

PYTHON

The Python logo consists of the word "PYTHON" in a bold, black, sans-serif font. To the right of the text is a green cartoon snake with a patterned body, a pink tongue, and large black eyes.

Дима Куприянов

@dokupriyanov

разработчик @ CI/CD



Преподаватели курса

Роман Афанаскин

Дмитрий Шуляк

Сергей Нинуа

Павел Лысак

Александр Непочатых

Дмитрий Куприянов

Немного о курсе & FAQ

- о чем курс?
- какой формат?
- зачем нужен курс?
- как будем оценивать?

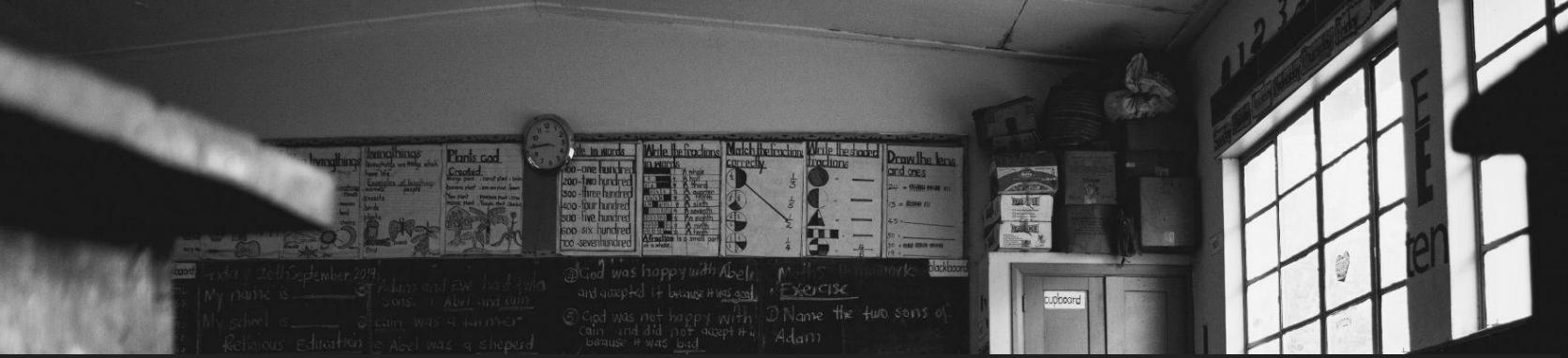
Правила игры 📖 :

- включите пожалуйста камеру 🙏
- не стесняйтесь спрашивать
- экспериментируйте! 🧐

План занятия :



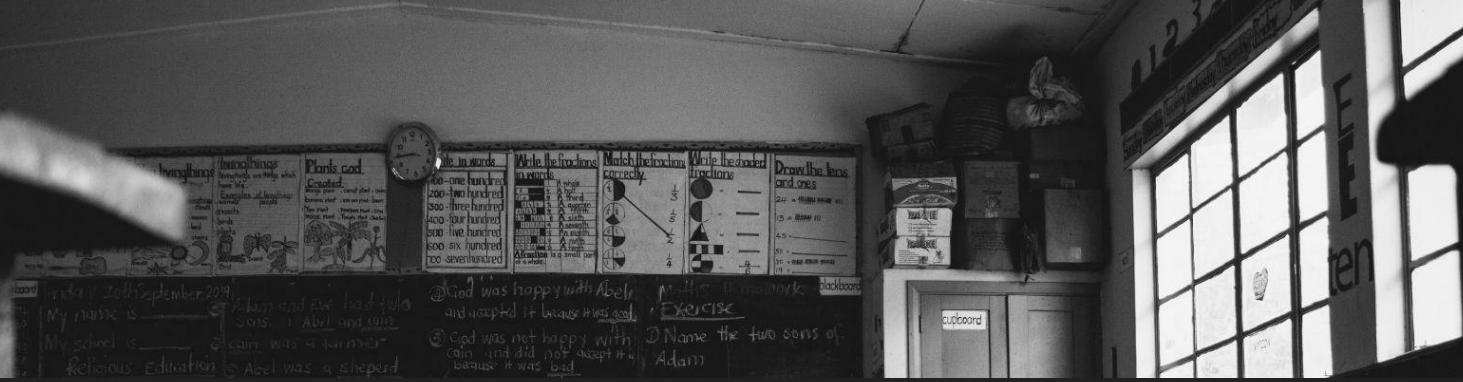
- знакомимся
- неизменяемые типы (immutable)
- coffee break ☕
- изменяемые типы (mutable)
- управляющие конструкции
- домашка 🦆



Блиц ⚡ :

- Есть ли у вас опыт программирования?
напишите в чат, на чём писали





Блиц ⚡ :

- Для чего, по вашему мнению, Python нужен аналитикам?



Блиц ⚡ :

- Почему Python так называется?



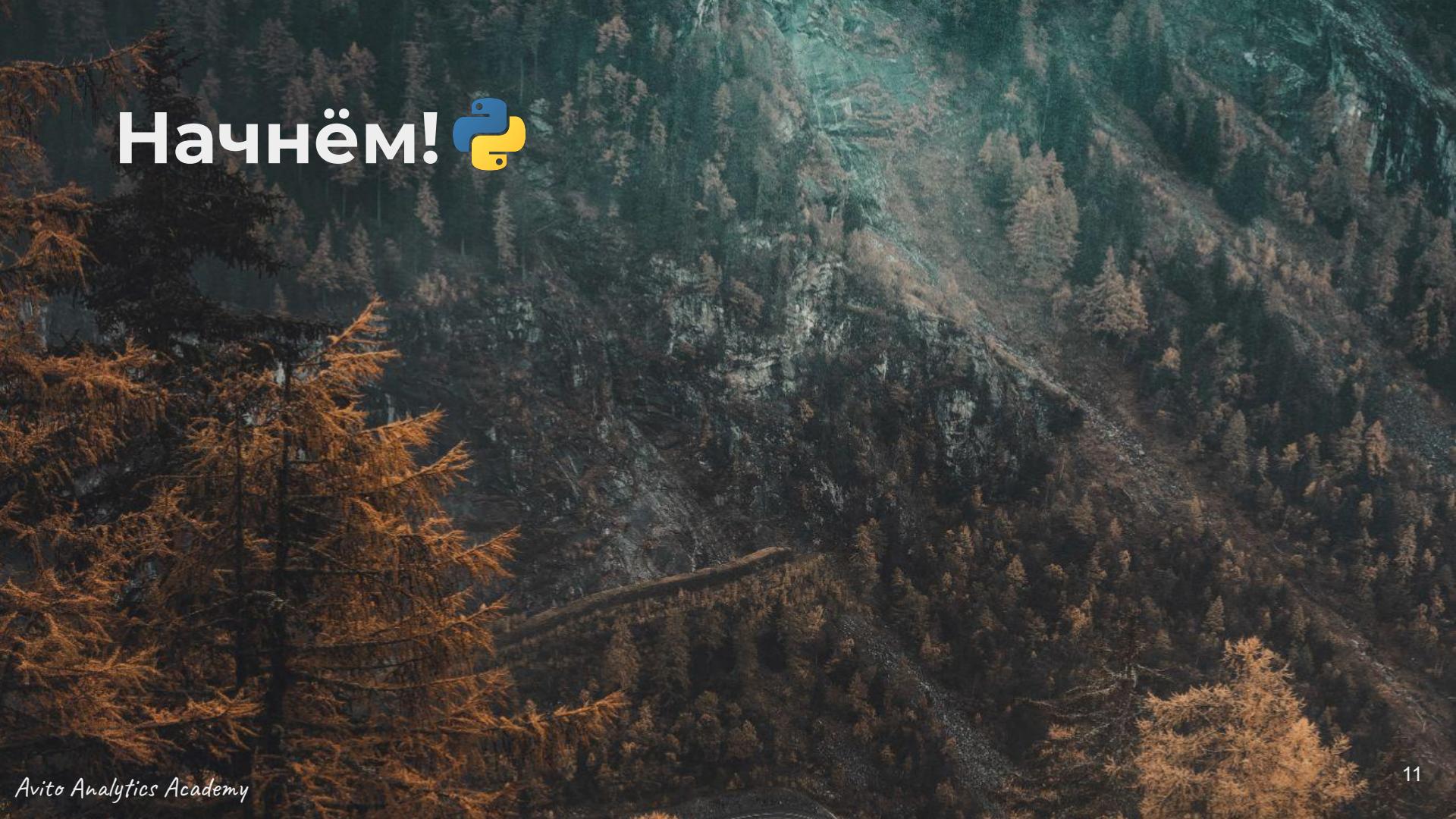


Подготовка



:

- IDE
 -  <https://code.visualstudio.com/>
 -  <https://www.jetbrains.com/pycharm/download/>
- Git
 -  [git https://git-scm.com/downloads](https://git-scm.com/downloads)



Начнём!

Пасхалочки



:

```
>>> import __hello__  
Hello World!
```



```
>>> import this  
The Zen of Python, by Tim Peters  
  
Beautiful is better than ugly.  
Explicit is better than implicit.  
Simple is better than complex.  
Complex is better than complicated.  
Flat is better than nested.  
Sparse is better than dense.  
Readability counts.  
Special cases aren't special enough to break the rules.  
Although practicality beats purity.  
Errors should never pass silently.  
Unless explicitly silenced.  
In the face of ambiguity, refuse the temptation to guess.  
There should be one-- and preferably only one --obvious way to do it.  
Although that way may not be obvious at first unless you're Dutch.  
Now is better than never.  
Although never is often better than *right* now.  
If the implementation is hard to explain, it's a bad idea.  
If the implementation is easy to explain, it may be a good idea.  
Namespaces are one honking great idea -- let's do more of those!
```



This.py

PEP (Python Enhancement Proposals) & PEP8 :

- PEPs: <https://www.python.org/dev/peps/>
- PEP8: <https://www.python.org/dev/peps/pep-0008/>

PEP8 🐍 :

```
# отступы имеют значение  
# отступ = 4 пробела [не Tab :)]  
  
def foo():  
    print('bar')
```

PEP8 :

```
# скобки и ';' не нужны
```

```
if (foo ≠ 0):  
    pass;
```

PEP8 🐍 :

```
# рекомендуемая длина строки по РЕР 8 – 79 символов

with open('/path/to/some/file/you/want/to/read') as file_1,
    open('/path/to/some/file/being/written', 'w') as file_2:
    file_2.write(file_1.read())
```

Hello World!

```
# сначала было слово и слово было  
print('Hello World!')
```

Объекты в Python



Всё в python - объект

Объекты в Python



Свойства:

- тип
- идентификатор
- значение

Пример:

- int
- 4313798960
- 1

Переменные в Python



Переменные - ссылки на объекты

```
>>> x = 1  
>>> y = 1  
>>> id(x)  
4313798960  
>>> id(y)  
4313798960
```

Переменные в Python



```
>>> x = 1
>>> y = 2
>>> x, y = y, x
>>> x
2
>>> y
1
```

Типизация в Python



Типизацию в Python часто называют "утиной". Почему?

Типизация в Python



“Если выглядит как int и крякает как int,
значит это наверное int”



Типизация в Python



```
foo = 1  
foo = 1.0  
foo = '1'  
foo = [1]  
foo = ['1']
```

- > int
- > float
- > string
- > list из int'ов
- > list из строк

Типы Python



bool
None
int
float
complex
string
tuple

list
dict
set

Полезные функции

```
# dir() возвращает все доступные атрибуты объекта

>>> dir(str)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper',
'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix',
'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',
'zfill']

# help() возвращает .__doc__() объекта

>>> help(print)
```

Неизменяемые типы

- bool
- None
- int
- float
- complex
- string
- tuple

string

```
# string - строки  
  
->>> x = 'quack-quack'  
->>> print(x)  
quack-quack
```

Виды строк

```
# unicode = default, включает в себя несколько алфавитов и emoji  
  
bird_one = u'duck'  
print(bird_one) # duck  
  
# bytes = байты  
  
bird_two = b'duck'  
print(bird_two) # b'duck'  
  
# raw - "сырые строки"  
  
bird_three = r'duck'  
print(bird_three) # duck
```

Операции над строками

```
# upper - делает все символы в строке заглавными  
bird_one = 'quack-quack'  
bird_one.upper() # QUACK-QUACK
```

```
# capitalize - делает первый символ в строке заглавным  
bird_one.capitalize() # Quack-quack
```

```
# encode - кодирует строку в binary  
bird_one.encode() # b'quack-quack'
```

Операции над строками

```
# upper - делает все символы в строке заглавными  
bird_two = b'QUACK QUACK'  
bird_two.lower() # b'quack quack'
```

```
# strip - вырезает переданный символ из строки. если символ не  
указан - удаляет пробелы  
bird_one.strip() # b'QUACKQUACK'
```

```
# decode - раскодирует строку в unicode  
bird_one.decode() # QUACK QUACK
```

Операции над строками

```
# split - разделяет строку на слова по разделителю  
# разделитель "по умолчанию" - пробел  
bird_three = 'QUACK QUACK'  
bird_three.split() # ['QUACK', 'QUACK']
```

```
# join - объединяет iterable объект в строку через разделитель  
'.'.join(bird_three) # 'Q,U,A,C,K, ,Q,U,A,C,K'
```

Форматирование строк

```
# "%s" % sub - олдскул
bird_three = 'QUACK QUACK %s'
bird_three % 'said duck' # QUACK QUACK said duck

# '{}'.format(sub) - недавнее прошлое
bird_three = 'QUACK QUACK {}'
bird_three.format('said duck') # QUACK QUACK said duck

# f-строки - новый способ форматирования, с Python 3.6
sub = 'said duck'
bird_three = f'QUACK QUACK {sub}' # QUACK QUACK said duck
```

Unicode Emoji

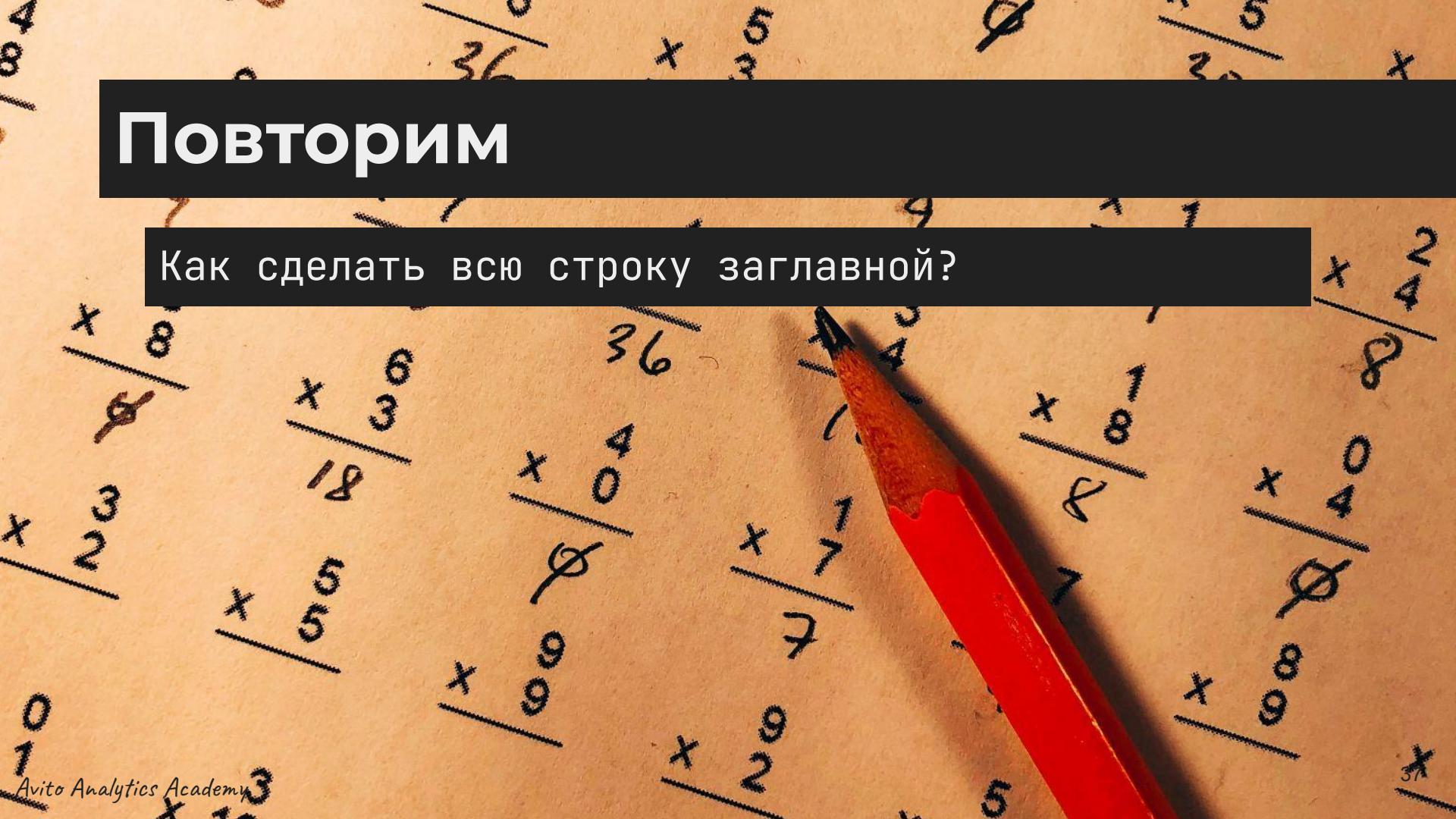
```
# Можно использовать \N для именованных  
# unicode-символов
```

```
>>> print('\N{Weary Cat Face}')
```



ПОВТОРИМ

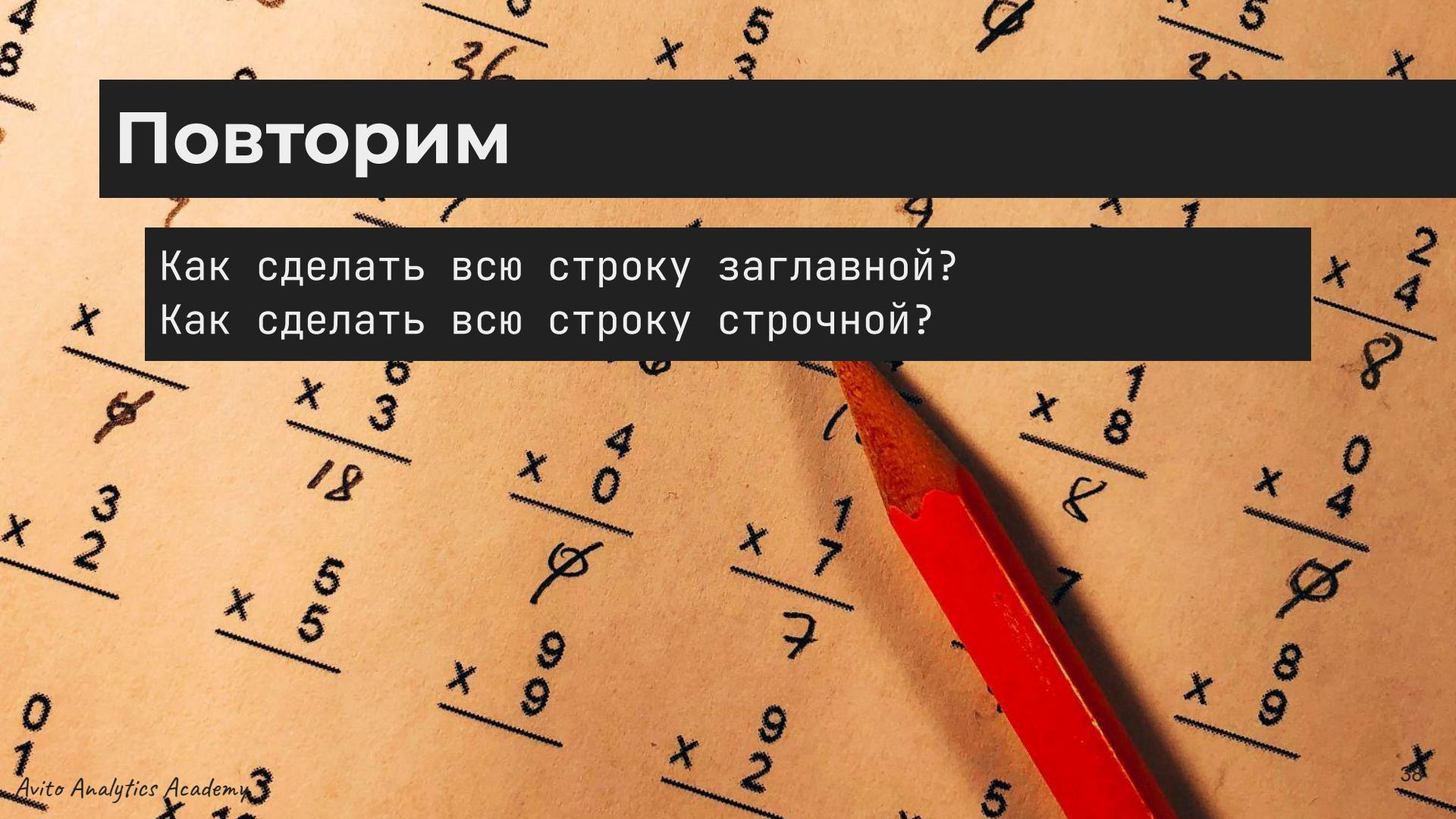
Как сделать всю строку заглавной?



Повторим

Как сделать всю строку заглавной?

Как сделать всю строку строчной?

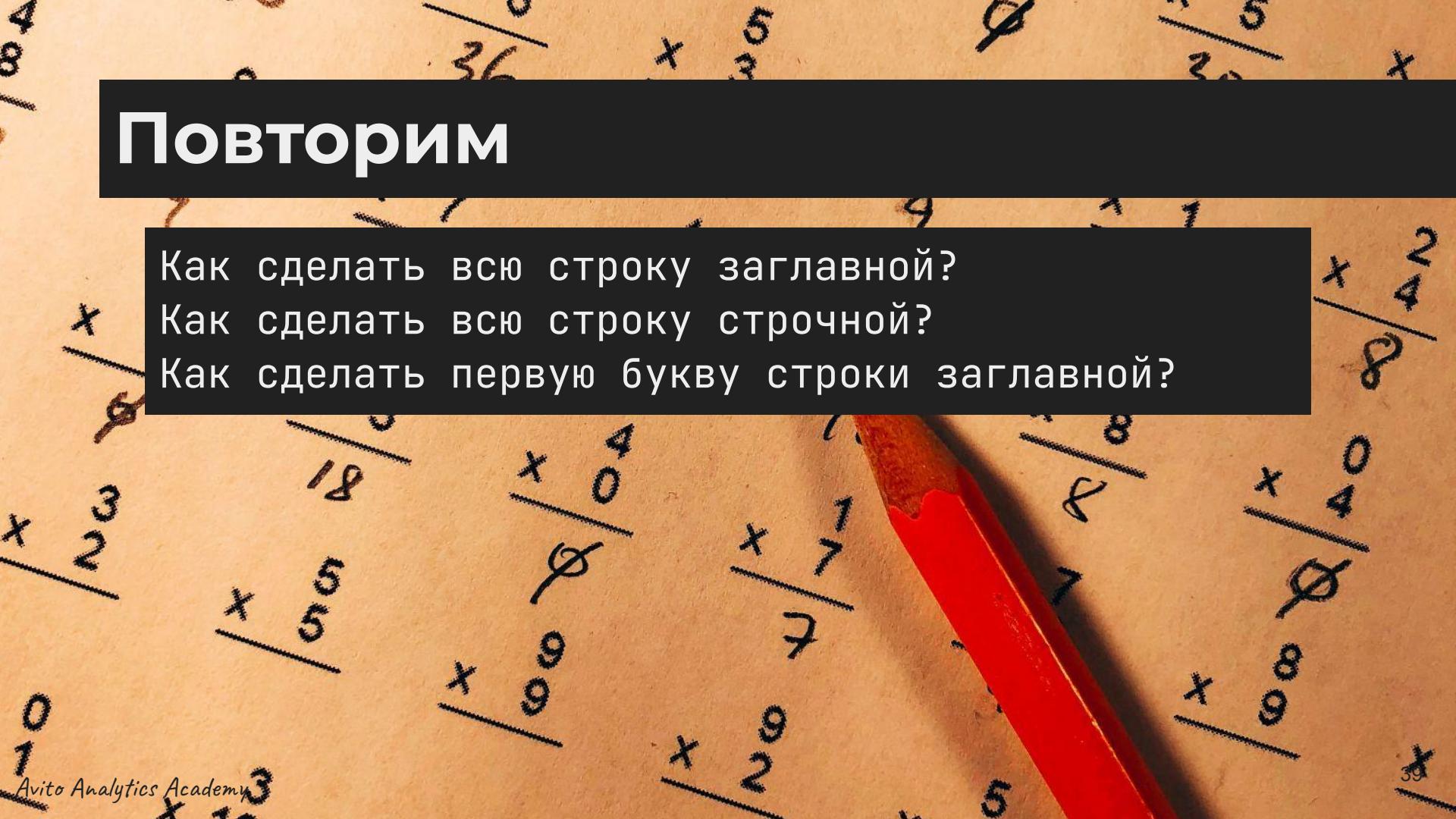


ПОВТОРИМ

Как сделать всю строку заглавной?

Как сделать всю строку строчной?

Как сделать первую букву строки заглавной?



ПОВТОРИМ

Как сделать всю строку заглавной?

Как сделать всю строку строчной?

Как сделать первую букву строки заглавной?

Какие виды форматирования запомнили?

bool

~Boolean - логический тип данных. Имеет только 2 значения:

- True
- False

Алиас к 1 и 0

bool

```
# проверка значения bool осуществляется через 'is'

>>> foo = True
>>> foo is not False
True

# примеры "алиасовости"

>>> True - 1 == False
>>> True - 1 == 0
>>> True + False == 1
```

Проверка на bool

True
False

= любое "ненулевое" значение
= любое "нулевое" значение

Нулевые значения

None
int
float
string
bool
list
dict
set

= None
= 0
= 0.0
= "" (пустая строка)
= False
= [] (пустой лист)
= {} или dict() (пустой словарь)
= set() (пустой сет)

Булевые операторы



and
or
not

= логическое "и"
= логическое "или"
= логическое отрицание

Булевые операторы



```
# boolean операторы ленивые

is_cozy = True
is_cheap = False
has_wifi = True

if is_cozy or is_cheap and has_wifi:
    print('This one is ok! 🍍')

if has_wifi or not is_cozy:
    print('No way!')
```

Булевые операторы



```
# можно использовать скобки
```

```
is_cozy = True  
is_cheap = False  
has_wifi = True
```

```
if not (has_wifi or is_cozy):  
    print('No way!')
```

Булевые операторы



```
# а если специально возвращать False?
```

```
class Foo
    def __bool__(self):
        return False

foo = Foo()
print(bool(foo) is False) # True
```

None █ :

```
# None - отсутствие значения  
# проверяется как и bool, через 'is'  
  
->>> None is None # True  
->>> None == None # True  
  
->>> birdie = 'duck'  
->>> birdie is not None # True
```

None □ :

может быть и значением 😎

```
>>> foo = {'bar': None}  
>>> foo.get('bar') is None # ключ существует  
>>> foo.get('key') is None # ключ не существует
```

Числа 100 :

int

float

complex

= int & long в других языках

= число с плавающей запятой

= комплексные

int

```
# в Python int - "универсальное" целое число

>>> foo = 42
>>> foo
42
>>> 42 ** 42
150130937545296572356771972164254457814047970
568738777235893533016064
```

float

```
# float - числа с плавающей запятой
```

```
>>> .42 == 0.42
```

```
True
```

```
>>> float(42) == 42.0
```

```
True
```

complex

```
# complex - комплексные числа  
# real + imag*1j
```

```
>>> 42j == complex(0, 42) == 0+42j  
True
```

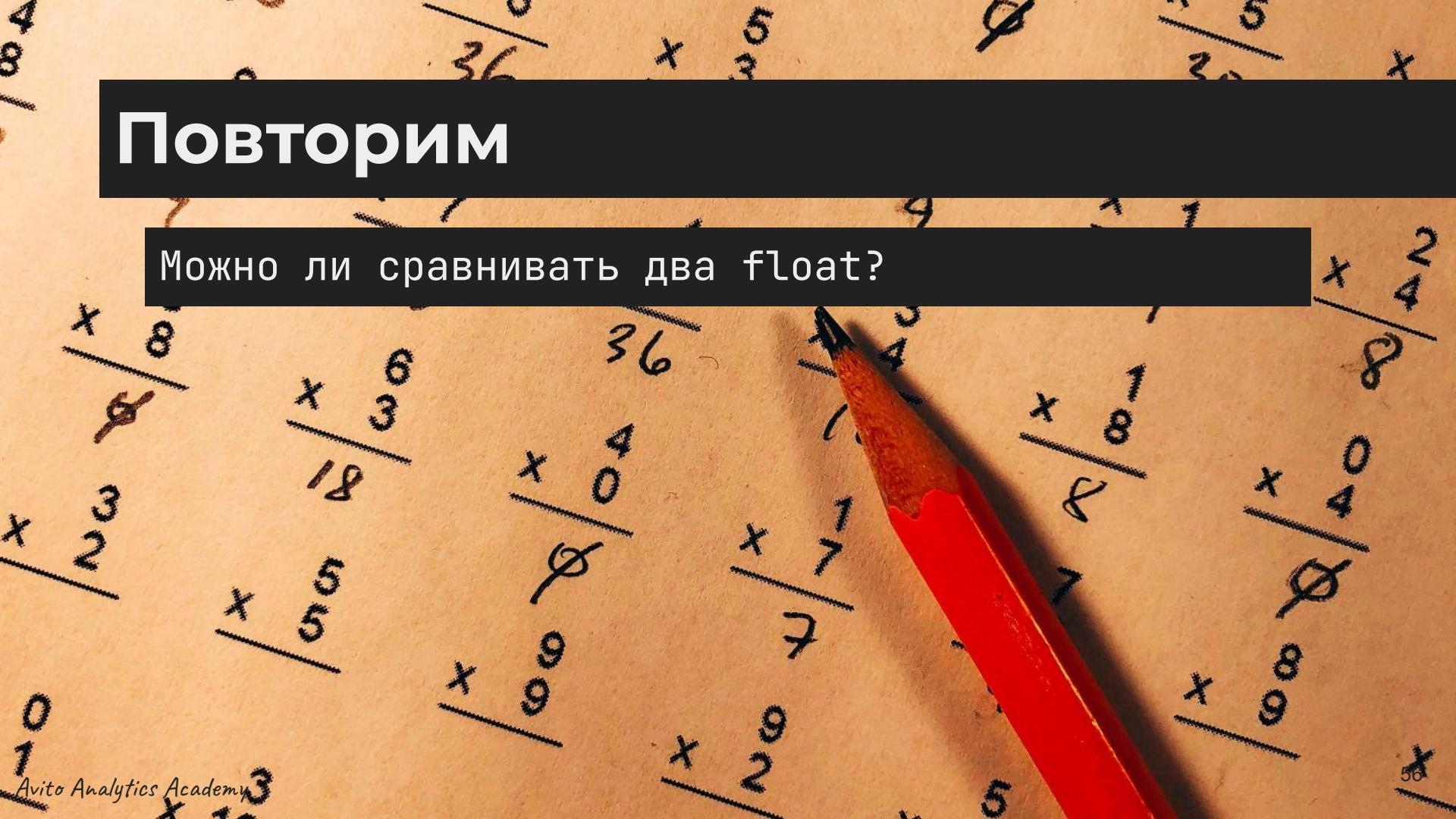
Операции над числами

```
# операции над числами

4 / 2 = 2.0      # деление
2 // 5 = 2        # целочисленное деление
5 // 2.0 = 2.0   # целочисленное деление
5 % 2 = 1         # остаток от деления
abs(-5) = 5       # абсолютное значение
2 ** 3 = 8        # возведение в степень
pow(2, 3) = 8    # возведение в степень
```

Повторим

Можно ли сравнивать два float?



ПОВТОРИМ

чему будет равно:

- $5 / 2$
- $5 // 2.0$
- $5 * 1.0$
- $5 + 1.0$

Кортежи

```
# tuple - кортеж
() = tuple()

# кортеж нельзя изменить
foo = (0, 1, 3)
foo[0] = 42    Out: TypeError
del foo[0]    Out:TypeError
```

Кортежи

```
# можно добавлять элементы и делать срезы,  
но будет создан новый объект
```

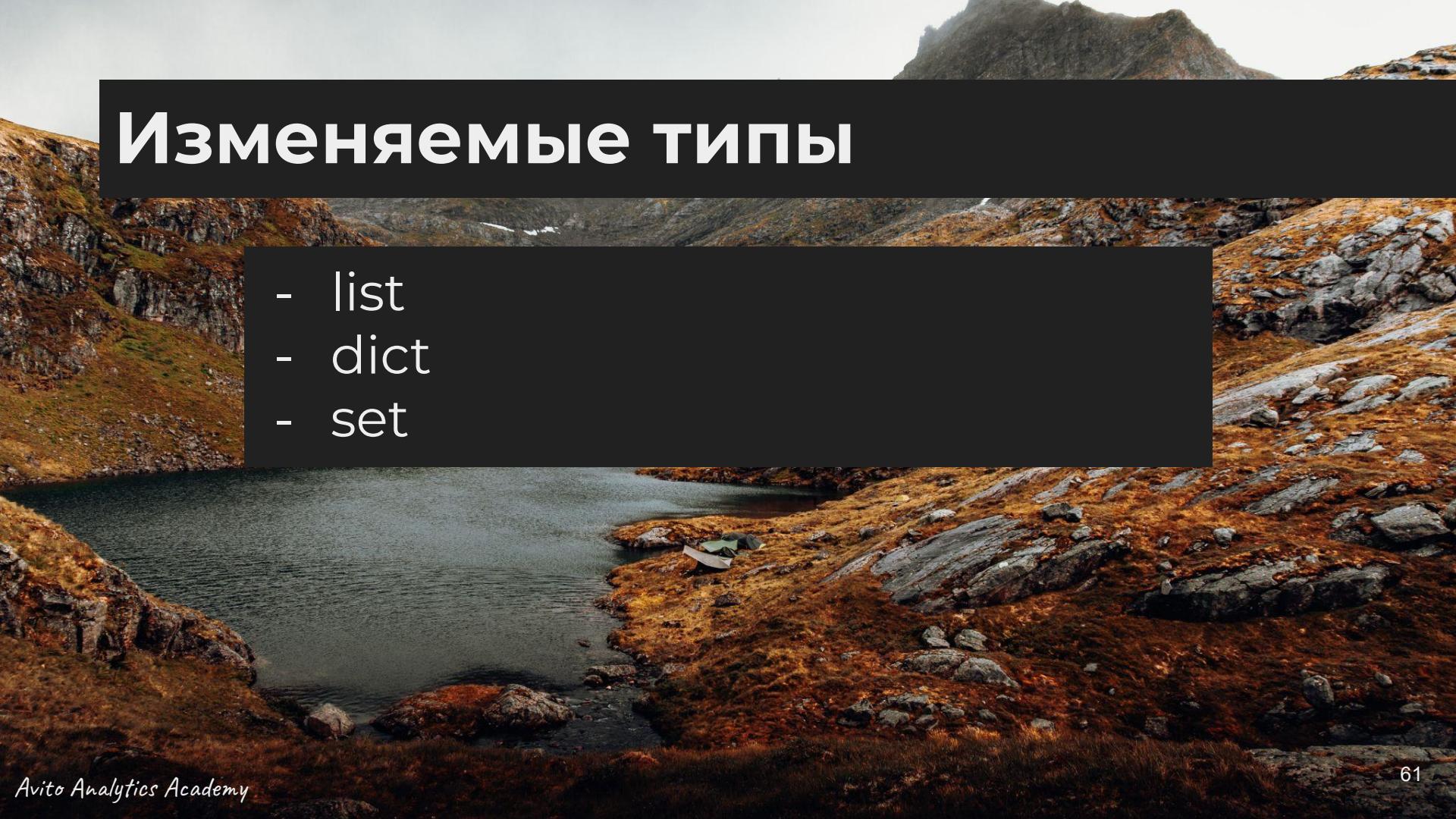
```
foo += (42, 'foo', 'bar')      # новый tuple  
foo[4:] = ('foo', 'bar')  
len(foo) = 6 and 42 in foo
```

ПЕРЕРЫВ



Изменяемые типы

- list
- dict
- set



Списки

list - список, упорядоченный набор элементов.
изменяемый. все элементы списка имеют индекс,
нумерация начинается с нуля.

```
x = list()  
x = []  
x = ['a', 'b', 'c']  
# проверка вхождения элемента в список  
print('a' in x) # True
```

Списки

```
# списки - изменяемый тип
foo = [0, 1, 2]
bar = [foo] * 3 # [[0, 1, 2], [0, 1, 2], [0, 1, 2]]
```

```
# переназначаем элемент с индексом [0][0]. т.к. тип
изменяемый, то поменялся и исходный объект
bar[0][0] = 9
```

```
foo = [9, 1, 2]
bar = [[9, 1, 2], [9, 1, 2], [9, 1, 2]]
```

Срезы

```
# списки позволяют делать срезы - выборку элементов  
по их индексам
```

```
x[start:end:step]
```

```
# start - с какого символа выбираем элементы
```

```
# end - до какого символа выбираем элементы
```

```
# step - шаг
```

Срезы

```
# списки позволяют делать срезы - выборку элементов по их индексам
```

```
x = ['a', 'b', 'c', 'd', 'e', 'f']
```

```
# выбираем все, кроме первого
```

```
x[1:] = ['b', 'c', 'd', 'e', 'f']
```

```
# выбираем все, с первого по четвёртый
```

```
x[1:4] = ['b', 'c', 'd']
```

Срезы

```
# списки позволяют делать срезы - выборку элементов по их индексам
```

```
x = ['a', 'b', 'c', 'd', 'e', 'f']
```

```
# выбираем каждый второй из интервала [1,4]
x[1:4:2] = ['b', 'd']
```

```
# такой срез инвертирует список
x[::-1] = ['f', 'e', 'd', 'c', 'b', 'a']
```

Операции со списками

```
x = ['a', 'b', 'c', 'd']
```

```
# добавляет элемент в конец списка
```

```
x.append('e')
```

```
x = ['a', 'b', 'c', 'd', 'e']
```

```
# расширяет список элементами из iterable-объекта
```

```
x.extend(['j', 'h', 'i'])
```

```
# x += ['j', 'h', 'i']
```

```
x = ['a', 'b', 'c', 'd', 'e', 'j', 'h', 'i']
```

Операции со списками

```
x = ['a', 'b', 'c', 'd']
```

```
# вытаскивает элемент списка по индексу и возвращает его
x.pop(0) # 'a'
x = ['b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
```

Словари

```
# dict - упорядоченная структура данных, состоящая из пар  
"ключ" - "значение"  
  
x = dict()  
x = {}  
x = {  
    'cat': 'meow',  
    'duck': 'quack',  
    'unicorn': 'does not exist',  
}  
print(x['duck']) # quack
```

Словари

Python 3.5

Упорядоченность ключей в словаре отсутствует.

Для получения упорядоченного словаря возможно использование словаря из модуля `collections`:

```
from collections import OrderedDict
```

Python 3.6

Порядок – частная реализация CPython.

Python 3.7

Порядок – часть языка.

Операции со словарями

```
# проверка на наличие элемента
```

```
>>> x = {'a': 1, 'b': 2}
```

```
>>> 'a' in x
```

```
True
```

Операции со словарями

```
# update  
  
x = {'a': 1, 'b': 2}  
  
# добавляет элементы в словарь  
x.update({'c': 3}) # {'a': 1, 'b': 2, 'c': 3}  
x.update(d=4) # {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

Операции со словарями

```
original = {'a': 1, 'b': 2}

# копирует ссылку на исходный объект
new = original # {'a': 1, 'b': 2}

# удаляет все элементы из словаря
new.clear()
print(new)      # {}
print(original) # {}
```

Операции со словарями

```
original = {'a': 1, 'b': 2}

# копирует ссылки на элементы исходного словаря в
# новый объект
new = original.copy() # {'a': 1, 'b': 2}

new.clear()
print(new)          # {}
print(original) # {'a': 1, 'b': 2}
```

Операции со словарями

```
sounds = {'duck': 'quack', 'cat': 'meow'}
```

```
# возвращает значение по ключу
duck_sound = sounds.get('duck', None) # 'quack'
print(sounds) # {'duck': 'quack', 'cat': 'meow'}
```

```
# возвращает элемент словаря по ключу и удаляет его
duck_sound = sounds.pop('duck') # 'quack'
print(sounds) # {'cat': 'meow'}
```

```
# возвращает последний item и удаляет его
last_sound = sounds.popitem() # ('cat', 'meow')
print(sounds) # {}
```

Операции со словарями

```
# setdefault

sounds = {'duck': 'quack', 'cat': 'meow'}

# если значение не существует в словаре - добавляет его
# возвращает значение добавляемого/существующего элемента
sounds.setdefault('cat', 'wuf')
print(sounds) # {'duck': 'quack', 'cat': 'meow'}

sounds.setdefault('t-rex', 'rawr')
print(sounds) # {'duck': 'quack', 'cat': 'meow', 't-rex': 'rawr'}
```

Операции со словарями

```
# keys/values/items

sounds = {'duck': 'quack', 'cat': 'meow'}

sounds.keys()
print(sounds) # dict_keys(['duck', 'cat'])

sounds.values()
print(sounds) # dict_values(['quack', 'meow'])

sounds.items()
print(sounds) # dict_items([('duck', 'quack'), ('cat', 'meow')])
```

Множества

```
# set - неупорядоченный набор уникальных элементов
```

```
x = set()  
y = {'a', 'b'}
```

Операции с множествами

```
x = {'a', 'b'}
```

```
# добавляет элементы в множества, поддерживает iterable  
x.update(['c']) # {'a', 'b', 'c'}
```

```
# добавляет элемент в множество  
x.add('d') # {'a', 'b', 'c', 'd'}
```

```
# удаляет все элементы из множества  
x.clear()
```

Операции с множествами

```
x = {'a', 'b', 'c', 'd'}
```

```
# удаляет элемент из множества по значению  
x.remove('d') # {'a', 'b', 'c'}
```

```
# удаляет элемент из множества по значению, если он там есть  
x.discard('d') # {'a', 'b', 'c'}
```

```
# удаляет случайный элемент из множества и возвращает его  
x.pop() # {'a', 'c'} → 'b'
```

Математические операции

```
sounds = {'quack', 'bark', 'meow', 'rawr'}  
bird_sounds = {'quack'}
```

```
# проверяет, что другой сет является подмножеством  
bird_sounds.issubset(sounds)
```

```
# проверяет, что другой сет является надмножеством  
sounds.issuperset(bird_sounds)
```

Математические операции

```
sounds = {'quack', 'meow', 'rawr'}  
bird_sounds = {'quack'}  
  
# возвращает пересечение множеств  
bird_sounds.intersection(sounds) # {'quack'}  
  
# возвращает True, если нет пересечений с объектом в параметре  
bird_sounds.isdisjoint([1]) # True  
  
# возвращает объединение множеств  
sounds.union(bird_sounds) # {'quack', 'meow', 'rawr'}
```

Математические операции

```
sounds = {'quack', 'meow', 'rawr'}  
bird_sounds = {'quack'}  
  
# возвращает разность множеств  
sounds.difference(bird_sounds) # {'meow', 'rawr'}  
  
# возвращает симметрическую разность множеств  
sounds.symmetric_difference(bird_sounds) # {'meow', 'rawr'}
```

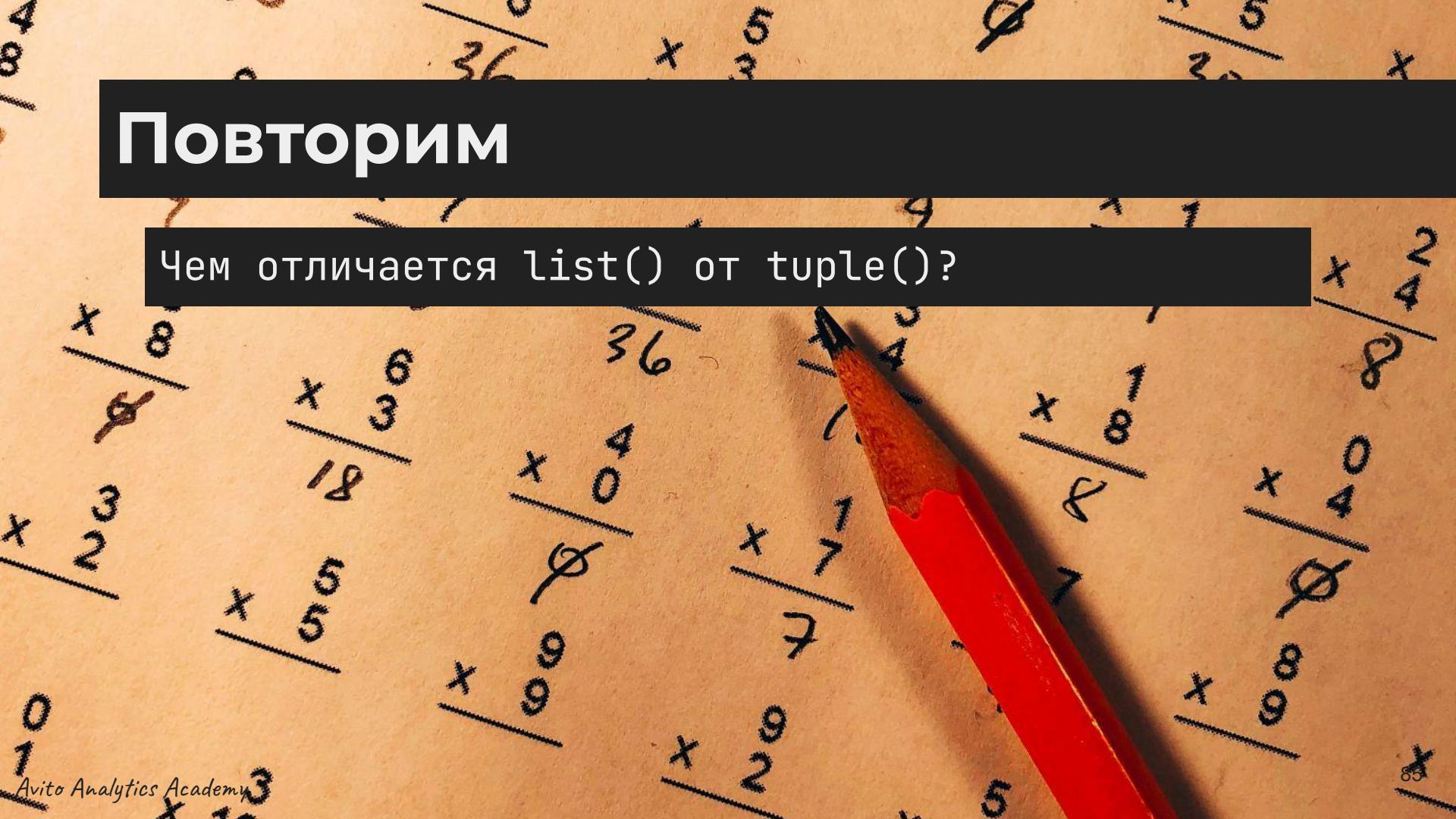
Нулевые значения

None
int
float
string
bool
list
dict
set

= None
= 0
= 0.0
= "" (пустая строка)
= False
= [] (пустой лист)
= {} или dict() (пустой словарь)
= set() (пустой сет)

ПОВТОРИМ

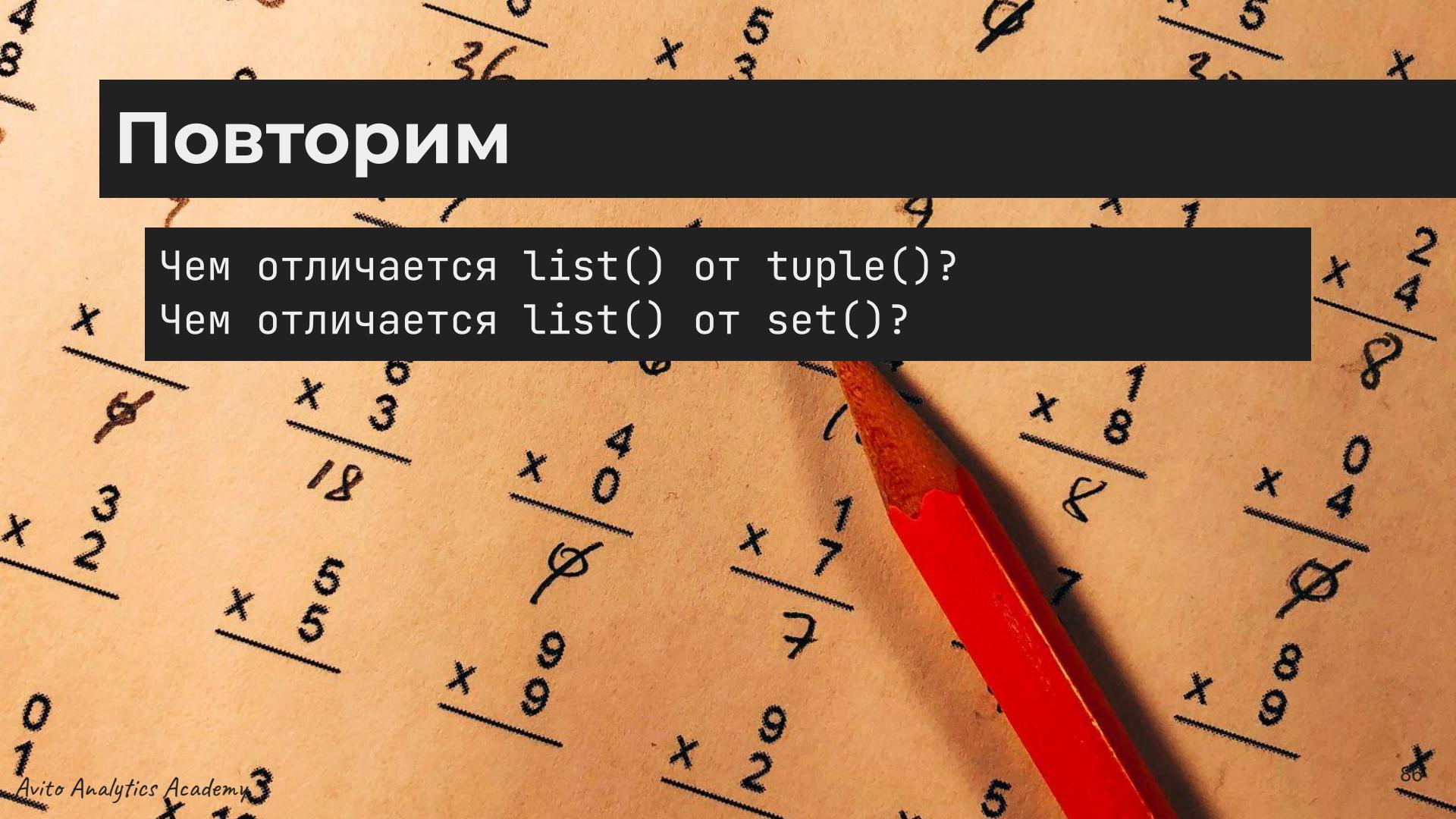
Чем отличается `list()` от `tuple()`?



ПОВТОРИМ

Чем отличается list() от tuple()?

Чем отличается list() от set()?



Повторим

Чем отличается list() от tuple()?

Чем отличается list() от set()?

Чем отличается set() от dict()?

ПОВТОРИМ

Чем отличается `list()` от `tuple()`?

Чем отличается `list()` от `set()`?

Чем отличается `set()` от `dict()`?

Как создать пустое множество?

Управляющие конструкции

условия

- if

циклы

- while
- for

Условия

```
if <что-то, что возвращает bool>:  
    <делаем что-то>  
  
elif <что-то, что возвращает bool>:  
    <делаем что-то другое>  
  
else:  
    <делаем что-то для всех остальных случаев>
```



Условия

```
quacks_like_a_duck = True
swims_like_a_duck = True
looks_like_a_duck = True

if quacks_like_a_duck and swims_like_a_duck and looks_like_a_duck:
    print('Это утка')
elif not quacks_like_a_duck:
    print('Странно оно крякает')
else:
    print('Это не утка')
```

Цикл while

```
# повторяет описанные в теле цикла инструкции, пока  
условие истинно
```

```
while True:  
    print('я никогда не завершусь')
```

Цикл while

```
# для while цикла должно существовать условие выхода,  
иначе он будет выполняться бесконечно
```

```
x = 1  
while x < 3: # цикл прервется при x=3  
    print(x)  
    x += 1
```

Цикл while

```
# для while цикла должно существовать условие выхода,  
иначе он будет выполняться бесконечно
```

```
x = 1  
while x < 3:  
    print(x)  
    if x == 2:  
        break # на моменте x==2 цикл прервется  
    x += 1
```

Цикл for

```
# цикл for проходит по элементам iterable объекта

for i in range(10):
    print(i)

for sound in ['quack', 'meow', 'whistle']:
    print(sound)
```

Цикл for

```
# можно итерироваться, например по спискам
for i in ['раз', 'два', 'три']:
    print(f'Печатаем элемент {i}')
```

```
# для доступа к индексу есть функция enumerate
for ind, val in enumerate(['раз', 'два', 'три']):
    print(f'Элемент с номером {ind}: {val}')
```

Цикл for

```
# по словарям тоже можно итерироваться
for k, v in {'foo': 1, 'bar': 2, 'baz': 3}.items():
    print(f'ключ: {k}, значение: {v}')
# можно итерироваться и только по значениям...
for v in {'foo': 1, 'bar': 2, 'baz': 3}.values():
    print(f'значение: {v}')
# ...и только по ключам. ~ .keys()
for k in {'foo': 1, 'bar': 2, 'baz': 3}:
    print(f'ключ: {k}')
```

Comprehensions

```
# list comprehensions
x = [k for k in range(10) if k%3 == 0]

# dictionary comprehensions
y = {k: 10 for k in range(5)}

# set comprehensions
z = set(k for k in ['apple', 'lemon', 'orange'] if k == 'apple')
```



ПОВТОРИМ

Что делает этот код?

```
while True:  
    print('Привет, Мир!👋')
```

ПОВТОРИМ

```
# Так не зацикливается
i = 1
while i <= 10:
    print('Привет, Мир!👋')
    i += 1
```

Package management

- ★ PIP - package installer for Python (<https://pypi.org/project/pip/>)
- Poetry (<https://python-poetry.org/>)

Package management

```
# установить пакет(ы)
$ pip install -r requirements.txt

Collecting requests==2.26.0
  Using cached requests-2.26.0-py2.py3-none-any.whl (62 kB)

Attempting uninstall: requests
  Found existing installation: requests 2.24.0
  Uninstalling requests-2.24.0:
    Successfully uninstalled requests-2.24.0
  Successfully installed requests-2.26.0
```

Package management

```
# установить пакеты из файла requirements.txt  
# requests==2.26.0  
# PyYAML≥5.4.1  
# ./local-wheels/ABC-0.0.2-py3-none-any.whl  
  
$ pip install -r requirements.txt
```

Установить iPython

```
# установить iPython
```

```
$ pip install iPython
```

Откуда брать данные?

- Файлики
- API
- Базы данных
- Ручной ввод

Откуда брать данные?

```
# чтение из файла

import json

with open('duck_tales.json', 'r') as f:
    raw_data = json.load(f)
    print(x)
```

Откуда брать данные?

```
# чтение из открытого API

import requests

response =
requests.get('https://www.7timer.info/bin/astro.php?lo
n=113.2&lat=23.1&ac=0&unit=metric&output=json&tzshift=
0')

print(response.json())
```

Откуда брать данные

```
# ручной ввод
```

```
input()
```

Домашка



```
# Guido van Rossum <guido@python.org>

def step1():
    print(
        'Утка малая 🦆 решила выпить зайти в бар. '
        'Взять ей зонтик? ☂ '
    )
    option = ''
    options = {'да': True, 'нет': False}
    while option not in options:
        print('Выберите:/{}/{}'.format(*options))
        option=input()

    if options[option]:
        return step2_umbrella()
    return step2_no_umbrella()

if __name__ == '__main__':
    step1()
```

Домашка



```
# Guido van Rossum <guido@python.org>

def step1(): # ← определение функции
    print(
        'Утка малая 🦆 решила выпить зайти в бар. '
        'Взять ей зонтик? 🌂 '
    )
    option = ''
    options = {'да': True, 'нет': False}
    while option not in options:
        print('Выберите:/{}/{}'.format(*options))
        option=input()

    if options[option]:
        return step2_umbrella() # ← возвращаем значение
    return step2_no_umbrella() # ← возвращаем значение

if __name__ == '__main__': # ← конструкция, ограничивающая исполнение
    step1() # ← вызов функции
```

Что дальше?

- Следующая большая тема - функции
- Но перед этим практика по сегодняшней теме
- Срок по домашке - до понедельника
- Методичка и презентация выложены на Stepic

Photos from unsplash.com

@eberhardgross
@evgenievgenief
@jonathanborba
@timothymeinberg
@danieljschwarz
@m_gatus
@shotbyrain
@davidclode
@catrionaobrian
@munjay
@chrisliverani
@anniespratt
@sinout

