

Функции, декораторы, встроенные функции

Антон Кухтичев

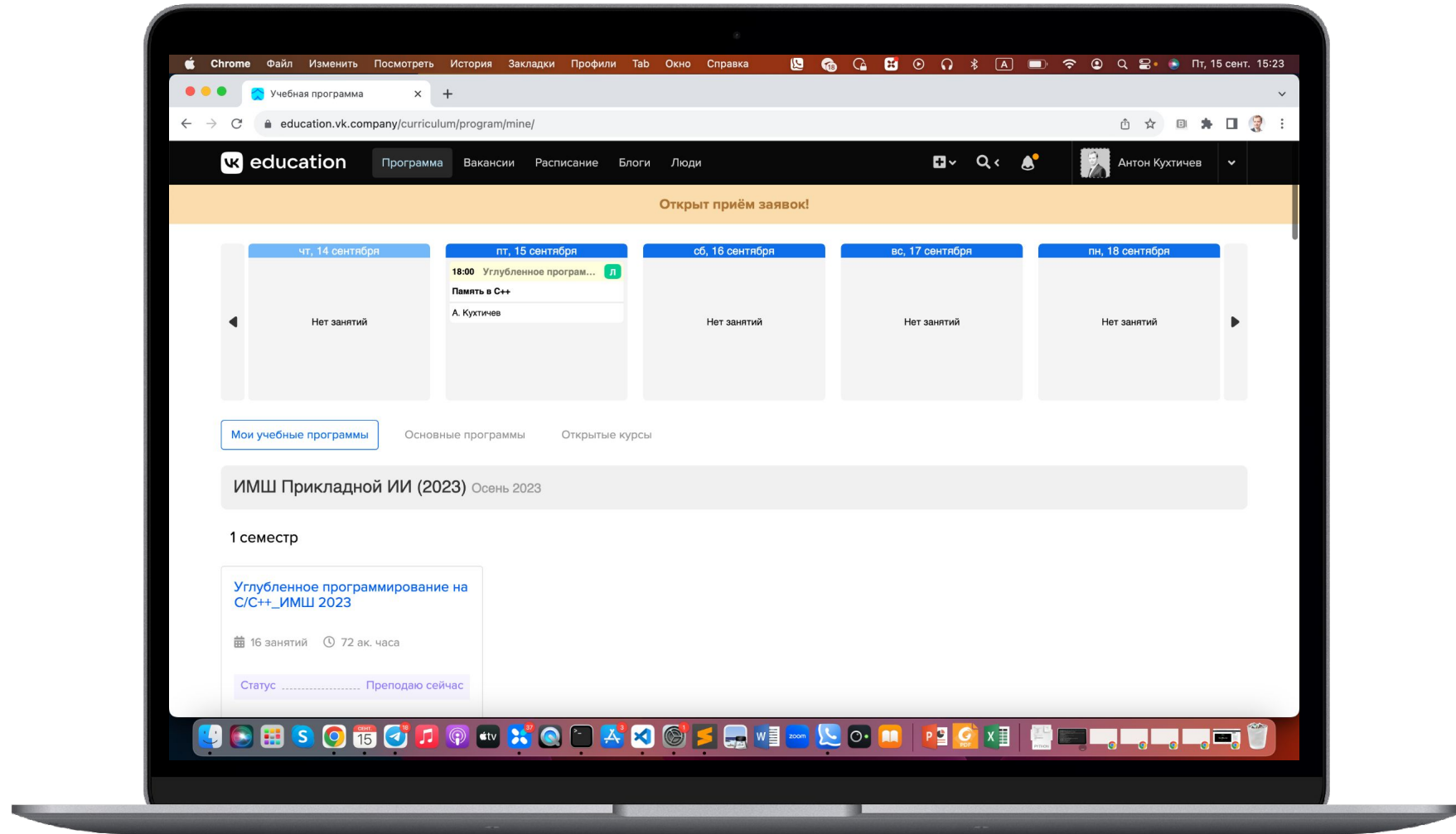


Содержание занятия

- Квиз
- Функции
- Аргументы функции
- Декораторы
- λ -функция
- Встроенные функции
- Тестирование

Напоминание отметиться на портале

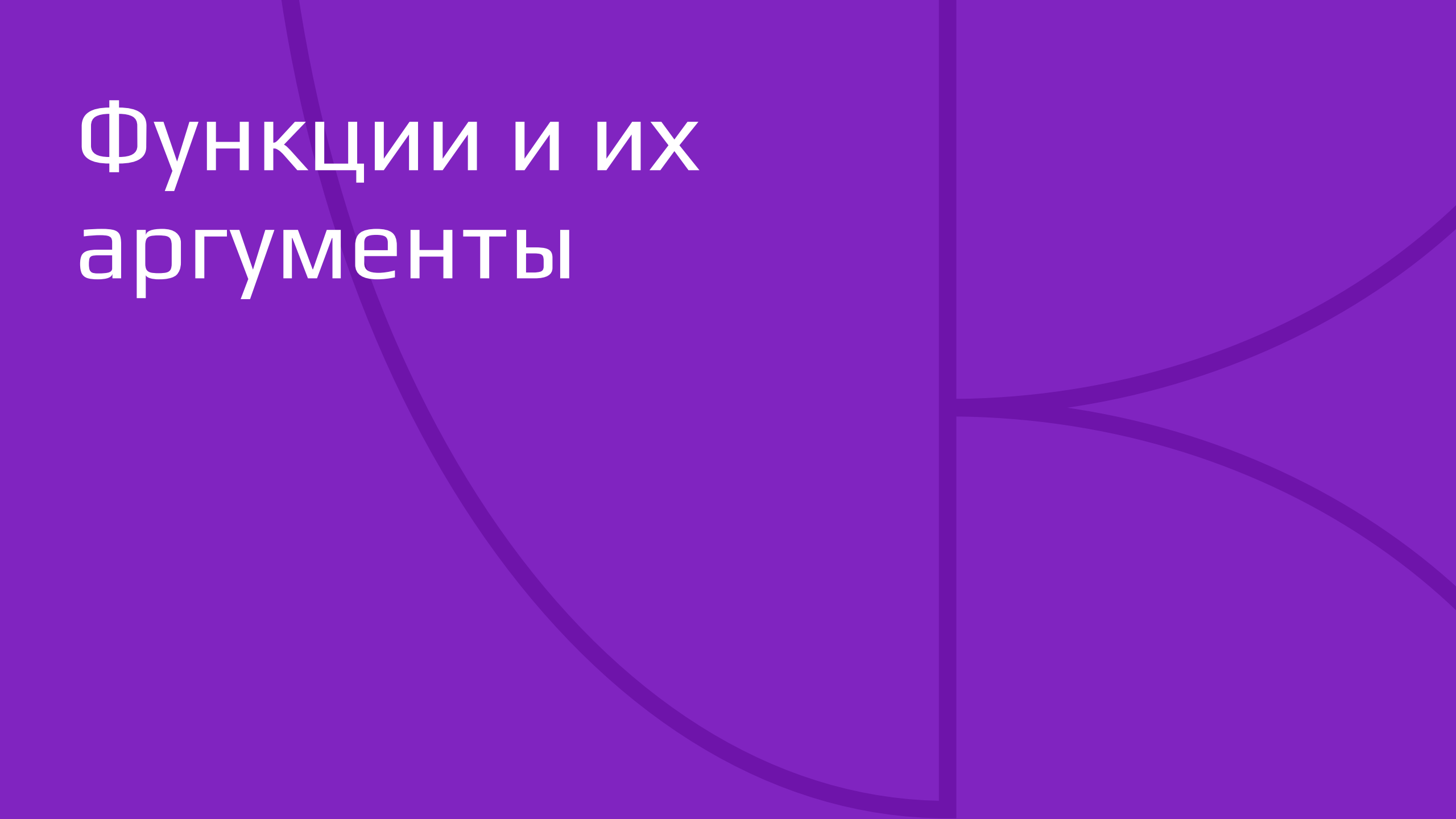
и оставить отзыв
после лекции



КВИЗ #1

<https://forms.gle/rZHRvuwmmpn47Hw5>

Функции и их аргументы

The background is a solid purple color. It features several white geometric lines: a large arc on the left side, a vertical line on the right side, and two curved lines on the right side that meet at a point, resembling a stylized 'Y' or a branching structure.

Функции

```
def square(x):  
    return x * x
```

```
>>> square(4)
```

```
16
```

Правила наименования:

- имя функции состоит из букв, чисел, знака подчёркивания (_);
- название функции не должно начинаться с цифры;
- лидирующий знак подчёркивания - соглашение, что функция приватная.



Аргументы функции

```
>>> def func(a, b, c=2): # c - необязательный аргумент
...     return a + b + c
```

- может принимать произвольное количество аргументов;
- а может и не принимать их вовсе;
- может принимать и произвольное число именованных аргументов;
- у аргументов может быть значение по умолчанию;



1. Эл Свейгарт. Python. Чистый код для продолжающих. Глава 8. Не используйте изменяемые значения для аргументов по умолчанию

Декоратор

```
def square(func):  
    def wrapper(*args, **kwargs):  
        # какая-то логика  
    return wrapper
```

Декоратор - это функция, принимающая единственный аргумент - другую функцию и выполняющая дополнительную логику.



λ-функция

```
f = lambda x: x * x
```

```
>>> f(2)
```

```
4
```

- Работают быстрее классических функций;
- Полезны в случае, когда нужна одноразовая функция;
- Потенциально повышают читаемость кода, а могут понизить.



Функции: параметры

```
def fn1(x, y=100): pass
```

```
def fn2(*args): pass
```

```
def fn3(**kwargs): pass
```

```
def fn4(*args, **kwargs): pass
```

```
def fn5(*, val): pass
```

```
def fn6(start, stop, /): pass
```

```
def fn7(pos1, /, pos2, pos3=3, *, kw1=11, **kwargs): pass
```



Функции: параметры

```
def make_function(name, *args, kw=12, **kwargs):  
    '''makes inner function'''  
    def inner(age=999):  
        print(f"{name=}, {age=}, {kw=}, {args=}, {kwargs=}")  
    return inner  
  
fn = make_function('skynet', 54, aim='term')  
  
fn()  
  
# name='skynet', age=999, kw=12, args=(54,), kwargs={'aim':  
# 'term'}
```

Функции

```
>>> fn.__dict__
```

```
{}
```

```
>>> fn.music = 'yes'
```

```
>>> fn.__dict__
```

```
{'music': 'yes'}
```

```
>>> fn.music
```

```
'yes'
```



Функции: атрибуты

__doc__ докстринг, изменяемое

```
>>> make_function.__doc__  
'makes inner function'
```

__name__ имя функции, изменяемое

```
>>> make_function.__name__  
'make_function'
```

__qualname__ полное имя функции, изменяемое

```
>>> make_function.__qualname__  
'make_function'
```

```
>>> fn.__qualname__
```

```
'make_function.<locals>.inner'
```



Функции: атрибуты

`__defaults__` кортеж дефолтных значений, изменяемое

```
>>> fn.__defaults__  
(999,)
```

`__kwdefaults__` словарь дефолтных значений кваргов, изменяемое

```
>>> make_function.__kwdefaults__  
{'kw': 12}
```

`__closure__` кортеж свободных переменных функции

```
>>> make_function.__closure__  
None  
  
>>> fn.__closure__[0].cell_contents  
(54,)
```

Пространство имён

*“Namespaces are one honking great idea --
let's do more of those!”*

Tim Peters (import this)

Пространство имён

Пространство имён — это совокупность определенных в настоящий момент символических имен и информации об объектах, на которые они ссылаются.

- Встроенное
- Глобальное
- Объемлющее
- Локальное

Пространство имён

Область видимости имени это часть программы, в которой данное имя обладает значением.

Интерпретатор определяет эту область в среде выполнения, основываясь на том, где располагается определение имени и из какого места в коде на него ссылаются.

1. Локальная
2. Объемлющая
3. Глобальная
4. Встроенная

Область видимости: LEGB

- `globals()`
- `locals()`
- `global`
- `nonlocal`

`__builtins__`

```
>>> hasattr(__builtins__, "dir")
```

```
True
```

```
>>> dir(__builtins__)
```

```
...
```

```
__builtins__
```

```
int float str bool tuple list dict set map zip filter range enumerate  
sorted min max reversed len sum all any globals locals callable dir  
type isinstance issubclass hasattr getattr setattr delattr
```



Встроенные функции

map

```
map(function, iterable, [iterable 2, iterable 3, ...])
```

```
def func(el1, el2):
```

```
    return '%s|%s' % (el1, el2)
```

```
list(map(func, [1, 2], [3, 4, 5])) # ['1|3', '2|4']
```

Применяет указанную функцию к каждому элементу указанной последовательности/последовательностей.

reduce

```
from functools import reduce

reduce(function, iterable[, initializer])

items = [1,2,3,4,5]

sum_all = reduce(lambda x,y: x + y, items)
```

Применяет указанную функцию к элементам последовательности, сводя её к единственному значению.



partial

```
partial(function, *args, **keywords)
```

```
>>> basetwo = partial(int, base=2)
```

```
>>> basetwo('10010')
```

18

Функция `partial` позволяет применять функцию частично. Получив на входе некоторую функцию, `partial` создаёт новый вызываемый объект, в котором некоторые аргументы исходной функции фиксированы.



filter

```
filter(function, iterable)
```

```
def is_even(x):
```

```
    return x % 2 == 0:
```

```
>>> print(list(filter(is_even, [1, 3, 2, 5, 20, 21])))
```

```
[2, 20]
```

Функция `filter` предлагает элегантный вариант фильтрации элементов последовательности. Принимает в качестве аргументов функцию и последовательность, которую необходимо отфильтровать.

zip

```
>>> a = [1, 2, 3]
>>> b = "xyz"
>>> c = (None, True)
>>> print(list(zip(a, b, c)))
[(1, 'x', None), (2, 'y', True)]
```

Функция `zip` объединяет в кортежи элементы из последовательностей переданных в качестве аргументов.



compile

```
compile(source, filename, mode, flag, dont_inherit, optimize)
```

```
# выполнение в exec
```

```
>>> x = compile('x = 1\nz = x + 5\nprint(z)', 'test', 'exec')
```

```
>>> exec(x)
```

```
# 6
```

```
# выполнение в eval
```

```
>>> y = compile("print('4 + 5 =', 4+5)", 'test', 'eval')
```

```
>>> eval(y)
```

```
# 4 + 5 = 9
```

exec

```
exec(obj[, globals[, locals]]) -> None
```

Динамически исполняет указанный код.



eval

```
eval(expression[, globals[, locals]])
```

- в `eval()` запрещены операции присваивания;
- `SyntaxError` также вызывается в случаях, когда `eval()` не удастся распарсить выражение из-за ошибки в записи;
- Аргумент `globals` опционален. Он содержит словарь, обеспечивающий доступ `eval()` к глобальному пространству имен;
- В `locals`-словарь содержит переменные, которые `eval()` использует в качестве локальных имен при оценке выражения.

Тестирование

“Тестирование показывает присутствие ошибок, а не их отсутствие.”

Эдсгер Дейкстра

Цели тестирования

- Проверка правильности реализации
- Проверка обработки нестандартных ситуаций и граничных условий
- Минимизация последствий
- Подготовка ко внесению изменений



Виды тестирования

- Unit-тесты (модульные тесты)
- Функциональное тестирование
- Системное тестирование
- Интеграционное тестирование
- Регрессионное тестирование
- Тестирование производительности
 - Нагрузочное
 - Стресс



TDD

TDD (Test Driven Development) – техника разработки ПО, основывается на повторении коротких циклов разработки: пишется тест, покрывающий желаемое изменение, затем пишется код, который позволит пройти тест, и далее проводится рефакторинг нового кода.

Степень покрытия тестами

coverage - библиотека для проверки покрытия тестами.

```
pip install coverage
```

```
coverage run tests.py
```

```
coverage report -m
```

```
coverage html
```



Инструменты тестирования в Python

- doctest
- unittest
- pytest
- factory_boy
- selenium



doctest

```
def multiply(a, b):  
    """  
    >>> multiply(4, 3)  
    12  
    >>> multiply("a", 3)  
    'aaa'  
    """  
    return a * b
```

```
python -m doctest <file>
```

unittest

```
class TestCase
```

- `def setUp(self):`

установки запускаются перед каждым тестом

- `def tearDown(self):`

очистка после каждого метода

- `def test_<название теста>(self):`

код теста

unittest: TestCase

```
import unittest

class TestString(unittest.TestCase):

    def test_upper(self):

        self.assertEqual("text".upper(), "TEXT")

if __name__ == "__main__":

    unittest.main()
```

unittest: набор assert*

- `assertEqual(a, b)`
- `assertNotEqual(a, b)`
- `assertTrue(x)`
- `assertFalse(x)`
- `assertIsNone(x)`
- `assertIs(a, b)`
- `assertIsNot(a, b)`
- `assertIn(a, b)`
- `assertIsInstance(a, b)`
- `assertLessEqual(a, b)`
- `assertListEqual(a, b)`
- `assertDictEqual(a, b)`
- `assertRaises(exc, fun, *args, **kwargs)`

unittest: mock

Mock — это объект-пустышка, который заменяет некий реальный объект (функцию, экземпляра части программы).

- Высокая скорость
- Избежание нежелательных побочных эффектов во время тестирования
- Позволяет задать специальное поведение в рамках теста

```
from unittest.mock import patch
```

```
class TestUserSubscription(TestCase):  
    @patch("users.views.get_status", return_value=True)  
    def test_subscription(self, get_status_mock):
```

unittest: mock

Атрибуты объекта Mock с информацией о вызовах

- `called` — вызывался ли объект вообще
- `call_count` — количество вызовов
- `call_args` — аргументы последнего вызова
- `call_args_list` — список всех аргументов
- `method_calls` — аргументы обращений к вложенным методам и атрибутам
- `mock_calls` — то же самое, но в целом и для самого объекта, и для вложенных

```
self.assertEqual(get_subscription_status_mock.call_count, 1)
```

unittest: запуск тестов

Найти и выполнить все тесты

```
python -m unittest discover
```

Тесты нескольких модулей

```
python -m unittest test_module1 test_module2
```

Тестирование одного кейса - набора тестов

```
python -m unittest tests.SomeTestCase
```

Тестирование одного метода

```
python -m unittest tests.SomeTestCase.test_some_method
```


factory_boy

Библиотека `factory_boy` служит для генерации разнообразных объектов (в т.ч. связанных) по заданным параметрам.

<https://factoryboy.readthedocs.io/en/stable/>

<https://faker.readthedocs.io/en/master/>

```
pip install factory_boy
```

selenium

Selenium WebDriver – это программная библиотека для управления браузерами. WebDriver представляет собой драйверы для различных браузеров и клиентские библиотеки на разных языках программирования, предназначенные для управления этими драйверами.

```
pip install selenium
```

selenium

- Требует конкретного драйвера для конкретного браузера (Chrome, Firefox и т.д.)
- Автоматическое управление браузером
- Поддержка Ajax
- Автоматические скриншоты

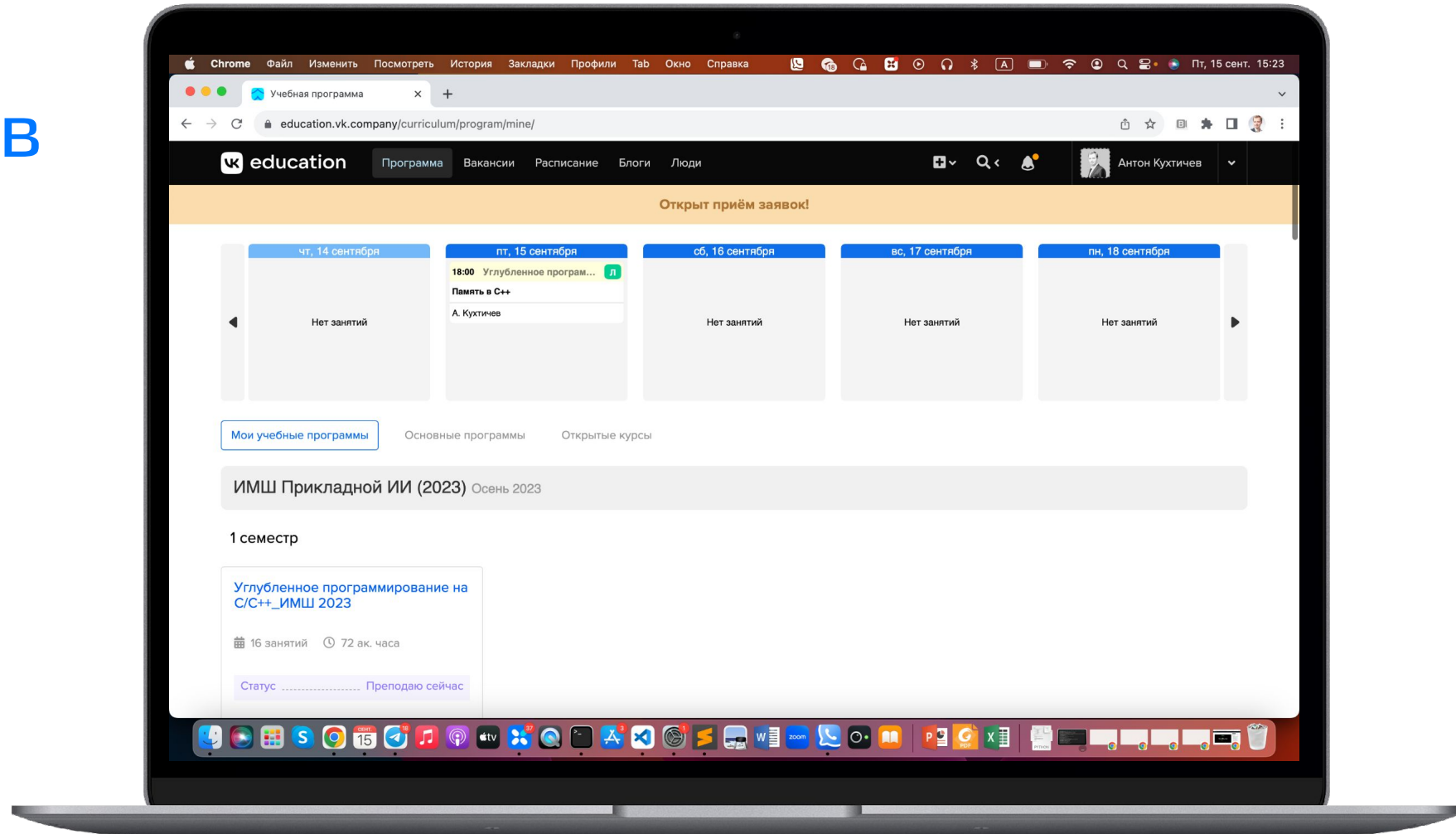
Домашнее задание

Домашнее задание

- Написать функцию, которая в качестве аргументов принимает:
 - строку json;
 - список обязательных полей (может быть None);
 - список ключевых слов, которые будут искаться в обязательных полях (может быть None);
 - функция-обработчик слов, которые встретились в списке ключевых слов;
- Написать декоратор;
- Использовать mock-объект при тестировании;
- Использовать factory boy (по умолчанию);
- Узнать степень покрытия тестами с помощью библиотеки coverage

Напоминание оставить отзыв

Это правда важно





Спасибо
за внимание!