

Хорошо слышно и видно?

Ставьте +, если все хорошо
Пишите, если есть проблемы

Вебинар
Структуры данных и функции



Y-LAB
UNIVERSITY

Преподаватель



Герман Гольцов

- Более 4 лет являюсь разработчиком на Python
- Создаю микросервисы, провожу реview кода, занимаюсь рефакторингом и оптимизацией
- Интересуюсь разработкой и реализацией архитектуры высоконагруженных веб-сервисов

Цели вебинара

После занятия вы сможете:

1

Уверенно работать с основными структурами данных



2

Разрабатывать функции и работать с аргументами

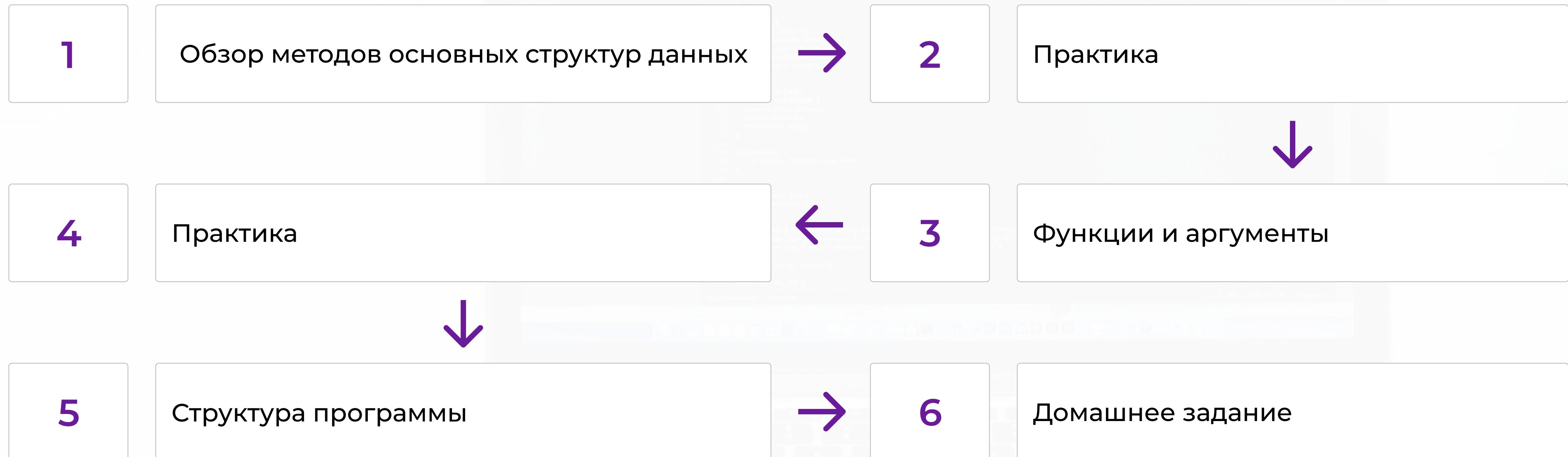


3

Правильно организовывать структуру программы



Маршрут вебинара



Правила вебинара

- Активно участвуйте
- Задавайте вопросы в чат
- Оффтоп вопросы можем обсуждать после занятия в общем чате

- Список (list)
- Кортеж (tuple)
- Множество (set)
- Словарь (dict)
- Стек (collections.deque)
- Очередь (collections.deque)
- Именованный кортеж (typing.NamedTuple)
- Типизированные массивы (array.array)
- ...

Списки

Методы работы. Индексы



```
my_list = ["Привет", 5, True]
```

```
my_list[0] # Привет
```

```
my_list[1] # 5
```

```
my_list[2] # True
```

```
my_list[-1] # True
```

```
my_list[-2] # 5
```

```
my_list[-3] # Привет
```

```
my_list[4] # IndexError: list index out of range
```

Списки

Методы работы. Срезы

```
my_list = ["Привет", 5, True]

my_list[0:2]    # ['Привет', 5]
my_list[0:1]    # ['Привет']
my_list[0:0]    # []

my_list[1:]    # [5, True]
my_list[:2]    # ['Привет', 5]
my_list[:]     # ['Привет', 5, True]

my_list[:-2]   # ['Привет']
my_list[-2:]   # [5, True]

my_list[::-2]  # ['Привет', True]
```

Списки

Методы работы. Вычисление длины списка



```
my_list = ["Привет", 5, True]
```

```
len(my_list) # 3
```

```
if my_list:  
    print("Список не пустой")  
else:  
    print("Список пустой")
```

```
if len(my_list) > 2:  
    print("В списке более двух элементов")  
else:  
    print("В списке не более двух элементов")
```

Списки

Методы работы. Добавление элемента в список



```
my_list = ["Привет", 5, True]
```

```
my_list.append("Новый элемент 1")
```

```
my_list # ['Привет', 5, True, 'Новый элемент 1']
```

```
my_list.insert(2, "Новый элемент 2")
```

```
my_list # ['Привет', 5, 'Новый элемент 2', True, 'Новый элемент 1']
```

```
my_list[3] = "Новый элемент 3"
```

```
my_list
```

```
# ['Привет', 5, 'Новый элемент 2', 'Новый элемент 3', 'Новый элемент 1']
```

Списки

Методы работы. Удаление элемента



```
my_list = ["Привет", 5, True]
```

```
my_list.pop() # True
```

```
my_list # ['Привет', 5]
```

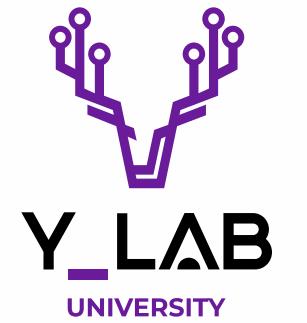
```
my_list.remove("Привет")
```

```
my_list # [5]
```

```
del my_list[0] # []
```

Списки

Зачем применять списки?



Списки подходят, когда нужно создать расширяемую упорядоченную структуру данных.
«Под капотом» списков находится динамический массив с ссылками на элементы.

Скорость операций над списком:

Вставка: $O(n)$.

Вставка в конец: в основном $O(1)$

Получение элемента: $O(1)$.

Удаление элемента: $O(n)$.

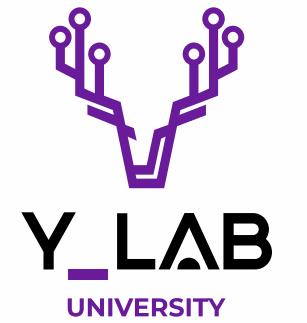
Проход: $O(n)$.

Получение длины: $O(1)$.

Подробнее об устройстве списка можно [почитать тут](#).

Кортежи

Методы работы. Индексы



```
my_tuple = ("Привет", 5, True)
```

```
my_tuple[0] # Привет
```

```
my_tuple[1] # 5
```

```
my_tuple[2] # True
```

```
my_tuple[-1] # True
```

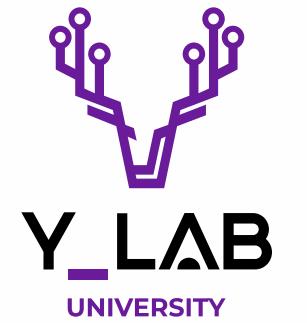
```
my_tuple[-2] # 5
```

```
my_tuple[-3] # Привет
```

```
my_tuple[4] # IndexError: tuple index out of range
```

Кортежи

Методы работы. Срезы



```
my_tuple = ("Привет", 5, True)

my_tuple[0:2]    # ('Привет', 5)
my_tuple[0:1]    # ('Привет',)
my_tuple[0:0]    # ()

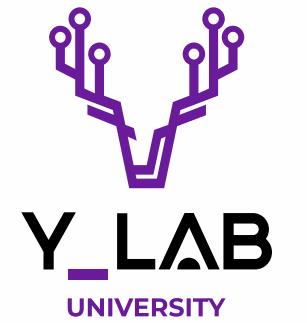
my_tuple[1:]    # (5, True)
my_tuple[:2]    # ('Привет', 5)
my_tuple[:]     # ('Привет', 5, True)

my_tuple[:-2]   # ('Привет',)
my_tuple[-2:]   # (5, True)

my_tuple[::-1]  # (True, 5, 'Привет')
```

Кортежи

Методы работы. Вычисление длины кортежа



```
my_tuple = ("Привет", 5, True)
```

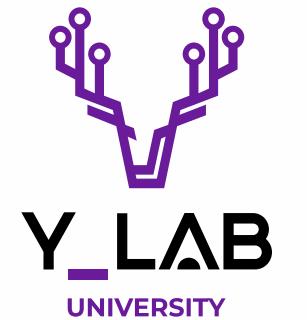
```
len(my_tuple) # 3
```

```
if my_tuple:  
    print("Кортеж не пустой")  
else:  
    print("Кортеж пустой")
```

```
if len(my_tuple) > 2:  
    print("В кортеже более двух элементов")  
else:  
    print("В кортеже не более двух элементов")
```

Кортежи

Зачем применять кортежи, когда есть списки?



Кортежи подходят, когда нужно создать статичную упорядоченную структуру данных.
«Под капотом» кортежи напоминают списки, но не выделяют пустые ячейки памяти для быстрого добавления элементов.

Скорость операций над кортежем:

Получение элемента: $O(1)$.

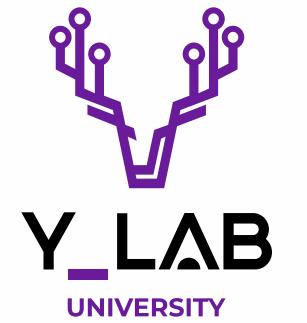
Проход: $O(n)$.

Получение длины: $O(1)$.

Подробнее об устройстве кортежи можно [почитать тут](#).

Множества

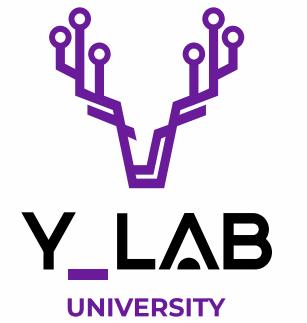
Методы работы. Добавление элементов



```
my_set = set()  
my_set.add("Новый элемент")  
my_set.add(5)  
my_set.add(True)  
my_set # {True, 'Новый элемент', 5}  
  
my_set.update([5, 5, 7]) # {True, 'Новый элемент', 5, 7}  
my_set.update((8, 9, 10)) # {True, 5, 7, 8, 9, 10, 'Новый элемент'}
```

Множества

Методы работы. Удаление элементов



```
my_set = {1, 2, 3}
```

```
my_set.remove(2)  
# {1, 3}
```

```
my_set.discard(3)  
# {1}
```

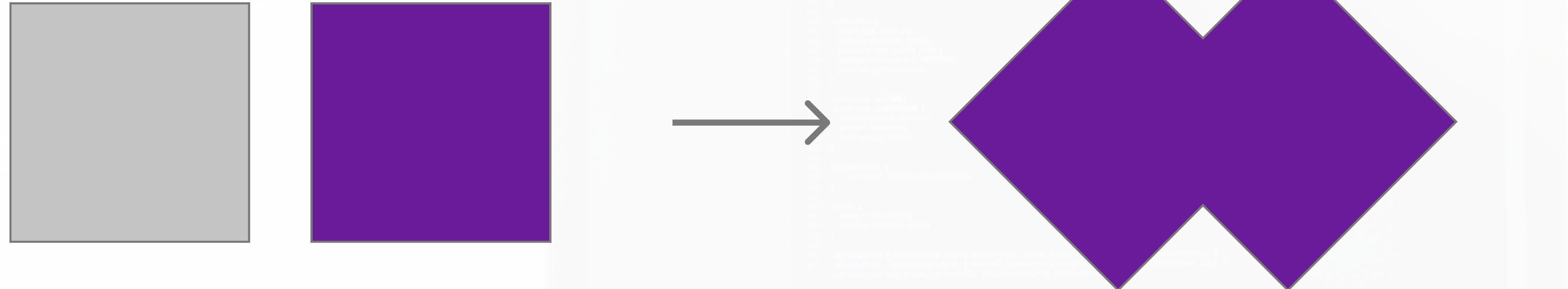
```
my_set = {1, 2, 3}
```

```
my_set.remove(123)  
# KeyError: 123
```

```
my_set.discard(123)  
# {1, 2, 3}
```

Множества

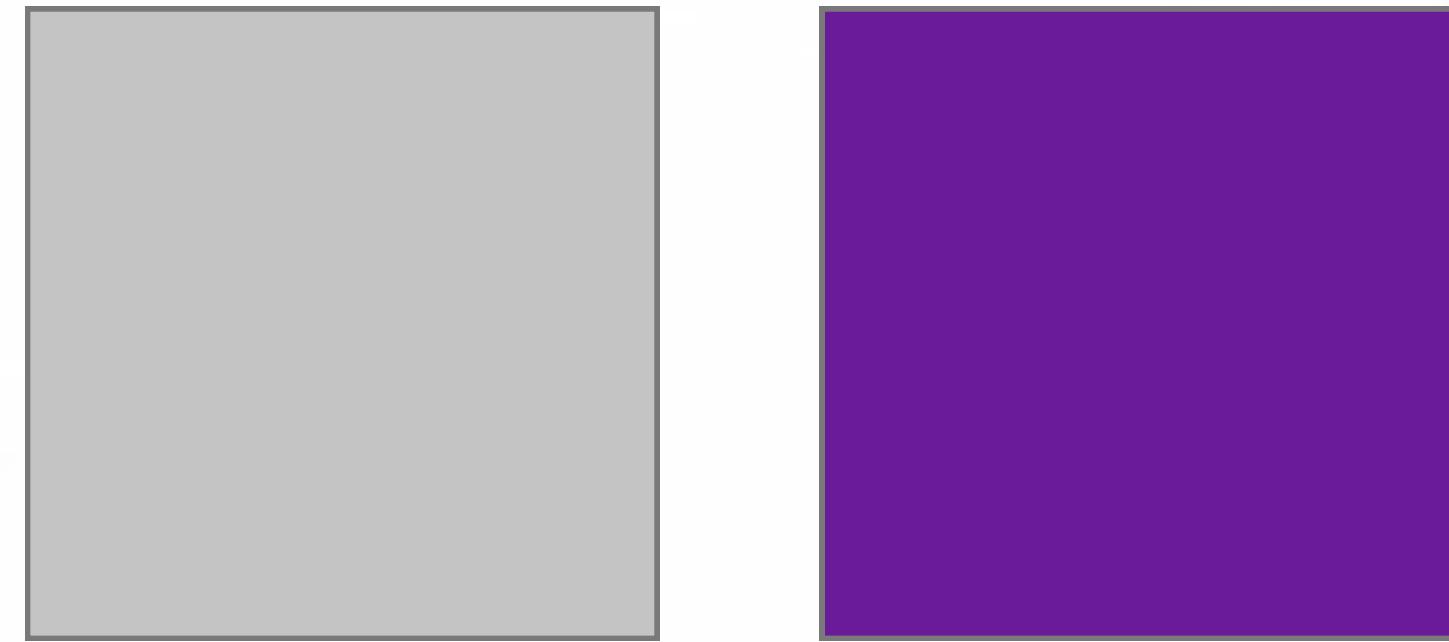
Методы работы. Объединение множеств



```
my_set_1 = {1, 2, 3}  
my_set_2 = {2, 3, 4}  
my_set_1.union(my_set_2) # {1, 2, 3, 4}  
my_set_1 | my_set_2 # {1, 2, 3, 4}
```

Множества

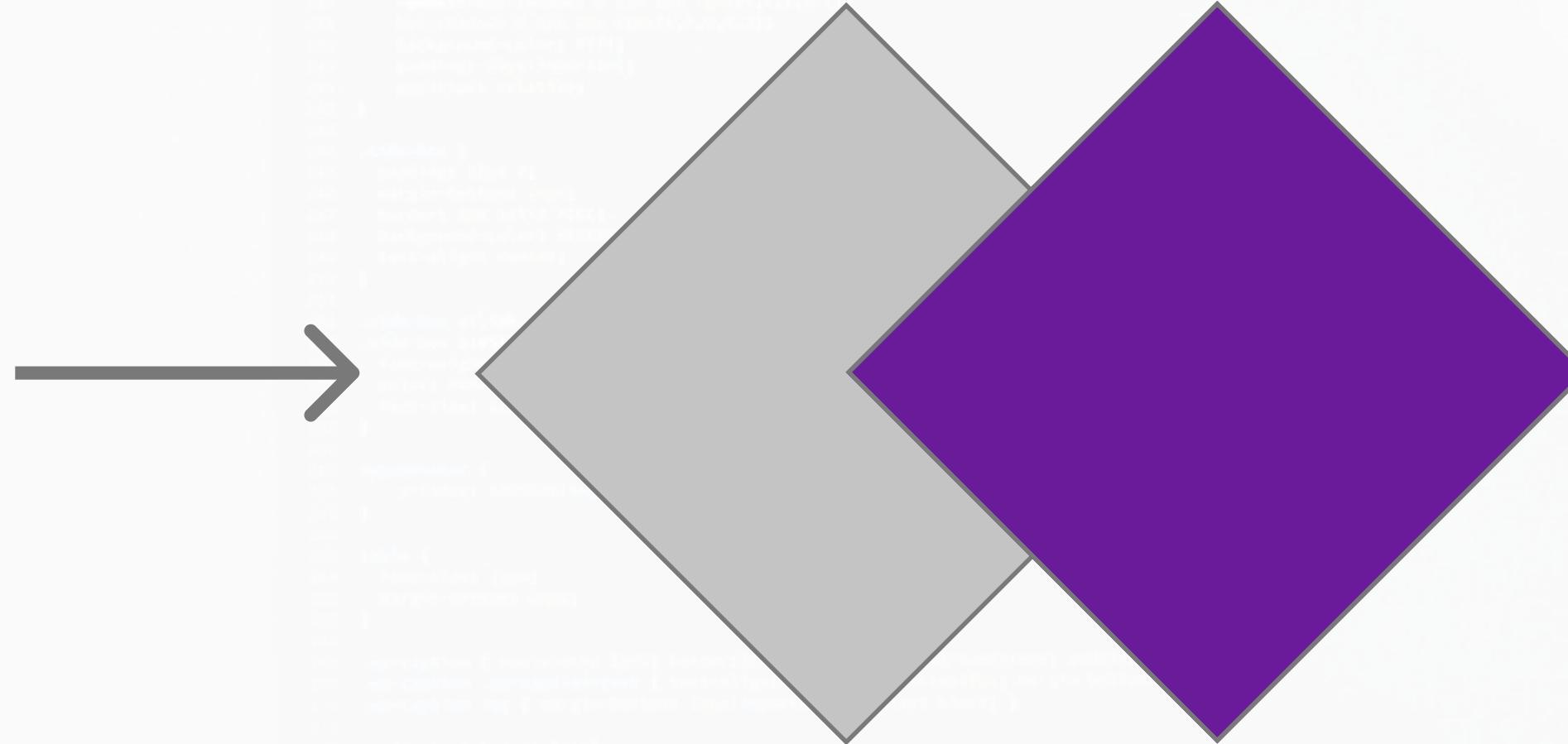
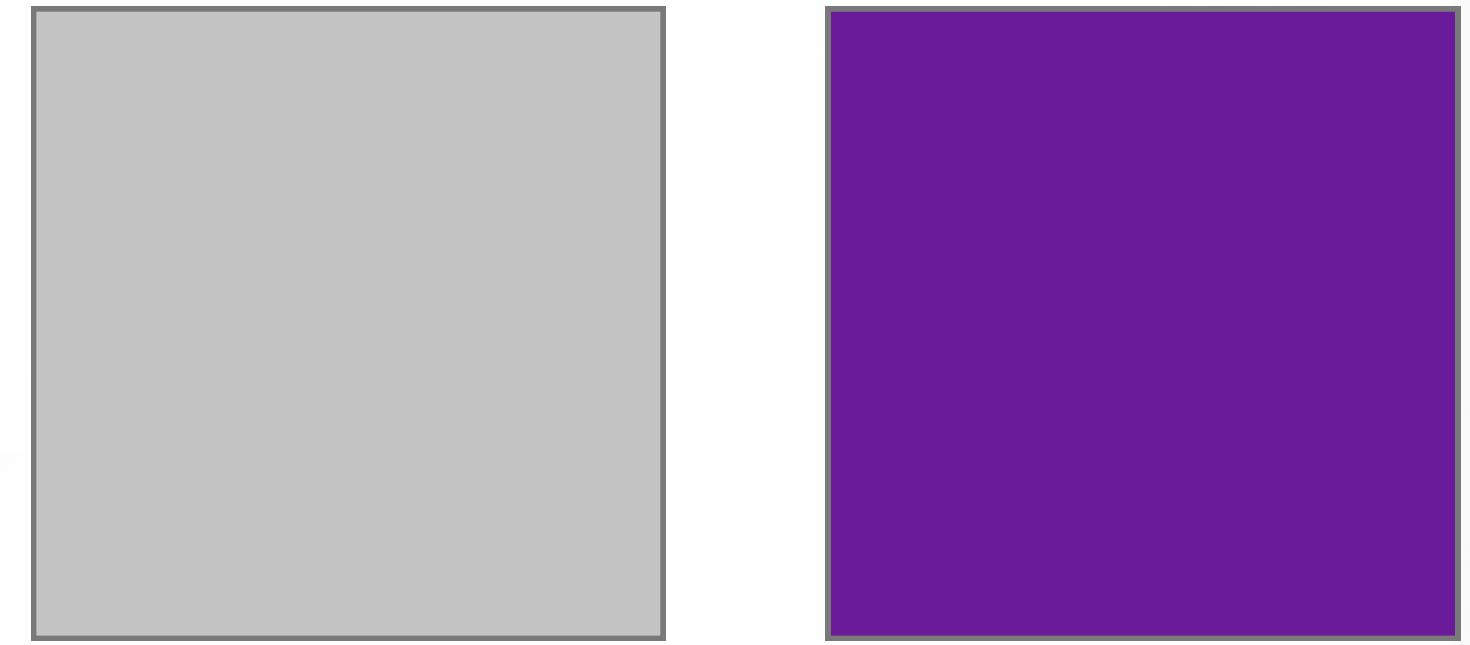
Методы работы. Пересечение множеств



```
my_set_1 = {1, 2, 3}  
my_set_2 = {2, 3, 4}  
my_set_1.intersection(my_set_2)    # {2, 3}  
my_set_1 & my_set_2    # {2, 3}
```

Множества

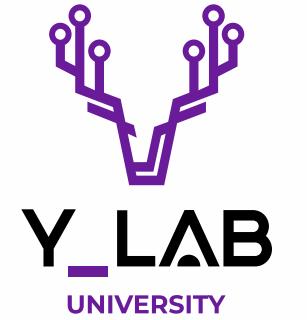
Методы работы. Вычитание множеств



```
my_set_1 = {1, 2, 3}  
my_set_2 = {2, 3, 4}  
my_set_1.difference(my_set_2) # {1}  
my_set_1 - my_set_2 # {1}
```

Множества

Зачем применять множества, когда есть списки?



Множества подходят, когда нужно создать расширяемую структуру уникальных данных с быстрой проверкой наличия/отсутствия элементов в ней.

«Под капотом» множеств находится хеш-таблица.

Скорость операций над множеством:

Проверить наличие элемента в множестве: $O(1)$.

Отличие множества A от B: $O(\text{длина A})$.

Пересечение множеств A и B: $O(\text{минимальная длина A или B})$.

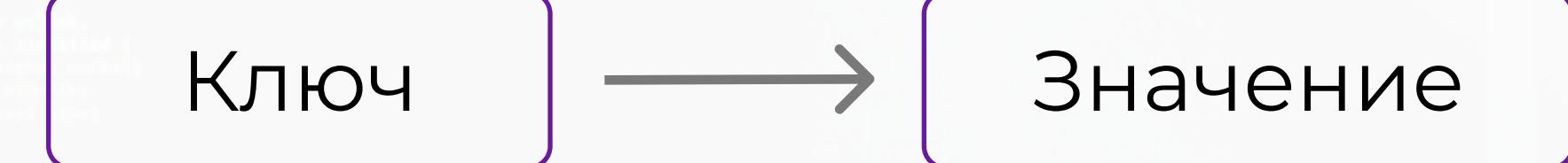
Объединение множеств A и B: $O(N)$, где N это длина (A) + длина (B).

Подробнее об устройстве множества можно [почитать тут](#).

Словари

Методы работы. Добавление элементов

```
my_dict = {  
    "ключ_1": "Привет",  
    "ключ_2": 5,  
}
```

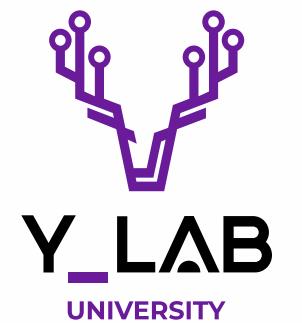


```
my_dict["new"] = "Новый элемент"  
# {'ключ_1': 'Привет', 'ключ_2': 5, 'new': 'Новый элемент'}
```

```
my_dict.update({"my_key": "Value", "new": "Updated value"})  
# {'ключ_1': 'Привет', 'ключ_2': 5, 'new': 'Updated value', 'my_key': 'Value'}
```

Словари

Методы работы. Получение значений



```
my_dict = {  
    "ключ_1": "Привет",  
    "ключ_2": 5,  
}
```

```
my_dict.get("ключ_1")    # Привет  
my_dict.get("ключ_10")   # None  
my_dict.get("ключ_10", "Значение по умолчанию")  
# Значение по умолчанию
```

```
my_dict["ключ_1"]      # 'Привет'  
my_dict["ключ_10"]     # KeyError: 'ключ_10'
```

Словари

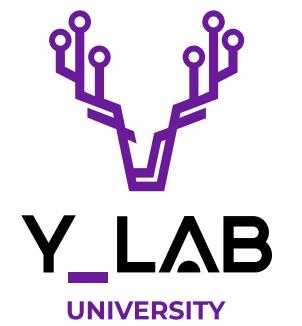
Методы работы. Получение значений



```
my_dict = {  
    "ключ_1": "Привет",  
    "ключ_2": 5,  
}  
  
my_dict.items() # dict_items([('ключ_1', 'Привет'), ('ключ_2', 5), ('ключ_3', True)])  
  
for key, value in my_dict.items():  
    print(key, value)  
  
# ключ_1 Привет  
# ключ_2 5
```

Словари

Методы работы. Получение значений



```
my_dict = {  
    "ключ_1": "Привет",  
    "ключ_2": 5,  
}
```

```
my_dict.keys()
```

```
for key in my_dict.keys():  
    print(key)
```

```
# Ключ_1  
# Ключ_2
```

```
for key in my_dict:  
    print(key)  
  
# Ключ_1  
# Ключ_2
```

```
my_dict.values()
```

```
for value in my_dict.values():  
    print(value)
```

```
# Привет  
# 5
```

Словари

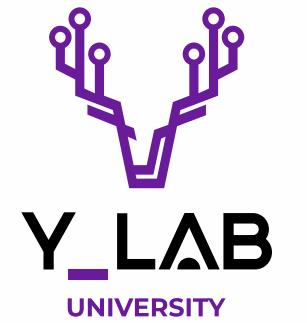
Методы работы. Удаление элементов



```
my_dict = {  
    "ключ_1": "Привет",  
    "ключ_2": 5,  
    "ключ_3": True  
}  
  
my_dict.popitem() # ('ключ_3', True)  
# {'ключ_1': 'Привет', 'ключ_2': 5}  
  
my_dict.pop("ключ_2") # 5  
# {'ключ_1': 'Привет'}  
  
del my_dict["ключ_1"]  
# {}  
  
my_dict.clear()
```

Словари

Зачем использовать словари?



Пожалуй, самый декларативный базовый тип.

Ключи словаря должны быть хешируемы и уникальны относительно друг друга.

«Под капотом» словарей находится хеш-таблица.

Скорость операций над словарями:

Получение элемента: $O(1)$.

Установка элемента: $O(1)$.

Удаление элемента: $O(1)$.

Проход по словарю: $O(n)$.

Подробнее об устройстве словаря можно [почитать тут](#).

- Объявление структур данных
- Методы работы со структурами данных

[Ссылка на практику](#)

ФУНКЦИЯ – это блок кода, который выполняет заданную последовательность действий.

ФУНКЦИИ являются объектами:

- их можно присваивать переменным
- хранить в структурах данных
- передавать другим функциям
- возвращать в качестве значений из функции

def – ключевое слово для объявления функции

Структура функции:

- название
- аргументы (опционально)
- тело
- возвращаемое значение (опционально)

def название (аргументы): } обязательно – двоеточие
тело функции

4 пробела

отсутствие символов
окончания строки

```
def my_function(text):  
    print('Привет из функции!')  
    return text.upper()  
  
print(  
    my_function('Это аргумент')  
)  
  
# Привет из функции!  
# ЭТО АРГУМЕНТ
```

Python

ФУНКЦИИ. Синтаксические особенности при вызове. Именованные аргументы.



```
def my_function(argument_1, argument_2, argument_3):  
    print(argument_1 + '!' + argument_2 + ' ' + argument_3 + '.')
```

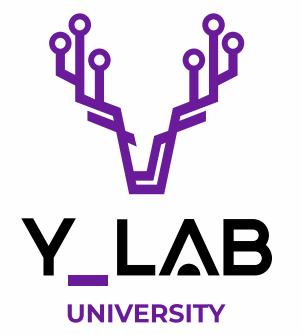
```
my_function(argument_1='Hello', argument_2='It\'s', argument_3='me')  
# Hello! It's me.
```

```
my_function(argument_1='Hello', argument_3='me', argument_2='It\'s')  
# Hello! It's me.
```

Python

ФУНКЦИИ. Синтаксические особенности при вызове.

Позиционные аргументы



```
def my_function(argument_1, argument_2, argument_3):  
    print(argument_1 + '!' + argument_2 + ' ' + argument_3 + '.')
```

```
my_function('Hello', 'It\'s', 'me')  
# Hello! It's me.
```

```
my_function('Hello', argument_3='me', argument_2='It\'s')  
# Hello! It's me.
```

```
my_function('It\'s', argument_1='Hello', argument_3='me')  
# TypeError: my_function() got multiple values for argument 'argument_1'
```

Python

ФУНКЦИИ. Синтаксические особенности при вызове.

Распаковка аргументов из перечисляемых объектов



```
def my_function(argument_1, argument_2, argument_3):
    print(argument_1, argument_2, argument_3)

list_arguments = [1, 2, 3]
tuple_arguments = (1, 2, 3)

my_function(*list_arguments)  # 1 2 3
my_function(*tuple_arguments) # 1 2 3

also_list = ['Еще можно так', *list_arguments, *tuple_arguments]
# ['Еще можно так', 1, 2, 3, 1, 2, 3]
```

Python

ФУНКЦИИ. Синтаксические особенности при вызове.

Распаковка аргументов из перечисляемых объектов



```
def my_function(argument_1, argument_2, argument_3):
    print(argument_1, argument_2, argument_3)

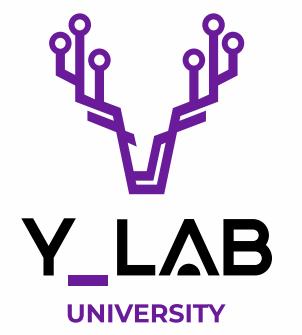
dict_arguments = {'argument_1': 1, 'argument_2': 2, 'argument_3': 3}

my_function(**dict_arguments)    # 1 2 3
my_function(*dict_arguments)    # argument_1 argument_2 argument_3
```

Python

Функции. Синтаксические особенности при определении.

Значения аргументов по умолчанию



```
def my_function(one=1, two=2, three=3):  
    print(f'{one + two + three = }')
```

```
def my_function_too(one, two=2, three=3):  
    print(f'{one + two + three = }')
```

```
my_function()  
# 1 + 2 + 3 = 6
```

```
my_function(2, 5)  
# 2 + 5 + 3 = 10
```

```
my_function_too(one=100)  
# one + two + three = 105
```

Python

Функции. Синтаксические особенности при определении.

Произвольные аргументы



- Символ `*` используется при распаковке итерируемых объектов.
- Функция принимает `*args` в виде кортежа, `**kwargs` – в виде словаря.
- Названия `args` и `kwargs` – просто соглашение. Можно давать любое наименование.
- Распаковка передает элементы итерируемого объекта как отдельные позиционные или именованные аргументы в вызванную функцию.

Python

ФУНКЦИИ. Синтаксические особенности при определении.

Произвольные аргументы



```
def my_function(argument, *args, **kwargs):  
    print(argument)  
    if args:  
        print(args)  
    if kwargs:  
        print(kwargs)
```

```
my_function(  
    'Привет!',  
    'Произвольный позиционный аргумент №1',  
    'Произвольный позиционный аргумент №2',  
    my_argument_1='Произвольный именованный аргумент №1',  
    my_argument_2='Произвольный именованный аргумент №2'  
)  
  
# Привет!  
# (  
#     'Произвольный позиционный аргумент №1',  
#     'Произвольный позиционный аргумент №2')  
# {  
#     'my_argument_1': 'Произвольный именованный аргумент №1',  
#     'my_argument_2': 'Произвольный именованный аргумент №2'  
# }
```

Аргументы функции со значением по умолчанию всегда идут после позиционных

```
def my_function(argument_1='Привет', argument_2, argument_3='я'):  
    print(argument_1 + '! ' + argument_2 + ' ' + argument_3 + '. ')
```

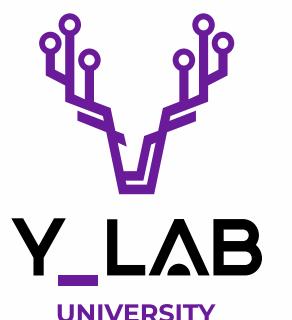
SyntaxError 😞

```
def my_function(argument_1, argument_2='Это', argument_3='я'):  
    print(argument_1 + '! ' + argument_2 + ' ' + argument_3 + '. ')
```



Python

Функции. Синтаксические особенности при расстановке аргументов



```
def my_function(argument_1, *args, argument_2, argument_3=None, **kwargs):
    print(f'args = {args} kwargs = {kwargs}')

# вызов функции minimumом аргументов
my_function(1, argument_2=2) # args = () kwargs = {}

# при вызове функций единственное правило по расположению:
# сначала позиционные, потом именованные

my_function(0, 1, argument_2=2, argument_3=3, argument4=4)
# args = (1,) kwargs = {'argument4': 4}
```

Перед kwargs
Всегда на последнем месте

Python

ФУНКЦИИ. ВЫСШИЙ ПОРЯДОК



```
def transform(text):  
    return text.upper()  
  
def transmute(func):  
    return func('Привет из transmute!')  
  
-----  
  
transmute(transform) # ПРИВЕТ ИЗ TRANSMUTE!
```

Python

ФУНКЦИИ. ВЫСШИЙ ПОРЯДОК



```
def transform(text):  
    def transmute(t):  
        return t.upper() + ' (из transmute)'  
    return transmute(text)
```

```
transform('Привет') # ПРИВЕТ (из transmute)
```

`input()` – функция для получения указанного пользователем значения.

Принимает в качестве аргумента строку для вывода сообщения пользователю о необходимости ввода значения.

Для преобразования в число строку необходимо конвертировать.

```
username = input('Введите ваше имя: ')
print(username)
```

```
>>> Введите ваше имя: Миша
Миша
```

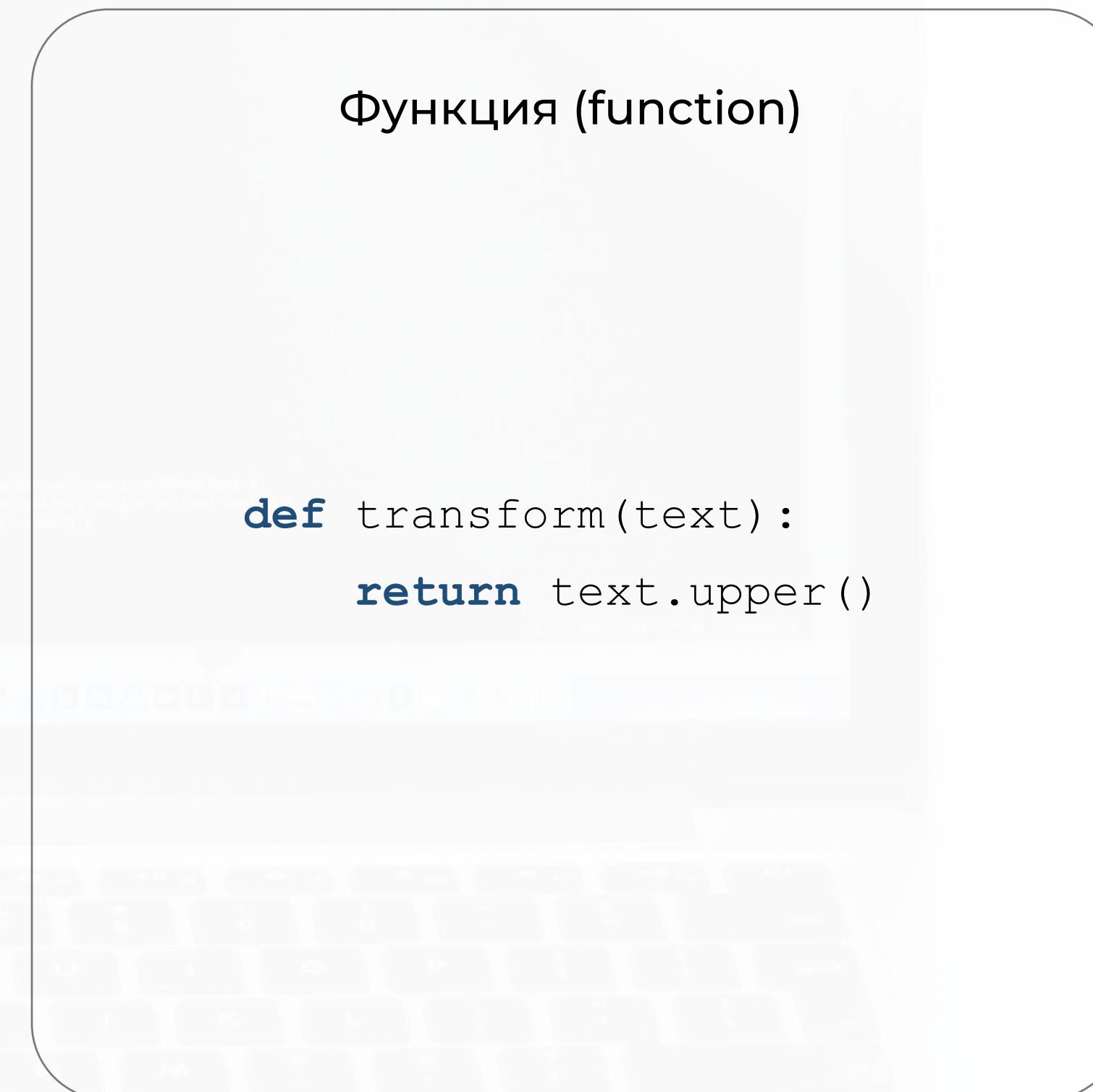
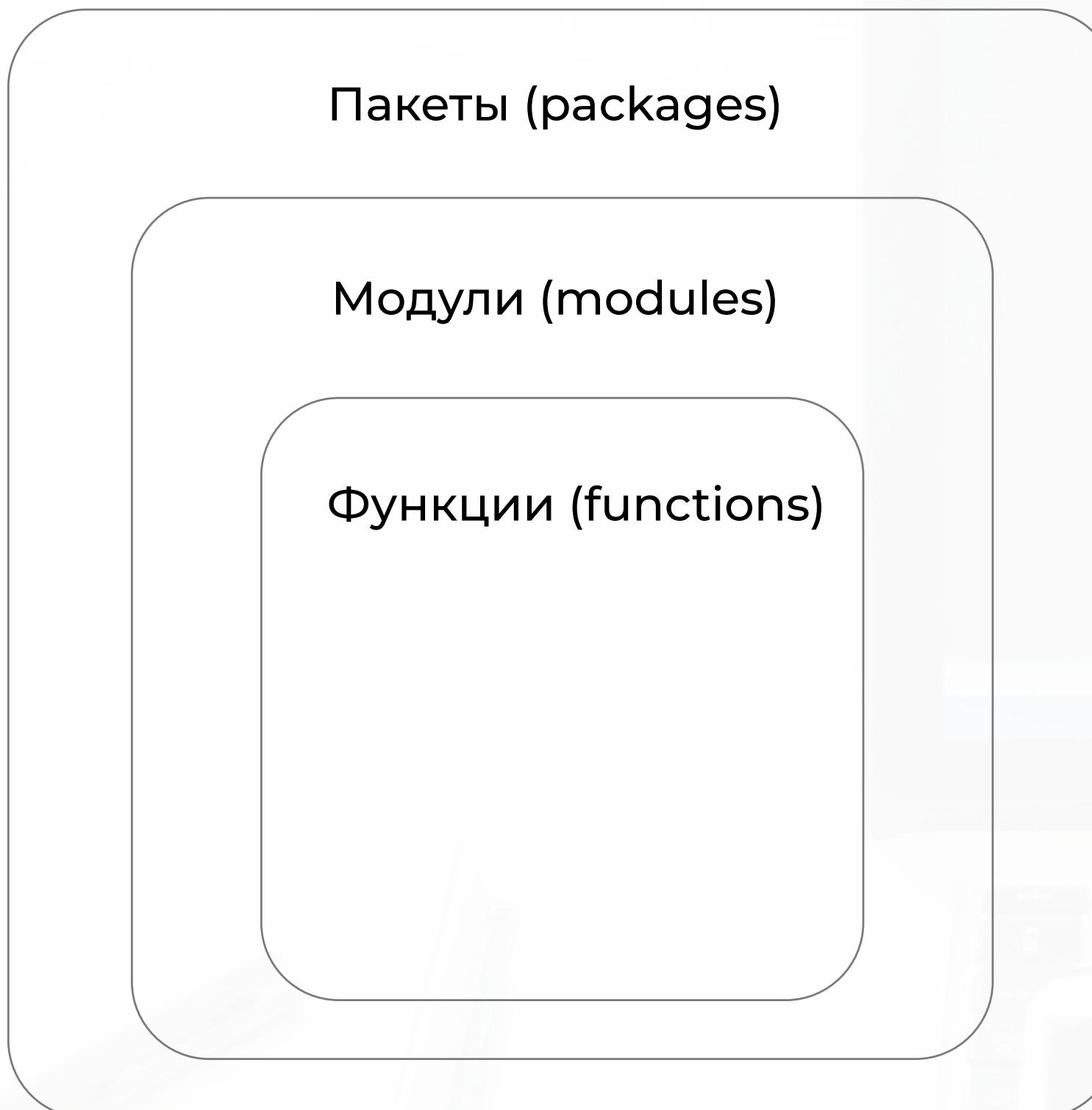
```
age = input('Сколько тебе лет? ')
print(age)
```

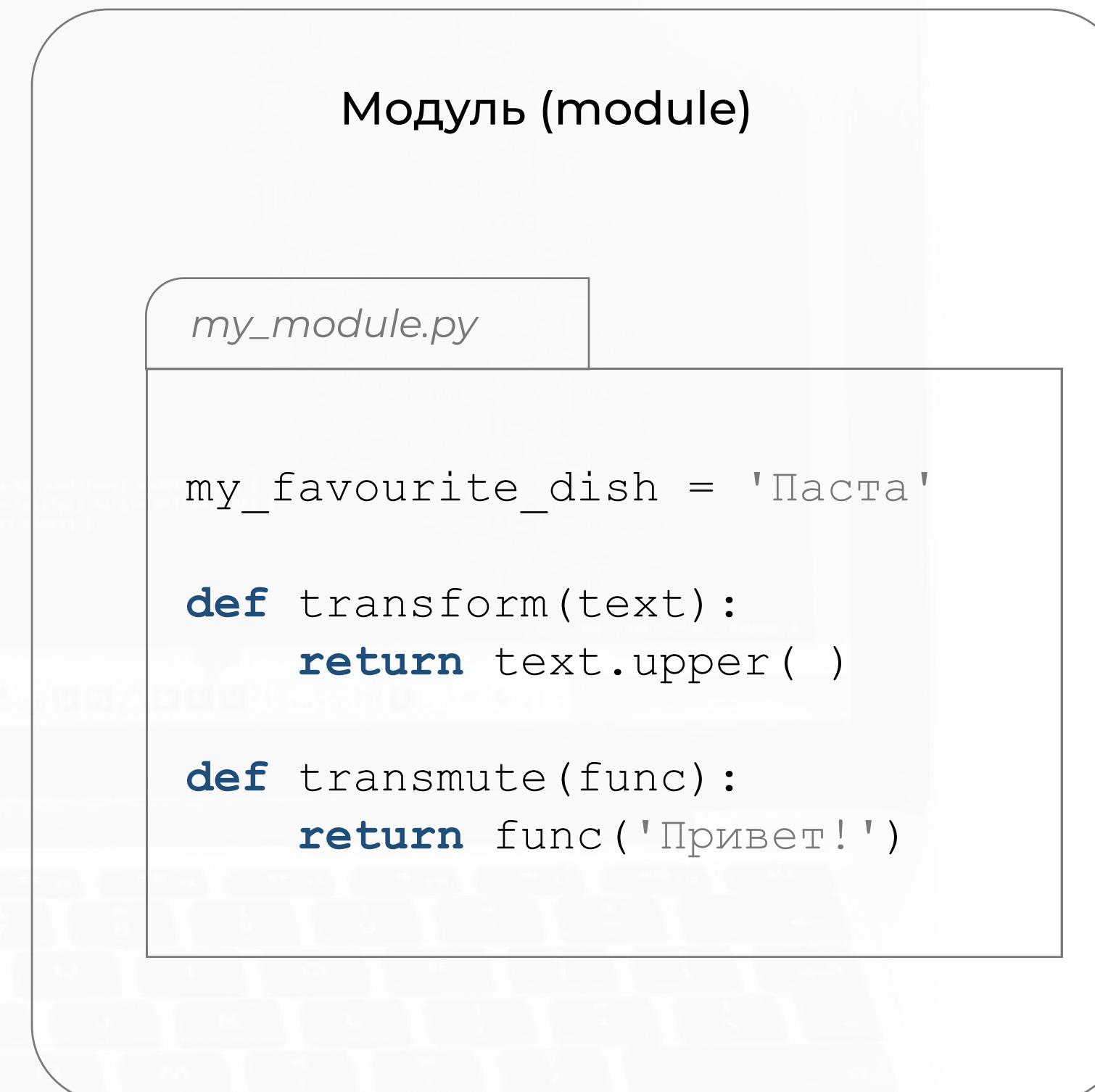
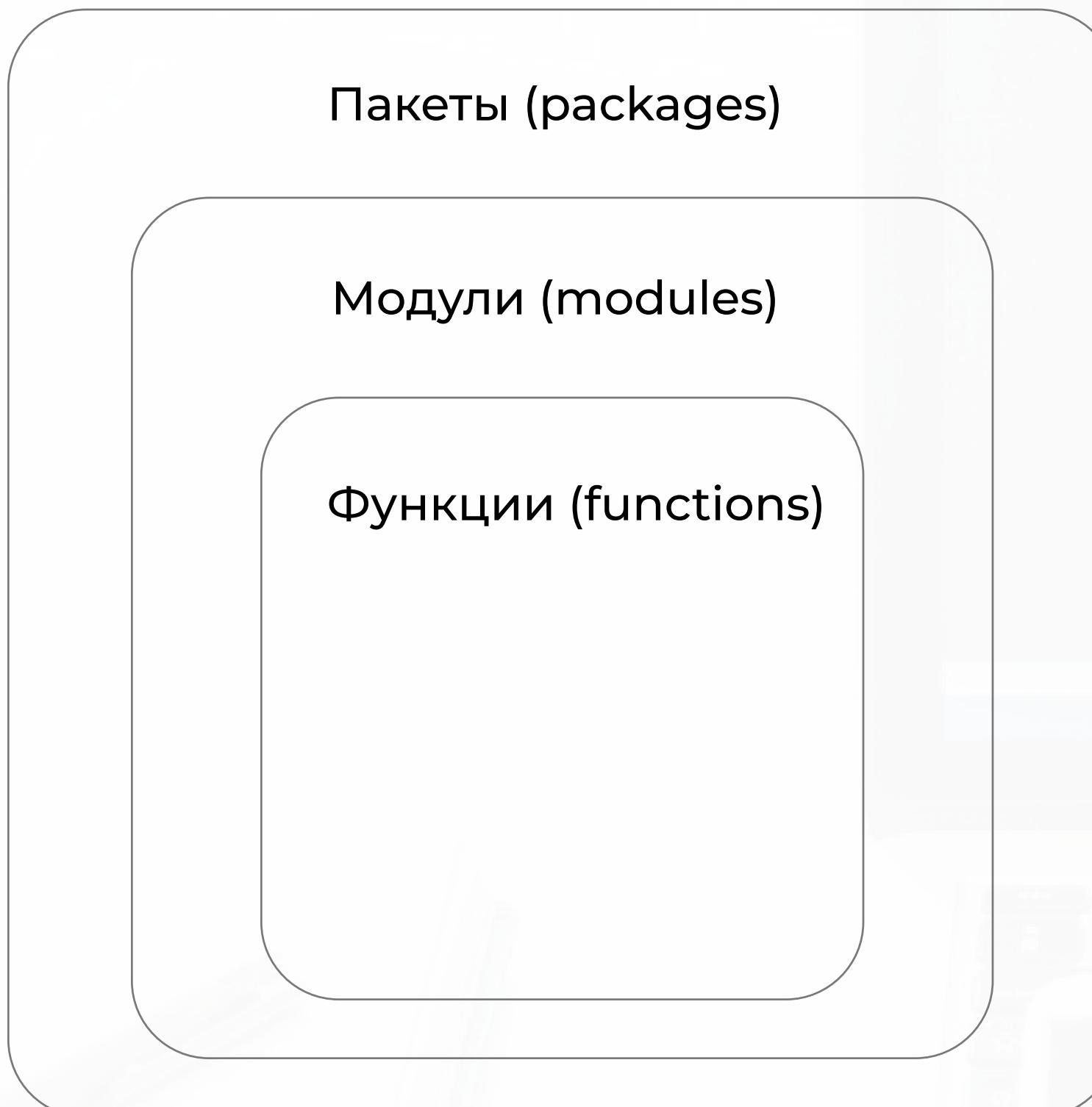
```
>>> Сколько тебе лет? 30
30
```

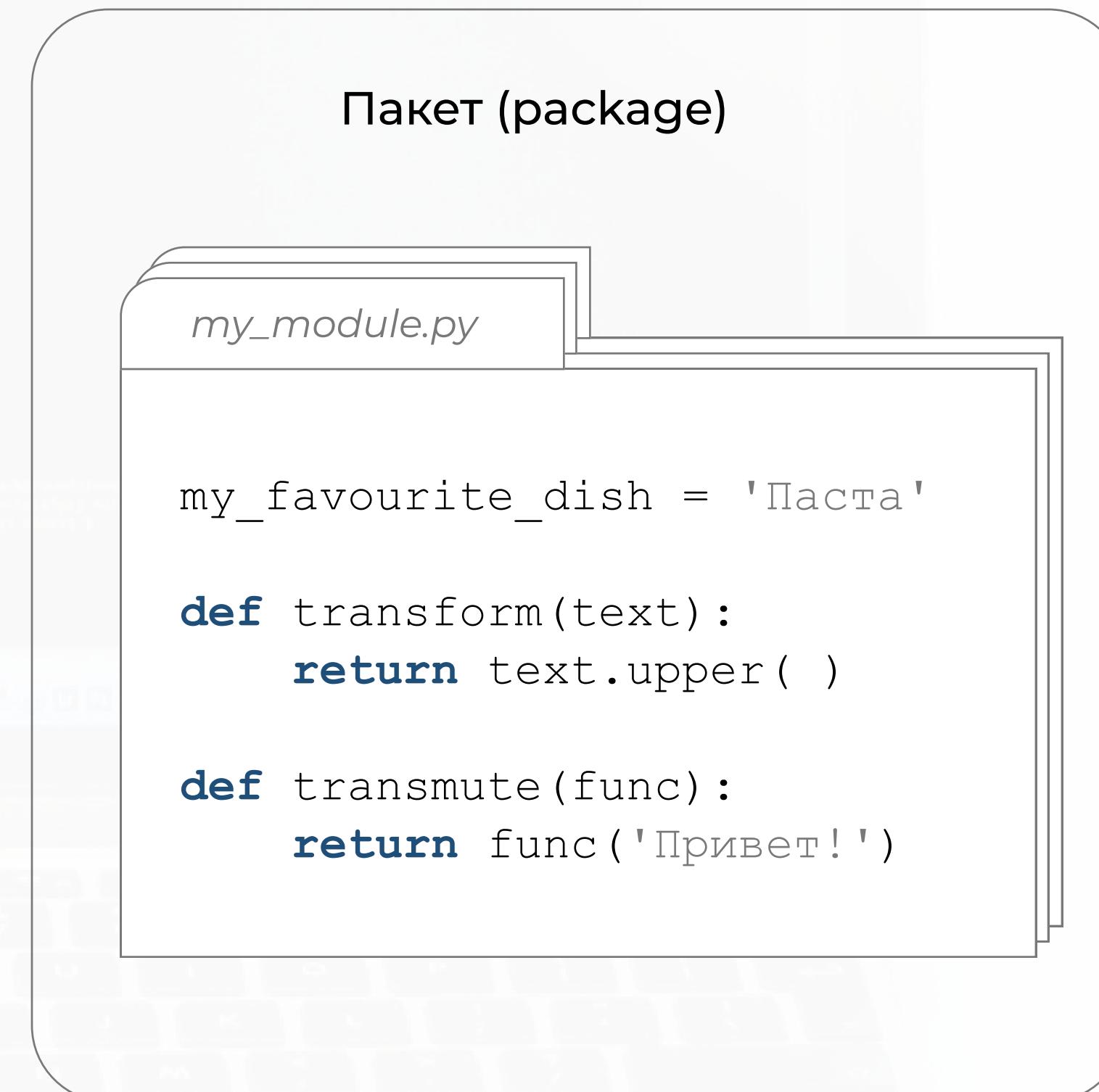
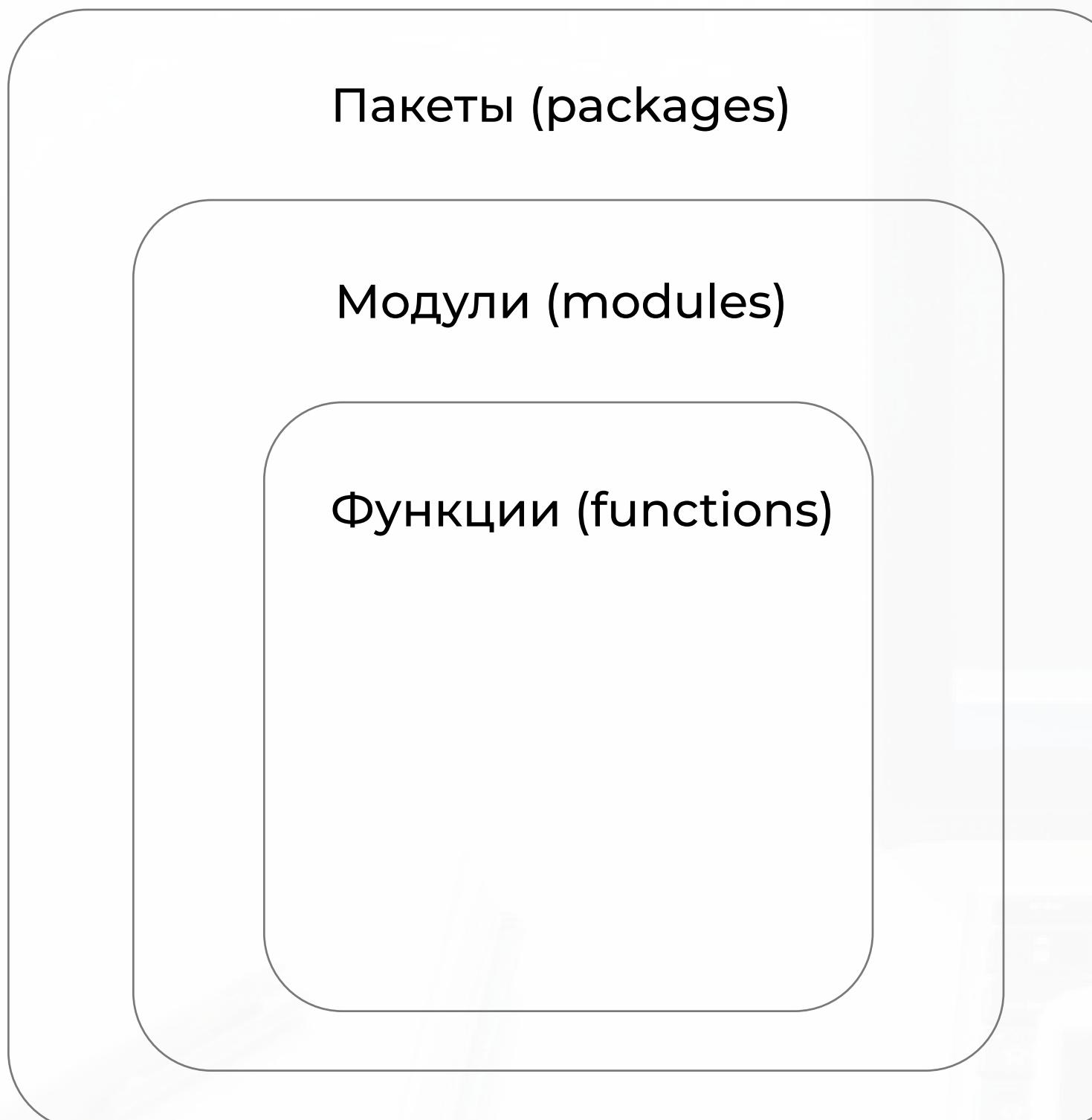
```
age = int(input('Сколько тебе лет? '))
print(age + 3)
```

- Вызов функций с именованными аргументами
- Вызов функций с позиционными аргументами
- Распаковка аргументов при вызове функций
- Определение функций со значениями по умолчанию
- Определение функций с произвольными аргументами
- Передача функции в качестве аргумента
- Вложенные функции
- Возврат функции в качестве аргумента
- Обработка ввода данных пользователем

[Ссылка на практику](#)







Python

Структура программы. Импорт



▼ package

 └ __init__.py
 └ module_1.py
 └ module_2.py
 └ module_3.py

module_1.py

```
def my_function():  
    print('Я из module_1!')
```

module_2.py

```
def my_function():  
    print('Я из module_2!')
```

module_3.py

```
def my_function():  
    print('Я из module_3!')
```

```
from package.module_1 import my_function  
from package.module_2 import my_function as my_function_2  
from package.module_3 import my_function as my_function_3
```

```
my_function() # Я из module_1!  
my_function_2() # Я из module_2!  
my_function_3() # Я из module_3!
```

Сыграем в игру - “Крестики-нолики”!

```
9 | X | 7
- | - | -
6 | 0 | 4
- | - | -
1 | 2 | 3
```

Очередь игрока "0":

Игрок "0", выберите ячейку с 1 по 9:

```
"""
Игра 'Крестики-нолики'

"""

import random

PLAY_BOARD = [str(num) for num in range(1, 10)]
PLAYERS_MARKS = ['X', 'O']

def display_board(board_list):
    """Генерация игрового поля"""

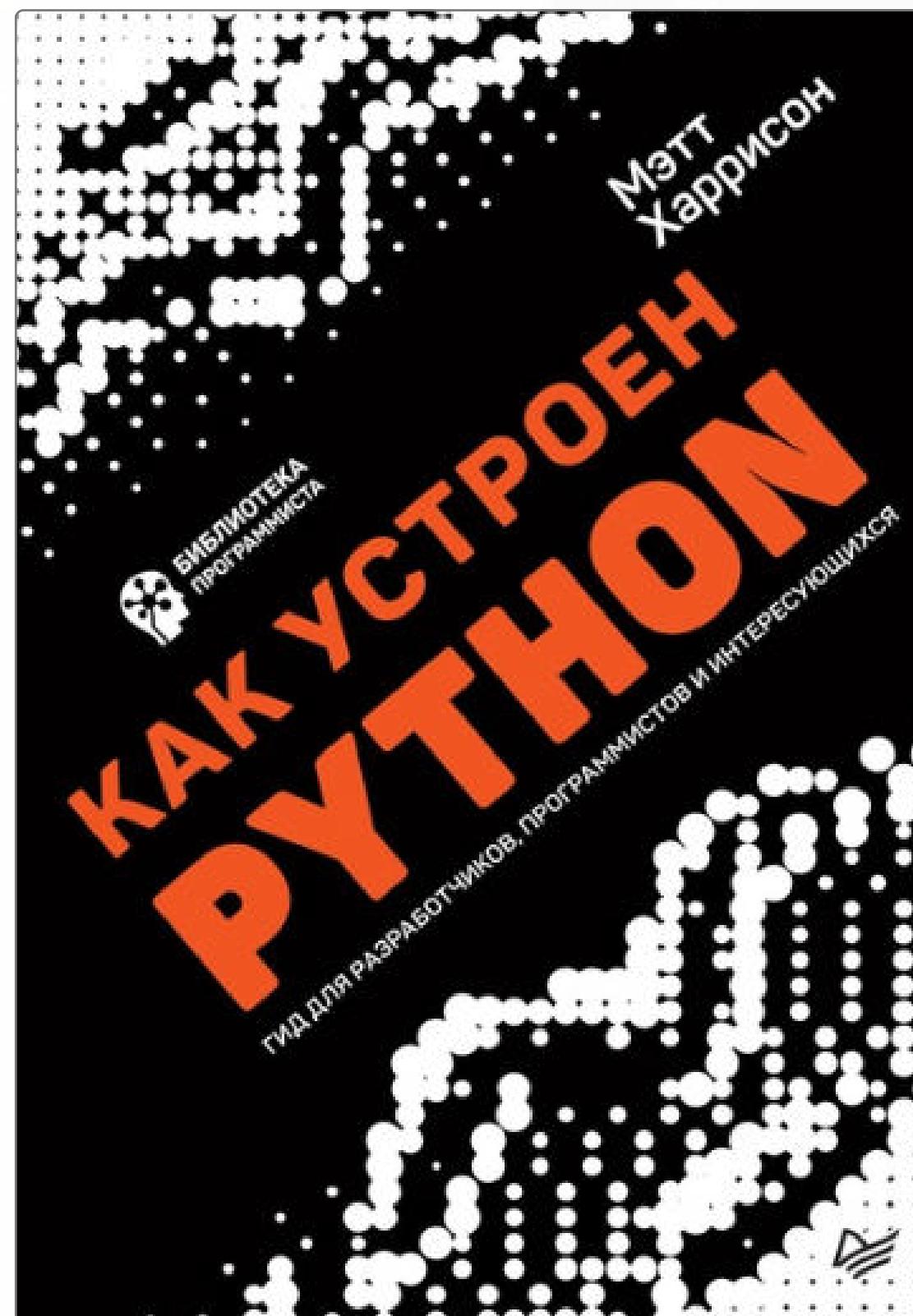
    print(board_list[8] + ' | ' + board_list[7] + ' | ' + board_list[6])
    print(' - | - | - ')
    print(board_list[5] + ' | ' + board_list[4] + ' | ' + board_list[3])
    print(' - | - | - ')
    print(board_list[0] + ' | ' + board_list[1] + ' | ' + board_list[2])
...
```

[Ссылка на практику](#)

<https://github.com/mnv/python-tic-tac-toe>

Python

Список материалов для изучения



Мэтт Харрисон: Как устроен Python.
Гид для разработчиков,
программистов и интересующихся



Бхаргава Адитья: Грекаем алгоритмы.
Иллюстрированное пособие для
программистов
и любопытствующих

Разработать игру «Обратные крестики-нолики» на поле 10×10 с правилом «Пять в ряд» – проигрывает тот, у кого получился вертикальный, горизонтальный или диагональный ряд из пяти своих фигур (крестиков/ноликов).

Игра должна работать в режиме «человек против компьютера».

Игра может быть консольной или поддерживать графический интерфейс (будет плюсом, но не требуется).

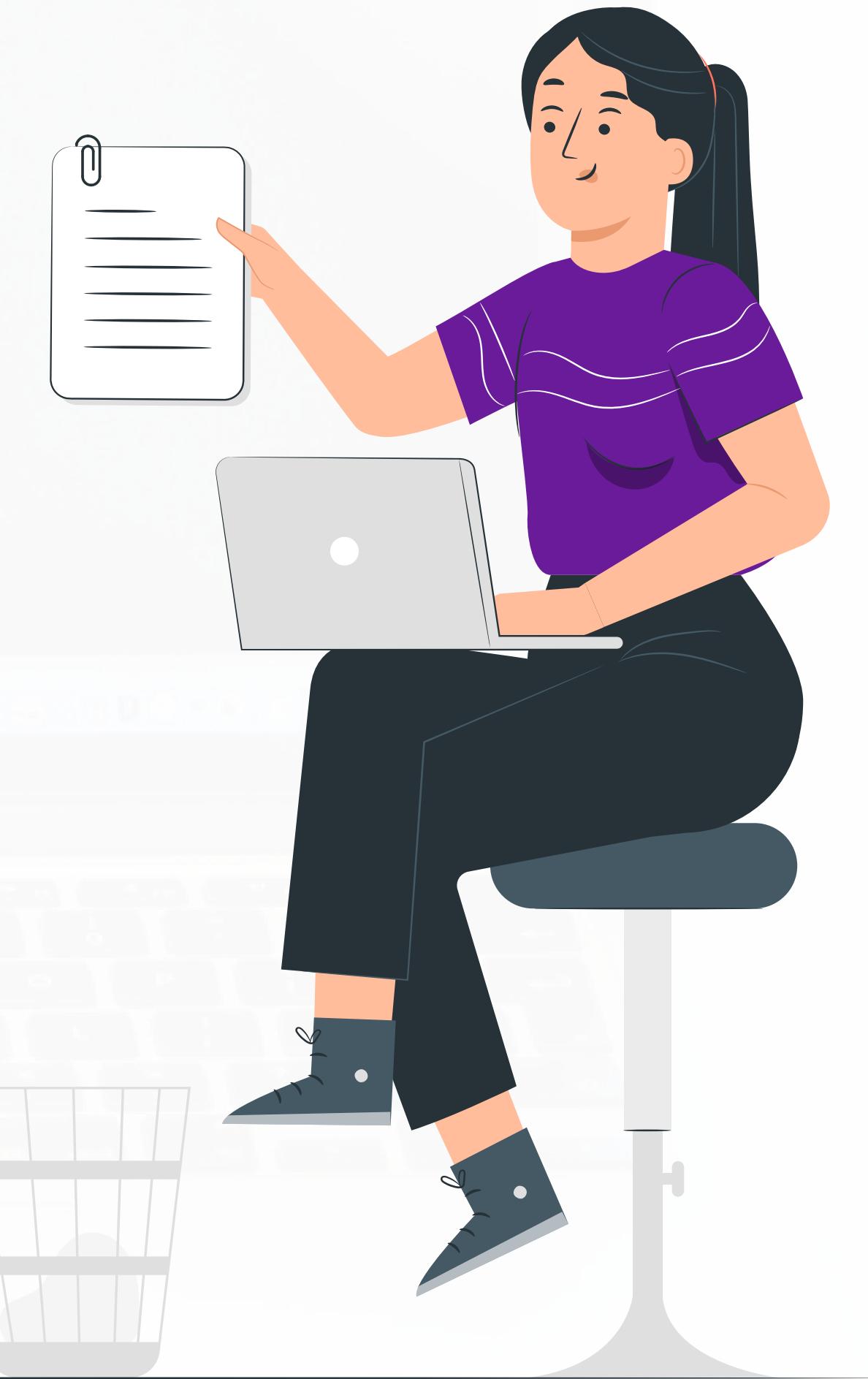
При разработке игры учесть принцип DRY (don't repeat yourself) – «не повторяйся». То есть минимизировать повторяемость кода и повысить его переиспользуемость за счет использования функций. Функции должны иметь свою зону ответственности.

- Достигли ли целей вебинара?
- Что запомнилось / понравилось?

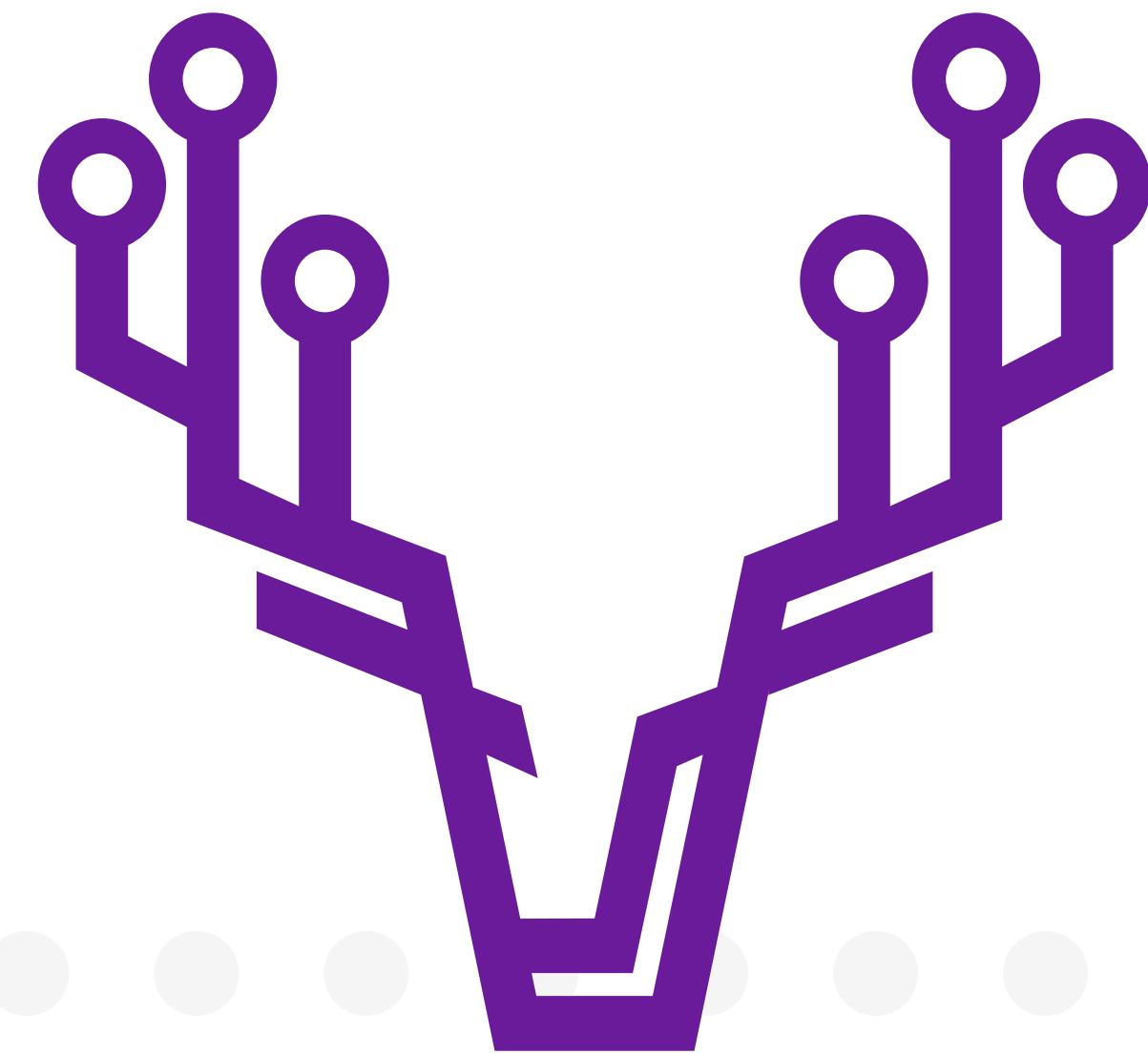
Обратная связь

Пожалуйста, пройдите опрос о занятии.
Нам очень важно ваше мнение!

Опрос о занятии



Спасибо за
внимание!



Y-LAB
UNIVERSITY