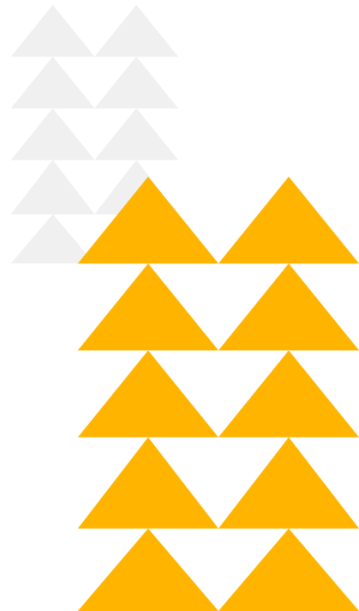
О чём поговорим?

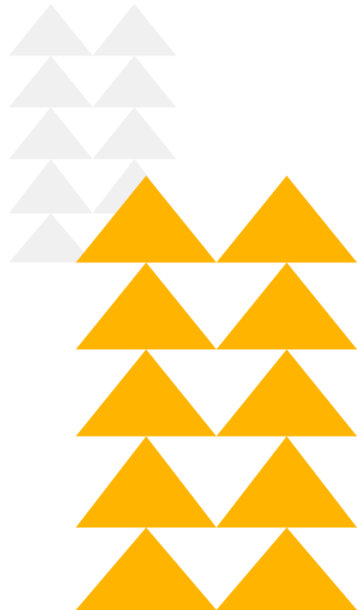
- Кто, когда и зачем придумал язык Python
- 2 или 3
- Интерпретаторы языка Python



Foreword for “Programming Python” (1st ed.)

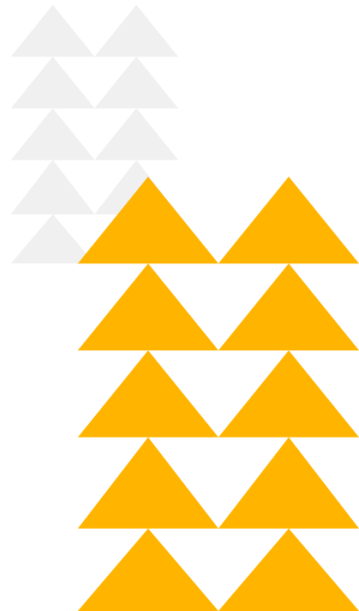
in December 1989, I was looking for a “hobby” programming project that would keep me occupied during the week around Christmas. My office […] would be closed, but I had a home computer, and not much else on my hands

Guido van Rossum



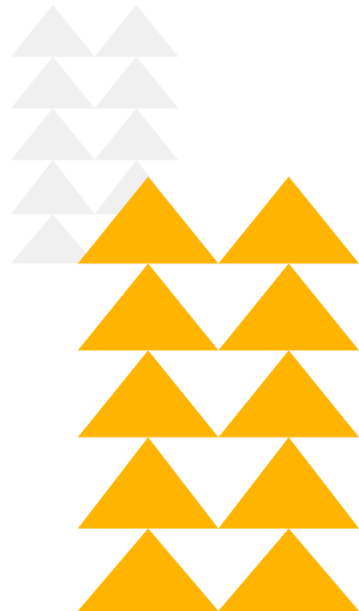
Источники вдохновения (ABC)

```
01. HOW TO RETURN words document:
02.     PUT {} IN collection
03.     FOR line IN document:
04.         FOR word IN split line:
05.             IF word not.in collection:
06.                 INSERT word IN collection
07.     RETURN collection
```



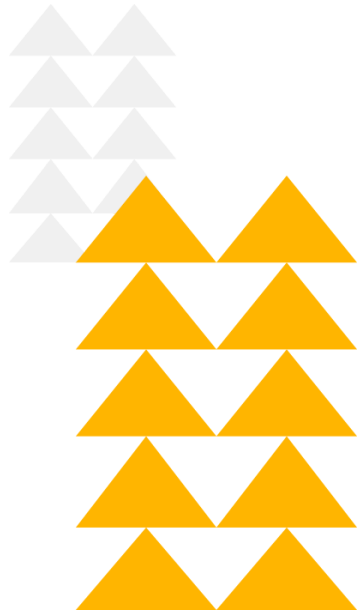
Источники вдохновения (Modula-3)

```
01. TRY
02.     DO.Something()
03. EXCEPT
04.     | IO.Error => IO.Put("An I/O error occurred.")
05. END;
```



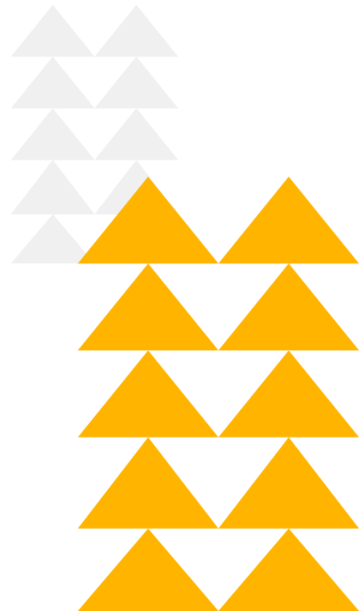
Какой язык хотел создать автор?

Простой, понятный, удобный и полезный язык с открытым исходным кодом



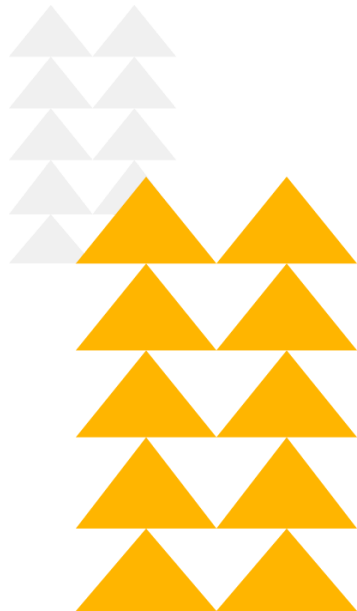
Что получилось в итоге?

```
01. def walk_recursively(dir):
02.     visited = []
03.     for root, dirs, files in os.walk(dir):
04.         visited.extend(
05.             os.path.join(root, file) for file in files
06.         )
07.     return visited
```



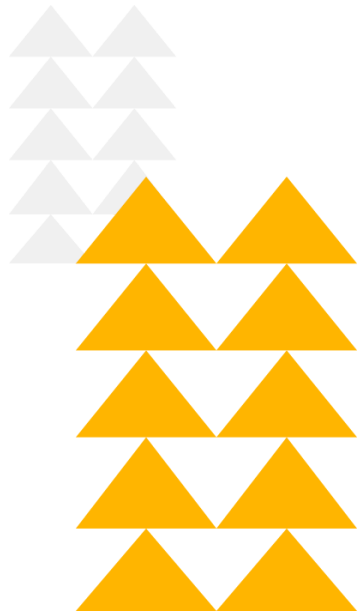
Динамический интепретируемый язык (1)

```
01. >>> def add(x, y):
02.         return x + y
03. >>> def bar(x):
04.         add(x, "1", "2")
05. >>> # ^^^ no error!
```



Динамический интепретируемый язык (2)

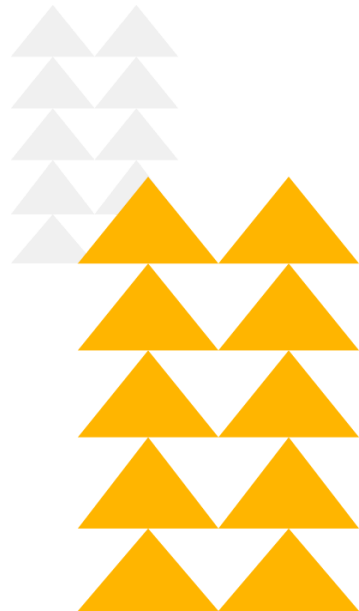
```
01. >>> add.__code__ = bar.__code__
02. >>> add(42)
03. Traceback (most recent call last):
04. File "<stdin>", line 1, in <module>
05. File "<stdin>", line 2, in bar
06. TypeError: bar() takes 1 positional argument but 3 [...]
```



Динамический **интепретируемый** язык

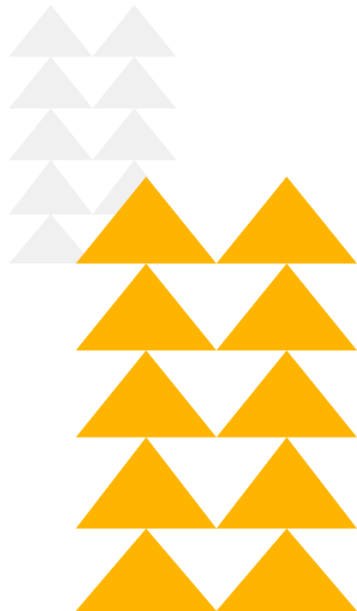
```
01. $ cat hello.py
02. message = "Hello, world!"
```

```
01. $ python -m dis hello.py
02.   3           0 LOAD_CONST               0 ('Hello, world!')
03.               2 STORE_NAME               0 (message)
04.               4 LOAD_CONST               1 (None)
05.               6 RETURN_VALUE
```



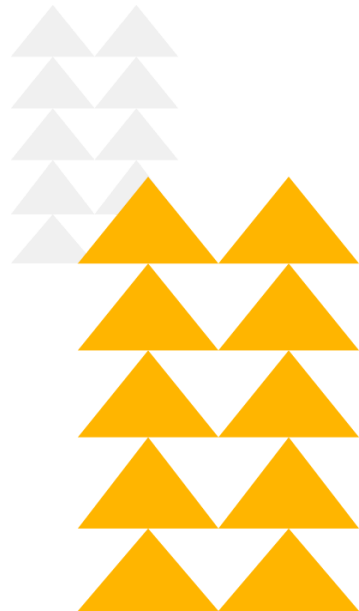
Интерпретаторы

- cpython
- pypy
- ironpython (C# VM)
- jython (Java VM)



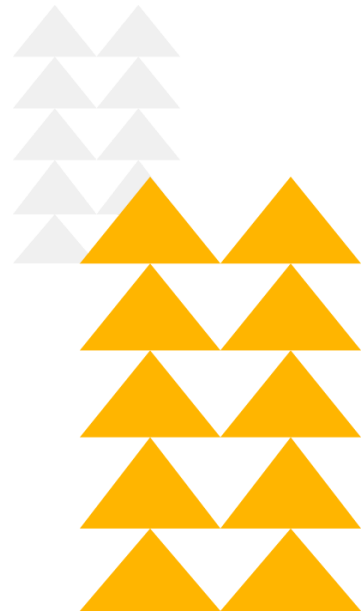
Отступы вместо скобочек (1)

```
>>> while True:
    print(42)
File "<stdin>", line 2
print(42)
^
IndentationError: expected an indented block
```



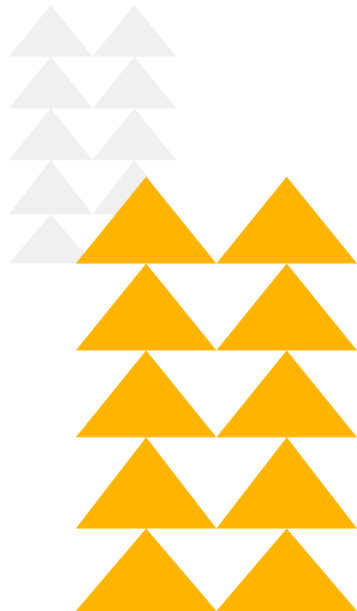
Всё есть объект (2)

```
01. >>> import hello
02. >>> type(hello)
03. <class 'module'>
04. >>> type(type(hello))
05. <class 'type'>
06. >>> type(type(type(hello)))
07. <class 'type'>
```



Типы языка (1)

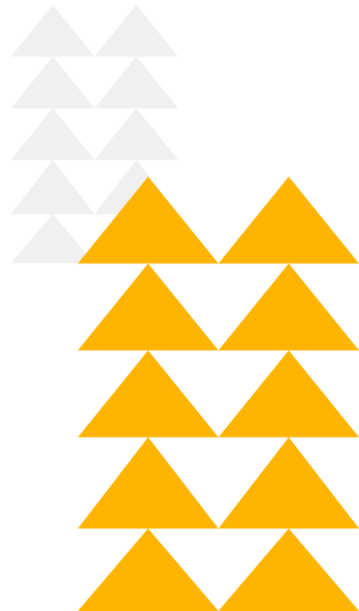
- Базовых типов не очень много:
  - `None`;
  - логические `True`, `False`;
  - числовые `int`, `float`, `complex`;
  - строковые `bytes`, `str`;
  - изменяемые коллекции `list`, `set`, `dict`,
  - неизменяемые коллекции `tuple`.



Типы языка (2)

Чаще всего над объектами из одной группы можно совершить конкретное действие одним очевидным способом

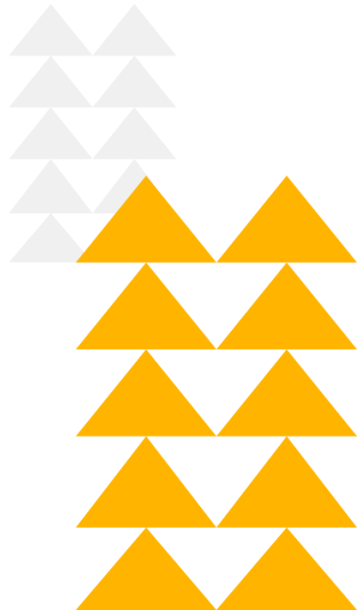
```
01. >>> 1 + 1
02. >>> 1. + 1.
03. >>> b"p" + b"y"
04. b'py'
05. >>> "p" + "y"
06. 'py'
```



В Python есть...

В Python есть многое из того, что так дорого программисту на императивном языке:

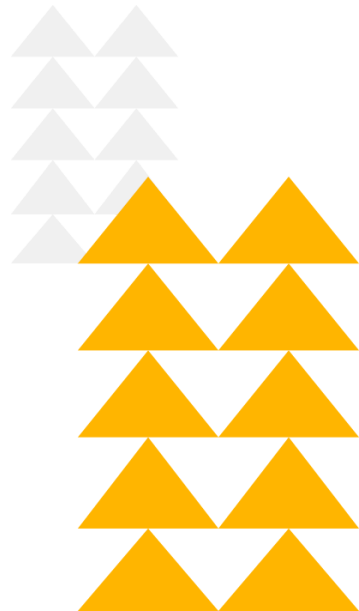
`if`, `for`, `while`, тернарный оператор.



В Python нет...

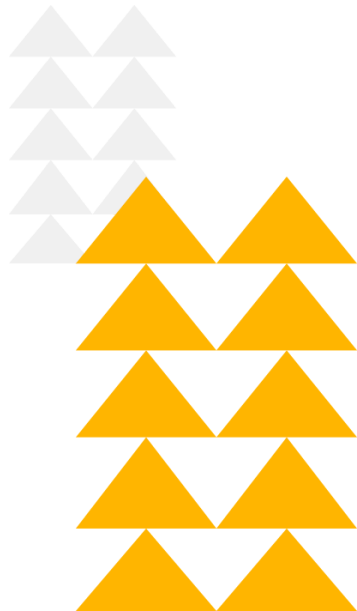
- циклов с пост-условием, потому что `while` вполне достаточно,
- операторов `switch` и `for` с явным счетчиком, потому что они имеют нетривиальную семантику,
- фигурных скобок для обозначения области видимости,

```
>>> from __future__ import braces
File "<stdin>", line 1
SyntaxError: not a chance
```



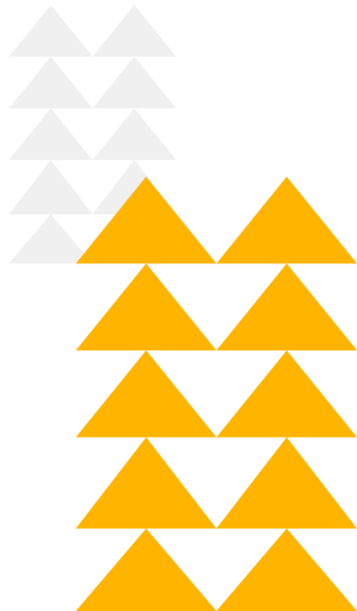
The Zen of Python, by Tim Peters

```
>>> 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.
```



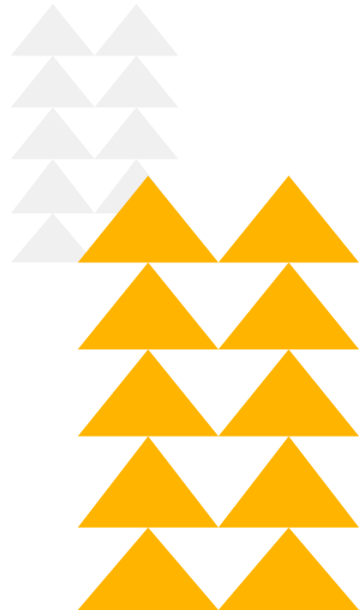
The Zen of Python, by Tim Peters (2)

```
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!
```



Функции в Python

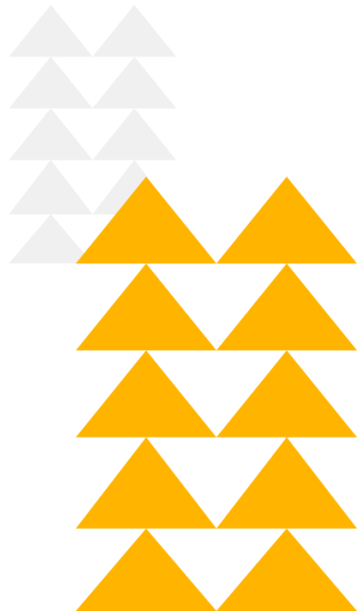
- Синтаксис объявления функций
- Упаковка и распаковка аргументов
- Ключевые аргументы и аргументы по умолчанию
- Распаковка и оператор присваивания
- Области видимости, правило LEGB, операторы global и nonlocal
- Функциональное программирование, анонимные функции
- Функции map, filter и zip
- Генераторы списков, множеств и словарей



Ограничения на имя функции

Можно использовать буквы, подчеркивание _ и цифры от 0 до 9, но цифра не должна стоять на первом месте.

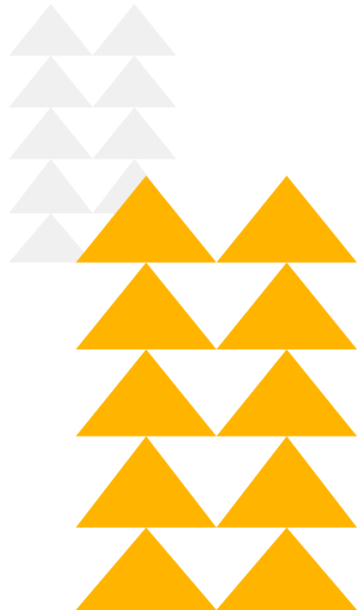
```
01. >>> def what_is_the_meaning_of_the_life():
02.         return 42
03.
04. >>> what_is_the_meaning_of_the_life()
05. 42
```



Синтаксис объявления функций (2)

`return` использовать не обязательно, по-умолчанию функция возвращает `None`

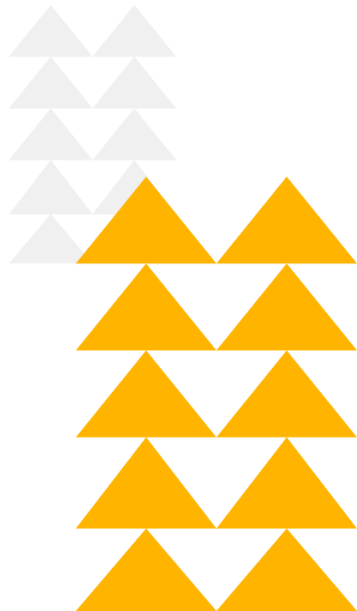
```
01. >>> def what_is_the_meaning_of_the_life():
02.         42
03.
04. >>> print(what_is_the_meaning_of_the_life())
05. None
```



Синтаксис объявления функций (3)

Для документации функции используются строковые литералы

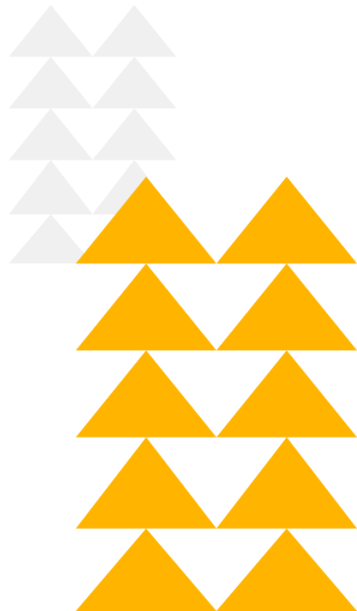
```
01. >>> def what_is_the_meaning_of_the_life():
02.         """I return 42"""
03.         return 42
```



Синтаксис объявления функций (4)

После объявления функции документация доступна через специальный атрибут `__doc__`

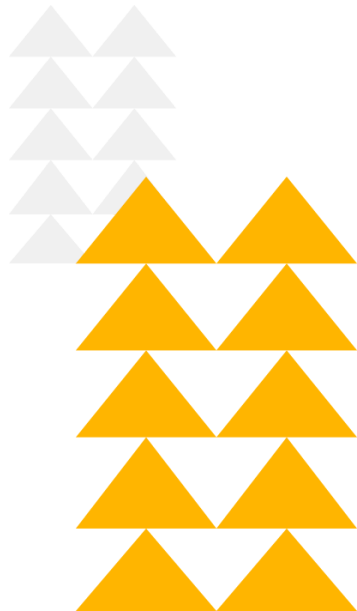
```
>>> what_is_the_meaning_of_the_life.__doc__
'I return 42'
```



Синтаксис объявления функций (5)

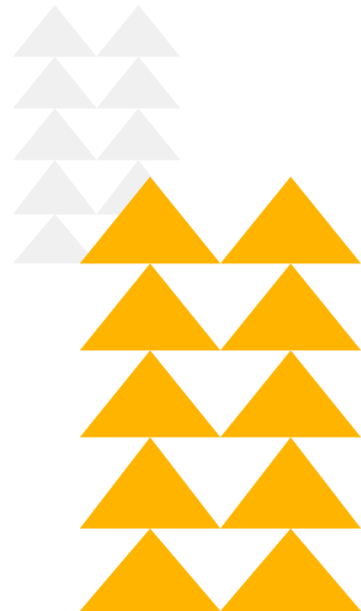
В интерпретаторе удобней пользоваться встроенной функцией `help`

```
>>> help(what_is_the_meaning_of_the_life)
```



Пример: min

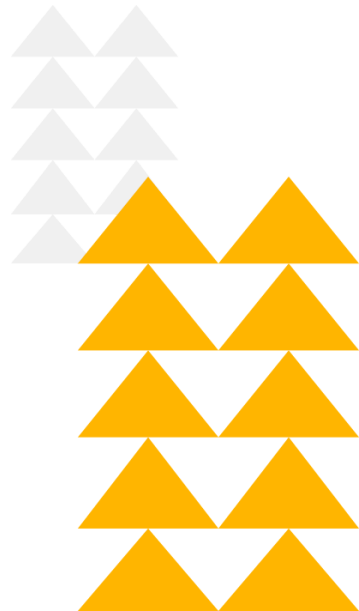
```
01. >>> def min(x, y):
02.         """Returns min of two values."""
03.         return x if x < y else y
04.
05. >>> min(-1, 4)
06. -1
07. >>> min(x=7, y=-7)
08. -7
09. >>> min(x=42, z=3)
10. Traceback (most recent call last):
11.   File "<stdin>", line 1, in <module>
12. TypeError: min() got an unexpected keyword argument 'z'
```



Упаковка аргументов (1)

Находим минимум произвольного количества аргументов

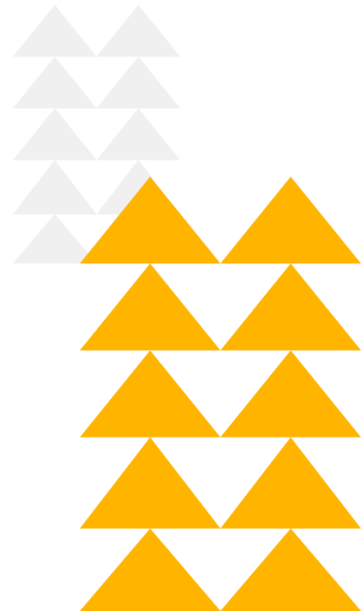
```
>>> def min(*values):
        min_value = float('inf')
        for value in values:
            if value < min_value:
                min_value = value
        return min_value

>>> min(1, 4, -5)
-5
```



Упаковка аргументов (2)

Как потребовать, чтобы в args был хотя бы один элемент?

```
01. >>> def min(first, *args):  # 'args' by convention
02.         min_value = first
03.         ...
04.
05. >>> min()
06. Traceback (most recent call last):
07. File "<stdin>", line 1, in <module>
08. TypeError: min() missing 1 required [...] argument: 'first'
```

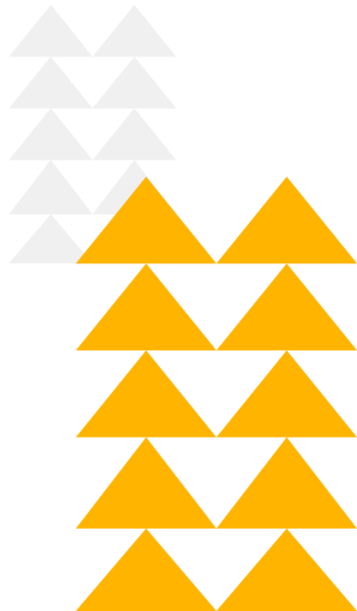


Упаковка аргументов (3)

Как применить функцию `min` к коллекции?

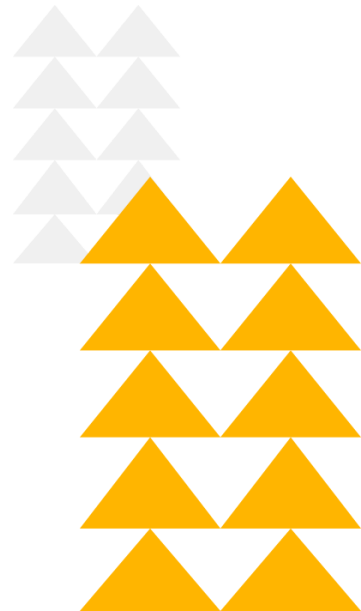
Синтаксис будет работать с любым объектом, поддерживающим протокол итератора.

```
>>> min(*{-5, 12, 13})
-5
>>> min(*[-5, 12, 13])
-5
>>> min(*(-5, 12, 13))
-5
```



Аргументы по умолчанию (1)

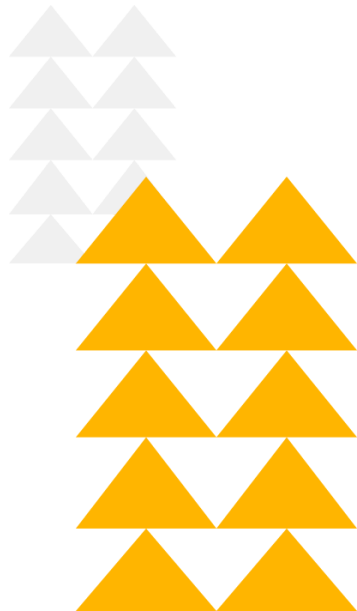
```
01. >>> def bounded_min(first, *args, lo=float("-inf"), hi=float("inf")):
02.         min_value = hi
03.         for value in (first, ) + args:
04.             if value < min_value and lo < value < hi:
05.                 min_value = value
06.         return max(min_value, lo)
07.
08. >>> bounded_min(-5, 12, 13, lo=0, hi=255)
09. 12
```



Аргументы по умолчанию (2)

В какой момент происходит инициализация ключевых аргументов со значениями по умолчанию?

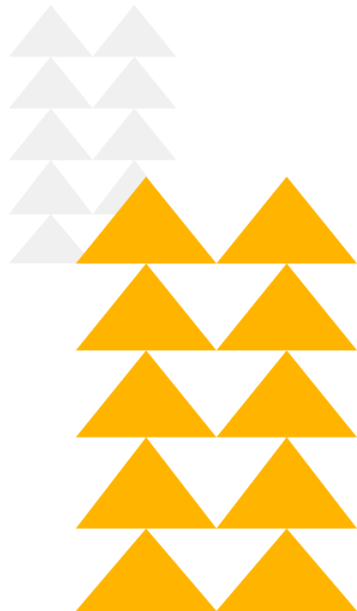
```
def unique(iterable, seen=set()):
    acc = []
    for item in iterable:
        if item not in seen:
            seen.add(item)
            acc.append(item)
    return acc
```



Аргументы по умолчанию (3)

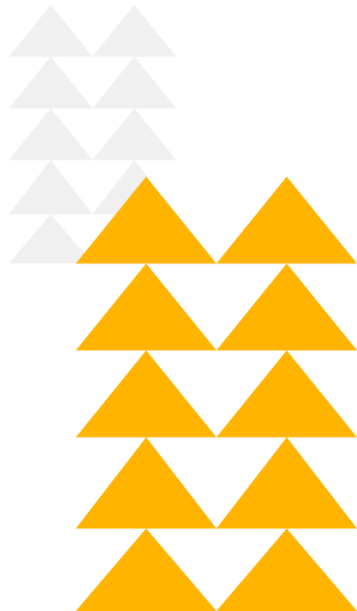
В какой момент происходит инициализация ключевых аргументов со значениями по умолчанию?

```
01. >>> xs = [1, 1, 2, 3]
02. >>> unique(xs)  # seen == {}
03. [1, 2, 3]
04. >>> unique(xs)  # seen == {1, 2, 3}
05. []
06. >>> unique.__defaults__
07. ({1, 2, 3},)
```



Правильная инициализация аргументов

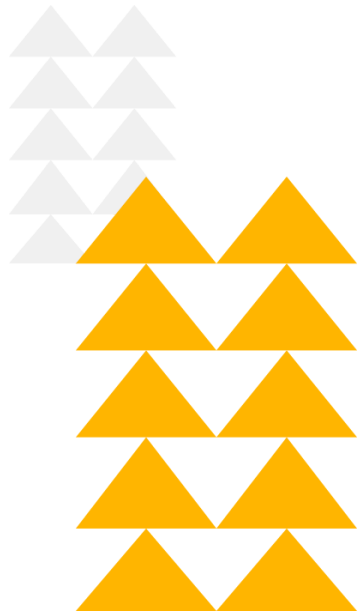
```
>>> def unique(iterable, seen=None):
        if seen is None:
            seen = set()
        acc = []
        for item in iterable:
            if item not in seen:
                seen.add(item)
                acc.append(item)
        return acc
>>> unique([1, 1, 2, 3])
[1, 2, 3]
>>> unique([1, 1, 2, 3])
[1, 2, 3]
```

Ключевые аргументы: и только ключевые

Если функция имеет фиксированную арность, то ключевые аргументы можно передавать без явного указания имени:

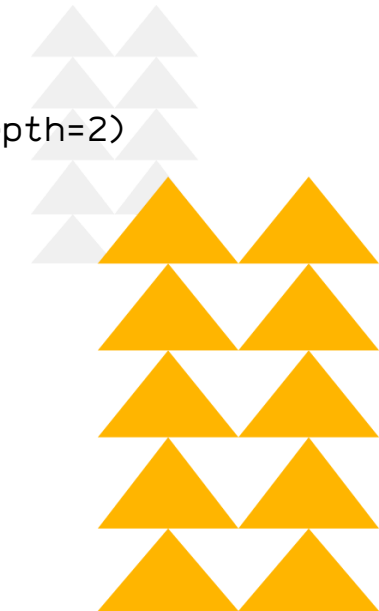
```
>>> def flatten(xs, depth=None):
    pass

>>> flatten([1, [2], 3], depth=1)
>>> flatten([1, [2], 3], 1)  # <- without name
```



Ключевые аргументы: и только ключевые

Можно явно потребовать, чтобы часть аргументов всегда передавалась как ключевые:

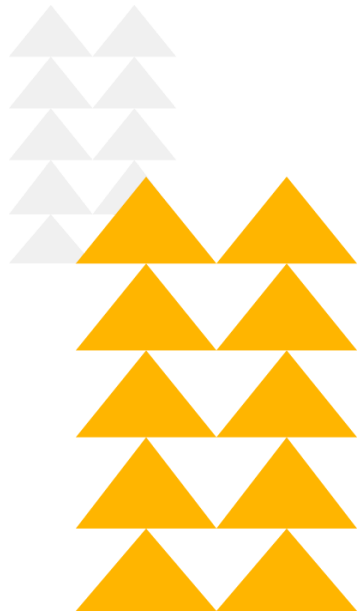
```
01. >>> def flatten(xs, *, depth=None):
02.     pass
03.
04. >>> flatten([1, [2], 3], 2)  # <- should be flatten([1, [2], 3], depth=2)
05. Traceback (most recent call last):
06. File "<stdin>", line 1, in <module>
07. TypeError: flatten() takes 1 positional argument [...]
```



Ключевые аргументы: упаковка и распаковка

Ключевые аргументы, аналогично позиционным, можно упаковывать и распаковывать:

```
01. >>> def runner(cmd, **kwargs):
02.     if kwargs.get("verbose", True):
03.         print("logging enabled")
04. >>> runner("mysqld", limit=42)
05. logging enabled
06. >>> runner("mysqld", **{"verbose": False})
07. >>> options = {"verbose": False}
08. >>> runner("mysqld", **options)
```

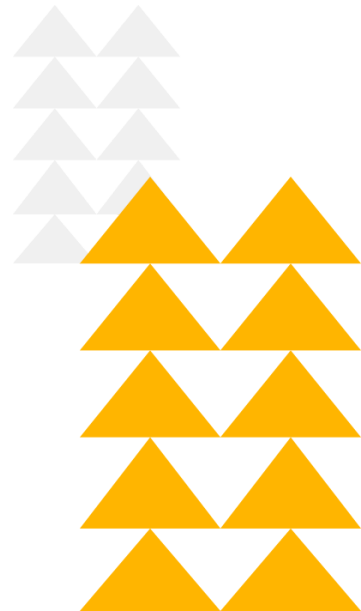


Распаковка и присваивание

```
01. >>> x, y, z = [1, 2, 3]
02. >>> x, y, z = {1, 2, 3} # unordered!
03. >>> x, y, z = "xyz"
```

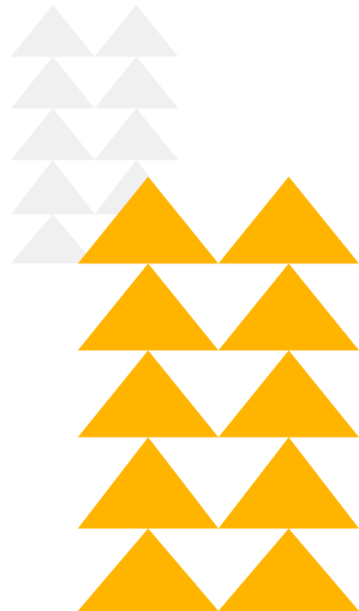
Скобки обычно опускают, но иногда они бывают полезны

```
01. >>> rectangle = (0, 0), (4, 4)
02. >>> (x1, y1), (x2, y2) = rectangle
```



Расширенный синтаксис распаковки

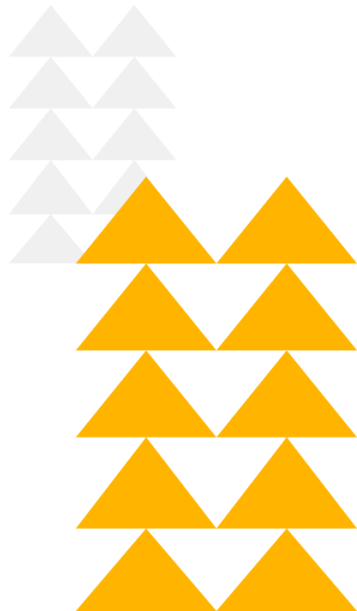
```
01. >>> first, *rest = range(1, 5)
02. >>> first, rest
03. (1, [2, 3, 4])
```



Расширенный синтаксис распаковки (2)

`*` можно использовать в любом месте выражения

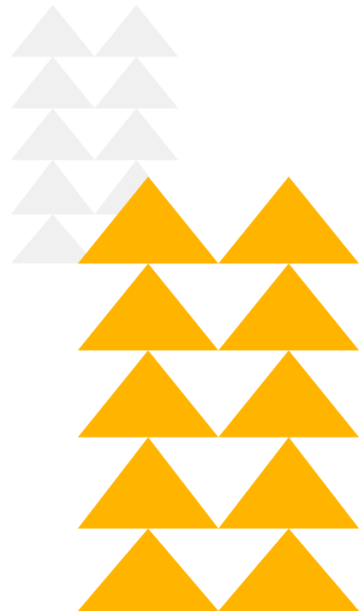
```
01. >>> first, *rest, last = range(1, 5)
02. >>> last
03. 4
04.
05. >>> first, *rest, last = [42]
06. Traceback (most recent call last):
07. File "<stdin>", line 1, in <module>
08. ValueError: need more than 1 values to unpack
```



Распаковка и цикл for

Синтаксис распаковки работает в цикле for, например:

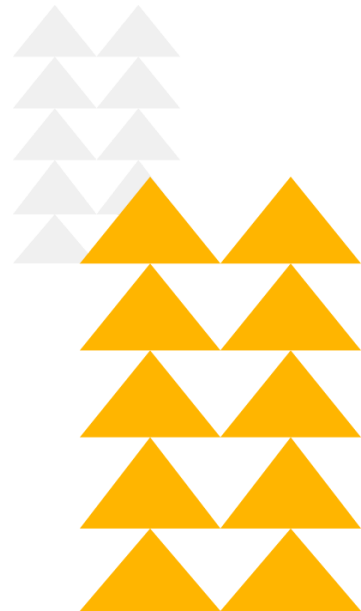
```
>>> for a, *b in [range(4), range(2)]:
        print(b)

[1, 2, 3]
[1]
```



Распаковка и байткод (1)

Присваивание в Python работает слева направо.

```
01. >>> dis.dis("first, *rest, last = ('a', 'b', 'c')")
02.   1           0 LOAD_CONST               4 (('a', 'b', 'c'))
03.               2 EXTENDED_ARG             1
04.               4 UNPACK_EX              257
05.               6 STORE_NAME               0 (first)
06.               8 STORE_NAME               1 (rest)
07.              10 STORE_NAME               2 (last)
08.              12 LOAD_CONST               3 (None)
09.              14 RETURN_VALUE
```

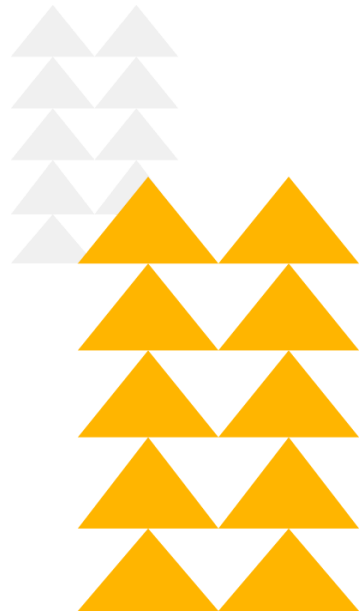


Распаковка и байткод (2)

Присваивание в Python работает слева направо.

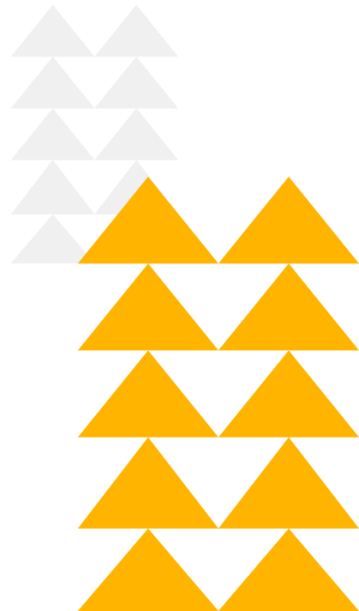
```
01. >>> x, (x, y) = 1, (2, 3)
02. >>> x  # ?
```

```
2
```



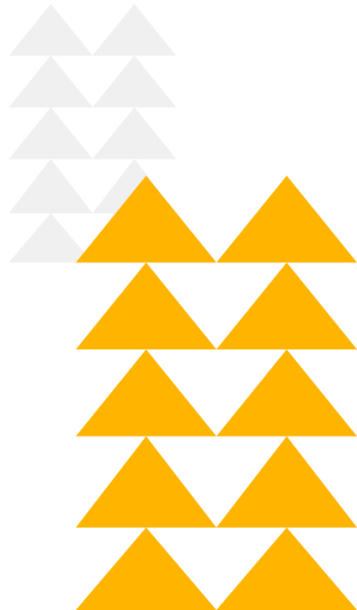
Распаковка и байткод (3)

```
01. >>> dis.dis("first, *rest, last = ['a', 'b', 'c']")
02.   1           0 LOAD_CONST               0 ('a')
03.               2 LOAD_CONST               1 ('b')
04.               4 LOAD_CONST               2 ('c')
05.               6 BUILD_LIST               3
06.               8 EXTENDED_ARG             1
07.              10 UNPACK_EX              257
08.              12 STORE_NAME               0 (first)
09.              14 STORE_NAME               1 (rest)
10.              16 STORE_NAME               2 (last)
11.              18 LOAD_CONST               3 (None)
12.              20 RETURN_VALUE
```

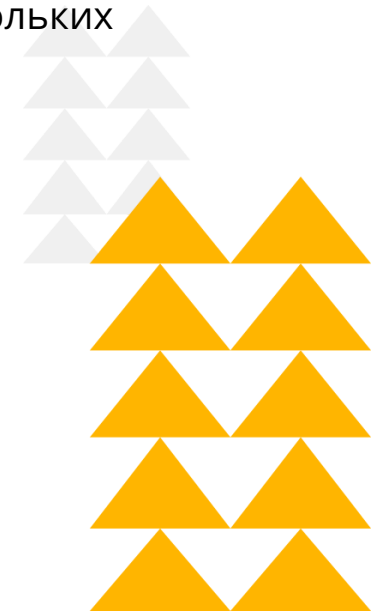
Распаковка и байткод (4)

Синтаксически схожие конструкции могут иметь различную семантику времени исполнения.



Take away: функции

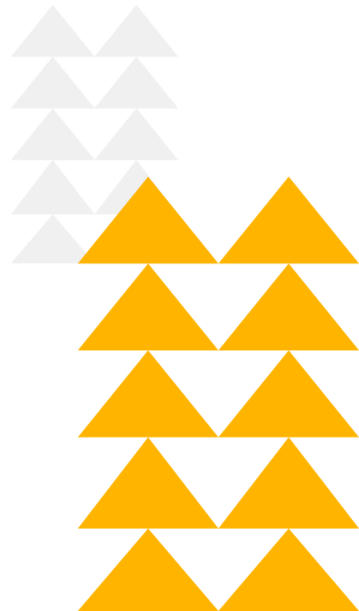
- Функции в Python могут принимать произвольное количество позиционных и ключевых аргументов
- Для объявления таких функций используют синтаксис упаковки, а для вызова синтаксис распаковки
- Синтаксис распаковки также можно использовать при присваивании нескольких аргументов и в цикле for



Области видимости

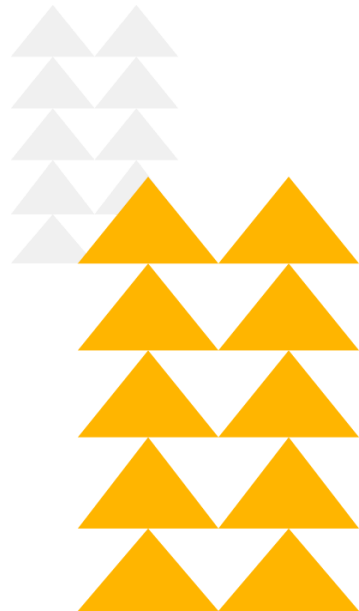
- Функции — объекты первого класса, то есть с ними можно делать всё то же самое, что и с другими значениями.
- Например, можно объявлять функции внутри других функций.

```
>>> def wrapper():
        def identity(x):
            return x
        return identity
>>> f = wrapper()
>>> f(42)
42
```



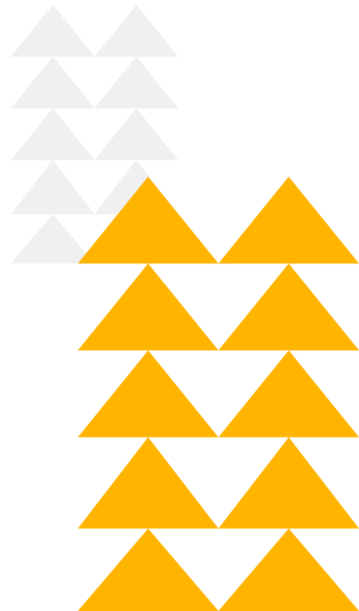
Пример: bounded_min

```
01. def make_min(*, lo, hi):
02.     def bounded_min(first, *rest):
03.         min_value = first
04.         for value in (first,) + rest:
05.             if value < min_value and lo < value < hi:
06.                 min_value = value
07.         return max(min_value, lo)
08.     return bounded_min
09.
10. >>> bounded_min = make_min(lo=0, hi=255)
11. >>> bounded_min(-1, -5, 42, 2)
12. 0
```



Области видимости

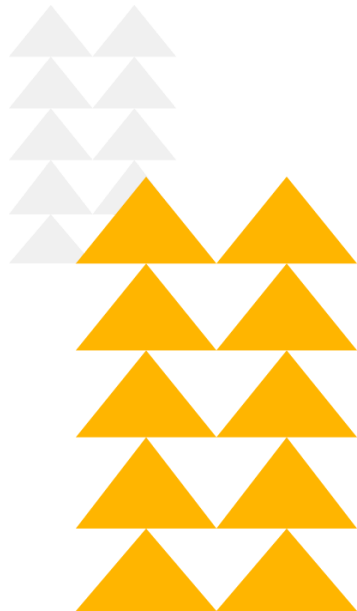
```
01. >>> min  # <- builtin
02. <built-in function min>
03. >>> min = 42  # <- global
04. >>> def f(*args):
05.         min = 2  # <- local for f
06.         def g():  # <- enclosing
07.             min = 4  # <- local for g
08.             print(min)
```



Области видимости (2)

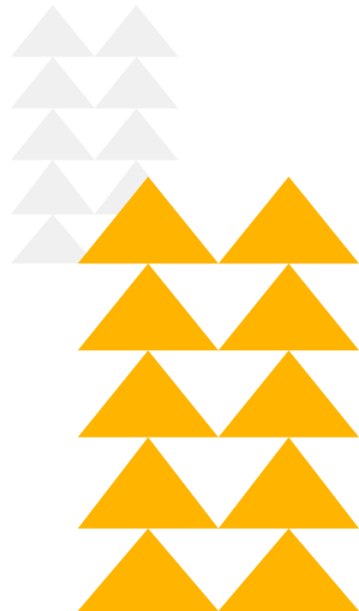
Поиск имени ведётся в четырёх областях видимости: локальной, затем в объемлющей функции (если такая имеется), затем в глобальной и, наконец, во встроенной.

```
L -> E -> G -> B
```



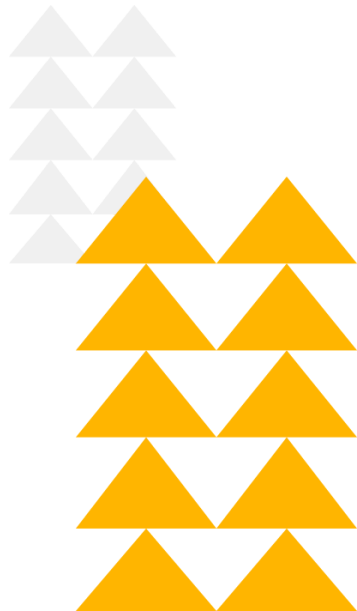
Области видимости: присваивание (1)

```
01. >>> min = 42
02. >>> def f():
03.         min += 1
04.         return min
05. >>> f()
06. Traceback (most recent call last):
07. File "<stdin>", line 1, in <module>
08. File "<stdin>", line 2, in f
09. UnboundLocalError: local variable 'min' referenced [...]
```



Области видимости: присваивание (2)

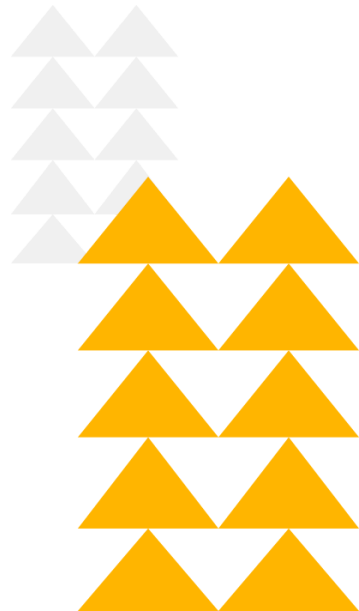
- По умолчанию операция присваивания создаёт локальную переменную
- Изменить это поведение можно с помощью операторов `global` и `nonlocal`



Области видимости: оператор global

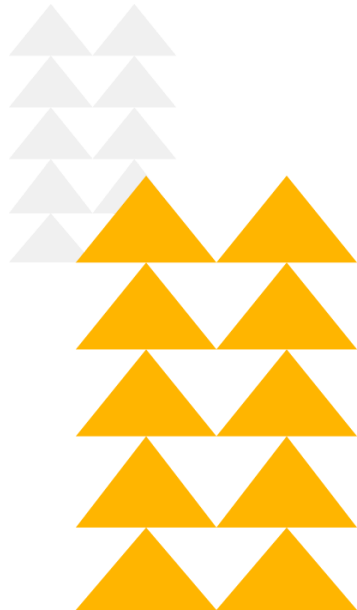
Позволяет модифицировать значение переменной из глобальной области видимости.

```
01. >>> min = 42
02. >>> def f():
03.         global min
04.         min += 1
05.         return min
06. >>> f()
07. 43
08. >>> f()
09. 44
```



Области видимости: оператор global

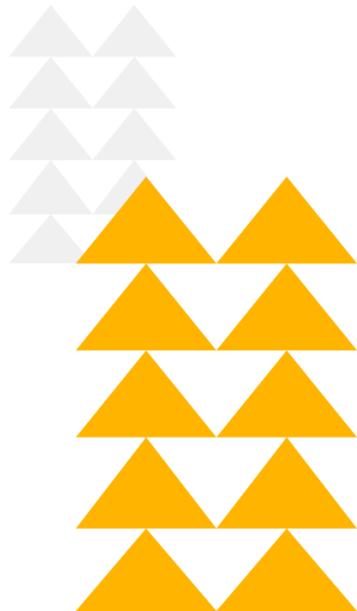
Использование global порочно, почти всегда лучше заменить global на thread-local объект.



Области видимости: оператор nonlocal

Позволяет модифицировать значение переменной из объемлющей области видимости

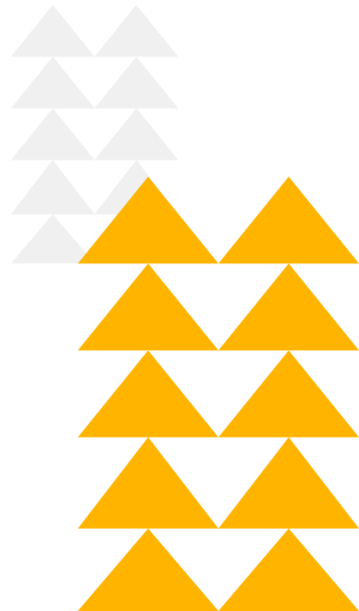
```
>>> def cell(value=None):
        def get():
            return value
        def set(new_value):
            nonlocal value
            value = new_value
        return get, set
>>> get, set = cell()
>>> set(42); get()
42
```

Замыкания

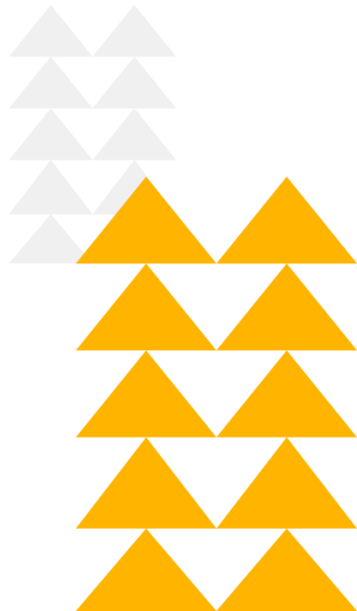
- Функции в Python могут использовать переменные, определенные во внешних областях видимости (`make_min`)
- Важно помнить, что поиск переменных осуществляется во время исполнения функции, а не во время её объявления

```
>>> def f():
        print(i)  # <- not declared
>>> for i in range(4):
        f()
0 1 2 3
```



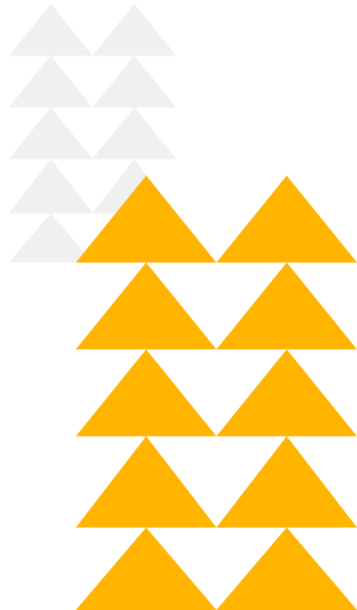
Take away: области видимости

- В Python четыре области видимости: встроенная, глобальная, объемлющая и локальная.
- Поиск имени осуществляется от локальной к встроенной (`LEGB`)
- При использовании операции присваивания имя считается локальным. Это поведение можно изменить с помощью операторов `global` и `nonlocal`.



Функциональное программирование

Python не функциональный язык, но в нём есть элементы функционального программирования.



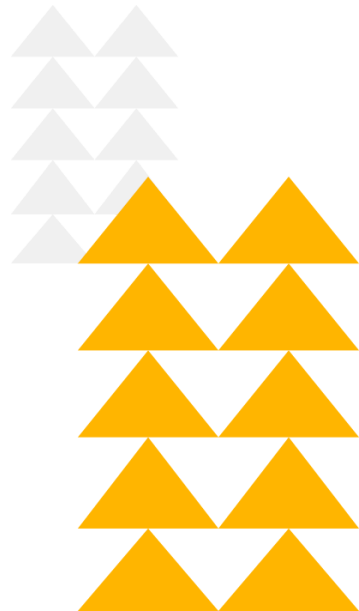
Анонимные функции

Анонимные функции имеют вид:

```
>>> lambda foo, *args, bar=None, **kwargs: expression
```

и эквивалентны по поведению:

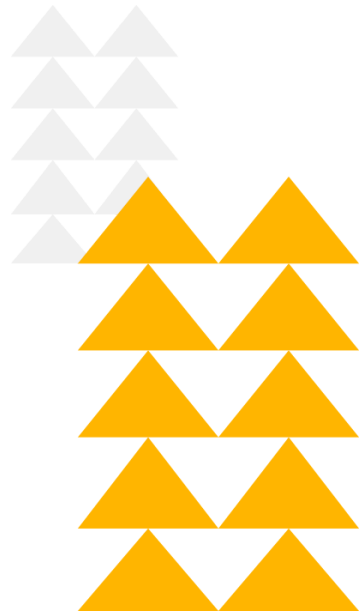
```
>>> def <lambda>(foo, *args, bar=None, **kwargs):
        return expression
```



map

Применяет функцию к каждому элементу последовательности

```
01. >>> map(indentity, range(4))
02. <map object at 0x100fc4c88>
03. >>> list(map(identity, range(4)))
04. [0, 1, 2, 3]
05. >>> set(map(lambda x: x % 7, [1, 9, 16, -1, 2, 5]))
06. {1, 2, 5, 6}
07. >>> map(lambda s: s.strip(), open("./HBA1.txt"))
08. <map object at 0x100fc4cc0>
```



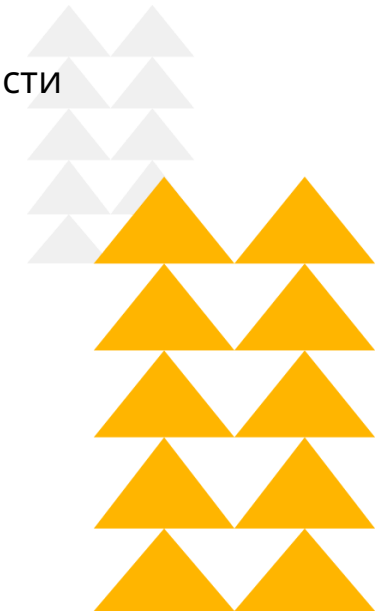
filter

Убирает из последовательности элементы, не удовлетворяющие предикату

```
>>> list(filter(lambda x: x % 2 != 0, range(10)))
[1, 3, 5, 7, 9]
```

Вместо предиката можно передать `None`, в этом случае в последовательности останутся только truthy значения

```
>>> xs = [0, None, [], {}, set(), "", 42]
>>> list(filter(None, xs))
[42]
```



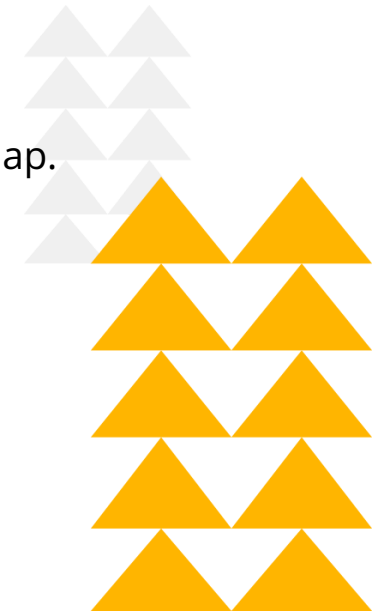
zip

Строит последовательность кортежей из элементов нескольких последовательностей

```
>>> list(zip("abc", range(3), [42j, 42j, 42j]))
[('a', 0, 42j), ('b', 1, 42j), ('c', 2, 42j)]
```

Поведение в случае последовательностей различной длины аналогично map.

```
>>> list(zip("abc", range(10)))
[('a', 0), ('b', 1), ('c', 2)]
```

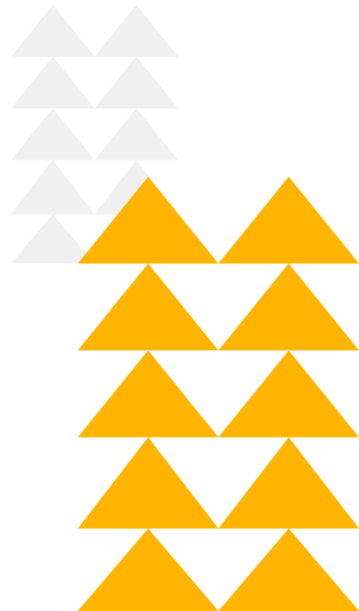


Генераторы списков

```
>>> [x ** 2 for x in range(10) if x % 2 == 1]
[1, 9, 25, 49, 81]
```

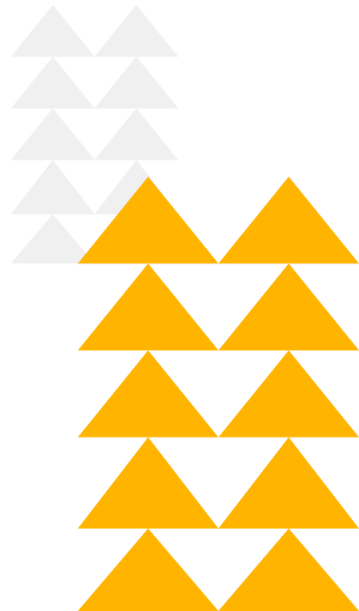
Могут быть вложенными

```
>>> nested = [range(5), range(8, 10)]
>>> [x for xs in nested for x in xs] # flatten
[0, 1, 2, 3, 4, 8, 9]
```



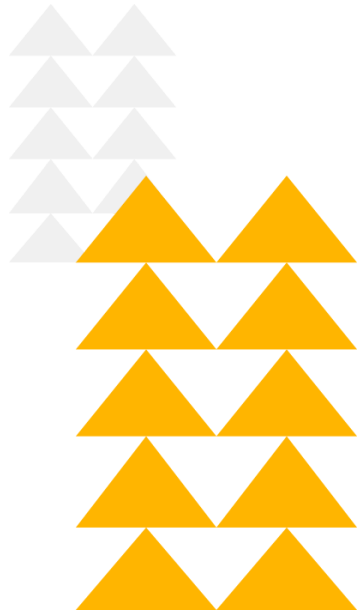
Генераторы множеств и словарей

```
>>> {x % 7 for x in [1, 9, 16, -1, 2, 5]}
{1, 2, 5, 6}

>>> date = {"year": 2014, "month": "September", "day": ""}
>>> {k: v for k, v in date.items() if v}
{'month': 'September', 'year': 2014}

>>> {x: x ** 2 for x in range(4)}
{0: 0, 1: 1, 2: 4, 3: 9}
```



Take away: функциональное программирование

- Наличие элементов функционального программирования позволяет компактно выражать вычисления.
- В Python есть типичные для функциональных языков:
  - анонимные функции `lambda`,
  - функции `map`, `filter` и `zip`,
  - генераторы списков, множеств и словарей



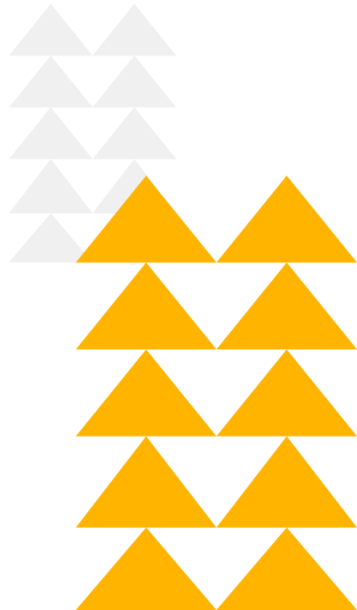
Синтаксис использования декораторов

Декоратор — функция, которая принимает другую функцию и что-то возвращает.

```
>>> @trace
    def foo(x):
        return 42
```

Аналогичная по смыслу версия без синтаксического сахара

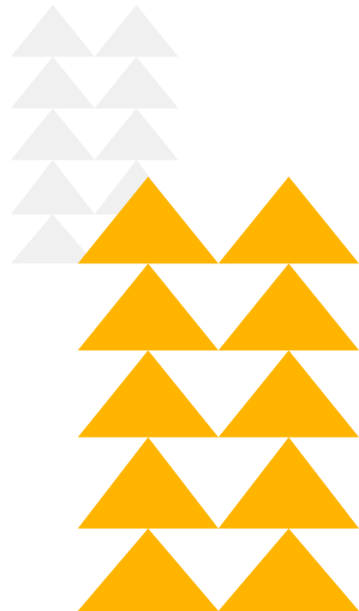
```
>>> def foo(x):
        return 42
>>> foo = trace(foo)
```

Пример: trace

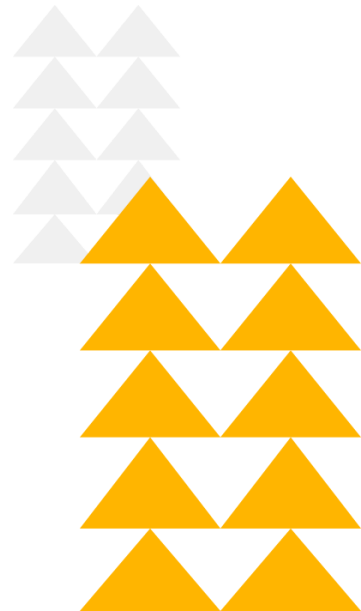
Декоратор `trace` выводит на экран сообщение с информацией о вызове декорируемой функции.

```
>>> def trace(func):
        def inner(*args, **kwargs):
            print(func.__name__, args, kwargs)
            return func(*args, **kwargs)
        return inner
```



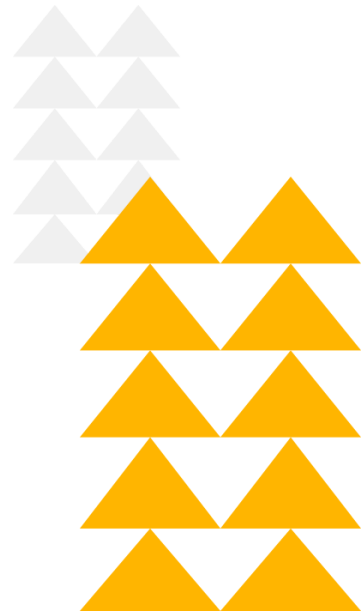
Пример: trace

```
01. >>> @trace
02.     def identity(x):
03.         """I do nothing useful."""
04.         return x
05.
06. >>> identity(42)
07. identity (42, ) {}
08. 42
```



Декораторы и help: проблема

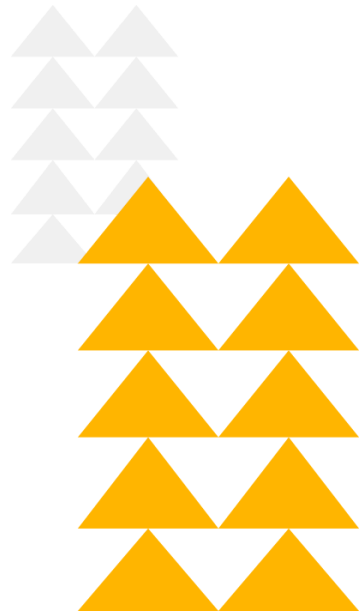
```
01. >>> help(identity)
02. Help on function inner in module __main__:
03. inner(*args, **kwargs)
04.
05. >>> identity.__name__, identity.__doc__
06. ('identity', 'I do nothing useful as well.')
07.
08. >>> identity = trace(identity)
09. >>> identity.__name__, identity.__doc__
10. ('inner', None)
```



Декораторы и help: решение

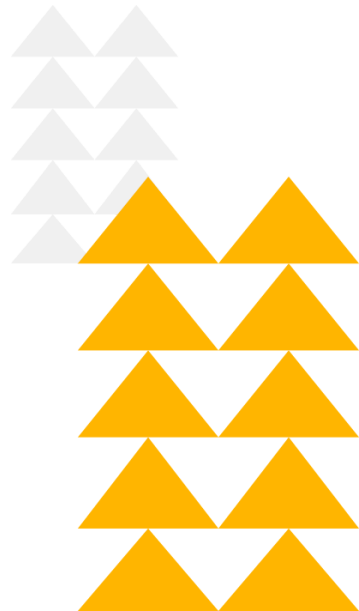
Давайте просто возьмём и установим “правильные” значения в атрибуты декорируемой функции:

```
01. >>> def trace(func):
02.         def inner(*args, **kwargs):
03.             print(func.__name__, args, kwargs)
04.             return func(*args, **kwargs)
05.         inner.__module__ = func.__module__
06.         inner.__name__ = func.__name__
07.         inner.__doc__ = func.__doc__
08.         return inner
```



Декораторы и help: решение

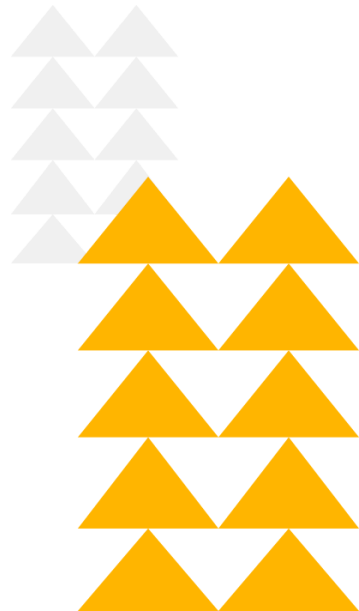
```
>>> identity = trace(identity)
>>> identity.__name__, identity.__doc__
('identity', 'I do nothing useful as well.')
```



Декораторы и help: functools

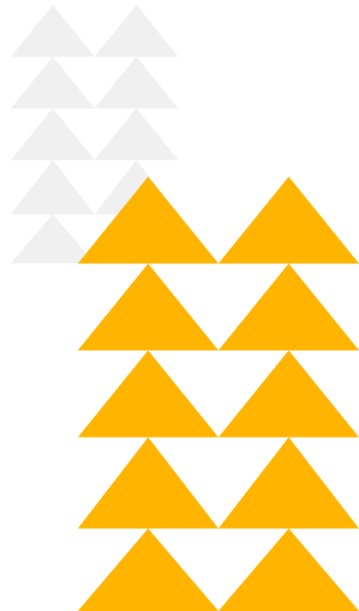
В модуле `functools` из стандартной библиотеки Python есть функция, реализующая логику копирования внутренних атрибутов.

```
01. >>> def trace(func):
02.         @functools.wraps(func)
03.         def inner(*args, **kwargs):
04.             print(func.__name__, args, kwargs)
05.             return func(*args, **kwargs)
06.         return inner
```



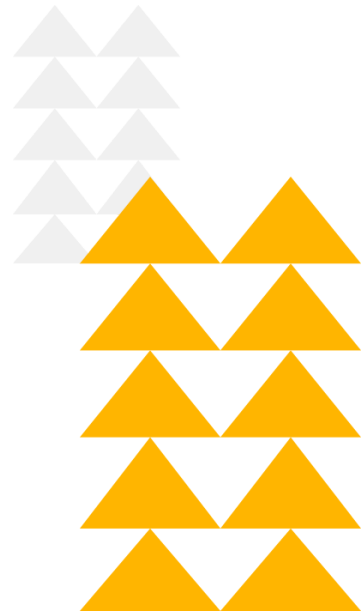
Декораторы с аргументами

```
01. def trace(handle):
02.     def wrapper(func):
03.         @functools.wraps(func)
04.         def inner(*args, **kwargs):
05.             print(func.__name__, args, kwargs, file=handle)
06.             return func(*args, **kwargs)
07.         return inner
08.     return wrapper
09.
10. @trace(handle=sys.stderr)
11. def identity(x):
12.     return x
```



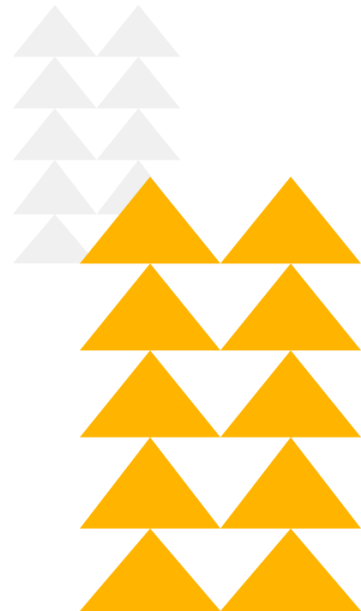
Пример: once

```
>>> def once(func):
        """
        Ensures the decorated function called only once.
        """
        @functools.wraps(func)
        def inner(*args, **kwargs):
            if not inner.called:
                func(*args, **kwargs)
                inner.called = True
        inner.called = False
        return inner
```



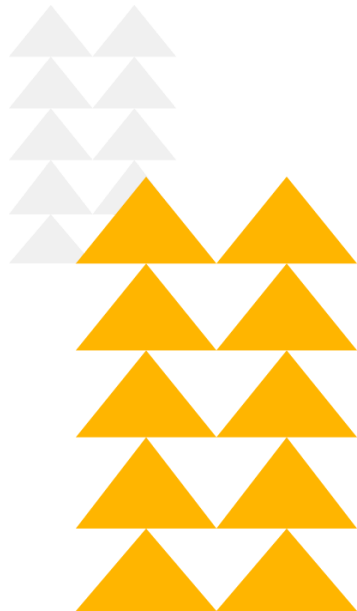
Пример: once (2)

```
01. >>> @once
02.     def initialize_settings():
03.         print("settings initialized")
04.
05. >>> initialize_settings()
06. settings initialized
07.
08. >>> initialize_settings()
```



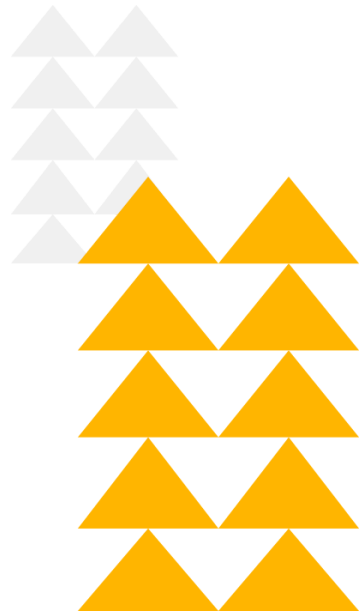
Take away: декораторы

- Декоратор — способ модифицировать поведение функции, сохраняя читаемость кода
- Декораторы бывают:
  - без аргиментов `@trace`
  - с аргументами `@trace(sys.stderr)`
  - с опциональными аргументами



Модуль functools

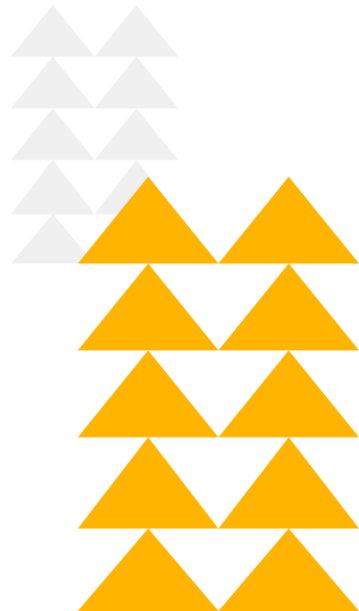
- `@wraps`
- `@partial`
- `@lru_cache`
- `@reduce`



partial

С помощью `partial` можно зафиксировать часть позиционных и ключевых аргументов в функции.

```
01. >>> sort_by_2nd_key = partial(sorted, key=operator.itemgetter(1))
02. >>> sort_by_2nd_key([("a", 4), ("b", 2)])
03. [('b', 2), ('a', 4)]
04.
05. >>> f = partial(sorted, [2, 3, 1, 4])
06. >>> f()
07. [1, 2, 3, 4]
```



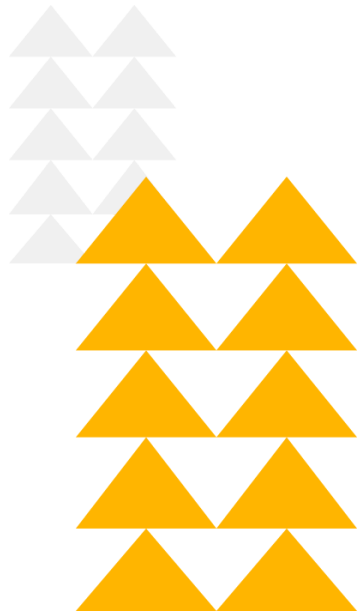
reduce

Функция reduce принимает три аргумента: бинарную функцию, последовательность и опциональное начальное значение. Вычисление `reduce(op, xs, initial)` можно схематично представить как:

```
>>> op(op(op(op(init, xs[0]), xs[1]), xs[2]), ...)
>>> op(op(op(xs[0], xs[1]), xs[2]), ...)
```

Пример

```
>>> reduce(merge, [[1, 2, 7], [5, 6], [0]])
[0, 1, 2, 5, 6, 7]
```

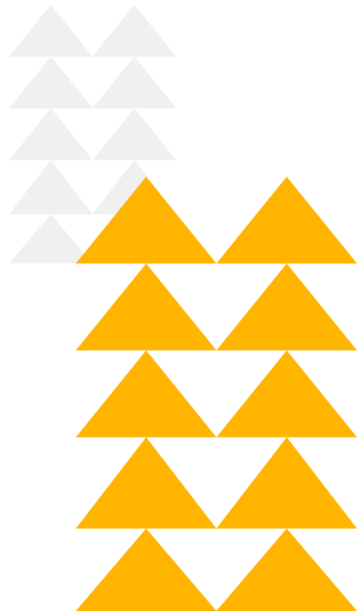
Встроенные коллекции

- Встроенных коллекций в Python немного: `tuple`, `list`, `set`, `frozenset` и `dict`.
- Мы уже кратко обсуждали базовые методы работы с ними:
  - Создать коллекцию можно с помощью литералов или конструктора типов
  - Функция `len` возвращает длину переданной коллекции.
  - elem `in` collection и elem `not in` collection проверяют, содержится ли в коллекции элемент elem,
  - Удалить элемент по ключу или по индексу можно с помощью оператора `del`.

Кортеж

С помощью слайсов можно выделить подпоследовательность в любой коллекции, в частности, в кортеже:

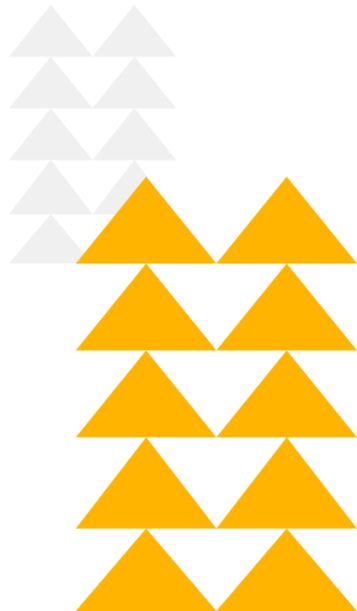
```
>>> person = ("George", "Carlin", "May", 12, 1937)
>>> name, birthday = person[:2], person[2:]
>>> name
('George', 'Carlin')
>>> birthday
('May', 12, 1937)
```



Слайсы в кортеже

Избавиться от “магических” констант помогут именованные слайсы:

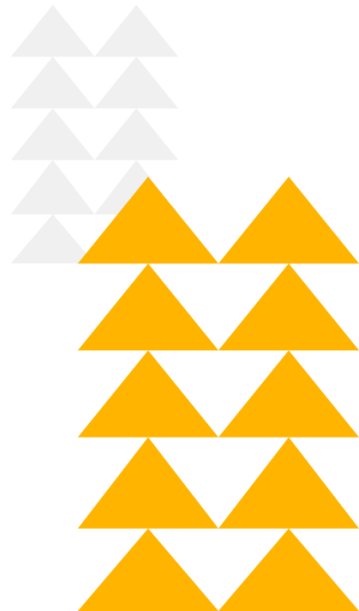
```
01. >>> NAME, BIRTHDAY = slice(2), slice(2, None)
02. (None, 2, None)
03. >>> person[NAME]
04. ('George', 'Carlin')
05. >>> person[BIRTHDAY]
06. ('May', 12, 1937)
```



Конкатенация кортежей

Кортежи можно конкатенировать с помощью бинарной операции +. Результатом конкатенации всегда является новый кортеж

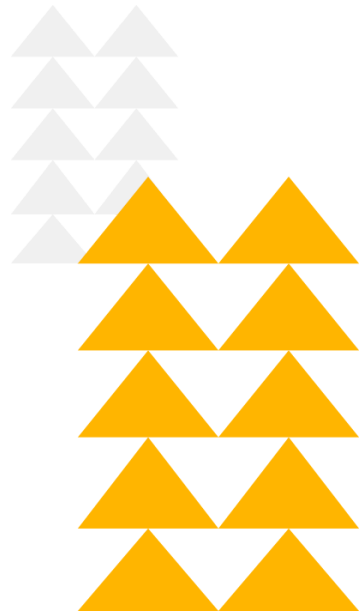
```
>>> xs, ys = (1, 2), (3)
>>> id(xs), id(ys)
(4411696072, 4411678472)
>>> id(xs + ys)
4411708104
```



Сравнение кортежей

Сравнение кортежей происходит в лексикографическом порядке, причём длина учитывается, только если одна последовательность является префиксом другой:

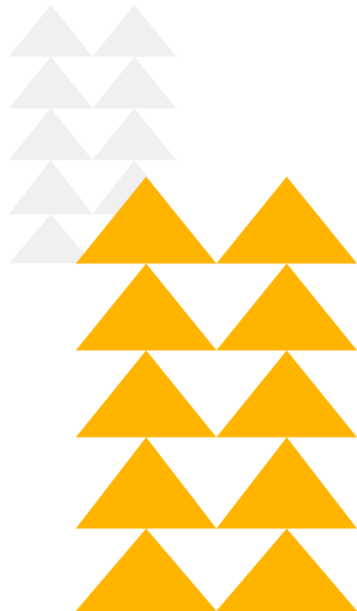
```
>>> (1, 2, 3) < (1, 2, 4)
True
>>> (1, 2, 3, 4) < (1, 2, 4)
True
>>> (1, 2) < (1, 2, 42)
True
```



collections.namedtuple

Функция `namedtuple` возвращает тип кортежа, специализированный на фиксированное множество полей:

```
01. >>> from collections import namedtuple
02. >>> Person = namedtuple(”Person”, [”name”, ”age”])
03. >>> p = Person(”George”, age=77)
04. >>> p._fields
05. (’name’, ’age’)
06. >>> p.name, p.age
```



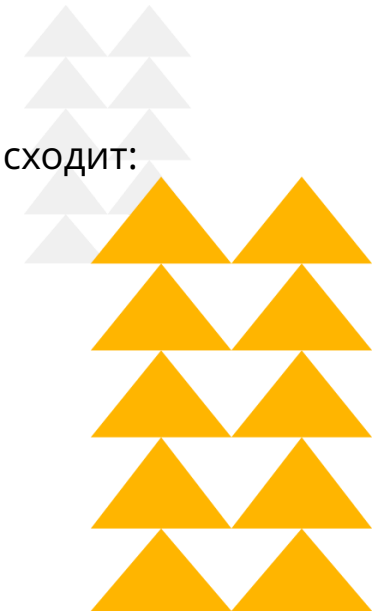
Список

Синтаксис инициализации создаёт список указанной длины и заполняет его начальным значением:

```
>>> [0] * 3
[0, 0, 0]
```

Важно понимать, что копирование начального значения при этом не происходит:

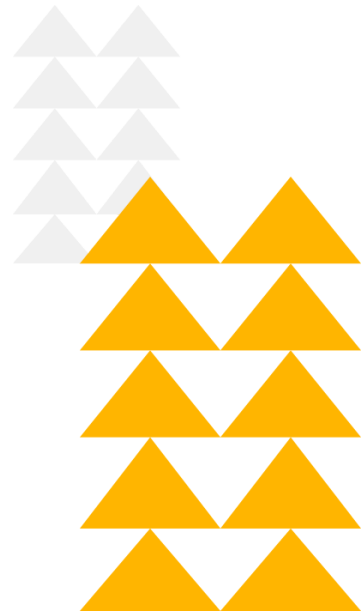
```
>>> chunks = [[0]] * 2  # 2 x 1 zero matrix
>>> chunks[0][0] = 42
>>> chunks
[[42], [42]]
```



Конкатенация списков

Конкатенация списков работает аналогично конкатенации кортежей: результатом всегда является новый список.

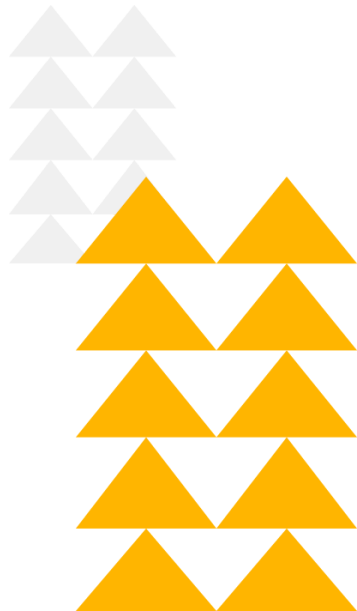
```
>>> xs, ys = [1, 2], [3]
>>> id(xs), id(ys)
(4411696072, 4411678472)
>>> id(xs + ys)
4411708104
```



Конкатенация списков (2)

В отличие от кортежей списки поддерживают inplace конкатенацию:

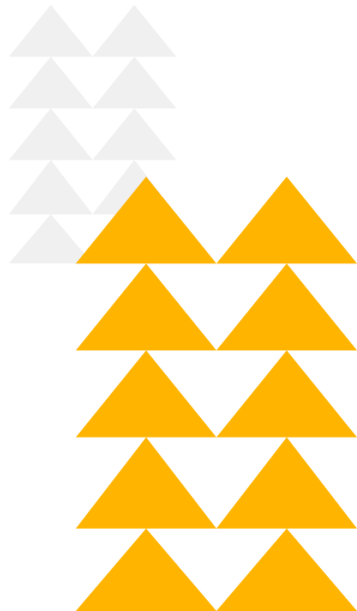
```
>>> xs, ys = [1, 2], [3]
>>> id(xs), id(ys)
(4411696072, 4411678472)
>>> xs += ys  # xs.extend(ys); xs = xs
>>> id(xs)
4411696072
```



Удаление элементов из списка

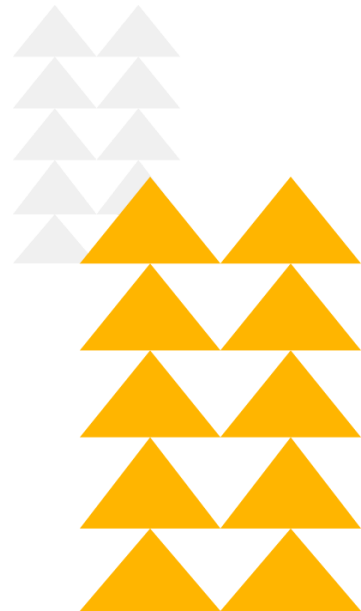
С помощью оператора `del` можно удалить не только один элемент по его индексу, но и целую подпоследовательность:

```
>>> xs = [1, 2, 3, 4]
>>> del xs[:2]
>>> xs
[3, 4]
>>> del xs[:]
>>> xs
[]
```



Список — это стек

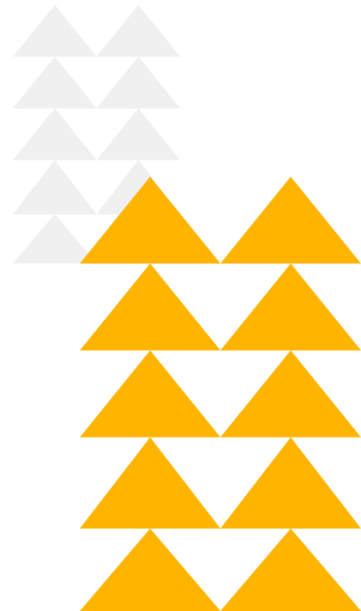
```
01. >>> stack = []
02. >>> stack.append(1)
03. >>> stack.append(2)
04. >>> stack.pop()
05. 2
06. >>> stack
07. [1]
```



Список — это очередь

```
01. >>> q = []
02. >>> q.append(1)
03. >>> q.append(2)
04. >>> q.pop(0)
05. 1
06. >>> q
07. [2]
```

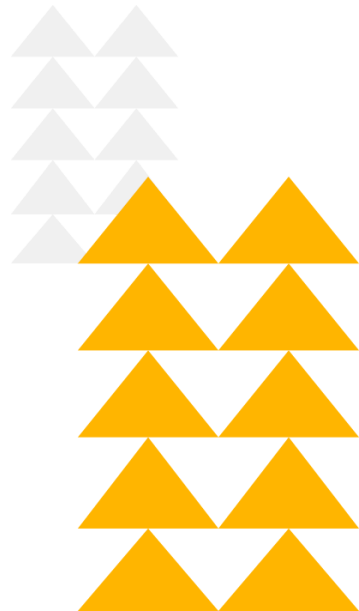
В чём проблема с `q.pop(0)`?



collections.deque

Добавление и удаление элемента с обеих сторон очереди работает за константное время, индексирование — за время, линейное от размера очереди.

```
>>> q = deque([1, 2, 3])
>>> q.appendleft(0)
>>> q.append(4)
>>> q
deque([0, 1, 2, 3, 4])
>>> q.popleft()
0
>>> q[0]
1
```



Множество

Множество в Python — это hash set, то есть оно может содержать только элементы, которые можно захешировать.

```
>>> {dict()}
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'
```

Объекты типа `frozenset` поддерживают все операции типа `set` кроме операций добавления и удаления элементов.

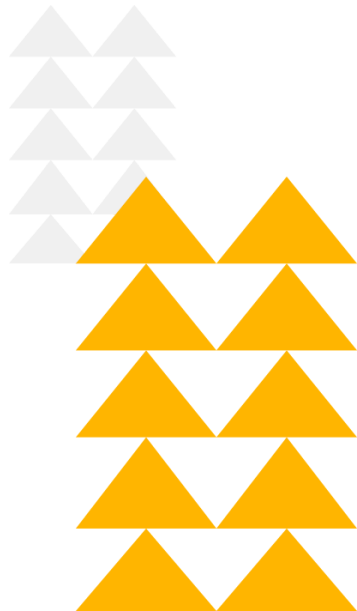
Словарь

Можно сконструировать словарь из последовательности ключей и значения по умолчанию:

```
>>> dict.fromkeys("abcd", 0)
{'d': 0, 'a': 0, 'b': 0, 'c': 0}
```

Эквивалентны ли эти два выражения?

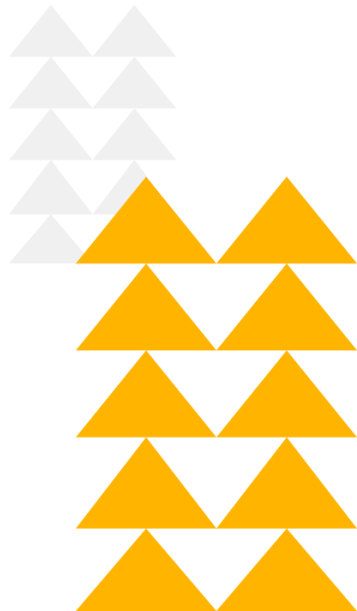
```
>>> dict.fromkeys('abc', [])
>>> {char: [] for char in 'abc'}
```



Проекции словаря

Методы `keys`, `values` и `items` возвращают проекции содержимого словаря:

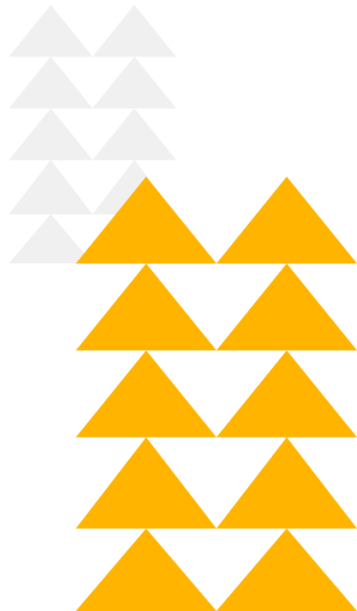
```
01. >>> d = dict.fromkeys(["foo", "bar"], 42)
02. >>> d.keys()
03. dict_keys(['foo', 'bar'])
04. >>> d.values()
05. dict_values([42, 42])
06. >>> d.items()
07. dict_items([('foo', 42), ('bar', 42)])
```



Проекции словаря (2)

Проекции можно использовать для итерации в цикле `for` или генераторе

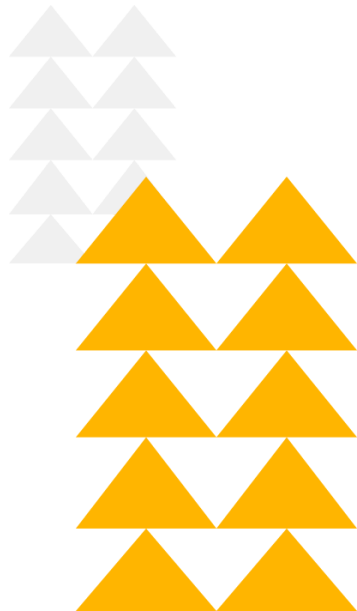
```
>>> {v for v in d.values()}
{42}

>>> 42 in d.values()
True
```



Проекции словаря (3)

Нельзя модифицировать содержимое словаря в процессе итерации

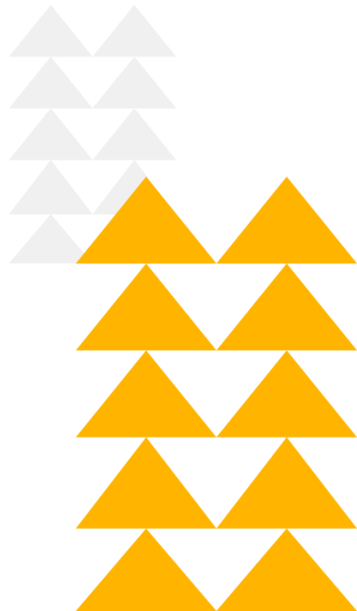
```
01. >>> for k in d:
02.         del d[k]
03. Traceback (most recent call last):
04. File "<stdin>", line 1, in <module>
05. RuntimeError: dictionary changed size during iteration
```



Добавление элементов в словарь

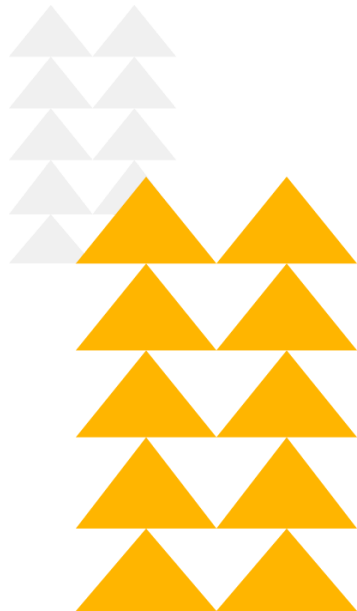
Метод `setdefault` позволяет за один запрос к хеш-таблице проверить, есть ли в ней значение по некоторому ключу и, если значения нет, установить его в заданное.

```
>>> d = {"foo": "bar"}
>>> d.setdefault("foo", "???")
'bar'
>>> d.setdefault("boo", 42)
42
>>> d
{'boo': 42, 'foo': 'bar'}
```



collections.defaultdict

```
01. >>> from collections import defaultdict
02. >>> g = defaultdict(set, **{"a": {"b"}, "b": {"c"}})
03. >>> g["c"].add("a")
04. >>> g
05. defaultdict(<class 'set'>, {'b': {'c'}, 'c': {'a'}, 'a': {'b'}})
```



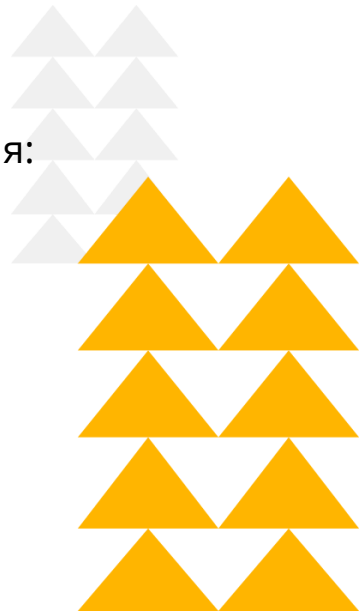
collections.OrderedDict

Порядок ключей в обычном словаре не определён:

```
>>> d = dict([("foo", "bar"), ("boo", 42)])
>>> list(d)
['boo', 'foo']
```

OrderedDict — словарь с ключами, упорядоченными по времени добавления:

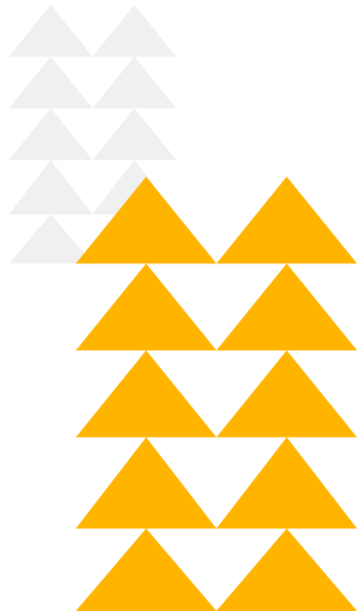
```
>>> from collections import OrderedDict
>>> d = OrderedDict([("foo", "bar"), ("boo", 42)])
>>> list(d)
['foo', 'boo']
```

collections.OrderedDict (2)

Изменение значения по ключу не влияет на порядок ключей в словаре:

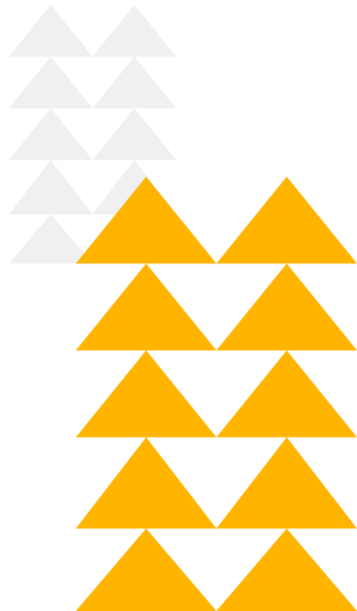
```
01. >>> d["boo"] = "???" # не изменит порядок ключей
02. >>> d["bar"] = "???"
03. >>> list(d)
04. ['foo', 'boo', 'bar']
```



collections.Counter

Специализация словаря для подсчёта объектов, которые можно захешировать:

```
01. >>> from collections import Counter
02. >>> c = Counter(["a", "a", "a", "b"])
03. >>> c["a"] += 1
04. >>> c
05. Counter({'a': 4, 'b': 1})
06. >>> c.most_common(1)
07. [('a', 4)]
```



Спасибо за внимание

Виталий Кудёлка

Разработчик программного обеспечения

vital.kudzelka@leverx.com

github.com/vitalk