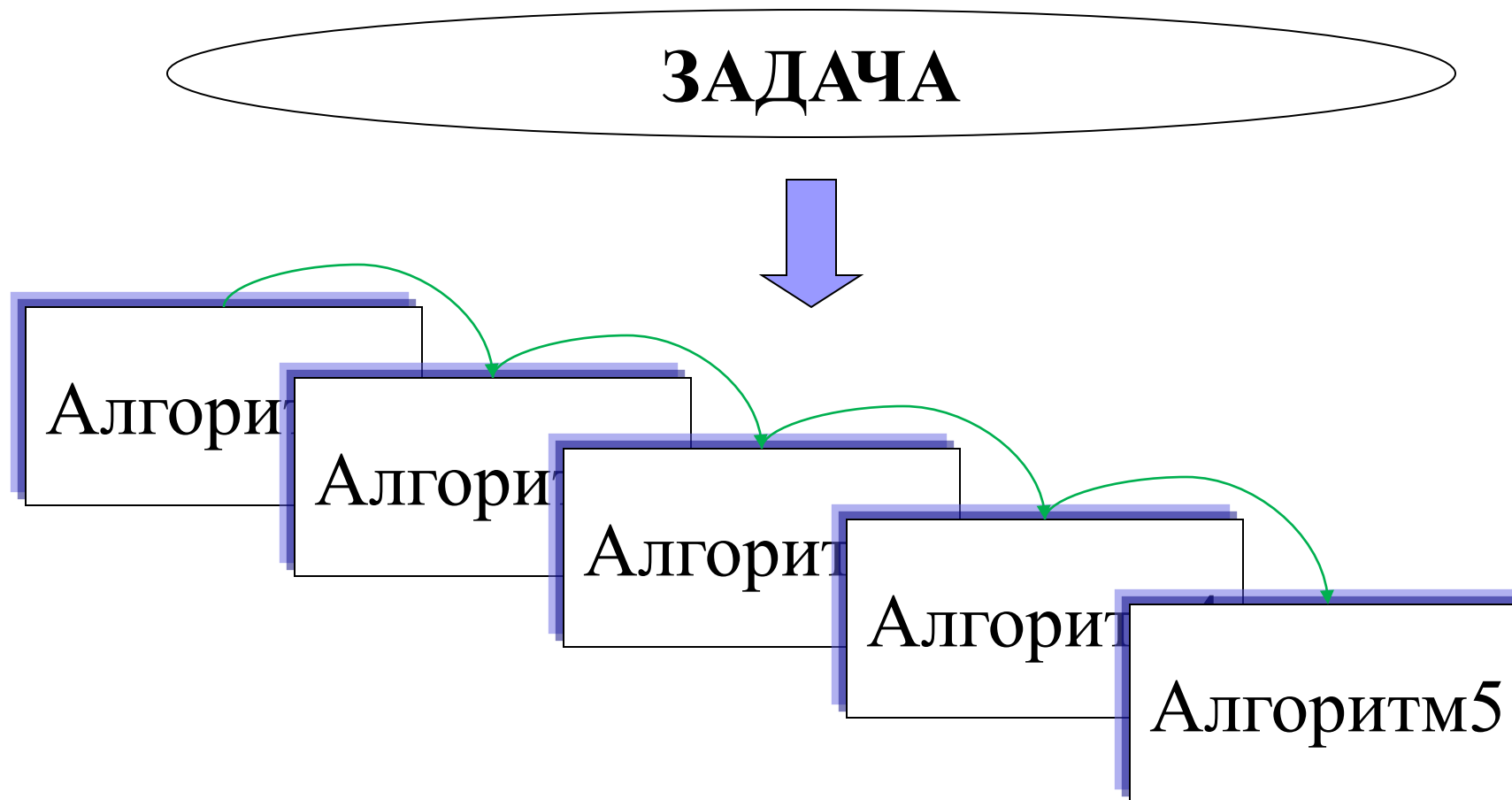
# О парадигмах программирования

- Программы имеют два основных аспекта:
  - набор алгоритмов
  - набор данных, которыми оперируют
- Императивное (процедурное) программирование
- Объектно-ориентированное программирование
- Функциональное программирование

# О парадигмах программирования

- Процедурная парадигма: задача непосредственно моделируется набором алгоритмов:
  - Реши, какие требуются процедуры; используй наилучшие доступные алгоритмы
- Парадигма абстрактных типов данных:
  - Реши, какие требуются типы; обеспечь полный набор операций для каждого типа
- ООП расширяет парадигму абстрактных типов данных механизмом *наследования* (повторного использования существующих объектов) и *динамического связывания* (повторного использования существующих интерфейсов)
- Функциональное программирование — процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании)

# Алгоритмическая декомпозиция



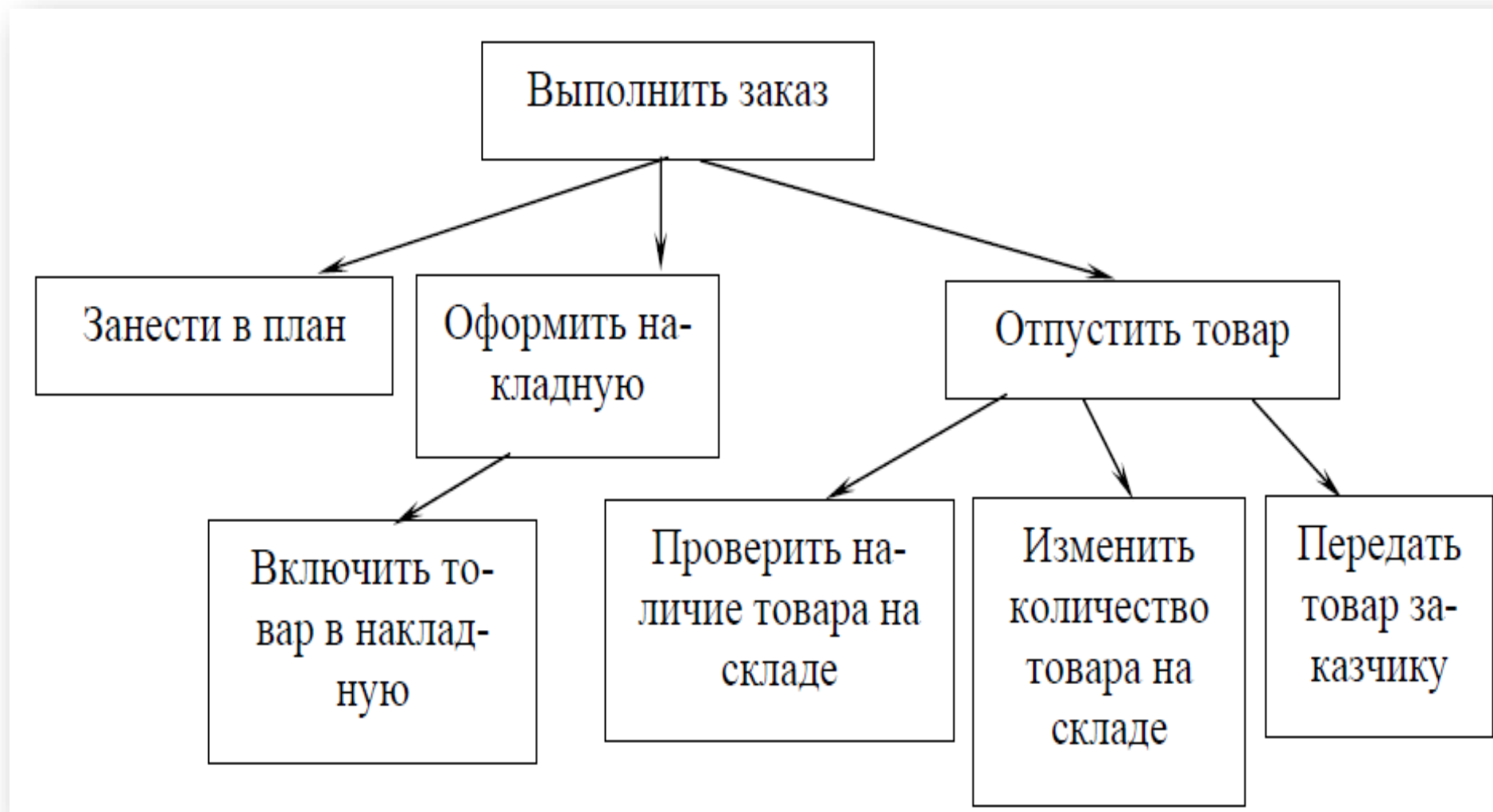
# Значение функций

- Проблема – увеличение размера и сложности программ
- Решение – разделение сложных и больших программ на небольшие легко управляемые части, называемые *функциями*:

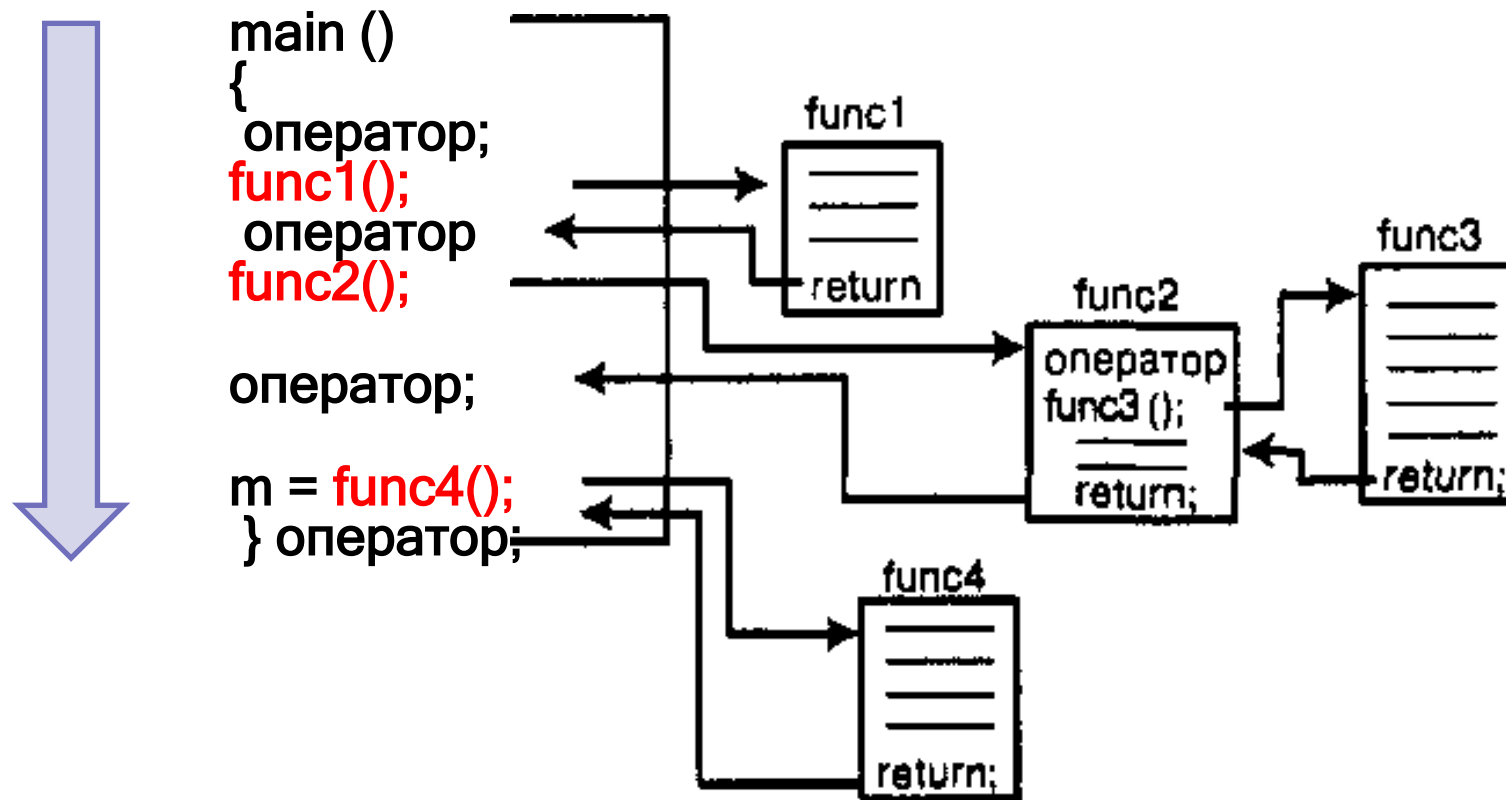
- ☐ каждая функция в программе должна выполнять определенную задачу

Если программе необходимо выполнить какую-либо задачу, то она *вызывает* соответствующую функцию, обеспечивая эту функцию информацией, которая ей понадобится в процессе обработки.

# Алгоритмическая декомпозиция. Пример



# Работа с функциями



- Когда программа вызывает функцию, управление переходит к телу функции, а затем выполнение программы возобновляется со строки, следующей после вызова



# Описание функции в Python

- Определение (описание) функции состоит из двух частей: заголовка и тела:

```
def <имя функции> (<список параметров>):  
    <тело функции>  
    return
```

- **список параметров** - состоит из перечня типов и имен параметров, разделенных запятыми, круглые скобки обязательны.
- **тело функции** – набор выражений.

# Процедуры или Функции?

**Процедура** – вспомогательный алгоритм, который выполняет некоторые действия.

- текст (алгоритм) процедуры записывается **до** её вызова в основной программе
- чтобы процедура заработала, нужно **вызвать** её по имени из основной программы или из другой процедуры

## Объявление процедуры:

*define*  
определить

```
def ErrorMessage() :  
    print("Ошибка в алгоритме")
```

ВЫЗОВ  
процедуры

```
n = int(input())  
if n < 0:  
    ErrorMessage()
```

# Процедуры или Функции?

**Функция** – это вспомогательный алгоритм, который **возвращает значение-результат** (число, символ или объект другого типа).

ВЫЗОВ  
функции

```
s = input()  
n = int(s)
```

Без  
параметра

С  
передачей  
параметра

# Локальные и глобальные переменные

глобальная переменная

```
a = 52
```

```
def qq():
```

```
    a = 1
```

```
    print(a)
```

```
qq()
```

```
print(a)
```

1

52

локальная переменная

```
a = 5
```

```
def qq():
```

```
    print(a)
```

```
qq()
```

5

```
a = 5
```

```
def qq():
```

```
    global a
```

```
    a = 1
```

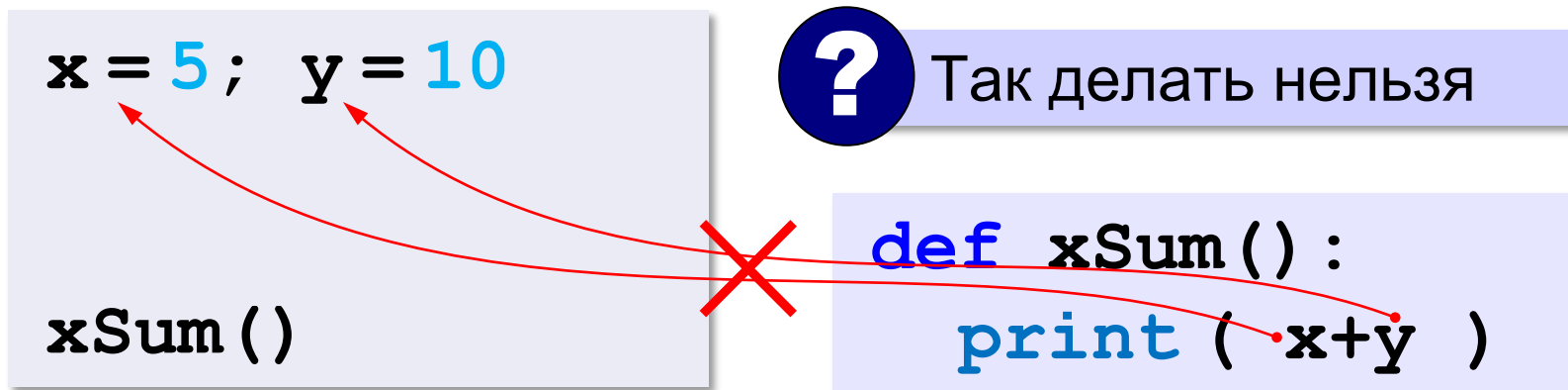
```
qq()
```

```
print(a)
```

1

работаем с  
глобальной  
переменной

# Процедура и глобальные переменные



- 1) процедура связана с глобальными переменными, нельзя перенести в другую программу
- 2) печатает сумму только `x` и `y`, нельзя напечатать сумму других переменных или сумму `x*y` и `3x`

? Как исправить?

передавать данные через параметры

# Процедура и глобальные переменные

Глобальные:

x	y
5	10
z	w
17	3

```
x = 5; y = 10
Sum2 ( x, y )
z = 17; w = 3
Sum2 ( z, w )
Sum2 ( z+x, y*w )
```

```
def Sum2 (a, b) :
    print ( a+b )
```

Локальные:

a	b	
5	10	15
17	3	20
22	30	52



- 1) процедура не зависит от глобальных переменных
- 2) легко перенести в другую программу
- 3) печатает сумму любых выражений

# К чему приведет процедурный подход?

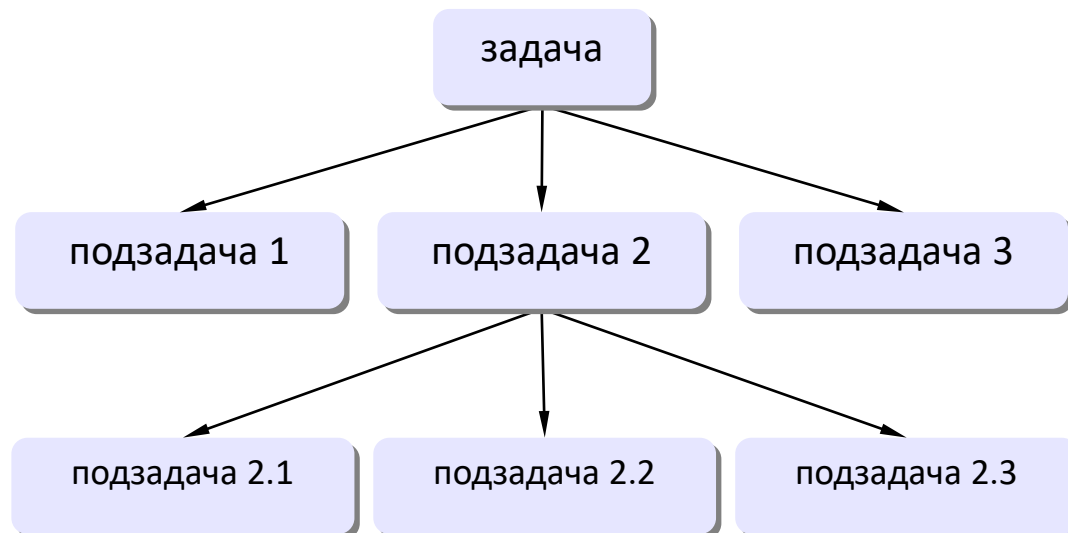


Главная проблема – **сложность!**

- программы из миллионов строк
- тысячи переменных и массивов

Э. Дейкстра: Человечество еще в древности придумало способ управления сложными системами: «**разделяй и властвуй**»

## Структурное программирование:



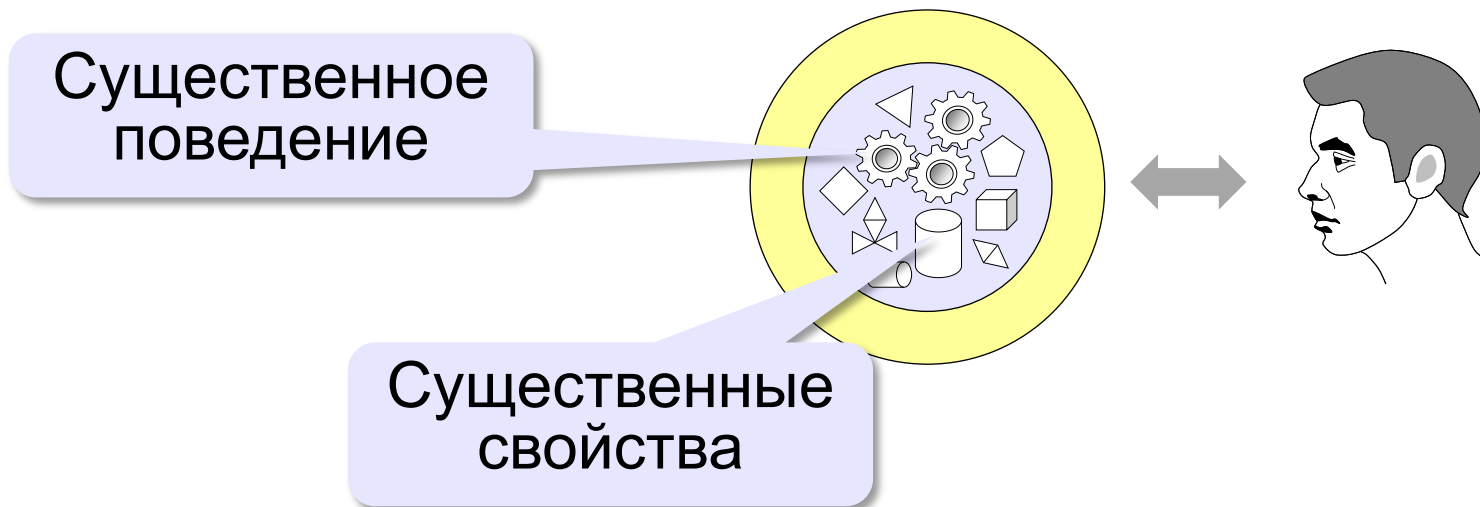
**декомпозиция по задачам**



человек мыслит иначе – объектами

# Восприятие внешнего мира

Как мы воспринимаем объекты?



**Абстракция** – это выделение существенных свойств объекта, отличающих его от других объектов.

**!** Разные цели – **разные модели!**



# Основная идея

- Разработка объектно-ориентированного ПО исходит из осознания того, что правильно построенные системы программной инженерии должны основываться на повторно используемых **компонентах** высокого качества, как это делается в других инженерных сферах

ОО-подход определяет, какую форму должны иметь эти компоненты: каждый из них должен быть основан на некотором типе объектов



# Объектный подход

- OOA (object oriented analysis)  
объектно-ориентированный анализ
- OOD (object oriented design)  
объектно-ориентированное проектирование
- OOP (object oriented programming)  
объектно-ориентированное программирование

# Объектно-ориентированный анализ

- Выделить объекты
- Определить их существенные свойства
- Описать поведение (команды, которые они могут выполнять)

**Объектом** можно назвать то, что имеет чёткие границы и обладает *состоянием* и *поведением*

- Состояние определяет поведение
  - ☐ лежащий человек не прыгнет
  - ☐ незаряженное ружье не выстрелит

# Объектная модель

Для объектно-ориентированного стиля концептуальная база -  
это **объектная модель**

- объектом может быть:

- ☐ автомобиль,
- ☐ человек и т.д.

- объекты обладают свойствами:

- ☐ цвет,
- ☐ размер и т.д.

- они обладают поведением:

- ☐ начинают функционировать
- ☐ меняют свое состояние в ответ на определенный набор внешних воздействий

Что делает объект объектом?

Не то, что он может иметь  
физического двойника,  
а то, что с ним можно  
манипулировать в программе,  
используя множество хорошо  
определенных операций,  
называемых методами

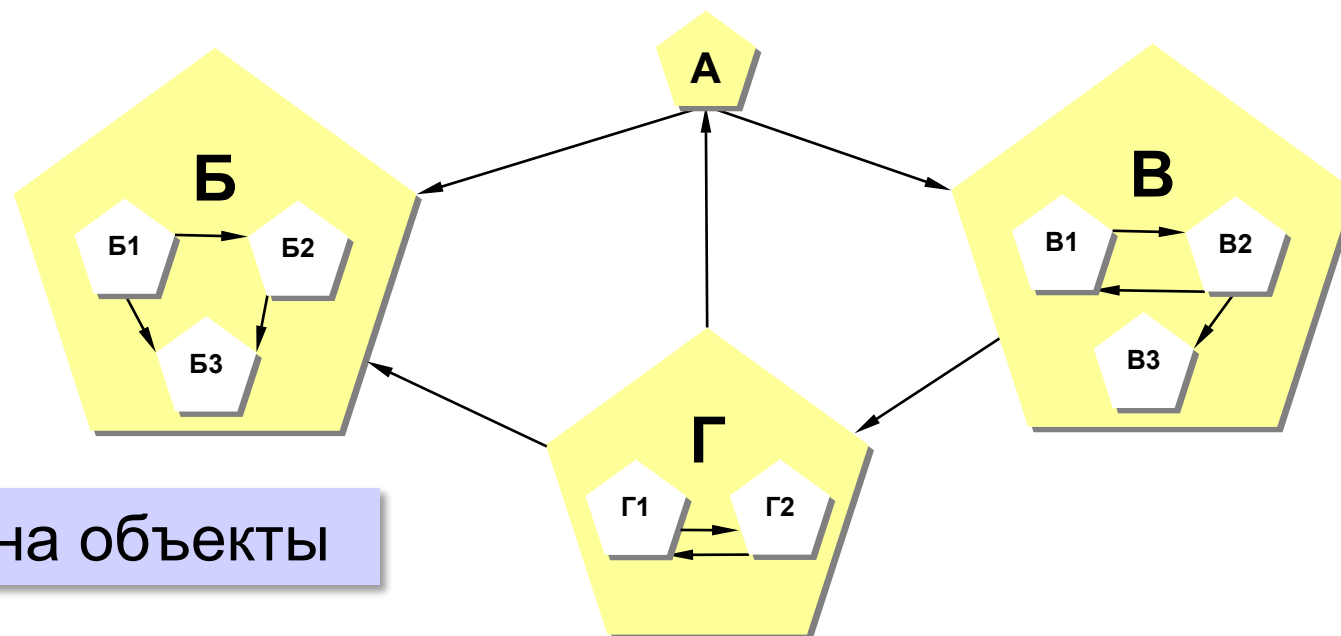
# Объектно-ориентированный подход

- Основополагающая идея:

- объединение **данных** и **действий**, производимых над **данными**, в единое целое, которое называется **объектом**

**Программа** – множество объектов (моделей), каждый из которых обладает своими свойствами и поведением, но его внутреннее устройство скрыто от других объектов.

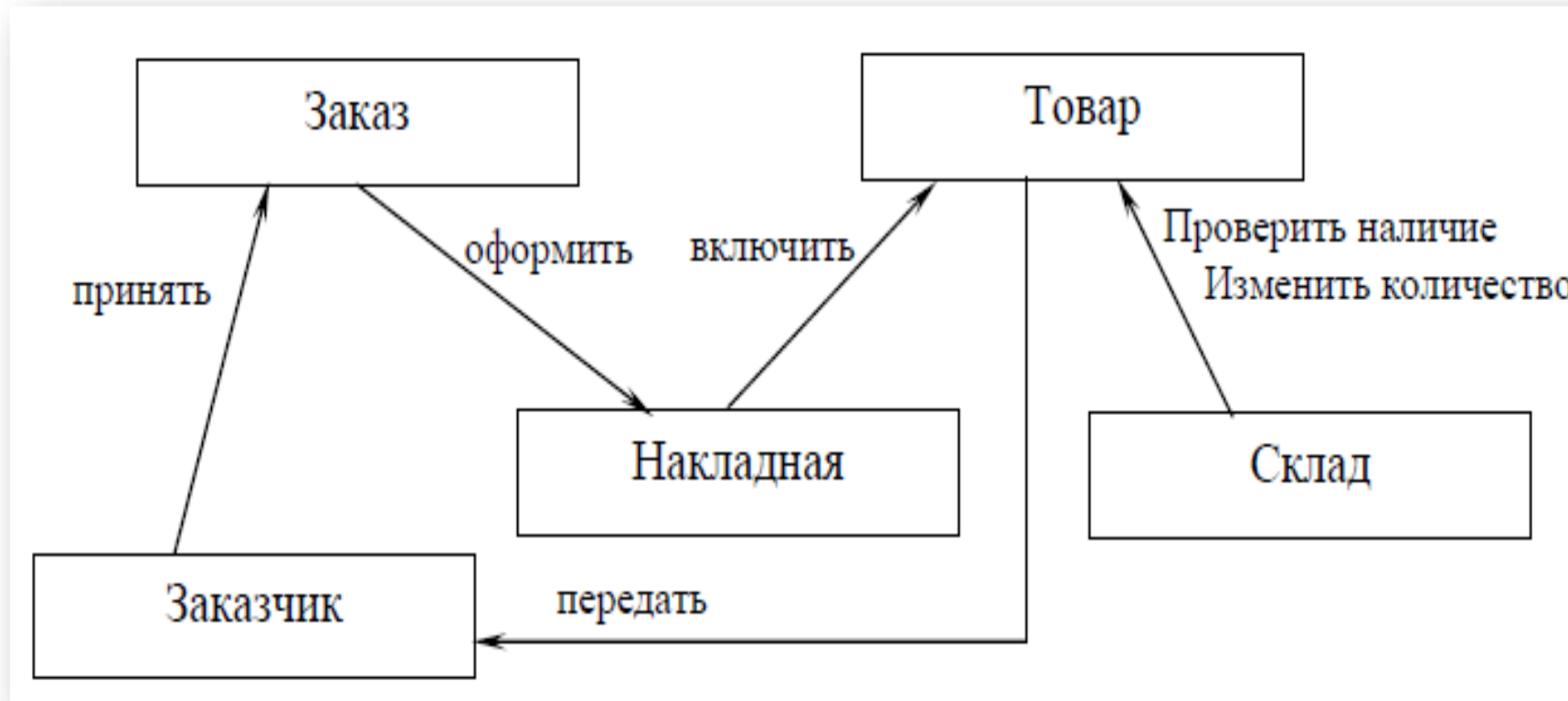
- Структура программы:



«Разделить» задачу на объекты

# Объектно-ориентированная декомпозиция

- Предметная область представлена как совокупность некоторых автономных объектов, которые взаимодействуют друг с другом, чтобы обеспечить функционирование всей системы в целом



# Объектно-ориентированный анализ и проектирование

- В процессе ОО-анализа основное внимание уделяется определению и описанию объектов (понятий) в терминах предметной области
  - Пример, в библиотечной информационной системе среди понятий должны присутствовать *Book* (книга), *Library* (библиотека) и *Patron* (клиент)
- В процессе ОО-проектирования определяются программные объекты и способы их взаимодействия с целью выполнения системных требований
  - Пример, в библиотечной системе программный объект *Book* может содержать атрибут *title* (название) и метод *getChapter* (определить номер главы)

# Объектно-ориентированный анализ и проектирование

- Пример. Объект «**Маршрут**»
- Вопросы, которые можно задавать:
  - ☐ Какова начальная точкой маршрута? Какова конечная точка?
  - ☐ Как передвигаемся на маршруте: пешком, на автобусе, на автомобиле, метро или маршрут смешанный?
  - ☐ Сколько этапов включает маршрут?
  - ☐ Какие линии метро используются, если они есть в маршруте?
- Запросы (queries) – методы, позволяющие получать свойства объекта
- Команды (commands) – позволяют изменять видимые (доступные) свойства объектов
  - ☐ Удалить (этап), Присоединить (этап), Добавить в начало (этап)



# Объектно-ориентированный анализ и проектирование

- Требуется добавить **методы программных классов**, описывающие передачу сообщений между объектами для удовлетворения требованиям
  - ✓ Вопрос определения способов взаимодействия объектов и принадлежности методов важен и не тривиален
- Применить принципы и шаблоны объектного проектирования для создания проектных моделей взаимодействия объектов
  - шаблоны проектирования **GRASP** – шаблоны распределения обязанностей

# Проектирование на основе обязанности

- Программные объекты имеют обязанности
- В UML обязанность (responsibility) определяется как  
*“контракт или обязательство”*
- Под обязанностью в контексте GRASP понимается некое *действие (функция)* объекта
- Обязанности описывают поведение объекта
- В общем случае два типа обязанностей:
  - Знание (knowing)
  - Действие (doing)

Проектирование на основе обязанностей  
responsibility-driven design — RDD

## Обязанности, относящиеся к действиям объекта

- Выполнение некоторых действий самим объектом, например, создание экземпляра или выполнение вычислений.
- Инициирование действий других объектов.
- Управление действиями других объектов и их координирование

### Пример.

- Объект Sale отвечает за создание экземпляра SalesLineItems (действие)

## Обязанности, относящиеся к знаниям объекта

- Наличие информации о закрытых инкапсулированных данных.
- Наличие информации о связанных объектах.
- Наличие информации о следствиях или вычисляемых величинах.

### Пример.

- Объект Sale отвечает за наличие информации о стоимости покупки (знание)

# Реализация обязанностей

- Обязанности реализуются посредством **методов**, действующих либо отдельно, либо во взаимодействии с другими методами и объектами
- Пример.
  - Для класса **Sale** можно определить один или несколько методов вычисления стоимости (метод **getTotal**).
  - Для выполнения этой обязанности объект Sale должен взаимодействовать с другими объектами, в том числе передавать сообщения **getSubtotal** каждому объекту **SalesLineItem** о необходимости предоставления соответствующей информации этими объектами

# Принципы и рекомендации ООАП

- GRASP – General Responsibility Assignment Software Patterns (Общие шаблоны распределения обязанностей в программных системах)
  - Information Expert
    - информационный эксперт, класс, у которого имеется информация, требуемая для выполнения обязанности
  - Creator
    - Назначить классу В обязанность создавать экземпляры класса А

Идеальный класс должен иметь лишь одну причину для изменения, обладать минимальным интерфейсом, правильно реализовывать наследование и предотвращать каскадные изменения в коде при изменении требований

# Принципы и рекомендации ООАП

- GRASP – General Responsibility Assignment Software Patterns (Общие шаблоны распределения обязанностей в программных системах)
  - High Cohesion
    - Распределение обязанностей, поддерживающее высокую степень сцепления
  - Low Coupling
    - Распределить обязанности таким образом, чтобы степень связанности оставалась низкой
  - Controller
    - Делегирование обязанностей по обработке системных сообщений другому классу

# Объектно-ориентированное программирование

- Объектно-ориентированное программирование – это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является **экземпляром** определенного **класса**, а классы образуют иерархию наследования
- Особенности:
  - ООП использует в качестве базовых элементов объекты, а не алгоритмы
  - каждый объект является экземпляром какого-либо определенного класса (играющего роль типа данных)
  - классы организованы иерархически

При ООП подходе: раскладываем программный код на составляющие, чтобы уменьшить его избыточность, и пишем новый код, адаптируя имеющийся программный код, а не изменяя его.



# Принципы и рекомендации ООАП

## ■ Принципы SOLID

- Single Responsibility Principle (Принцип единственной обязанности)
- Open/Closed Principle (Принцип открытости/закрытости)
- Liskov Substitution Principle (Принцип подстановки Лисков)
- Interface Segregation Principle (Принцип разделения интерфейсов)
- Dependency Inversion Principle (Принцип инверсии зависимостей)

# Принципы и рекомендации ООАП

## ■ Шаблоны GoF (Gang-of-Fou)

### □ Структурные паттерны

- Применяются при компоновке системы на основе классов и объектов (Adapter, Facade, Decorator, Proxy)

### □ Порождающие паттерны

- Предназначены для создания объектов, позволяя системе оставаться независимой как от самого процесса порождения, так и от типов порождаемых объектов (Factory Method, Abstract Factory,

### □ Паттерны поведения

- Описывают правильные способы организации взаимодействия между используемыми объектами (Template Method, Strategy)

# OOPS Concept

Objects

Class

Polymorphism

Encapsulation

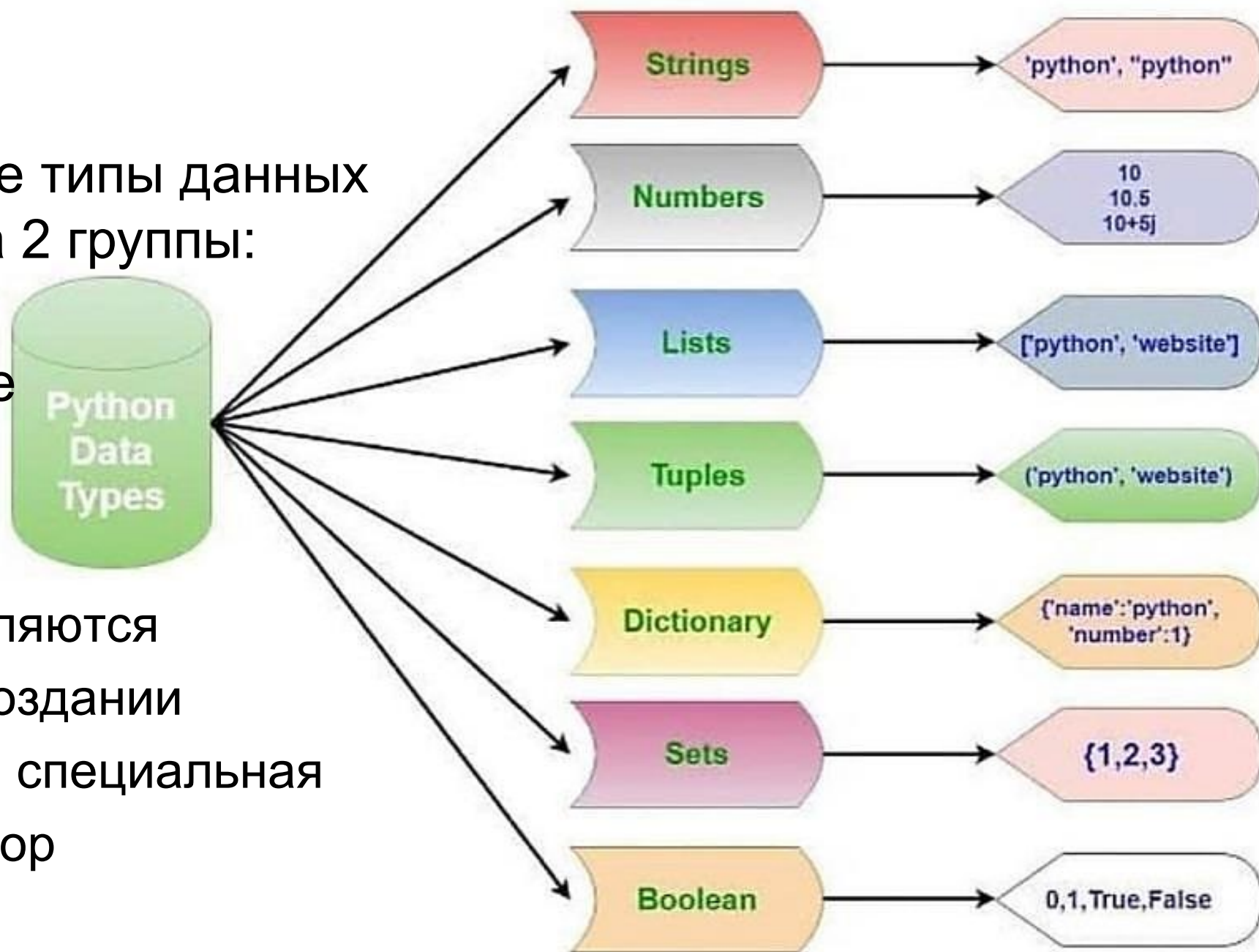
Inheritance

# Типы данных

В Python встроенные типы данных подразделяются на 2 группы:

- скалярные
- структурированные

Все типы в Python являются **объектами** – при создании объекта вызывается специальная функция – конструктор





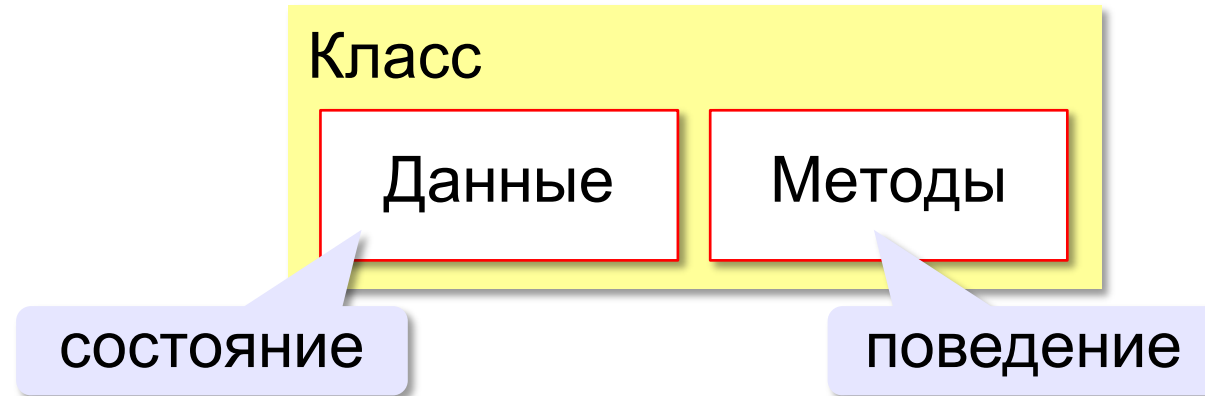
# ЛЕКЦИЯ 2. КЛАССЫ И ОБЪЕКТЫ

## *Учебные вопросы:*

1. Определение класса.
2. Атрибуты и методы класса.
3. Конструктор и деструктор класса.

# Класс

- Класс — это структура данных, объединяющая состояние (поля) и действия (методы и другие функции-члены)



- Класс предоставляет определения для динамически создаваемых **экземпляров** класса (**объектов класса**)
  - Классы поддерживают механизмы наследования и полиморфизма, которые позволяют создавать производные классы, расширяющие функциональные возможности базового класса

# Определение класса

- Класс определяется с помощью ключевого слова **class**
- Создается новое пространство имен и используется в качестве локальной области видимости при выполнении инструкций в теле определения

```
class ИмяКласса:  
    код_тела_класса
```

По окончании выполнения определения создается объект (объект-класс)

# Определение класса

Это объект и поэтому:

- его можно присвоить переменной,
- его можно скопировать,
- можно добавить к нему атрибут,
- его можно передать функции в качестве аргумента

Можно поместить определение класса в одну из ветвей инструкции *if* или в тело функции

```
if paramF > 0:
    class Person:
        lev = "1 уровень" # атрибут класса
        def display_info():
            print("Level: ", Person.lev)
else:
    ...
```



# Классы и объекты

- Создание экземпляра класса использует запись вызова функций
- Созданный объект связывают с переменной:  
**имя\_объекта = ИмяКласса([параметры])**

```
person1 = Person()  
person1.display_info()
```

Если у объекта **person1** нет своего собственного метода **display\_info()**, он ищется в классе **Person**

# Пример. Класс. Точка на плоскости

```
class Point:
```

```
    def __init__(self, x=0, y=0):  
        self.x = x  
        self.y = y
```

Конструктор

```
    def distance_from_origin(self):  
        return math.hypot(self.x, self.y)
```

Новый метод

```
    def __eq__(self, other):  
        return self.x == other.x and self.y == other.y
```

```
    def __str__(self):  
        return "({0.x!r}, {0.y!r})".format(self)
```

Стандартные (встроенные) методы

# Пример. Стек на базе списка

```
class Stack:
    def __init__(self):
        """Инициализация стека"""
        self._stack = []

    def top(self):
        """Возвратить вершину стека (не удаляя элемент)"""
        return self._stack[-1] # -1 индекс последнего элемента

    def pop(self):
        """Снять со стека элемент"""
        return self._stack.pop()

    def push(self, x):
        """Поместить элемент на стек"""
        self._stack.append(x)

    def __len__(self):
        """Количество элементов в стеке"""
        return len(self._stack)

    def __str__(self):
        """Представление в виде строки"""
        return " ; ".join(["%s" % e for e in self._stack])
```

Список

Метод списка

Метод списка

# Атрибуты и методы

```
class Staff601:
    course = '6.01'
    building = 34
    room = 501

    def salutation(self):
        return self.role + ' ' + self.name
```

```
pat = Staff601()
print("pat.course", pat.course)
```

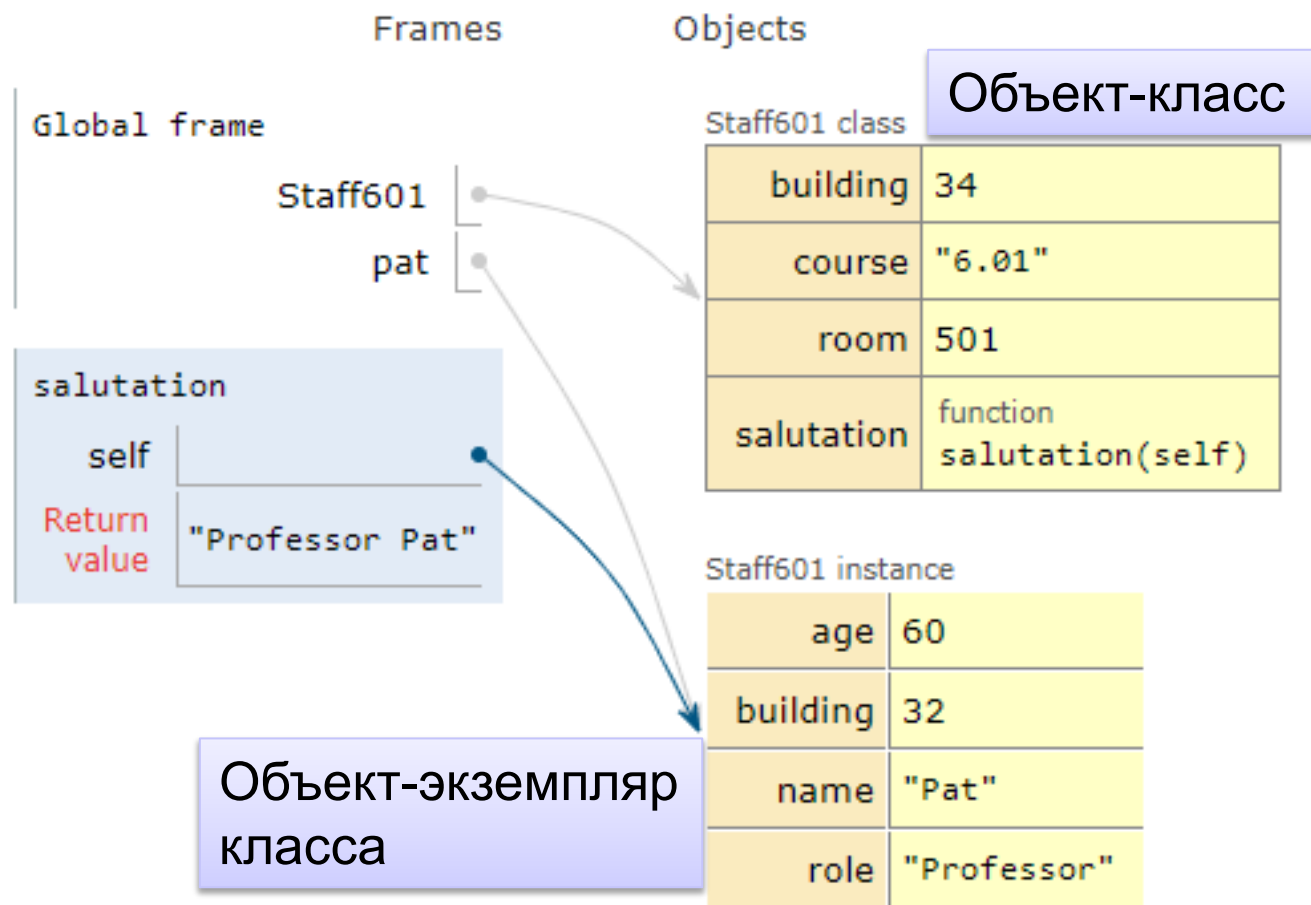
```
pat.name = 'Pat'
pat.age = 60
pat.role = 'Professor'
```

```
print("pat.building", pat.building)
pat.building = 32
print("pat.building", pat.building)
```

```
print("pat.salutation()", pat.salutation())
print("Staff601.salutation(pat)", Staff601.salutation(pat))
```

Print output (drag lower right corner to resize)

```
pat.course 6.01
pat.building 34
pat.building 32
```



# Атрибуты и методы

```
class Staff601:
    course = '6.01'
    building = 34
    room = 501

    def salutation(self):
        return self.role + ' ' + self.name
```

```
pat = Staff601()
print("pat.course", pat.course)
```

```
pat.name = 'Pat'
pat.age = 60
pat.role = 'Professor'
```

```
print("pat.building", pat.building)
pat.building = 32
print("pat.building", pat.building)
```

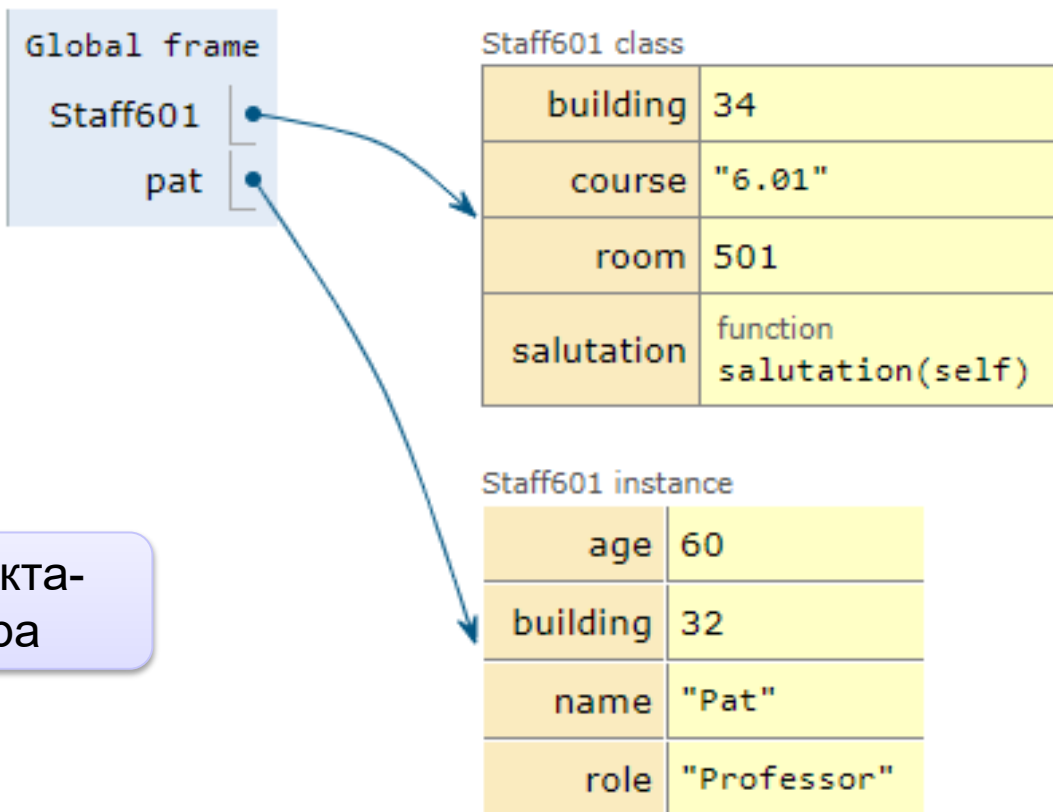
```
print("pat.salutation()", pat.salutation())
print("Staff601.salutation(pat)", Staff601.salutation(pat))
```

Print output (drag lower right corner to resize)

```
pat.course 6.01
pat.building 34
pat.building 32
pat.salutation() Professor Pat
Staff601.salutation(pat) Professor Pat
```

Frames

Objects



- Атрибуты хранятся в специальном словаре и к нему можно обратиться по имени `__dict__`:

```
Nclass.__dict__  
ob.__dict__
```

- Если обратиться к атрибуту, которого не существует, то будет возбуждено исключение **AttributeError**

- `hasattr(obj, attr_name)` - проверить наличие атрибута `attr_name` в объекте `obj`.
  - Если атрибут присутствует, то функция возвращает `True`, иначе `False`.
- `getattr(obj, attr_name[, default_value])` - получить значение атрибута `attr_name` в объекте `obj`.
  - Если атрибут не был найден, то будет возбуждено исключение `AttributeError`. Можно указать значение по умолчанию `default_value`, которое будет возвращено, если атрибута не существует.
- `setattr(obj, attr_name, value)` - изменить значение атрибута `attr_name` на `value`. Если атрибут не существовал, то он будет создан.

# Конструктор

- Конструктором класса называют метод, который автоматически вызывается при создании объектов
- По умолчанию создается автоматически

```
class Person:  
    name = "Иван"  
  
p = Person()  
print(p.name) # Иван
```

ВЫЗОВ  
конструктора



# Конструктор

- В Python роль конструктора играет метод `__init__()`
- Необходимость конструкторов связана с тем, что часто объекты должны иметь собственные свойства сразу
- Конструктор класса не позволит создать объект без обязательных полей
- Первый его параметр – **self** – ссылка на сам только что созданный объект, остальные параметры – атрибуты объекта

ВЫЗОВ  
конструктора

```
class Person:  
    def __init__(self, name):  
        self.name = name  
  
person1 = Person("Иван")
```

# Конструктор

- В Python роль конструктора играет метод `__init__()`
- Можно создать объединенное поле

```
class Person:
    def __init__(self, first_name, last_name):
        self.full_name = (first_name, last_name)

# создание объектов класса
person3 = Person("Петр", "Иванов")

print(*person3.full_name)
```

ВЫЗОВ  
конструктора

# Конструктор

- В Python роль конструктора играет метод `__init__()`
- Если надо допустить создание объекта, даже если никакие данные в конструктор не передаются, то в таком случае параметрам конструктора класса задаются *значения по умолчанию*
- Первый его параметр – **self** – ссылка на сам только что созданный объект, остальные параметры – атрибуты объекта

ВЫЗОВ  
конструктора

```
class Person:
    def __init__(self, name="Иван"):
        self.name = name

person1 = Person("Петр")
person2 = Person()
```

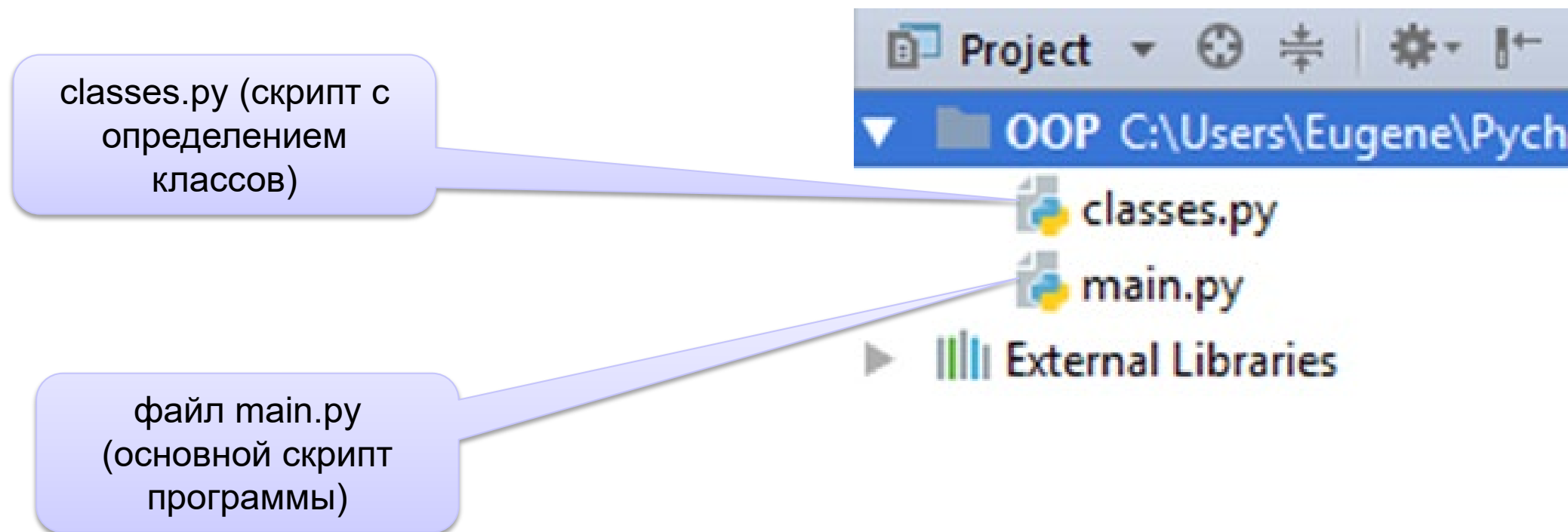
# Деструктор

- Деструктор вызывается, когда объект уничтожается
- В языке программирования Python объект уничтожается, когда:
  - исчезают все связанные с ним переменные,
  - им присваивается другое значение, в результате чего связь со старым объектом теряется,
  - он удаляется с помощью команды **del**

```
class Person:  
    # конструктор  
    def __init__(self, name):  
        self.name = name  
    # деструктор  
    def __del__(self):  
        print(self.name, "удален из памяти")
```

# Определение классов в отдельных модулях

- Как правило, классы размещаются в отдельных модулях и затем импортируются в основной скрипт программы



# Перегрузка функций посредством сигнатур вызова?

- Поскольку в Python отсутствуют объявления типов, эта концепция в действительности неприменима – полиморфизм в языке Python основан на интерфейсах объектов, а не на типах

Инструкция `def` просто присваивает объект некоторому имени в области видимости класса и поэтому будет сохранено только последнее определение метода

```
class C:  
    def meth(self, x):  
        ...  
    def meth(self, x, y, z):  
        ...
```