# 2025-10-01

# Программирование (Pyton) — оргвопросы

## Трек 1: Хочу выучить Питон

1. Посещение занятий
2. Выполнение и сдача лабораторных работ

## Трек 2: Хочу сдать дисциплину, при этом Питон знаю или выучу самостоятельно

1. Работа с преподавателем в режиме консультации
2. Использование Python как основного языка разработки программного продукта (MVP), которым предполагается отчитаться по дисциплине
3. Должен быть разработан проект с описанием архитектуры, API (если предусмотрено), желательно в какой-либо нотации, например, UML или BPMN
4. Должны быть написаны автотесты
5. Проект должен быть подготовлен к развертыванию через средства CI/CD

# Объективно-ориентированное программирование в Python

Объектно-ориентированная парадигма представляет собой методологию разработки, в которой семантические важные элементы (сущности) представляются в виде объектов со своим набором свойств и каким-либо образом между собой взаимодействуют.

Каждый объект принадлежит к какому-либо **классу**.

**Класс** — модель объекта определенного типа, которая описывает его внутреннюю структуру и то, какие методы и алгоритмы доступны при взаимодействии объекта этого типа с другими объектами (как такого же типа, так и иных типов).

Базовым классом в Python является класс `object`. От него наследуются все классы (в том числе происходит неявно наследование, если вы создаете свой класс и явно не наследуете его ни от чего).

Чтобы посмотреть набор свойств и методов *любого* класса, можно вызвать метод `dir()`

```
dir(object)

'''
[out]:
```

```
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__']
'''
```

```
class Pet: # в отличие от имен переменных, функций/методов и т.п., имена
классов в Питоне подчиняются CamelCase
    def __init__(self, tail:str='black', ears:tuple[str, str]=('black',
'black')) -> None:
        self.tail = tail
        self.ears = ears

class Cat(Pet):
    def __init__(self, tail:str, ears:tuple[str, str],
whiskers:str='white') -> None:
        super().__init__()
        self.whiskers = whiskers
```

Класс могут наследоваться друг от друга, при этом **дочерний класс** получает доступ к *публичной* структуре и методам **родительского класса**, при этом может добавлять в свою структуру новые элементы и методы.

Объединенные такими связями классы образуют **иерархию классов** в рамках отдельно взятого программного продукта.

## Основные принципы ООП в Python

Рассмотрим основные принципы ООП в экосистеме Pyhon:

> **Об абстракции**
>
> Если нужны абстрактные классы, то они вынесены в отдельный модуль стандартной библиотеки — `abc` (Abstract Base Classes).
> Документация: abc — Abstract Base Classes — Python 3.13.7 documentation

**Полиморфизм** — способность метода обрабатывать данные разных типов.

```
def divider(a:int|float|str, b:int|float|str) -> float:
    if isinstance(a, str):
        a = float(a)
    if isinstance(b, str):
        b = float(b)
    return a / b
```

**Инкапсуляция** — возможность изолирования/сокрытия конкретных элементов структуры и/или методов класса от внешних воздействий. Это разделяет методы и свойства на публично доступные и скрытые.

Явные модификаторы защиты ( `protected` , `private` , `public` ) в Python отсутствуют. Для имитации их поведения предназначены специальные правила именования свойств и методов.

- `_single_leading_underscore` : указание на то, что свойство или метод должен использоваться *только внутри* класса или модуля. `from M import *` такие объекты не импортирует, а многие среды разработки скрывают такие свойства и методы при подсказках.

> ⚠ **Warning**
>
> Использование `*` при импорте само по себе является плохой практикой

- `single_trailing_underscore_` : конвенциональные имена для атрибутов и методов, чьи названия совпадают со встроенными объектами языка (например `class_` вместо `class` в `Tkinter.Toplevel(master, class_='ClassName')` )
- `__double_leading_underscore` : name mangling — внутри класса такие атрибуты и методы в памяти всегда записываются как `_ИмяКласса__Метод` , например: `class FooBar` , метод `__tea()` , в памяти и во всех ссылках он будет зафиксирован как `_FooBar__tea` ; к нему нельзя обратиться как к `__tea` , только как к `_FooBar__tea`
- `__double_leading_and_trailing_underscore__` : «dunder»-методы, «магические» методы, объекты и атрибуты. **Никогда** не нужно изобретать свои магические методы, только использовать уже определенные в языке. К ним относятся, например, `__init__` , `__import__` , `__file__` , `__name__` , `__str__` , `__repr__` ; все методы, определяющие поведение операторов сравнения, арифметических операторов, оператора присваивания и т.п.

```
class Foo:
    def __init__(self):
        self.public_var = 'Публичная (public) переменная'
        self._protected_var = 'Защищенная (protected) переменная'
        self.__private_var = 'Приватная/частная (private) переменная'

'''
Вызов метода dir()

['_Foo__private_var', '__class__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
```

```
'__subclasshook__', '__weakref__', '_protected_var', 'public_var']
'''

a = Foo()
a._protected_var = 'Типа защищенная переменная'
print(a._protected_var) # 'Типа защищенная переменная'
```

В современном Питоне помимо соглашений об именовании с подчеркиваниями для создания «защищенных» свойств активно используются декораторы `@property`, `@имя_свойства.setter` и `@имя_свойства.deleter`

```
class Boo:
    def __init__(self, price:float) -> None:
        self._price = price

    @property
    def price(self):
        print('Getting price')
        return self._price

    @price.setter
    def price(self, new_price):
        print(f'Setting price, {new_price=}')
        self._price = new_price

    @price.deleter
    def price(self):
        print('Deleting price')
        del self._price
```

## Сценарии использования своего `__new__`

Когда нам необходимо изменить «реальный» конструктор, то необходимо использовать переопределение метода `__new__`:

```
class MyClass:
    def __new__(cls) -> MyClass:
        print('Создаем экземпляр класса в памяти')
        return super(MyClass, cls).__new__(cls)
    def __init__(self) -> None:
        print('Инициализируем')

a = MyClass()

'''
[out]:
Создаем экземпляр класса в памяти
```

```
Инициализируем
'''
```

Когда имеет смысл применять свой `__new__`:

1. **Создание дочерних классов от неизменяемых (иммутабельных) типов**, например, от `tuple`, т.к. их настройка должна происходить на этапе создания экземпляра, после того, как экземпляр иммутабельного типа создан, менять его по определению не получится
2. **Реализация паттерна Singleton**: т.к. в таком паттерне необходима гарантия единственного экземпляра, то и контроль этого должен производиться на этапе создания этого экземпляра.

## Датаклассы, валидируемые модели данные и type hinting (аннотации типов)

Аннотации типов являются **строгой рекомендацией** для всех новых проектов.

PEP 484 – Type Hints — официальное предложение по улучшению (PEP), в котором представлены и описаны аннотации типов:

```python
# Python 3.9+
from typing import List, Dict, Tuple, Union

def process(data: List[Union[int, str]]) -> Tuple[str, Dict[str, int]]:
    ...

# Python 3.10+ — рекомендуемый современный стиль
def process(data: list[int | str]) -> tuple[str, dict[str, int]]:
    ...
```

Все это направлено на более структурный подход и более жесткие требования к типизации в Питоне, хотя на уровне интерпретатора все еще типизация была и остается динамической.

Такой подход окупает себя при разработке с проектированием структур данных и их валидацией.

Для этого есть ряд библиотек и модулей (как в стандартной поставке Питона, так и сторонние):

- dataclasses — Data Classes — Python 3.13.7 documentation
- attrs 25.3.0 documentation
- Welcome to Pydantic - Pydantic
- marshmallow 4.0.1 documentation

# Лабораторная работа №2: Основы ООП в Python

**Общие требования ко всем программам:**

1. **Чистота кода:** Код должен быть читаемым: использовать осмысленные имена переменных и методов, добавлять комментарии, разделять логические блоки. Имена классов должны подчиняться стилю `CamelCase`.
2. **Инкапсуляция:** Использовать соглашения об именовании для защиты данных: `_protected_var` для указания на защищенный атрибут и `__private_var` для сокрытия имени атрибута (name mangling).
3. **Строковое представление объектов:** Все классы должны иметь реализованные "магические" методы `__str__()` (для человеко-читаемого представления) и `__repr__()` (для официального, часто позволяющего воссоздать объект). Будьте готовы пояснить разницу между ними.
4. **Обработка некорректного ввода:** Программы должны устойчиво работать при любом пользовательском вводе.

## Задание 1: Таск-трекер в консоли

**Требования к реализации:**

1. Реализуйте класс `Task` для представления одной задачи (атрибуты: описание, статус выполнения, категория).
2. Реализуйте класс `TaskTracker`, который будет хранить коллекцию задач и управлять ими.
3. Данные должны сохраняться в JSON-файл при завершении программы и загружаться из него при запуске (используйте модуль `json`).
4. Реализуйте функционал:
   - Добавление новой задачи с произвольным описанием и категорией.
   - Отметка задачи как выполненной.
   - Вывод списка всех задач.
   - *(Бонус)* Поиск по задачам и вывод всех задач в указанной категории.

**Пример интерфейса:**

```
- [x] Задание выполнено #pstu
- [ ] Еще задача #work
- [ ] И еще задача #pstu
```

## Задание 2: Трекер бюджета в консоли

**Требования к реализации:**

1. Реализуйте класс `Transaction` для представления одной финансовой операции (атрибуты: описание, сумма, тип [доход/расход], категория).
2. Реализуйте класс `BudgetTracker`, который будет хранить историю операций и баланс.
3. Данные должны сохраняться в JSON-файл и загружаться из него.
4. Реализуйте функционал:
    - Добавление новой операции с указанием суммы и категории.
    - Расчет и отображение текущего баланса.
    - *(Бонус)* Установка и проверка месячных лимитов для категорий расходов.

## Задание 3: Очередь и стек

**Требования к реализации:**

1. Реализуйте класс `Queue` (First In, First Out, FIFO) со следующими методами:
    - `enqueue(item)` — добавление элемента в конец очереди.
    - `dequeue()` — удаление и возврат элемента из начала очереди.
    - `peek()` — просмотр первого элемента без его удаления.
2. Реализуйте класс `Stack` (Last In, First Out, LIFO) со следующими методами:
    - `push(item)` — добавление элемента на вершину стека.
    - `pop()` — удаление и возврат элемента с вершины стека.
    - `peek()` — просмотр верхнего элемента без его удаления.

## Задание 4: Модель шестизвенного манипулятора

**Требования к реализации:**

1. Создайте иерархию классов, описывающих разные виды сервоприводов, с **минимум 3 уровнями** (например, `Двигатель` -> `ВращательныйДвигатель` -> `СинхронныйСервопривод`).
2. Базовый класс должен определять общие атрибуты (например, `угол_поворота`, `скорость_вращения`, `ускорение`), а классы-потомки — добавлять специфичные.
3. Реализуйте строковое представление классов с помощью методов `__str__()` и `__repr__()`.
4. Перегрузите операторы сравнения (магические методы `__eq__`, `__lt__` и др.), чтобы позволить сравнивать экземпляры классов по одному из параметров (например, по мощности).
5. Реализуйте упрощенную модель шестизвенного манипулятора, состоящего из нескольких сервоприводов.
6. Реализуйте функцию перемещения манипулятора в пространстве через перегрузку арифметических операторов (например, `__add__` для сложения с вектором перемещения).

## Задание 5: Иерархия классов с абстрактными методами

**Требования к реализации:**

1. Используя модуль `abc`, создайте абстрактный базовый класс `Vehicle` с абстрактными методами `get_max_speed` (максимальная скорость) и `get_vehicle_type` (тип транспортного средства).
2. Создайте абстрактный подкласс `RoadVehicle`, который добавляет абстрактный метод `get_engine_type`.
3. Создайте конкретные классы `Car` и `Bicycle`, реализующие все унаследованные абстрактные методы. Для `Car` тип двигателя может быть "бензиновый" или "электрический", а для `Bicycle` — "мускульная сила".

**Пример:**

```
car = Car("седан", "электрический")
print(car.get_vehicle_type()) # Должно вывести: "Автомобиль"
print(car.get_engine_type()) # Должно вывести: "электрический"
```

## Задание 6: Метапрограммирование

**Требования к реализации:**

1. Создайте базовый класс `Plugin`, в котором переопределен метод `__init_subclass__`. При создании нового класса-потомка он должен автоматически добавляться в реестр плагинов (например, в словарь `PluginRegistry`).
2. Каждый плагин должен иметь уникальный атрибут `name`.
3. Реализуйте метод `execute` в базовом классе, который выбрасывает исключение `NotImplementedError`.
4. Создайте 2-3 конкретных плагина (например, `UpperCasePlugin`, `ReversePlugin`), которые реализуют метод `execute` (например, преобразуют строку).

**Пример:**

```
print(PluginRegistry) # Должно вывести: {'upper': <class
'__main__.UpperCasePlugin'>, 'reverse': <class '__main__.ReversePlugin'>}
plugin = PluginRegistry["upper"]()
print(plugin.execute("hello")) # Должно вывести: "HELLO"
```