# Конспект по книге *"Fluent Python"*

Anton S Khomyakov

# Список разделов

# Глава 2. Коллекции

## Сравнение перечисляемых типов

| *Критерий сравнения* | **tuple** | **list** | **set** |
|---|---|---|---|
| *Нотация* | a = (1, 2, 3) | b = [1, 2, 3] | c = {1, 2, 3} |
| *Название* | Кортеж | Список | Множество |
| *Хэшируем* | + | - | - |
| *Упорядоченность* | Всегда упорядоченный список объектов | Всегда упорядоченный список объектов | До 3.6 словари `dict` и множества `set` не сохраняли порядок, но начиная с 3.7 **официально упорядочены** |
| *Дубликаты* | Может содержать дубликаты | Может содержать дубликаты | Не содержит дубликатов |
| *Индексация* | + | + | - |
| *Размер* | Фиксированны | Динамический | Динамический |

| *Критерий сравнения* | **tuple** | **list** | **set** |
|---|---|---|---|
| *Подходит для* | Последовательность не планируется изменять; Если нужно поочередно перебирать неизменную последовательность элементов; Нужна последовательность элементов для ее назначения в качестве ключа словаря. Поскольку списки — это изменяемый тип данных, их нельзя применять в качестве ключей словаря; Важна скорость выполнения операций с последовательностью : из-за отсутствия возможности изменения, кортежи работают куда быстрее списков; | Последовательность планируется изменять; Планируется добавлять новые элементы или удалять старые | Базовая структура типа данных “множество” — это хеш-таблица (Hash Table). Поэтому множества очень быстро справляются с проверкой элементов на вхождение, например содержится ли объект `x` в последовательности `a_set`. Идея заключается в том, что поиск элемента в хэш-таблице — это операция `O(1)`, то есть операция с постоянным временем выполнения. |

По сути, если не нужно хранить дубликаты, то множество будет лучшим выбором, чем список.

# Выводы

*“Преждевременная оптимизация — корень всех зол”.*

Итак, самое главное, что вам стоит запомнить по поводу списков, кортежей и множеств:

- Если необходимо хранить дубликаты, то выбирайте список или кортеж.
- Если НЕ планируется изменять последовательность после ее создания, то выбирайте кортеж, а не список.
- Если НЕ нужно хранить дубликаты, то воспользуйтесь множеством, так как они значительно быстрее определяют наличие объекта в последовательности. :experimental: :doctype: book :icons: font :myprojectbasedir: {asciidoctorconfigdir} :doctestdir: {asciidoctorconfigdir}/../../source/doctest :sourcedir: {asciidoctorconfigdir}/../../source :imagesoutdir: ./images

# Глава 3. Словари и множества

Все словари наследуют класс **collections.abc.Mapping**. Ключи должны быть хэшируемые. Включать метод *hash*() и *eq*() Объект называется хэшируемым, если он обладает хэш-значением, которое не изменяется на протяжении всей жизни объекта и допускает сравнение с другими объектами.

*Способы инициализации словаря*

```
a = dict(one=1, two=2, three=3)
b = {'one': 1, 'two': 2, 'three': 3}
c = dict(zip(['one','two','three'],[1,2,3]))
d = dict([('two', 2), ('one', 1), ('three', 3)])
e = dict({'two': 2, 'one': 1, 'three': 3})
a == b == c == d == e
True
```

# Глава 4. Тексты и байты.

Всё что нужно знать о байтах: главное это то, что существует 2 основных типа отображения двоичных последовательностей: изменяемы тип bytes, появившийся в py3 и не изменяемы тип bytearray. Каждый элемент bytes или bytearray - целое число от 0 до 255

# Глава 5. Полноправные функции

## Семь видов вызываемых объектов

Оператор () можно применять не только к функциям, определённым пользователями. Что бы понять является ли объект вызываемым, воспользуейтесь функцией:

```
callable()
```

*Table 1. Вызываемые элементы Python*

| Функция | Описание |
|---|---|
| *Пользовательские функции* | Создаются при помощи выражения `def` или `lambda`-выражения |
| *Встроенные функции* | Функции написанные на C (в случае CPython), например `len` или `time.strftime` |
| *Методы* | Функции определённые в теле класса |
| *Встроенные методы* | Метода написанные на C, например dict.get |
| *Классы* | При вызове класса выполняется свой метод `new`, что бы создать экземпляр, затем вызывает метод `init` для его инициализации и, наконец, возвращает экземпляр вызывающей программе |
| *Экземпляры классов* | Если в классе определен метод `call`, то его экземпляры можно вызвать, как функции |
| *Генераторные функции* | Функции или методы, в которых используется ключевое слово `yield`. При вызове генераторная функция возвращает объект-генератор |

Учитывая разнообразие вызываемых типов в Python, самый безопасный способ узнать, является ли объект вызываемым, - воспользоваться встроенной функцией callable()

## Пользовательские вызываемые типы

*Пример создания класса с реализованным методом `__call__`*

```
import random


class BingoCage:
    """
    Экземпляр этого класса строится из любого иттерируемого объекта и
    хранит внутри себя список элементов в случайном порядке. При вызове
    экземпляра из списка удаляется один элемент.
    """

    def __init__(self, items=None):
        """
        Метод __init__ принимает произвольный иттерируемый объект;
        Создание локальной копии предотвращает изменение списка, переданного
        в качестве аргумента.
        """
        self._items = list(items)
        random.shuffle(self._items)  # Метод shuffle гарантированно работает, т.к.
self._items объект тип list.

    def pick(self):
        """
        Основной метод.
        """
        try:
            return self._items.pop()
        except IndexError:
            # Возбудить исключение со специальным сообщением, если список self._items
пустой.
            raise LookupError('pick from empty BingoCage')

    def __call__(self):
        """
        Позволяет писать просто bingo() вместо bingo.pick()
        :return:
        """
        return self.pick()
```

*Демонстрация BingoCage*

```
>>> from source.bingocall import BingoCage
>>> bingo = BingoCage(range(3))
>>> bingo.pick()
2
>>> bingo()
1
>>> callable(bingo)
True
```

Объект `bingo` можно вызвать как функцию, и встроенная функция `callable(…)` распознает его как вызываемый объект

*Пример классного разбора именованых и не именовых аргументов функции*

```
def tag(name, *content, cls=None, **attrs):
    """
    Функция tag генерирует HTML; чисто именованый аргумент cls
    для передачи аттрибута "class". Это обходное решение необходимо,
    т.к. в Python class - Зарезервированное слово.
    """
    print(name)
    if cls is not None:
        attrs['class'] = cls
    if attrs:
        attr_str = ''.join(' %s="%s"' % (attr, value) for attr, value in sorted(attrs
.items()))
    else:
        attr_str = ''
    if content:
        return '\n'.join('<%s%s>%s</%s>' % (name, attr_str, c, name) for c in content)
    else:
        return '<%s%s />' % (name, attr_str)
```

# Получение информации о параметрах

- У объекта-функции есть аттрибут `defaults`, в котором хранится кортеж со значениями по умолчанию позиционных и именованных параметров.
- Значения чисто именованных аргументов находятся в `kwdefaults`
- Сами имена параметров находятся в атрибуте `code`, который содержится ссылку на объект `code` с множеством своих собственных параметров

```
>>> from source.tag import tag
>>> tag.__code__.co_varnames
('name', 'cls', 'content', 'attrs')
>>> tag.__code__.co_argcount
1
```

# inspect.signature

Метод `inspect.signature` возвращает объект inspect.Signature, у которого есть аттрибут `parameters`, позволяющий прочитать упорядоченное отображение имен на объекы типа `inspect.Parameter`. У каждого объекта `Parameter` есть набор аттрибутов, например: `name`, `default` и `kind`. Специально значение `inspect._empty` обозначающий параметры, не имеющие значения по-умолчанию.

```
>>> from inspect import signature
>>> sig = signature(help)
>>> sig
<Signature (*args, **kwds)>
>>> str(sig)
'(*args, **kwds)'
>>> for name, param in sig.parameters.items(): print(param.kind, ':', name, '=',
param.default)
...
VAR_POSITIONAL : args = <class 'inspect._empty'>
VAR_KEYWORD : kwds = <class 'inspect._empty'>
>>> sig = signature(open)
>>> for name, param in sig.parameters.items(): print(param.kind, ':', name, '=',
param.default)
...
POSITIONAL_OR_KEYWORD : file = <class 'inspect._empty'>
POSITIONAL_OR_KEYWORD : mode = r
POSITIONAL_OR_KEYWORD : buffering = -1
POSITIONAL_OR_KEYWORD : encoding = None
POSITIONAL_OR_KEYWORD : errors = None
POSITIONAL_OR_KEYWORD : newline = None
POSITIONAL_OR_KEYWORD : closefd = True
POSITIONAL_OR_KEYWORD : opener = None
```

У объекта `inspect.Signature` имеется метод `bind`, который принимает любое количество атрибутов и связывает их с параметрами, указанных в сигнатуре, следуя обычным правилам сопоставления фактических аргументов с формальными параметрами.

Каркас может использовать эту возможность для проверки аттрибутов до фактического вызова функции.

*Связывание сигнатуры функции `tag` со словарём аргументов*

```
>>> import inspect
>>> from source.tag import tag
>>> sig = inspect.signature(tag)
>>> my_tag = {
... 'name'  : 'img',
... 'title' : 'Sunset Boulevard',
... 'src'   : 'sunset.jpg',
... 'cls'   : 'framed'
... }
>>> bounds_args = sig.bind(**my_tag)
>>> bounds_args
<BoundArguments (name='img', cls='framed', attrs={'title': 'Sunset Boulevard', 'src':
'sunset.jpg'})>
>>> for name, value in bounds_args.arguments.items(): print(name, '=', value)
...
name = img
cls = framed
attrs = {'title': 'Sunset Boulevard', 'src': 'sunset.jpg'}
>>> del my_tag['name']
>>> bounds_args = sig.bind(**my_tag)
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    bounds_args = sig.bind(**my_tag)
  File "C:\Users\User\AppData\Local\Programs\Python\Python38-32\lib\inspect.py", line
3025, in bind
    return self._bind(args, kwargs)
  File "C:\Users\User\AppData\Local\Programs\Python\Python38-32\lib\inspect.py", line
2940, in _bind
    raise TypeError(msg) from None
TypeError: missing a required argument: 'name'
>>>
```

На этом примере видно, как модель данных Python - посредством модуля inspect - раскрывает механизм, которым пользуется сам интерпретатор дя связывания аргументов с формальными параметрами при вызове функии.

# Аннотация функций

*Аннотированная функция clip*

```
def clip(text: str, max_len: 'int > 0' = 80) -> str:  # Аннотированное объявление
функции
    """
    Return text clipped at the last space before or after max_len

    :param text:
        Переменная с текстом
    :param max_len:
        Максимальная длина возвращаемой строки
    :return:
        Возращает строку обрезаную до последнего пробела или до максимальной длины.
    """
    end = None
    if len(text) > max_len:
        space_before = text.rfind(' ', 0, max_len)
        if space_before >= 0:
            end = space_before
        else:
            space_after = text.rfind(' ', max_len)
            if space_after >= 0:
                end = space_after
    if end is None:  # No spaces were found
        end = len(text)
    return text[:end].rstrip()
```

- У любого аргумента в объявлении функции может быть выражение аннотации, которому предшествует `:`.
- Если у аргумента есть значение по-умолчанию, то аннотация располагается между именем и знаком `=`.
- Что-бы аннотировать возвращаемое значение, поместите `->` и вслед за ним выражение между знаком `)` и двоеточием в конце объявления функции.
- Аннотации никак не обрабатываются. Они просто сохраняются в аттрибуте функции `__annotations__` тип `dict`

```
>>> from source.clip_annot import clip
>>> clip.__annotations__
{'text': <class 'str'>, 'max_len': 'int > 0', 'return': <class 'str'>}
```

# Пакеты для функционального программирования

## Модуль `operator`

Модуль `operator` включает в себя функции для выборки элементов из последовательностей и чтения атрибутов объектов: `itemgetter` и `attrgetter` строят специализированные функции

для выполнения этих действий.

*Результат применения `itemgetter` для сортировки списка кортежей*

```
from operator import itemgetter

metro_data = [
    ('Tokyo', 'JP', 36.933, (35, 139)),
    ('Delhi NCR', 'IN', 21.935, (28, 77)),
    ('Mexico City', 'MX', 20.142, (19, -99)),
    ('New York-Newark', 'US', 20.104, (40, -74)),
    ('Sao Paulo', 'BR', 19.649, (-23, -46)),
]

for city in sorted(metro_data, key=itemgetter(1)):
    print(city)
```

```
py .\metro_data.py
('Sao Paulo', 'BR', 19.649, (-23, -46))
('Delhi NCR', 'IN', 21.935, (28, 77))
('Tokyo', 'JP', 36.933, (35, 139))
('Mexico City', 'MX', 20.142, (19, -99))
('New York-Newark', 'US', 20.104, (40, -74))
```

## Фиксация аргументов с помощью `functools.partial`

В модуле `functools` собраны некоторые функции высшего порядка. Из них наиболее широко известна функция `reduce`. Помимо неё, особенно полезна функция `partial` и её вариация `partialmethod`.

- `functools.partial` — функция высшего порядка. Позволяет применять функцию "частично". Получив на вход некоторую функцию, `partial` создает новый вызываемый объект, в котором некоторые аргументы исходной функции фиксированы. Функция `partial` принимает в первом аргументе вызываемый объект, а за ним - произвольное число позиционных и именованных аргументов, подлежащих связыванию.

*Построение вспомогательной функции нормализации Unicode-строк с помощью partial*

```
>>> import functools
>>> import unicodedata
>>> nfc = functools.partial(unicodedata.normalize, 'NFC')
>>> s1 = 'café'
>>> s2 = 'cafe\u0301'
>>> nfc(s1) == nfc(s2)
True
>>> s1 == s2
False
```

- `functools.partialmethod` — делает тоже самое, что и `partial`, но предназначена для работы с методами.

## Спецификации и статьи по пройденному материалу:

1. PEP 3102 — Keyword-Only Arguments
2. PEP 3107 — Function Annotations
3. PEP 362 — Function Signature Object
4. Functional Programming HOWTO

# Глава 6. Реализация паттернов проектирования с помощью полноправных функций.

## Практический пример: переработка паттерна Стратегия.

*В книге "Паттерны проектирования" паттерн* **Стратегия** *описывается следующим образом:*

Определить семейство алгоритмов, инкапсулировать каждый из них и сделать их взаимозаменяемыми. Стратегия позволяет заменять алгоритмы независимо от использующих его клиентов.

Наглядный пример применения паттерна Стратегия к коммерческой задаче — вычисление скидок на заказы в соответствии с характеристиками заказчика или результатами анализа заказанных позиций. Рассмотрим интернет-магазин со следующими правилами формирования скидок:

- Заказчику, имеющему не менее 1000 баллов лояльности, предоставляется глобальная скидка `5%` на весь заказ;
- На позиции, заказанные в количестве не менее 20 штук, предоставляется скидка `10%`
- На заказы, содержащие не мене 10 различных позиций, предоставляется глобальная скидка `7%`

Для простоты предложим, что к каждому заказчику может быть применена только одна скидка.

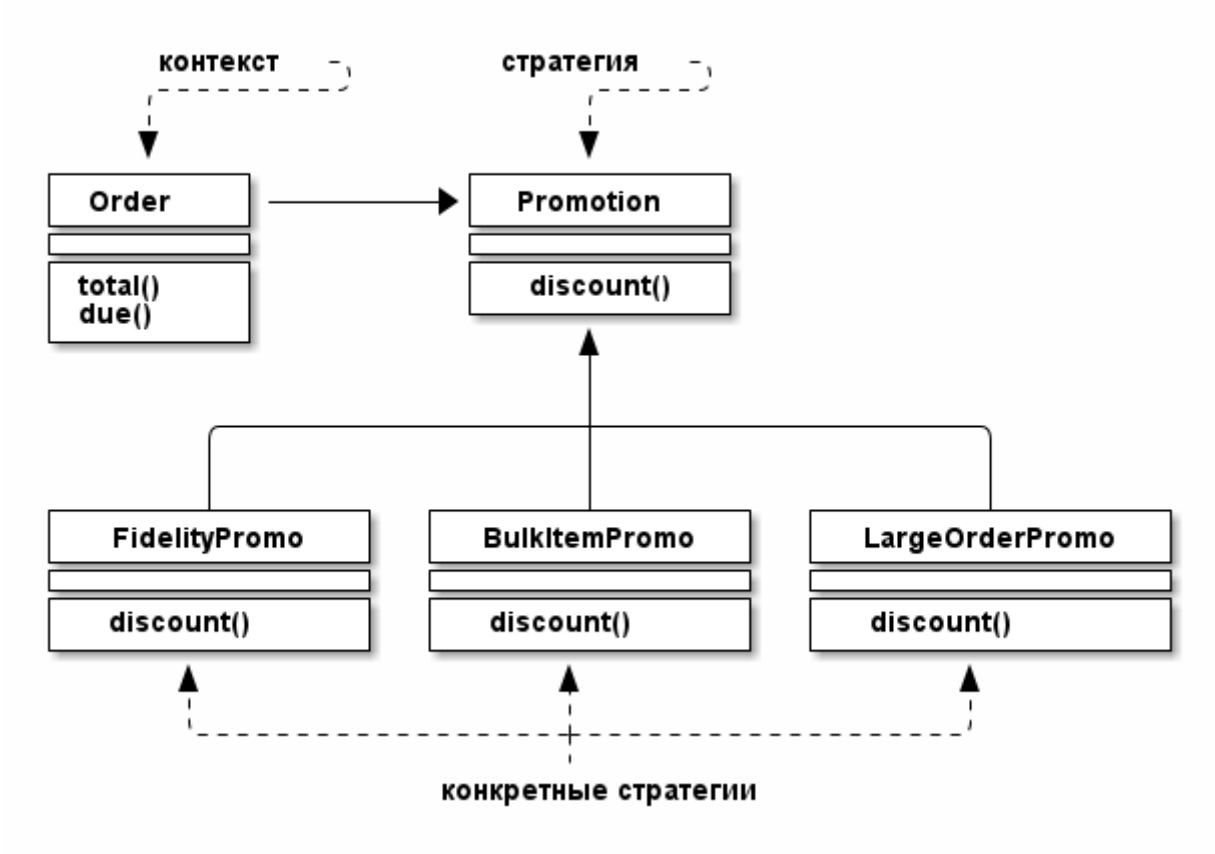



*Figure 1. UML-диаграмма классов для обработки заказов*

*Контекст*

Представляет службу, делегируя часть вычислений взаимозаменяемым компонентам, реализующих различные алгоритмы. В примере Интернет-магазина контекстом является класс `Order`, который конфигурируется для применения поощрительной скидки по одному из нескольких алгоритмов.

*Стратегия*

Интерфейс, общий для всех компонентов, реализующих различные алгоритмы. В нашем примере эту роль играет абстрактный класс `Promotion`.

*Конкретные стратегии*

Один из конкретных подклассов Стратегии. В нашем примере реализованы три конкретные стратегии: `FidelityPromo`, `BulkItemPromo`, `LargeOrderPromo`.

В нашем примере система, перед тем как создать объект заказ, должна каким-то образом определит стратегию предоставления скидки и передать ее конструктору класса `Order`. Вопрос о выборе стратегии не является предметом данного паттерна.

*Реализация класса `Order` с помощью взаимозаменяемых стратегий*

```
from abc import ABC, abstractmethod
from collections import namedtuple

Customer = namedtuple('Customer', 'name fidelity')
```

```
class LineItem:

    def __init__(self, product, quantity, price):
        self.product = product
        self.quantity = quantity
        self.price = price

    def total(self):
        return self.price * self.quantity


class Order:
    """
    Контекст
    """
    def __init__(self, customer, cart, promotion=None):
        self.customer = customer
        self.cart = list(cart)
        self.promotion = promotion

    def total(self):
        if not hasattr(self, '__total'):
            self.__total = sum(item.total() for item in self.cart)
        return self.__total

    def due(self):
        if self.promotion is None:
            discount = 0
        else:
            discount = self.promotion.discount(self)
        return self.total() - discount

    def __repr__(self):
        return f'<Order total: {self.total():.2f} due: {self.due():.2f}>'


class Promotion(ABC):
    """
    Стратегия: абстрактный базовый класс
    """
    @abstractmethod
    def discount(self, order):
        """
        Вернуть скидку в виде положительной суммы в долларах
        :param order:
        :return:
        """
```

```
class FidelityPromo(Promotion):
    """
    5%-я скидка для заказчиков, имеющих не менее 1000 баллов лояльности
    """
    def discount(self, order):
        return order.total() * .05 if order.customer.fidelity >= 1000 else 0


class BulkItemPromo(Promotion):
    """
    10%-я скидка для каждой позиции LineItem, в которой заказано не мене 20 единиц
    """
    def discount(self, order):
        discount = 0
        for item in order.cart:
            if item.quantity >= 20:
                discount += item.total() * .1
        return discount


class LargeOrderPromo(Promotion):
    """
    7%-я скидка для заказов, включающих не менее 10 различных позици
    """
    def discount(self, order):
        distinct_items = {item.product for item in order.cart}
        if len(distinct_items) >= 10:
            return order.total() * .07
        return 0
```

Пример работает без нареканий, но ту же функциональность можно реализовать в Python гораздо короче, воспользовавшись функциями как объектами.

# Функционально-ориентированная стратегия.

Каждая конкретная стратегия в примере — это класс с одним методом `discount`. Сильно напоминают функции. В следующем примере код переработан — конкретные классы заменены функциями, а абстрактный класс `Promo` исключен

*Класс `Order`, в котором реализован в виде функций*

```
from collections import namedtuple

Customer = namedtuple('Customer', 'name fidelity')


class LineItem:

    def __init__(self, product, quantity, price):
        self.product = product                      # Наименование
```

```
        self.quantity = quantity                    # Количество
        self.price = price                          # Цена

    def total(self):
        return self.price * self.quantity           # Общая стоимость позиции


class Order:                                        # Контекст

    def __init__(self, customer, cart, promotion=None):
        self.customer = customer
        self.cart = cart
        self.promotion = promotion

    def total(self):
        if not hasattr(self, '__total'):
            self.__total = sum(item.total() for item in self.cart)
        return self.__total

    def due(self):
        if self.promotion is None:
            discount = 0
        else:
            discount = self.promotion.discount(self)
        return self.total() - discount

    def __repr__(self):
        return f'<Order total: {self.total():.2f} due: {self.due():.2f}>'


def fidelity_promo(order):
    """
    5%-я скидка для заказчиков, имеющих не менее 1000 баллов лояльности
    """
    return order.total() * .05 if order.customer.fidelity >= 1000 else 0


def bulk_item_promo(order):
    """
    10%-я скидка для каждой позиции LineItem, в которой заказано не мене 20 единиц
    """
    discount = 0
    for item in order.cart:
        if item.quantity >= 20:
            discount += item.total() * .1
    return discount


def large_order_promo(order):
    """
    7%-я скидка для заказов, включающих не менее 10 различных позици
```

```
    """
    distinct_items = {item.product for item in order.cart}
    if len(distinct_items) >= 10:
        return order.total() * .07
    return 0
```

# Выбор наилучшей стратегии: простой подход

*Реализация функции выбора наилучшая стратегии*

```
from source.strategy_3 import fidelity_promo, bulk_item_promo, large_order_promo
promos = [fidelity_promo, bulk_item_promo, large_order_promo]
# promos - список стратегий реализованный в виде функций


def best_promo(order):
    """
    :param order: список покупок
    :return: максимально возможную скидку из params
    """
    return max(promo(order) for promo in promos)
```

Фишка в том что бы воспринимать функцию как объект, который можно передавать в виде параметра.

В данном коде возможна тонка ошибка. При написании новой стратегии, возможно забыть добавить её в список `promos`.

# Поиск стратегии в модуле

`globals()` - возвращает словарь, представляющий текущую таблицу глобальных символов. Это всегда словарь текущего модуля.

*Список promos строится путем перебора глобального пространства имен*

```
promos = [globals()[name] for name in globals() if name.endswith('_promo') and name !=
'best_promo']
"""
promos - перебираем все имена в словаре,
возвращенном функцией global(), оставляем
только те что с суффикосом _promo и не best_promo
"""


def best_promo(order):
    """
    :param order: список покупок
    :return: максимально возможную скидку из params
    """
    return max(promo(order) for promo in promos)
```

# Паттерн Команда

Цель **Команды** - разорвать связь между объектом, инициировавшим операцию (Инициатором) и объектом, который её реализует (Получатель).

В примере Инициаторы - это пункты меню в графическом редакторе, а Получателем - редактируемый документ или само приложении.

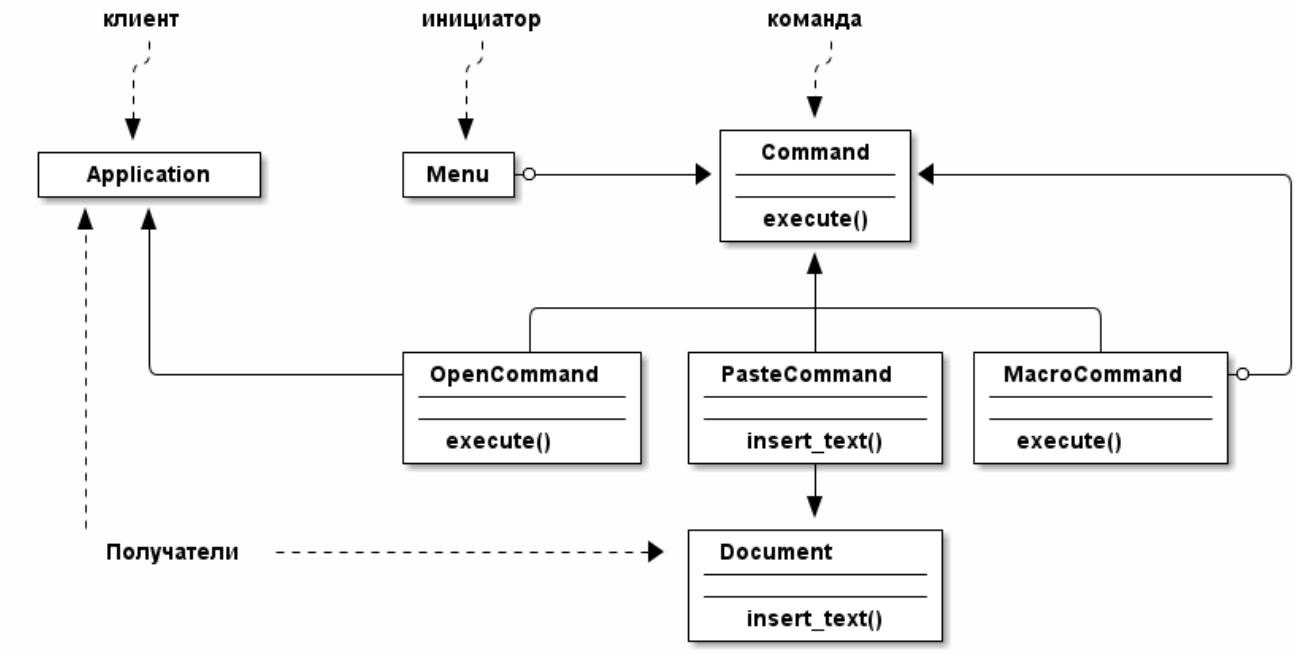



*Figure 2. UML-диаграмма классов для управляемого меню текстового редактора, реализованного с помощью паттерна Команда. У каждой команды может быть свой получатель: объект, выполняющий действие. Для команды `PasteCommand` получателем является `Document`, а для `OpenCommand`- приложение.*

Идея в том, что бы между инициатором и исполнителем поместить объект `Command`, который реализует интерфейс с одним методом `execute()`, вызывающим какой-то метод Получателя для выполнения желаемой операции. Таким образом, Инициатор ничего не знает об интерфейсе Получателя, так что, написав подкласс `Command`, можно адаптировать различные получатели. Инициатор конфигурируется конкретной командой и вызывает ее метод `execute`.

Класс `MacroCommand`, который может содержать последовательность команд; его метод `execute()` вызывает одноименные методы каждой хранимой команды.

Вместо передачи объекта `Command`, мы можем передать обычную функцию. Реализовать можно через специальный метод `__call__`.

Тогда `MacroCommand` будут вызываемыми объектами, содержащими список функций для последующего вызова.

```
class MacroCommand:
    """
    Команда, выполняющая список команд
    """
    def __init__(self, commands):
        self.commands = list(commands)

    def __call__(self, *args, **kwargs):
        for command in self.commands:
            command()
```

# Ссылочки и дополнительные материалы

- Паттерны поректирования в динамических языках.

```
#!./venv/Scripts/python.exe
# -*- coding: utf-8 -*-
"""
Паттерн СТРАТЕГИЯ определяет семейство алгоритмов, инкапсулирует
каждый из них и обеспечивает их взаимозаменяемость. Он позволяет
модифицировать алгоритмы независимо от их использования на стороне клиента.
"""
COLLEAGUES = ('Павел Клименко', 'Павел Румянцев', 'Николай Ластовский',
              'Кирилл Кулешов', 'Сергей Мирук', 'Алёна Ларина')


class Layout:
    """
    Этот класс поддерживает только один алгоритм: табуляция. Функция,
    реализующая этот алгоритм, ожидает получить счетчик строк и после-
    довательность элементов, а возвращает результат в виде таблицы.
    """
    def __init__(self, tabulator):
        self.tabulate = tabulator

    def tabulate(self, rows, items):
        return self.tabulator(rows, items)


def main():
    """
    В этой функции создаются 2 объекта Layout, параметризованные
    различными функциями-табуляторами. Для каждого формата печатается
    таблица с 2,3,4,5 строками
    :return:
    """
    htmlLayout = Layout(html_tabulator)
    for rows in range(2, 6):
        print(htmlLayout.tabulate(rows, COLLEAGUES))
    textLayout = Layout(text_tabulator)
    for rows in range(2, 6):
        print(textLayout.tabulate(rows, COLLEAGUES))
```

# Глава 7. Декораторы и функции замыкания

Декоратор - это вызываемый объект, принимающий в качестве аргумента другую функцию.

## Главное, что нужно знать о декораторах:

- Тот факт, что они властны заменить декорируемую функцию другой;
- Выполняется сразу после загрузки модуля;

## Паттерн Стратегия, дополненный декоратором

*Список* `promos` *заполняется декоратором* `promotion`

```
from collections import namedtuple

Customer = namedtuple('Customer', 'name fidelity')


class LineItem:

    def __init__(self, product, quantity, price):
        self.product = product                      # Наименование
        self.quantity = quantity                    # Количество
        self.price = price                          # Цена

    def total(self):
        return self.price * self.quantity           # Общая стоимость позиции


class Order:                                        # Контекст

    def __init__(self, customer, cart, promotion=None):
        self.customer = customer
        self.cart = cart
        self.promotion = promotion

    def total(self):
        if not hasattr(self, '__total'):
            self.__total = sum(item.total() for item in self.cart)
        return self.__total

    def due(self):
        if self.promotion is None:
            discount = 0
        else:
```

```
            discount = self.promotion.discount(self)
        return self.total() - discount

    def __repr__(self):
        return f'<Order total: {self.total():.2f} due: {self.due():.2f}>'


promos = []


def promotion(promo_func):
    promos.append(promo_func)
    return promo_func


@promotion
def fidelity(order):
    """
    5%-я скидка для заказчиков, имеющих не менее 1000 баллов лояльности
    """
    return order.total() * .05 if order.customer.fidelity >= 1000 else 0


@promotion
def bulk_item(order):
    """
    10%-я скидка для каждой позиции LineItem, в которой заказано не мене 20 единиц
    """
    discount = 0
    for item in order.cart:
        if item.quantity >= 20:
            discount += item.total() * .1
    return discount


@promotion
def large_order(order):
    """
    7%-я скидка для заказов, включающих не менее 10 различных позици
    """
    distinct_items = {item.product for item in order.cart}
    if len(distinct_items) >= 10:
        return order.total() * .07
    return 0


def best_promo(order):
    """
    Выбрать максимально возможную скидку
    """
    return max(promo(order) for promo in promos)
```

По сравнению с другими решениями, у этого есть несколько преимуществ:

- Функции, реализующие стратегии вычисления скидки могут избавиться от суффикса `_promo`
- Декоратор `@promotion` явно описывает назначение декорируемой функции и без труда позволяет временно отменить предоставление ссылки: достаточно закомментировать декоратор
- Стратегии скидки можно определить в других модулях.

# Правила видимости переменных

Видимость локальных переменных определяется при компилировании байт-кода и если одноименная переменная определена в теле функции, то она будет считаться локальной.

# Замыкания

Замыкание вступает в игру только при наличии вложенной функции.

Замыкание — это функция с расширенной областью видимости, которая охватывает все не глобальные переменные, имеющие ссылки в теле функции, хотя они в нем не определены.

Эту идею довольно трудно переварить, поэтому пример.

Рассмотрим функцию `avg`, которая вычисляет среднее продолжающегося ряда чисел, например, среднюю цену закрытия биржевого товара за всю историю торгов. Каждый день ряд пополняется новой ценой, а при вычислении среднего учитываются все прежние цены.

*Если начать с чистого листа, то функция `avg` можно было бы использовать следующим образом:*

```
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

*Вопрос на подумать*

Откуда берется `avg` и где она хранит предыдущие значения?

### Реализация Average основанная на классах.

*average_oo.py: класс для вычисления накопительного среднего значения*

```
class Averager:

    def __init__(self):
        self.series = []

    def __call__(self, new_value):
        self.series.append(new_value)
        total = sum(self.series)
        return total/len(self.series)
```

*Класс* `Averager` *создает вызываемые объекты*

```
>>> from source.average_oo import Averager
>>> avg = Averager()
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

*Результат тестирования*

```
>>> import doctest
>>> doctest.testfile('./source/doctest/avg_oo.txt')
TestResults(failed=0, attempted=5)
```

## Функциональная реализация с использованием функции высшего порядка `make_averager`

```
def make_averager():
    """
    При обращении к make_averager возвращается объект-функция averager.
    При каждом вызове averager добавляет переданный аргумент в конец
    списка series и вычисляет текущее среднее.
    :return:
    """
    series = []

    def averager(new_value):
        series.append(new_value)
        total = sum(series)
        return total/len(series)

    return averager
```

Обратите внимание на сходство обоих примеров: мы обращаемся к `Averager` и к `make_averager` что бы получить вызываемый объект `avg`, который обновляет временной ряд и вычисляет среднее значение.

Совершенно ясно, где хранит историю объектов `avg` класса `Averager`: в аттрибуте экземпляра `self.series`. Но где находится `series` функции `avg` из второго примера?

Внутри `averager` переменная `series` является *свободной*. Этот технический термин обозначает, что переменная не связана в локальной области видимости.

**Python** хранит имена локальных и свободных переменных в аттрибуте `__code__`, который представляет собой откомпилированное тело функции.

*Инспекция функции, созданной функцией* `make_averager`

```
>>> from source.average import make_averager
>>> avg = make_averager()
>>> avg.__code__.co_varnames
('new_value', 'total')
>>> avg.__code__.co_freevars
('series',)
```

Привязка переменной `series` хранится в аттрибуте `__closure__` возвращенной функцией `avg`.

Каждому элементу `avg.__closure__` соответствует имя в `avg.__code__.co_freevars`. Эти элементы называются *ячейками(cells)*, и у каждого из них есть атрибут `cell_contents`, где можно найти само значение.

*Инспекция функции, созданной функцией* `make_averager` *(продолжение)*

```
```

*Резюмируем:*

**Замыкание** — это функция, которая запоминает привязку свободных переменных, существовавшие на момент определения функции. Так что их можно использовать впоследствии при вызове функции, когда область видимости, в которой она была определена уже не существует.

Единственная ситуация, когда функции может понадобиться доступ к внешней не глобальной переменной,- это когда она вложена в другую функцию.

## Объявление `nonlocal`

Приведенная ранее реализация `make_averager` не эффективна. Мы храним в переменной все значения и каждый раз вычисляем их сумму при каждом вызове `averager`. Лучше было бы

хранить предыдущую сумму и количество элементов, тогда зная два числа мы можем вычислить среднее.

*В Pyton 3 было добавлено nonlocal*

**nonlocal** позволяет пометить переменную как свободную, даже если ей присваивается значение внутри функции. В таком случае изменяется привязка, хранящаяся в замыкании.

*Правильная реализация идеи*

```
def make_averager():
    count = 0
    total = 0

    def averager(new_value):
        nonlocal total, count
        count += 1
        total += new_value
        return total/count

    return averager
```

# Реализация простого декоратора

*Простой декоратор для вывода времени выполнения функции*

```
import functools
import time


def clock(func):
    """
        Декоратор functools.wraps копирует аргументы
        декорируемой функции.
    """
    @functools.wraps(func)
    def clocked(*args, **kwargs):
        t0 = time.perf_counter()
        result = func(*args, **kwargs)
        elapsed = time.perf_counter() - t0
        name = func.__name__
        arg_list = []
        if args:
            arg_list.append(', '.join(repr(arg) for arg in args))
        if kwargs:
            pairs = [f'{k}={w}' for k, w in sorted(kwargs.items())]
            arg_list.append(', '.join(pairs))
        arg_string = ', '.join(arg_list)
        print(f'[{elapsed:0.8f}] {name}({arg_string} -> {result})')     # [0.00000120]
factorial(1 -> 1)
        return result
    return clocked
```

*Использование декоратора clock*

```
# clock_decorator_demo.py
import time
from clock_decorator import clock


@clock
def snooze(seconds):
    time.sleep(seconds)


@clock
def factorial(n):
    return 1 if n < 2 else n*factorial(n-1)


if __name__ == '__main__':
    print('*' * 40, 'Calling snooze(.123)')
    snooze(.123)
    print('*' * 40, 'Calling factorial(5)')
    print('5! =', factorial(5))
```

*Результат выполнения*

```
PS C:\Users\User\PycharmProjects\Python. Design patterns and other\source> py
.\clock_decorator_demo.py
**************************************** Calling snooze(.123)
[0.13673660] snooze(0.123 -> None)
**************************************** Calling factorial(5)
[0.00000120] factorial(1 -> 1)
[0.00017130] factorial(2 -> 2)
[0.00031230] factorial(3 -> 6)
[0.00047800] factorial(4 -> 24)
[0.00066200] factorial(5 -> 120)
5! = 120
```

# Декораторы в стандартной библиотеке

Два самых любопытных декоратора в стандартной библиотеке - `lru_cache` и совсем новый `singledispatch` *(Python >= 3.4)*, оба определены в `functools`.

## Кэширование с помощью `functools.lru_cache`

*Декоратор `functools.lru_cache` очень полезен на практике*

Он реализует "запоминание" (memorization): прием оптимизации, смысл которого заключается в запоминании дорогостоящих вычислений, позволяет избежать повторных вычислений с теми же аргументами, что и раньше.

*Пример использования кэширования*

```
import functools
from contextlib import redirect_stdout
from clock_decorator import clock


#@functools.lru_cache()
@clock
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-2) + fibonacci(n-1)


if __name__ == '__main__':
    with open('./doctest/fibo_demo_out.txt', 'a') as f:
        with redirect_stdout(f):
            print(fibonacci(10))
```

На примере функция уже декорирована. Для сравнения, вот выводы декорированной и не декорированной функции `fibonacci`:

*Вывод скрипта без использования `lru_cache`. Очевидны лишние вычисления.*

```
[0.00000050] fibonacci(0 -> 0)
[0.00000040] fibonacci(1 -> 1)
[0.00003930] fibonacci(2 -> 1)
[0.00000030] fibonacci(1 -> 1)
[0.00000030] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000870] fibonacci(2 -> 1)
[0.00001650] fibonacci(3 -> 2)
[0.00006430] fibonacci(4 -> 3)
[0.00000020] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000720] fibonacci(2 -> 1)
[0.00001430] fibonacci(3 -> 2)
[0.00000020] fibonacci(0 -> 0)
[0.00000030] fibonacci(1 -> 1)
[0.00000710] fibonacci(2 -> 1)
[0.00000030] fibonacci(1 -> 1)
```

```
[0.00000240] fibonacci(0 -> 0)
[0.00000030] fibonacci(1 -> 1)
[0.00001180] fibonacci(2 -> 1)
[0.00001890] fibonacci(3 -> 2)
[0.00003320] fibonacci(4 -> 3)
[0.00005450] fibonacci(5 -> 5)
[0.00012600] fibonacci(6 -> 8)
[0.00000020] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000730] fibonacci(2 -> 1)
[0.00001410] fibonacci(3 -> 2)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000710] fibonacci(2 -> 1)
[0.00000020] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000030] fibonacci(1 -> 1)
[0.00000700] fibonacci(2 -> 1)
[0.00001380] fibonacci(3 -> 2)
[0.00002810] fibonacci(4 -> 3)
[0.00004980] fibonacci(5 -> 5)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000710] fibonacci(2 -> 1)
[0.00000020] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000710] fibonacci(2 -> 1)
[0.00001460] fibonacci(3 -> 2)
[0.00002830] fibonacci(4 -> 3)
[0.00000020] fibonacci(1 -> 1)
[0.00000030] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000720] fibonacci(2 -> 1)
[0.00001410] fibonacci(3 -> 2)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000740] fibonacci(2 -> 1)
[0.00000020] fibonacci(1 -> 1)
[0.00000050] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000810] fibonacci(2 -> 1)
[0.00001680] fibonacci(3 -> 2)
[0.00003080] fibonacci(4 -> 3)
[0.00005170] fibonacci(5 -> 5)
[0.00008660] fibonacci(6 -> 8)
[0.00014380] fibonacci(7 -> 13)
[0.00027750] fibonacci(8 -> 21)
[0.00000030] fibonacci(1 -> 1)
[0.00000030] fibonacci(0 -> 0)
```

```
[0.00000020] fibonacci(1 -> 1)
[0.00000700] fibonacci(2 -> 1)
[0.00001400] fibonacci(3 -> 2)
[0.00000030] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000710] fibonacci(2 -> 1)
[0.00000020] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000890] fibonacci(2 -> 1)
[0.00001620] fibonacci(3 -> 2)
[0.00003080] fibonacci(4 -> 3)
[0.00005150] fibonacci(5 -> 5)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000720] fibonacci(2 -> 1)
[0.00000030] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00002470] fibonacci(2 -> 1)
[0.00003160] fibonacci(3 -> 2)
[0.00004530] fibonacci(4 -> 3)
[0.00000020] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000700] fibonacci(2 -> 1)
[0.00001380] fibonacci(3 -> 2)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000700] fibonacci(2 -> 1)
[0.00000020] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000030] fibonacci(1 -> 1)
[0.00001540] fibonacci(2 -> 1)
[0.00002220] fibonacci(3 -> 2)
[0.00003610] fibonacci(4 -> 3)
[0.00005660] fibonacci(5 -> 5)
[0.00010850] fibonacci(6 -> 8)
[0.00016680] fibonacci(7 -> 13)
[0.00000020] fibonacci(0 -> 0)
[0.00000030] fibonacci(1 -> 1)
[0.00000730] fibonacci(2 -> 1)
[0.00000030] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000710] fibonacci(2 -> 1)
[0.00001430] fibonacci(3 -> 2)
[0.00002820] fibonacci(4 -> 3)
[0.00000020] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
```

```
[0.00000700] fibonacci(2 -> 1)
[0.00001410] fibonacci(3 -> 2)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000700] fibonacci(2 -> 1)
[0.00000020] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000700] fibonacci(2 -> 1)
[0.00001390] fibonacci(3 -> 2)
[0.00002730] fibonacci(4 -> 3)
[0.00004810] fibonacci(5 -> 5)
[0.00008580] fibonacci(6 -> 8)
[0.00000020] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000830] fibonacci(2 -> 1)
[0.00001530] fibonacci(3 -> 2)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000710] fibonacci(2 -> 1)
[0.00000030] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000730] fibonacci(2 -> 1)
[0.00001580] fibonacci(3 -> 2)
[0.00003140] fibonacci(4 -> 3)
[0.00005330] fibonacci(5 -> 5)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000710] fibonacci(2 -> 1)
[0.00000020] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000030] fibonacci(1 -> 1)
[0.00000700] fibonacci(2 -> 1)
[0.00001440] fibonacci(3 -> 2)
[0.00002800] fibonacci(4 -> 3)
[0.00000020] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000730] fibonacci(2 -> 1)
[0.00001400] fibonacci(3 -> 2)
[0.00000030] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000930] fibonacci(2 -> 1)
[0.00000020] fibonacci(1 -> 1)
[0.00000030] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00003070] fibonacci(2 -> 1)
[0.00003770] fibonacci(3 -> 2)
[0.00005370] fibonacci(4 -> 3)
```

```
[0.00007440] fibonacci(5 -> 5)
[0.00010920] fibonacci(6 -> 8)
[0.00016910] fibonacci(7 -> 13)
[0.00026170] fibonacci(8 -> 21)
[0.00043520] fibonacci(9 -> 34)
[0.00072080] fibonacci(10 -> 55)
55
```

*Вывод скрипта с* `lru_cache`.

```
[0.00000140] fibonacci(0 -> 0)
[0.00000040] fibonacci(1 -> 1)
[0.00005920] fibonacci(2 -> 1)
[0.00000070] fibonacci(3 -> 2)
[0.00007080] fibonacci(4 -> 3)
[0.00000060] fibonacci(5 -> 5)
[0.00008260] fibonacci(6 -> 8)
[0.00000060] fibonacci(7 -> 13)
[0.00009310] fibonacci(8 -> 21)
[0.00000070] fibonacci(9 -> 34)
[0.00010440] fibonacci(10 -> 55)
55
```

`lru_cache` необходимо вызывать как функцию со скобками. `functools.lru_cache()`. Причина в том, что декоратор принимает конфигурационные параметры.

*Полная сигнатура*

`functools.lru_cache(max_size=128, typed=False)`

- `maxsize` — сколько результатов хранить (*для достижения результата maxsize* = $n^2$).
- `type` - если стоит True, то результаты разных типов будут храниться порознь.

# Одиночная диспетчеризация и обобщенные функции

`functools.singledispatch` - (Python >= 3.4) позволяет каждому модулю вносить свой вклад в общее решение, так, что пользователь может легко добавить специализированную функцию, даже не имея возможности изменить класс.

Обычная функция, декорированная `singledispatch` становится *обобщенной функцией*: группой функций, выполняющих одну и ту же логическую операцию по-разному в зависимости от типа первого аргумента.

*Декоратор `functools.singledispatch` создает функцию `htmlize.register` для объединения нескольких функций в одну обобщенную.*

```
from collections import abc
import html
import numbers
from functools import singledispatch


@singledispatch             # Помечает базовую функцию, которая обрабатывает obj
def htmlize(obj):
    content = html.escape(repr(obj))
    return f'<pre>{content}</pre>'


@htmlize.register(str)      # Каждая специальная функция снабжается декоратором
def _(text):                # Имена функций не существенны
    content = html.escape(text).replace('\n', '<br>\n')
    return f'<pre>{content}</pre>'


@htmlize.regiter(numbers.Integral)
def _(n):
    return f'<pre>{n} (0x{0:x})</pre>'


@htmlize.register(tuple)
@htmlize.register(abc.MutableSequence)
def _(seq):
    inner = '</li>\n<li>'.join(htmlize(item) for item in seq)
    return f'<ul>\n<li>{inner}</li>\n<li>'
```

Замечательное свойство данного декоратора в том, что специализированные функции можно зарегистрировать в любом месте системы, в любом модуле. Если в последствии, вы добавите модуль, содержащий новый пользовательский тип, то без труда сможете новую специализированную функцию для обработки данного типа.

*Возможности этого декоратора шире, подробнее можно почитать тут:*

- PEP-0443 Single-dispatch generic function

*Дополнительный пример диспетчеризации.*

```
from datetime import date, datetime, time
from functools import singledispatchmethod


class Formatter:
    """
    Пример использования декоратора для диспетчеризациия
    функций в зависимости от типа аргумента

    https://martinheinz.dev/blog/50
    """
    @singledispatchmethod
    def format(self, args):
        raise NotImplementedError(f'Can not format value of type {type(args)}')

    @format.register
    def _(self, args: date):
        return args.strftime('%d/%m/%Y')

    @format.register
    def _(self, args: datetime):
        return args.strftime('%d/%m/%Y %H:%M:%S')

    @format.register
    def _(self, args: time):
        return args.strftime('%H:%M:%S')

    def __call__(self, args):
        return self.format(args)


if __name__ == '__main__':
    f = Formatter()
    samples = (date(2021, 2, 14), time(12, 59, 50), datetime(2021, 2, 14, 12, 59, 50))
    print(list(f(sample) for sample in samples))
```

# Композиция декораторов

Когда два декоратора `@d1` и `@d2` применяются к одной и той же функции `f` в указанном порядке, получается то же самое, что и в результате композиции `f=d1(d2(f))`

# Параметризованные декораторы

Для реализации параметризованного декоратора, необходимо создать *фабрику декораторов*. Т.е. создать функцию, которая возвращает декоратор.

# Ссылочки

- A Python module for decorators, wrappers and monkey patching.
- pip install decorator
- Python Decorator Library
- PEP 443 — Single-dispatch generic functions
- PEP 3104 — Access to Names in Outer Scopes
- The Correct Way to Overload Functions in Python :experimental: :doctype: book :icons: font :myprojectbasedir: {asciidoctorconfigdir} :doctestdir: {asciidoctorconfigdir}/../../source/doctest :sourcedir: {asciidoctorconfigdir}/../../source :imagesoutdir: ./images

# Глава 8. Ссылки на объекты

Для правильного понимания присваивания в Python всегда сначала читайте правую часть, ту, где объект создается или извлекается. Уже после этого переменная в левой части связывается с объектом — как приклеенная к нему этикетка.

Поскольку переменные — это просто этикетки, ничего не мешает наклеить на объект несколько этикеток. В этом случае образуются *синонимы*.

## Выбор между `==` и `is`

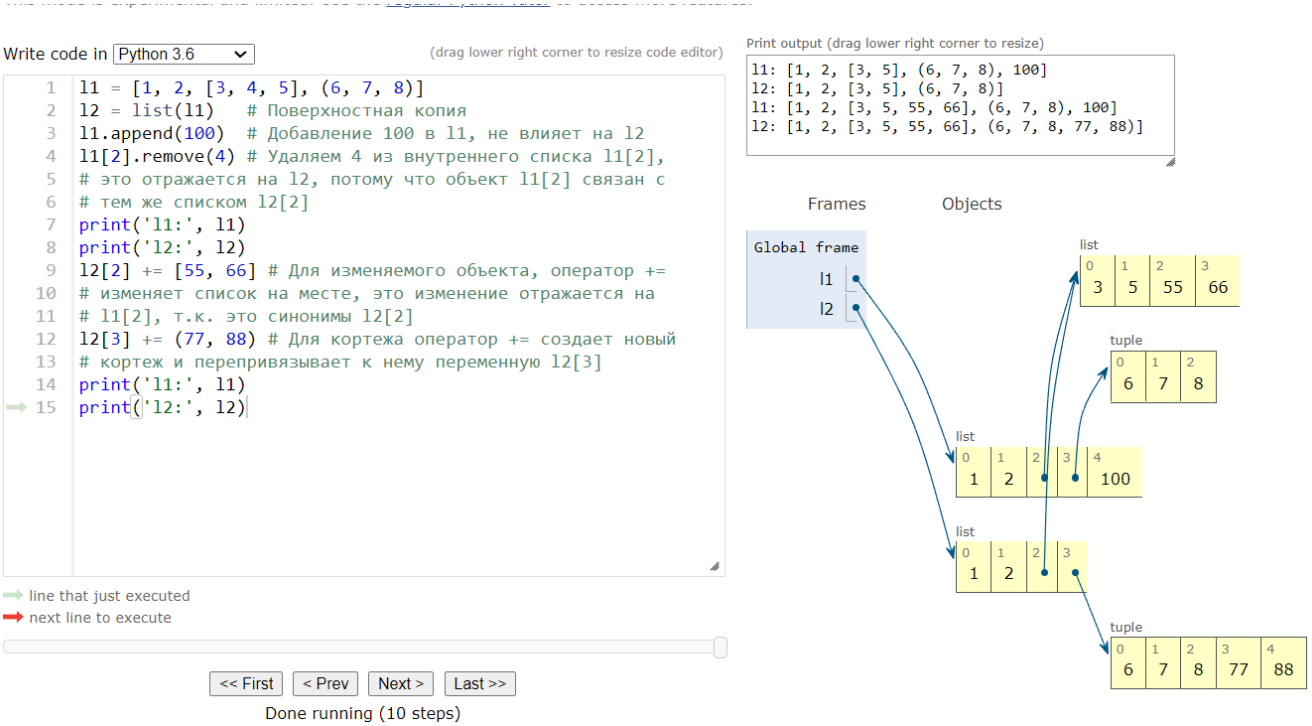
Оператор `==` сравнивает значение объектов (хранящихся в нем данных), а оператор `is` — их `id`.

## Относительная неизменность кортежей

Кортежи, как и большинство коллекций в Python, — списки, словари, множества и т.д. — хранят ссылки на объекты. Если элементы, на которые указывают ссылки, изменяемы, то их можно модифицировать, хотя сам кортеж останется неизменяемый.

## По умолчанию копирование поверхностное

*Создание поверхностной копии списка, содержащий другой список;*




Теперь должно быть понятно, что создать поверхностную копию легко, но это не всегда то что нужно.

Для значений по умолчанию `def func(a=default):` необходимо устанавливать значение None, а не изменяемые пустые объекты типа `[]`. Это может привести к изменению объекта установленного по умолчанию и во вновь созданных объектах будут фантомные объекты.

*Ошибка при назначении по умолчанию изменяемого объекта*

Write code in Python 3.6 (drag lower right corner to resize code editor)

```
1  class HauntedBuss:
2      def __init__(self, passangers=[]):
3          self.passangers = passangers
4
5      def pick(self, name):
6          self.passangers.append(name)
7
8      def drop(self, name):
9          self.passangers.remove(name)
10
11 bus1 = HauntedBuss(['Олеся', 'Артём'])
12 print(bus1.passangers)
13 bus1.pick('Антон')
14 bus1.drop('Олеся')
15 print(bus1.passangers)
16 bus2 = HauntedBuss()
17 bus2.pick('Оксана')
18 print(bus2.passangers)
19 bus3 = HauntedBuss()    # Объект, новый но уже
20 # содержит в списке пассажира из bus2
21 print(bus3.passangers)
```

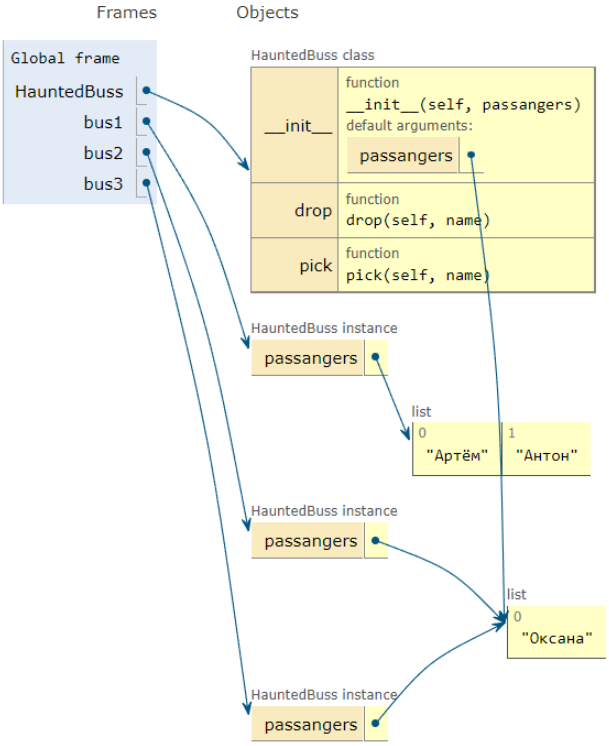
line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Done running (29 steps)

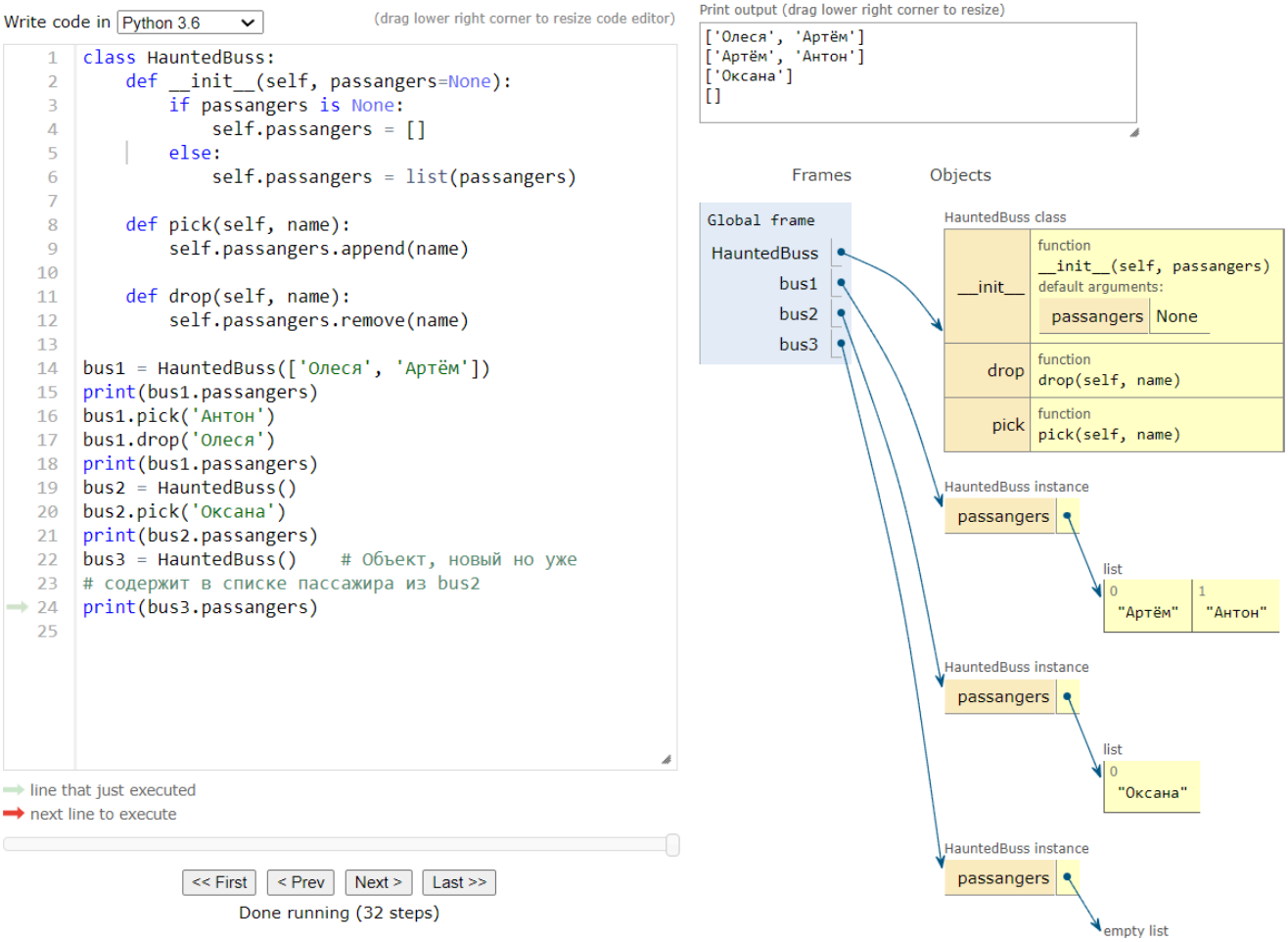
Print output (drag lower right corner to resize)

```
['Олеся', 'Артём']
['Артём', 'Антон']
['Оксана']
['Оксана']
```




При написании функции, принимающий изменяемый параметр, необходимо тщательно обдумывать, ожидает ли принимающая сторона, что аргумент может быть изменён.

*Пример правильной реализации*




Важное замечание. В объекте автобуса при инициализации создается копия списка. Это исключает ошибки связанной с изменением объекта переданного при инициализации.

```
class HauntedBuss:
    def __init__(self, passangers=None):
        if passangers is None:
            self.passangers = []
        else:
            self.passangers = list(passangers)
            # Если тут сделать просто self.passangers = passangers,
            # эти объекты будут синонимами.

    def pick(self, name):
        self.passangers.append(name)

    def drop(self, name):
        self.passangers.remove(name)
```

*doctest для проверки корректного автобуса*

```
>>> from source.buss import HauntedBuss
>>> bus1 = HauntedBuss(['Олеся', 'Артём'])
>>> bus1.passangers
['Олеся', 'Артём']
>>> bus1.pick('Антон')
>>> bus1.drop('Олеся')
>>> bus1.passangers
['Артём', 'Антон']
>>> bus2 = HauntedBuss()
>>> bus2.pick('Оксана')
>>> bus2.passangers
['Оксана']
>>> bus3 = HauntedBuss()
>>> bus3.passangers
[]
>>> footboal_team = ['Артем', 'Гриша', 'Пётр', 'Антон', 'Лакки']
>>> bus4 = HauntedBuss(footboal_team)
>>> bus4.passangers
['Артем', 'Гриша', 'Пётр', 'Антон', 'Лакки']
>>> bus4.drop('Антон')
>>> footboal_team
['Артем', 'Гриша', 'Пётр', 'Антон', 'Лакки']
>>> bus4.passangers
['Артем', 'Гриша', 'Пётр', 'Лакки']
```

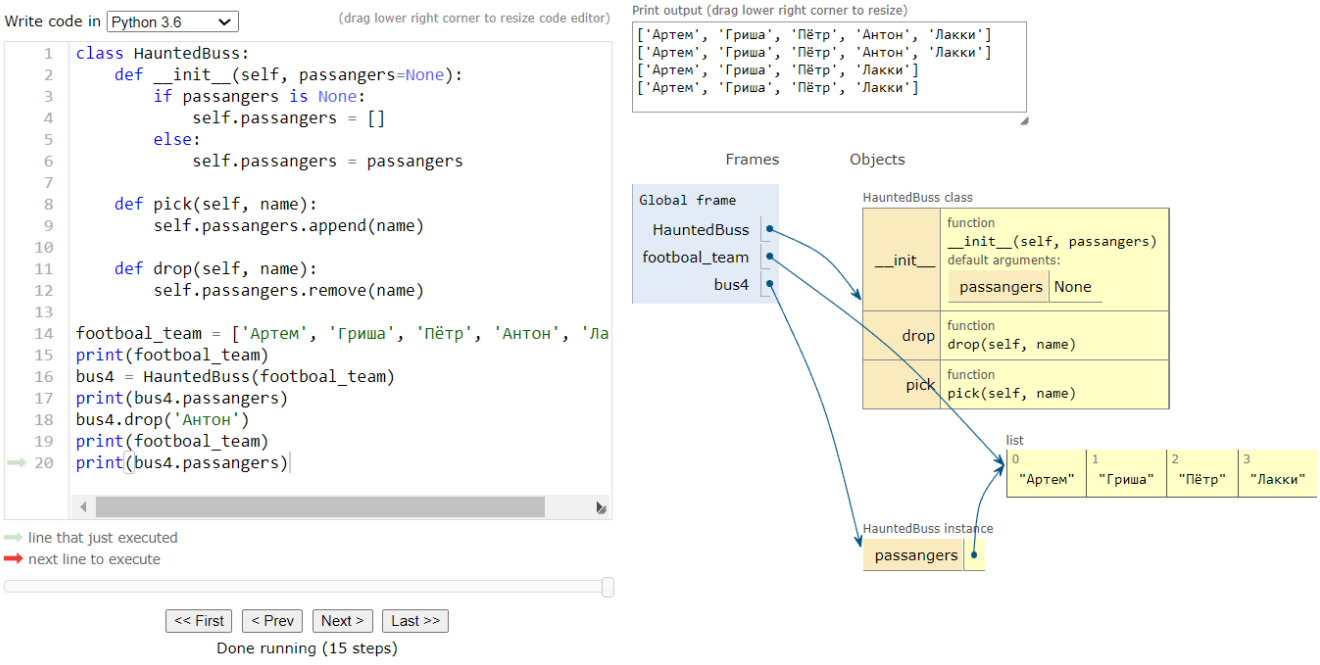
*Если сделать вот так:*

```
self.passangers = passangers
```

*То футбольная команда потеряет одного игрока вместе с выходом из автобуса.*

```
doctest.testfile('./source/doctest/buss_correct.txt')
**********************************************************************
File ".\source\doctest\buss_correct.txt", line 21, in buss_correct.txt
Failed example:
    footboal_team
Expected:
    ['Артем', 'Гриша', 'Пётр', 'Антон', 'Лакки']
Got:
    ['Артем', 'Гриша', 'Пётр', 'Лакки']
**********************************************************************
1 items had failures:
   1 of  17 in buss_correct.txt
***Test Failed*** 1 failures.
TestResults(failed=1, attempted=17)
```

*Визуализация потери игрока*




Если метод специально не предназначен для изменения объекта, полученного в качестве аргумента, то стоит дважды подумать создавая синоним аргумента, просто присваивая его атрибуту экземпляра в своём классе. Если сомневаетесь, делайте копию. Клиенты обычно будут только рады.

# del и сборка мусора

Предложение `del` удаляет имена, а не объекты. Это может привести к отсутствию ссылок и может быть удалён сборщиком мусора. Привязывание переменной к другому объекту так же может обнулить количество ссылок на объект, что так же приведёт к уничтожению.

Существует специальный метод `__del__`, но он не приводит к уничтожению экземпляра. И Вы не должны вызывать его вручную. Метод `__del__` вызывается интерпретатором **Python** непосредственно перед уничтожением объекта, давая ему возможность освободить внешние ресурсы.

# Слабые ссылки

Наличие ссылок — вот что удерживает объект в памяти. Слабые ссылки на объект не увеличивают счётчик ссылок. Слабая ссылка не препятствует уничтожению объектов.

Слабые ссылки хороши для хранения хэш значений.

**weakref** - модуль для работы с множествами и слабыми ссылками.

`weakref.ref` создает слабую ссылку на объект. :experimental: :doctype: book :icons: font :myprojectbasedir: {asciidoctorconfigdir} :doctestdir: {asciidoctorconfigdir}/../../source/doctest :sourcedir: {asciidoctorconfigdir}/../../source :imagesoutdir: ./images

# Глава 9. Объекты в духе Python

## Представление объекта

- `repr()` — вернуть строку, представляющую объект в виде, удобном для разработчика.
- `str()` — вернуть строку, представляющую объект в виде, удобном для пользователя.

Для поддержки repr и str мы должны реализовать методы `__repr__` и `__str__`. Существует еще два специальным метода для поддержки альтернативных представлений объектов `__bytes__` и `__format__`

## И снова класс вектора

*class Vector2d. Пока, что реализованы только специальные методы*

```
import math
from array import array


class Vector2d:
    """
    :typecode: -- аттрибут класса для преобразования
    экземпляра Vector2d последовательность байтов и обратно.

    >>> v1 = Vector2d(3, 4)
    >>> print(v1.x, v1.y)
    3.0 4.0
    >>> x, y = v1
    >>> x, y
    (3.0, 4.0)
    >>> v1
    Vector2d(3.0, 4.0)
    >>> v1_clone = eval(repr(v1))
    >>> v1 == v1_clone
    True
    >>> print(v1)
    (3.0, 4.0)
    >>> octets = bytes(v1)
    >>> abs(v1)
    5.0
    >>> bool(v1), bool(Vector2d(0, 0))
    (True, False)
    """
    typecode = 'd'

    def __init__(self, x, y):
        """
        Преобразование x, y в тип float в методе инициализации
```

```
        позволяет на ранней стадии обнаруживать ошибки. Это
        полезно когда конструктор вызывается с не подходящими
        аргументами
        """
        self.x = float(x)
        self.y = float(y)

    def __iter__(self):
        """
        Наличие __iter__ делает объект итерируемым.
        Благодаря ему работает распаковка (пр.: x, y = my_vector).
        В данном случае реализация при помощи генераторного
        выражения, который отдает компоненты поочередно.
        """
        return (i for i in (self.x, self.y))

    def __repr__(self):
        """
        Метод __repr__ строит строку, интерполируя компоненты с
        помощью синтаксиса {!r} для получения их представления,
        возвращаемого функцией repr; Поскольку Vector2d - итерируемый
        объект, *self поставляет компоненты x и y функции format.
        """
        class_name = type(self).__name__
        return "{}({!r}, {!r})".format(class_name, *self)

    def __str__(self):
        """
        Из итерируемого объекта легко построить кортеж для отображения
        в виде упорядоченной пары.
        """
        return str(tuple(self))

    def __bytes__(self):
        """
        Для генерации объекта типа bytes мы преобразуем typecode
        в bytes и конкатенируем с объектом bytes, полученным
        преобразованием массива, который построен путём обхода
        экземпляра.
        """
        return (bytes([ord(self.typecode)]) +
                bytes(array(self.typecode, self)))

    def __eq__(self, other):
        """
        Для быстрого сравнения всех компонентов мы строим кортеж
        их операндов. Это работает, когда операнды являются экземплярами
        класса Vector2d, но не без проблем.
        """
        return tuple(self) == tuple(other)
```

```
    def __abs__(self):
        """
        Модулем вектора называется длина гипотенузы прямоугольного
        треугольника, где катеты x и y.
        """
        return math.hypot(self.x, self.y)

    def __bool__(self):
        """
        Метод __bool__ вызывает abs(self) для вычисления модуля, а
        затем преобразует полученное значение в тип bool, так что
        0.0 преобразуется в False, а любое отличное от нуля в True
        """
        return bool(abs(self))

    def __format__(self, format_spec=''):
        """
        :param format_spec: применяется к каждому компоненту вектора
        с помощью встроенной функцией format и строит итерируемый объект,
        порождающий отформатированные строки.
        :return: Подставляем отформатированные строки в шаблон (x, y).
        """
        components = (format(c, format_spec) for c in self)
        return '({}, {})'.format(*components)

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

*У экземпляров Vector2d есть несколько представлений.*

```
>>> from source.vector2d_v0 import Vector2d
>>> v1 = Vector2d(3, 4)
>>> print(v1.x, v1.y)
3.0 4.0
>>> x, y = v1
>>> x, y
(3.0, 4.0)
>>> v1
Vector2d(3.0, 4.0)
>>> v1_clone = eval(repr(v1))
>>> v1 == v1_clone
True
>>> print(v1)
(3.0, 4.0)
>>> octets = bytes(v1)
>>> octets
b'd\x00\x00\x00\x00\x00\x00\x08@\x00\x00\x00\x00\x00\x00\x10@'
>>> abs(v1)
5.0
>>> bool(v1), bool(Vector2d(0, 0))
(True, False)
```

Метод `__eq__` работает для операндов типа Vector2d, но возвращает True и в случае, когда экземпляр Vector2d сравнивается с другими итерируемыми объектами, содержащими точно такие же числовые значения (например `Vector(3, 4) == [3, 4]`). Считать ли это ошибкой или нет, зависит от точки зрения.

У нас имеется довольно полный набор базовых методов, но одного не хватает: восстановления объекта Vector2d из двоичной последовательности, порожденного функцией bytes.

# Альтернативный конструктор

*Добавлен метод класса frombytes*

```
import math
from vector2d_v0 import Vector2d


class Vector2dv1(Vector2d):
    def __init__(self, x, y):
        super().__init__(x, y)

    @classmethod
    def from_bytes(cls, octets):
        """
```

```
        Метод класса снабжён декоратором classmethod
        Аргумент self отсутствует; вместо него в аргументе
        cls передается сам класс.
        >>> v1 = Vector2d(3, 4)
        >>> v2 = Vector2dv1.from_bytes(bytes(v1))
        >>> v1 == v2
        True
        >>> v2
        Vector2dv1(3.0, 4.0)
        >>> v1
        Vector2d(3.0, 4.0)
    """
    typecode = chr(octets[0])   # Читаем typecode из первого байта
    memv = memoryview(octets[1:]).cast(typecode)
    """
        Создаем объект memoryview из двоичной последовательности
        октетов и приводим его к типу typecode
        Распаковываем memoryview, получившийся в результате приведения
        типа, и получаем пару аргументов, необходимых конструктору
    """
    return cls(*memv)

def angle(self):
    """
    Данный метод для получения угла.
    :return: atg
    """
    return math.atan2(self.y, self.x)

def __format__(self, format_spec=''):
    """
    Переопределю метод для возможности вывода в полярных координатах.
    :param format_spec: если оканчивается на 'p': полярные координаты.
    :return: Подставляет строки во внешний формат.

    >>> format(Vector2dv1(1,1), '.3ep')
    '<1.414e+00, 7.854e-01>'
    >>> format(Vector2dv1(1,1), '0.5fp')
    '<1.41421, 0.78540>'
    >>> format(Vector2dv1(1,1), '0.5f')
    '(1.00000, 1.00000)'
    >>> format(Vector2dv1(1,1), '0.5fp')
    '<1.41421, 0.78540>'
    """
    if format_spec.endswith('p'):
        format_spec = format_spec[:-1]
        coords = (abs(self), self.angle())
        outer_fmt = '<{}, {}>'
    else:
        coords = self
        outer_fmt = '({}, {})'
```

```
        components = (format(c, format_spec) for c in coords)
        return outer_fmt.format(*components)


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

*Пример использования альтернативного конструктора*

```
>>> from source.vector2d_v0 import Vector2d
>>> v1 = Vector2d(3, 4)
>>> print(v1.x, v1.y)
3.0 4.0
>>> x, y = v1
>>> x, y
(3.0, 4.0)
>>> v1
Vector2d(3.0, 4.0)
>>> v1_clone = eval(repr(v1))
>>> v1 == v1_clone
True
>>> print(v1)
(3.0, 4.0)
>>> octets = bytes(v1)
>>> octets
b'd\x00\x00\x00\x00\x00\x00\x08@\x00\x00\x00\x00\x00\x00\x10@'
>>> abs(v1)
5.0
>>> bool(v1), bool(Vector2d(0, 0))
(True, False)
```

## Декораторы `classmethod` и `staticmethod`

`@classmethod` — определяет метод на уровне класса, а не отдельного экземпляра. Данный декоратор изменяет способ вызова метода таким образом, что в качестве первого аргумента передается сам класс, а не экземпляр. Типичное применение — альтернативные конструкторы, подобные `frombytes` из примера выше.

`@staticmethod` — изменяет метод так, что он не получает в первом аргументе ничего специального. По существу, статический метод — это просто обычная функция, определенная в теле класса, а не на уровне модуля.

*Метод `__format__` позволяет расширить мини-язык форматирования. В нашем случае возможность вывода вектора в полярных координатах.*

```
>>> from source.vector2d_v1 import Vector2dv1
>>> format(Vector2dv1(1,1), '.3ep')
'<1.414e+00, 7.854e-01>'
>>> format(Vector2dv1(1,1), '0.5fp')
'<1.41421, 0.78540>'
>>> format(Vector2dv1(1,1), '0.5f')
'(1.00000, 1.00000)'
>>> format(Vector2dv1(1,1), '0.5fp')
'<1.41421, 0.78540>'
```

# Хэшируемый класс Vector2d

До сих пор класс Vector2d не был хэшируемым и мы не могли поместить его во множество.

Что бы класс стал хэшируемым необходимо реализовать метод `__hash__`

Необходим еще метод `__eq__`, но он у нас уже есть.

Нужно сделать класс не изменяемым. Мы добьемся этого сделав `x` и `y` аттрибутами только на чтение.

*Пример реализации закрытых свойств*

```
from vector2d_v1 import Vector2dv1


class Vector(Vector2dv1):
    __slots__ = ('__x', '__y')
    """
    Реализация класса Vector с не изменяемыми атрибутами
    """
    def __init__(self, x, y):
        """
        Используем __ что бы сделать атрибуты закрытыми
        """
        self.__x = float(x)
        self.__y = float(y)

    @property   # Определяет метод чтения свойств
    def x(self):
        return self.__x

    @property
    def y(self):
        return self.__y

    def __iter__(self):
        return (i for i in (self.x, self.y))
```

Теперь когда вектор не изменяемый, мы должны реализовать `__hash__`.

*Пример реализации метода хэширования.*

```
    def __hash__(self):
        """
        >>> v1 = Vector(3,4)
        >>> v2 = Vector(3.1, 4.2)
        >>> hash(v1), hash(v2)
        (7, 384307168202284039)
        >>> set([v1, v2])
        {Vector(3.1, 4.2), Vector(3.0, 4.0)}
        """
        return hash(self.x) ^ hash(self.y)


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

В рекомендациях по специальному методу `__hash__` (3.10.1 Documentation » The Python Language Reference » 3. Data model) рекомендуется объединять хэши компонентов при помощи поразрядного оператора **ИСКЛЮЧАЮЩЕЕ ИЛИ** (^)

## Экономия памяти с помощью аттрибута `__slots__`

*Определяя в классе аттрибут __slots__ мы говорим интерпретатору: "Это все аттрибуты экземпляра в данном классе"*

```
class Vector(Vector2dv1):
    __slots__ = ('__x', '__y')
```

Тогда Python помещает их в кортеже-подобную структуру в каждом экземпляре, что позволяет избежать накладных рассходов по использованию словаря `__dict__`

# Глава 10. Рубим, нарезаем и перемешиваем последовательности

## Vector, попытка N1: совместимость с Vector2d

В этой главе мы напишем класс Vector для многомерного массива.

*Vector, попытка □1: совместимость с Vector2d*

```
import math
import reprlib
from array import array


class Vector:
    typecode = 'd'

    def __init__(self, components):
        """
        В "защищённом" атрибуте экземпляра self._comoponents
        хранится массив array компонент Vector

        >>> Vector([3.1, 4.2])
        Vector([3.1, 4.2])
        >>> Vector((3, 4, 5))
        Vector([3.0, 4.0, 5.0])
        >>> Vector(range(10))
        Vector([0.0, 1.0, 2.0, 3.0, 4.0, ...])
        """
        self._components = array(self.typecode, components)

    def __iter__(self):
        """
        Что бы была возможность иттерировать объект
        возвращаем иттератор основаный на _components

        >>> for i in Vector([1, 2, 3, 4]):
        ...     print(i)
        ...
        1.0
        2.0
        3.0
        4.0
        """
        return iter(self._components)

    def __repr__(self):
        """
        Используем reprlib.repr() для получения представления
```

```
        self._components ограниченной длины

        >>> v1 = Vector((1, 2, 3, 4, 5))
        >>> repr(v1)
        'Vector([1.0, 2.0, 3.0, 4.0, 5.0])'
        >>> v1 = Vector([1, 2, 3, 4, 5, 6, 7, 8])
        >>> repr(v1)
        'Vector([1.0, 2.0, 3.0, 4.0, 5.0, ...])'
        """
        components = reprlib.repr(self._components)
        # Удаляем префикс array('d' и закрываем скобку ),
        # перед тем как подставить строку в вызов конструктора.
        components = components[components.find('['):-1]
        return 'Vector({})'.format(components)

    def __str__(self):
        """
        >>> v1 = Vector((1, 2, 3, 4, 5))
        >>> str(v1)
        '(1.0, 2.0, 3.0, 4.0, 5.0)'
        """
        return str(tuple(self))

    def __bytes__(self):
        """
        Строим объект bytes из self._components

        >>> v1 = Vector([1,1])
        >>> bytes(v1)
        b'd\\x00\\x00\\x00\\x00\\x00\\x00\\xf0?\\x00\\x00\\x00\\x00\\x00\\x00\\xf0?'
        """
        return bytes([ord(self.typecode)]) + bytes(self._components)

    def __eq__(self, other):
        return tuple(self) == tuple(other)

    def __abs__(self):
        """
        Метод hypot больше не применим, поэтому
        вычисляем сумму квадратов компонент и извлекаем
        из неё квадратный корень.

        >>> v1 = Vector([3, 4])
        >>> abs(v1)
        5.0
        """
        return math.sqrt(sum(x * x for x in self))

    def __bool__(self):
        """
        >>> bool(Vector([0, 0]))
```

```
        False
        >>> bool(Vector([1, 1]))
        True
        """
        return bool(abs(self))

    @classmethod
    def from_bytes(cls, octets):
        typecode = chr(octets[0])
        memv = memoryview(octets[1:]).cast(typecode)
        return cls(memv)


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

`reprlib.repr` - эта функция порождает безопасное представление длинной или рекурсивной структуры путём ограничения длины выходной строки с заменой отброшенного окончания многоточием `...`. При написании метода `__repr__` я мог бы вывести упрощённое отображение `components` с помощью такого выражения: `reprlib.repr(list(self._components)`. Но это было бы расточительно, поскольку пришлось бы копировать каждый элемент `self._components` в `list` только для того, что бы использовать `list repr`. Вместо этого можно применить `reprlib.repr` непосредственно к `self._components`, а затем отбросить все символы, оказавшиеся вне квадратный скобок `[]`.

Поскольку метод `repr` используется для отладки, он никогда не должен возбуждать исключение. Если в `__repr__` происходит какая-то ошибка, Вы должны обработать её сами и сделать всё возможное, что бы показать пользователю нечто разумное, позволяющее идентифицировать объект.

# Протоколы и динамическая типизация

В ООП ***протоколом*** называется неформальный интерфейс, определённый только в документации, но не в коде. Например, протокол последовательности в Python подразумевает только наличие методов `__len__` и `__getitem__`. Любой класс `Spam`, в котором есть такие методы со стандартной сигнатурой и семантикой, можно использовать всюду, где ожидается последовательность. Является `Spam` подклассом какого-то другого класса или нет, роли не играет.

*Пример реализации последовательности*

```
import collections
Card = collections.namedtuple('Card', ['rank', 'suit'])
"""
collections.namedtuple - используется для коструирования простого класса,
представляющего одну карту.
Можно использовать для построения классов, содержащий только аттрибуты и никаких
методов. прим.: запись БД.
"""


class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'черви буби крести пики'.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits for rank in self.ranks]

    def __len__(self):
        return len(self._cards)

    def __getitem__(self, item):
        return self._cards[item]
```

Любому опытному программисту на Python достаточно одного взгляда, что бы понять что это именно класс последовательность. Несмотря на то, что он является подклассом `object`. Мы говорим, что он *является последовательностью*, потому, что он *ведёт себя* как последовательность. **А только это и важно.**

Такой подход получил название ***"Динамическая типизация"***. В оригинале используется термин ***"duck typing"***.

# Vector, попытка N2: последовательность допускающая срезку

*Реализация методов __len__ и __getitem__ для поддержки операций срезки*

```
from vector_nd import Vector


class Vector2(Vector):
    def __len__(self):
        """
        >>> v1 = Vector2([3, 4, 5])
        >>> len(v1)
        3
        """
        return len(self._components)

    def __getitem__(self, item):
        """
        В данной реализации есть проблема. Срез будет
        объектом класс list. Было бы лучше, если срез
        был бы объектом класса Vector.

        >>> v1 = Vector2([3, 4, 5])
        >>> v1[0], v1[-1]
        (3.0, 5.0)
        >>> v7 = Vector2(range(7))
        >>> v7[1:4]
        array('d', [1.0, 2.0, 3.0])
        """
        return self._components[item]


if __name__ == '__main__':
    import doctest
```

Если мы хотим, что бы срезы `Vector` тоже были объектом класс `Vector`, то не должны делегировать получение среза классу `array`.

## Как работает срезка

*Изучение поведения `__getitem__` и срезов*

```
class MySeq:
    def __getitem__(self, item):
        """
        :param item: - индекс в последовательности.
        :return: в данном случае __getitem__ просто
        возвращает то, что ему передали.

        >>> s = MySeq()
        >>> s[1] # Один индекс, ничего нового.
        1
        >>> s[1:4] # Нотация 1:4 преобразуется в:
        slice(1, 4, None)
        >>> s[1:4:2] # slice(1, 4, 2) означает: начать с 1, закончить на 4, шаг 2.
        slice(1, 4, 2)
        >>> s[1:4:2, 9] # Сюрприз: при наличии запятых внутри [] метод __getitem__
принимает кортеж.
        (slice(1, 4, 2), 9)
        >>> s[1:4:2, 7:9] # Этот кортеж даже может содержать несколько объектов среза.
        (slice(1, 4, 2), slice(7, 9, None))
        """
        return item


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

*Теперь приглядимся повнимательнее к `slice`:*

```
>>> slice
<class 'slice'>
>>> dir(slice)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__',
'__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'indices', 'start',
'step', 'stop']
```

`slice.indices` — возвращает "нормализованный" кортеж.

indices(...) S.indices(len) → (start, stop, stride)

```
Assuming a sequence of length len, calculate the start and stop
indices, and the stride length of the extended slice described by
S. Out of bounds indices are clipped in a manner consistent with the
handling of normal slices.
```

## Метод `__getitem__` с учётом срезов

*Не окончательная версия*

```
from vector_nd import Vector


class Vector2(Vector):
    def __len__(self):
        """
        >>> v1 = Vector2([3, 4, 5])
        >>> len(v1)
        3
        """
        return len(self._components)

    def __getitem__(self, item):
        """
        В данной реализации есть проблема. Срез будет
        объектом класс list. Было бы лучше, если срез
        был бы объектом класса Vector.

        >>> v1 = Vector2([3, 4, 5])
        >>> v1[0], v1[-1]
        (3.0, 5.0)
        >>> v7 = Vector2(range(7))
        >>> v7[1:4]
        array('d', [1.0, 2.0, 3.0])
        """
        return self._components[item]
```

*Реализация срезов*

```
import functools
import itertools
import numbers
import operator
import math
from vector_nd import Vector


class VectorND2(Vector):
    shortcut_names = 'xyzt'

    def __len__(self):
        return len(self._components)

    def __getitem__(self, item):
        """
        >>> v7 = VectorND2(range(8))
        >>> v7[-1]
        7.0
        >>> v7[-2]
        6.0
        >>> v7[1:4]
        Vector([1.0, 2.0, 3.0])
        >>> v7[-1:]
        Vector([7.0])
        """
        cls = type(self)  # Получаем класс экземпляра
        if isinstance(item, slice):  # Если индекс - это срез, то
            return cls(self._components[item])  # вызываем класс cls для построения
экземпляра Vector
        elif isinstance(item, numbers.Integral):  # абстрактный базовый класс, для
гибкости ;)
            return self._components[item]  # Если индекс число, то просто возвращаем
элемент
        else:
            msg = '{cls.__name__} indices must be integers'
            raise TypeError(msg.format(cls=cls))
```

# Vector, попытка N3: доступ к динамическим атрибутам.

Идея сводится к реализации метода `__getattr__` для создания атрибутов класса, которые будут обращаться к элементам вектора. (пр. `self.x → self[0]`, `self.y → self[1]`).

*Релизация методов для установки и получения динамических аттрибутов*

```
    def __getattr__(self, item):
        """
        >>> v1 = VectorND2(range(10))
        >>> v1.x, v1.y, v1.z, v1.t
        (0.0, 1.0, 2.0, 3.0)
        """
        cls = type(self)
        if len(item) == 1:
            pos = cls.shortcut_names.find(item)
            if 0 <= pos < len(self._components):
                return self._components[pos]
        msg = '{.__name__!r} object has no attribute {!r}'
        raise AttributeError(msg.format(cls, item))

    def __setattr__(self, key, value):
        """
        Для исключения ошибок связанной с попыткой установить новую переменную
        из зарезервированных имен x, y, z, t

        >>> v1 = VectorND2(range(10))
        >>> v1.x = 10
        Traceback (most recent call last):
            ...
            raise AttributeError(msg)
        AttributeError: readonly attribute 'x'

        """
        cls = type(self)
        if len(key) == 1:  # если аттрибут односимвольный
            if key in cls.shortcut_names:  # и входит в переменную с именами
                error = 'readonly attribute {attr_name!r}'  # ошибка установки
            elif key.islower():
                error = 'can\'t set attributes \'a\' to \'z\' in {cls_name!r}'
            else:
                error = ''
            if error:
                msg = error.format(cls_name=cls.__name__, attr_name=key)
                raise AttributeError(msg)
        super().__setattr__(key, value)
```

Часто с методом `__getattr__` приходится писать `__setattr__`, что бы избежать несогласованного поведения объекта. Если бы мы решили допустить изменение компонент, то могли бы реализовать метод `__setitem__`, что бы можно было писать `v[0] = 1.1`, и (или) метод `__setattr__`, что бы работала конструкция `v.x = 1.1`. Но сам класс **Vector** должен оставаться неизменяемый, потому что далее мы собираемся сделать его хэшируемым.

# Vector, попытка N4: хэширование и ускорение оператора ==

*Релизация методов для установки и получения динамических аттрибутов*

```
    def __eq__(self, other):
        """
        Реализация оператора сравнения сделана для экономии ресурсов
        при сравнении векторов с большим количеством измерений.

        Функция zip порождает генератор кортежей, содержащих соответственно
        элементы каждого переданного итерируемого объекта.

        Сравнение длины в предыдущем предложении необходимо, потому, что
        zip без предупреждения перестает порождать значения, как только
        хотя бы один входящий аргумент будет исчерпан.

        """
        if len(self) != len(other):
            return False
        for a, b in zip(self, other):
            if a != b:
                return False
        return True

    def __hash__(self):
        """
        Подаем выражение hashes на вход reduce вместе с функцией xor -
        для вычисления итогового хэш-значения; третий элемент равный 0 -
        инициализатор.
        """
        hashes = map(hash, self._components)
        return functools.reduce(operator.xor, hashes, 0)
```

*Удивительная функция* **zip**

Позволяет параллельно обходить 2 и более итерируемых объекта: она возвращает кортежи, которые можно распаковать в переменные, — по одной для каждого входного объекта

**zip** *за работой*

```
>>> zip(range(3), 'ABC') # zip возвращает генератор, который порождает кортежи по
запросу
<zip object at 0x0000024C12157EC0>
>>> list(zip(range(3), 'ABC'))
[(0, 'A'), (1, 'B'), (2, 'C')]
>>> list(zip(range(3), 'ABC', [0.0, 0.1, 0.2]))
[(0, 'A', 0.0), (1, 'B', 0.1), (2, 'C', 0.2)]
>>> from itertools import zip_longest
# itertools.zip_longest подставляет вместо отсутствующих значений необязательный
аргумент fillvalue. Поэтому генерирует кортежи пока не окажется исчерпанным самый
длинный итерируемый объект. Функция zip напротив бы остановилась без предупреждения
если бы закончился один из объектов.
>>> list(zip_longest(range(3), 'ABC', [0.0, 0.1, 0.2, 0.3], fillvalue=666))
[(0, 'A', 0.0), (1, 'B', 0.1), (2, 'C', 0.2), (666, 666, 0.3)]
```

# Vector, попытка N5: форматирование.

Можно взять метод форматирования из 2D вектора, но взамен подсчёта полярных координат, мы будем использовать *"гиперсферические"* координаты (название связанно с тем, что в пространствах размерностью больше 4 и выше сферы называются гиперсферами). Соответственно специальный суффикс форматной строки `p` мы заменним на `h`.

Для объекта Vector в 4D пространстве код `h` порождает представление вида `<r, Φ1, Φ2, Φ3>`, где:

- **r** — модуль вектора
- **Φ1, Φ2, Φ3** - угловые координаты

*Релизация метода формат для отображения в гиперсферических координатах*

```
    def angle(self, n):
        """
        Вспомогательная функция для вычисления угловой координаты.
        :param n: элементы вектора
        :return: значние угловой координаты

        >>> v1 = VectorND2(range(10))
        >>> v1.angle(2)
        1.5115267439240436
        >>> v1.angle(1)
        1.5707963267948966


        """
        r = math.sqrt(sum(x * x for x in self[n:]))
        a = math.atan2(r, self[n-1])
```

```
        if (n == len(self) - 1) and (self[-1] < 0):
            return math.pi * 2 - a
        else:
            return a

    def angles(self):
        """
        Генераторное выражение для вычисления
        всех угловых координат по запросу.
        """
        return (self.angle(n) for n in range(1, len(self)))

    def __format__(self, format_spec=''):
        """
        Переопределю метод для возможности вывода в гиперсферических координатах.
        :param format_spec: если оканчивается на 'h': гиперсферические координаты.
        :return: Подставляет строки во внешний формат.

        >>> v1 = VectorND2(range(4))
        >>> v1.x
        0.0
        >>> v1.y
        1.0
        >>> format(v1, '.3eh')
        '<3.742e+00, 1.571e+00, 1.300e+00, 9.828e-01>'
        >>> format(v1, '.3e')
        '(0.000e+00, 1.000e+00, 2.000e+00, 3.000e+00)'
        >>> format(v1, '.3f')
        '(0.000, 1.000, 2.000, 3.000)'
        >>> format(v1, '.3fh')
        '<3.742, 1.571, 1.300, 0.983>'
        """
        if format_spec.endswith('h'):
            format_spec = format_spec[:-1]
            # используем itertools.chain для порождения генераторного
            # выражения, которое перебирает модуль и угловые координаты
            coords = itertools.chain([abs(self)], self.angles())
            outer_fmt = '<{}>'
        else:
            coords = self
            outer_fmt = '({})'
        components = (format(c, format_spec) for c in coords)
        return outer_fmt.format(', '.join(components))
```

# Глава 11. Интерфейсы: от протоколов до абстрактных классов.

ABC, подобно дескрипторам и метаклассам, предназначены для разработки каркасов. Поэтому лишь малая часть пишущих на Python может создавать ABC, не налагая ненужных ограничений на своих коллег.

## Интерфейсы и протоколы в культуре Python

***Интерфейс*** — подмножество открытых методов объекта, которое позволяет ему играть определённую роль в системе.

***Протоколы*** — это интерфейсы, но поскольку они не формализованы — определены лишь путём документирования и соглашения — то не могут быть строго поддержаны как формальные интерфейсы

Одним из самых фундаментальных интерфейсов в Python — протокол последовательности.

## Python в поисках следов последовательностей.

Интерпретатор Python прилагает все усилия, стараясь обработать объекты, представляющие самую минимальную реализацию протокола последовательностей.

Sequence — Последовательность

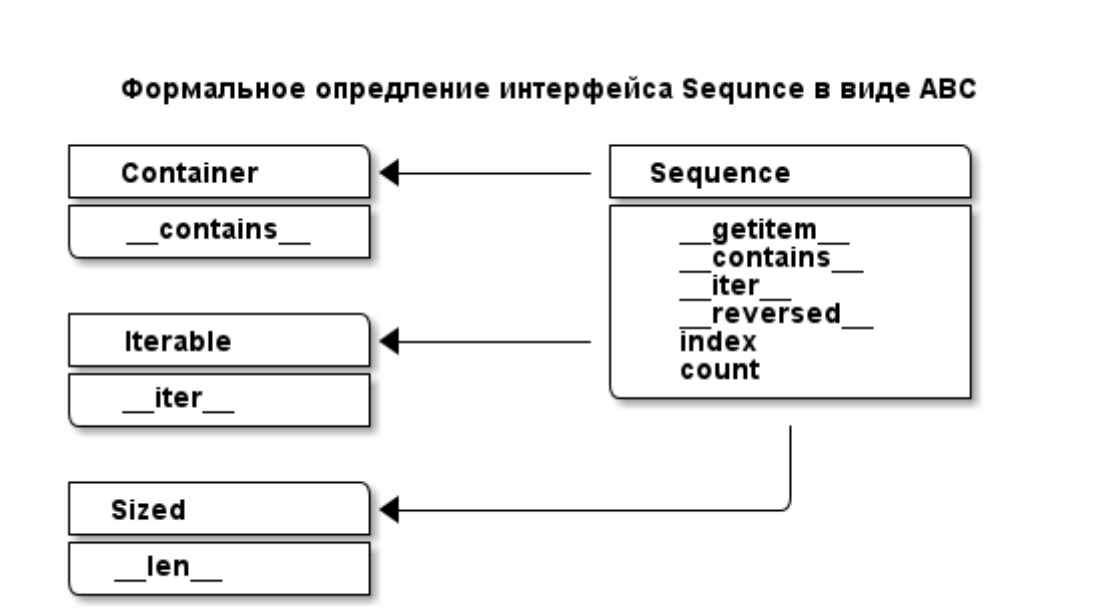



*Figure 3. UML-диаграмма абстрактного базового класса* **Sequence** *(последовательность)*

*Частичная реализация протокола последовательности: метода `__getitem__` достаточно для доступа к элементам, итерирования и реализации оператора `in`*

```
>>> class Foo:
...     def __getitem__(self, pos):
...         return range(0, 30, 10)[pos]
...
>>> f = Foo()
>>> f[1]
10
>>> for i in f: print(i)
0
10
20
>>> 20 in f
True
>>> 15 in f
False
```

Метода `__iter__` в классе **Foo** нет, однако его экземпляры являются итерируемыми объектами, потому что даже в случае отсутствия `__iter__` Python, обнаружив метод `__getitem__`, пытается обойти объект, вызывая этот метод, с целочисленными индексами, начиная с 0. Поскольку Python достаточно умён, что бы обойти объект Foo, он может также реализовать оператор `in`, даже если в классе нет метода `__contains__`: для этого достаточно просто обойти объект в поисках элемента.

# Monkey patching или партизанское латание.

```
>>> from deck import FrenchDeck         # Пример партизанского латания
>>> def set_card(deck, position, card): # метод это всего лишь функция
...     deck._cards[position] = card    # Данная функция знает об deck._cards
...
>>> FrenchDeck.__setitem__ = set_card   # Латаем функцию к специальному методу
>>> deck = FrenchDeck()
>>> from random import shuffle
>>> shuffle(deck)                    # Теперь наша колода поддерживает перетасовку
>>> deck[:5]
[Card(rank='J', suit='кресты'), Card(rank='6', suit='кресты'), Card(rank='8',
suit='черви'), Card(rank='2', suit='пики'), Card(rank='J', suit='буби')]
```

Как пример динамической типизации можно подчеркнуть протокол `random.shuffle`. Ему всё равно какой аргумент ему передан, лишь бы он мог реализовать метод изменения последовательности.

Вызов `isinstance(obj, cls)` приемлем, при условии, что `cls` — абстрактный базовый класс, т.е. метаклассом `cls` является `abc.ABCMeta`

# Создание подкласса ABC

Важно воспользоваться уже существующим абстрактным классом. Потому что всё уже написано до нас.

Поэтому что бы реализовать колоду карт возьмём уже существующий `collections.MutableSequence`

```
Help on class MutableSequence in module collections.abc:
class MutableSequence(Sequence)
 |  All the operations on a read-write sequence.
 |
 |  Concrete subclasses must provide __new__ or __init__,
 |  __getitem__, __setitem__, __delitem__, __len__, and insert().
 |
 |  Порядок разрешения метода:
 |      MutableSequence
 |      Sequence
 |      Reversible
 |      Collection
 |      Sized
 |      Iterable
 |      Container
 |      builtins.object
 |
 |  Определенные здесь методы:
 |
 |  __delitem__(self, index)
 |
 |  __iadd__(self, values)
 |
 |  __setitem__(self, index, value)
 |
 |  append(self, value)
 |      S.append(value) -- append value to the end of the sequence
 |
 |  clear(self)
 |      S.clear() -> None -- remove all items from S
 |
 |  extend(self, values)
 |      S.extend(iterable) -- расширить последовательность, добавив элементы из
повторяющегося
 |
 |  insert(self, index, value)
 |      S.insert(index, value) -- insert value before index
 |
 |  pop(self, index=-1)
 |      S.pop([index]) -> item -- remove and return item at index (default last).
 |      Raise IndexError if list is empty or index is out of range.
 |
```

```
|  remove(self, value)
|      S.remove(value) -- remove first occurrence of value.
|      Raise ValueError if the value is not present.
|
|  reverse(self)
|      S.reverse() -- reverse *IN PLACE*
|
|  ----------------------------------------------------------------------
|  Данные и другие атрибуты, определенные здесь:
|
|  __abstractmethods__ = frozenset({'__delitem__', '__getitem__', '__len_...
|
|  ----------------------------------------------------------------------
|  Методы, унаследованные от Sequence:
|
|  __contains__(self, value)
|
|  __getitem__(self, index)
|
|  __iter__(self)
|
|  __reversed__(self)
|
|  count(self, value)
|      S.count(value) -> integer -- return number of occurrences of value
|
|  index(self, value, start=0, stop=None)
|      S.index(value, [start, [stop]]) -> integer -- return first index of value.
|      Raises ValueError if the value is not present.
|
|      Supporting start and stop arguments is optional, but
|      recommended.
|
|  ----------------------------------------------------------------------
|  Методы класса, унаследованные от Reversible:
|
|  __subclasshook__(C) from abc.ABCMeta
|      Abstract classes can override this to customize issubclass().
|
|      This is invoked early on by abc.ABCMeta.__subclasscheck__().
|      It should return True, False or NotImplemented.  If it returns
|      NotImplemented, the normal algorithm is used.  Otherwise, it
|      overrides the normal algorithm (and the outcome is cached).
|
|  ----------------------------------------------------------------------
|  Методы, унаследованные от Sized:
|
|  __len__(self)
|
|  ----------------------------------------------------------------------
|  Методы класса, унаследованные от Iterable:
```

```
 |
 |  __class_getitem__ = GenericAlias(...) from abc.ABCMeta
 |      Represent a PEP 585 generic type
 |
 |      E.g. for t = list[int], t.__origin__ is list and t.__args__ is (int,).
```

**FrenchDeck2** *подкласс* `abc.MutableSequence`

```
from abc import ABC
from collections import abc, namedtuple

Card = namedtuple('Card', ['rank', 'suit'])
"""
Card -- мини-класс карты

namedtuple(typename, field_names, *, rename=False, defaults=None, module=None)
    Возвращает новый подкласс кортежа с именованными полями.

    >>> Point = namedtuple('Point', ['x', 'y'])
    >>> Point.__doc__                   # docstring for the new class
    'Point(x, y)'
    >>> p = Point(11, y=22)             # instantiate with positional args or keywords
    >>> p[0] + p[1]                     # indexable like a plain tuple
"""


class FrenchDeck2(abc.MutableSequence, ABC):
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamond clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits for rank in self.ranks]

    def __len__(self):
        return len(self._cards)

    def __getitem__(self, item):
        return self._cards[item]

    def __setitem__(self, key, value):
        """
        Данный метод, всё что нужно для поддержки
        тасования (random.shuffle)
        :param key: - позиция карты
        :param value: - карта
        >>> deck = FrenchDeck2()
        >>> from random import shuffle
        >>> shuffle(deck)
        """
        self._cards[key] = value
```

```
    def __delitem__(self, key):
        """
        Что бы создать подкласс abc.MutableSequence
        необходимо так же реализовать метод удаления
        карты из колоды.
        :param key: - позиция карты
        """
        del self._cards[key]

    def insert(self, index: int, value: Card) -> None:
        """
        Так же необходимо реализовать третий абстрактный
        метод abc.MutableSequence
        """
        self._cards.insert(index, value)
```

На этапе импорта Python не проверяет, реализованы ли абстрактные методы. Это происходит только на этапе выполнения. И тогда если абстрактный метод не реализован, мы получим исключение `TypeError`. Поэтому мы обязаны реализовать `__delitem__` и `insert`, хотя они нам не нужны в примерах.

# ABC в стандартной библиотеке

Большая часть ABC находятся в:

- `collections.abc`
- `numbers`
- `io`

### Группы классов в модуле `collections.abc`:

- `Iterable`, `Container`, `Sized` — Любая коллекция должна либо наследовать какому-то из этих ABC классов, либо реализовывать совместимые протоколы.
  - `Iterable` — поддерживает итерирование методом `__iter__`
  - `Container` — поддерживает оператор `in` методом `__contains__`
  - `Sized` — поддерживает функцию `len()` методом `__len__`
- `Sequence`, `Mapping`, `Set` — Это основные типы неизменяемых коллекций, и у каждого есть изменяемый подкласс.
- `MappingView` — В Python3 объекты, возвращённые отображение метода `.items()`, `.keys()` и `.value()`, наследуют классы `ItemsView`, `KeysView` и `ValuesView()` соответственно. Первые 2 также наследуют богатый интерфейс класса `Set` со всеми операторами для работы над множествами.
- `Callable`, `Hashable` — основное назначение - поддержка функций вызова и хеширования с помощью `isinstance`

- `Iterator` — является подклассом `Iterable`

### Числовая башня `numbers`

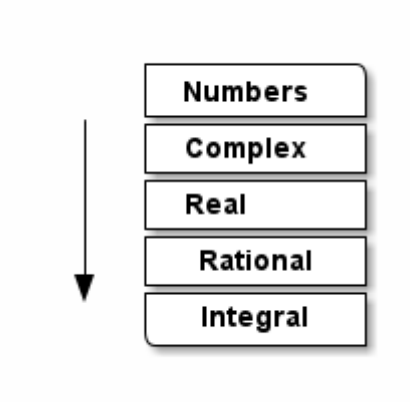



*Figure 4. В пакете `numbers` определена иерархия ABC, называемая "числовой башней"*

Таким образом, если нужно проверить, является ли объект целым числом, вызывайте `isinstance(x, numbers.Integral)`. Этот метод вернёт `True` для типов `int`, `bool` и прочих целочисленных типов предоставленными внешними библиотеками, которые зарегистрируют свои типы как подклассы `abc` модуля `numbers`.

## Определение и использование ABC

Пусть требуется отображать на сайте рекламные объявления в случайном порядке, но при этом не повторять никакое объявление, пока будут показаны все имеющиеся из набора. Допустим мы разрабатываем рекламную систему под названием `ADAM`. Одно из требований — поддержать предоставляемые пользователем классы случайного выбора без повторений. Что бы у пользователя не было сомнений, что понимается под "случайным выбором", мы определим ABC.

Наш ABC будет называться **Tombola** — это итальянское название игры в Bingo.

В ABC `Tombola` определены 2 абстрактных метода:

- `.load(…)`: поместить элементы в контейнер
- `.pick()`: извлечь случайный элемент из коллекции и вернуть его.

И еще 2 конкретных метода:

- `.loaded()`: вернуть `True`, если в коллекции имеется хотя бы 1 элемент
- `.inspect()`: вернуть отсортированный кортеж `tuple`, составленный из элементов, находящихся в контейнере, не изменяя его содержимое. Внутреннее упорядочивание не сохраняется.

*Tombola - ABC с двумя абстрактными и двумя конкретными методами.*

```
import abc
```

```
class Tombola(abc.ABC):
    """
    Что бы определить ABC, создаем подкласс `abc.ABC`

    doctest показывает не возможность создания экземпляра
    подкласса `Tombola` не реализующего абстрактные методы.

    >>> class Fake(Tombola):
    ...     def pick(self):
    ...         return 13
    ...
    >>> Fake
    <class '__main__.Fake'>
    >>> f = Fake()
    Traceback (most recent call last):
    ...
        f = Fake()
    TypeError: Can't instantiate abstract class Fake with abstract methods load, pick
    """
    @abc.abstractmethod
    def load(self, iterable):
        """
        Абстрактный метод помечен декоратором @abc.abstractmethod
        и зачастую содержит в теле только строку документации.

        :param iterable: добавляемый элемент в коллекцию.
        :return:
        """

    @abc.abstractmethod
    def pick(self):
        """
        Строка документации сообщает программисту, реализующему
        метод, что в случае отсутствия элементов, необходим
        возбудить `LookupError`.

        :return: случайный элементы из коллекции.
        Этот метод должен возбуждать исключение
        `LookupError`, если контейнер пуст.
        """

    def loaded(self):
        """
        ABC может содержать конкретные методы.

        :return: `True` если хотя бы 1 элемент, иначе `False`.
        """
        return bool(self.inspect())     # Конкретные методы ABC,
        # должны зависеть только от открытого интерфейса данного.
```

```
    def inspect(self):
        """
        :return: отсортированный кортеж, содержащий находящиеся
        в контейнере элементы
        """
        items = []
        while True:
            """
            Мы не знаем, как в конкретных подклассах будут храниться
            элементы, но можем построить `inspect` опустошив объект
            `Tombola` с помощью последовательного обращения методом
            `.pick()`...
            """
            try:
                items.append(self.pick())
            except LookupError:
                break
        self.load(items)
        """
        ...а затем, с помощью `.load(...)` вернуть все элементы
        обратно.
        """
        return tuple(sorted(items))


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

У абстрактного метода может существовать реализация. Но даже если так, подклассы все равно обязаны переопределить его, однако имеют право вызывать абстрактный метод через `super()`, расширяющего имеющуюся функциональность, вместо того, что бы реализовывать еге с нуля. Подробнее об использовании @abstractmethod кури мануал abc — Abstract Base Classes

## Синтаксические детали ABC

Лучший способ объявить ABC — сделать его подклассом `abc.ABC` или какого-нибудь другого ABC. Однако `abc.ABC` появился только Python >= 3.4 и если используешь более раннюю версию то придется писать так: `class Tombola(metaclass=abc.ABCMeta)`.

*Именованный аргумент* `metaclass=` *был введён в Python 3. В Python 2 нужно было использовать атрибут класса* `__metaclass__`*:*

```
class Tombols(object):  # Это для Python 2
    __metaclass__ = abc.ABCMeta
    # ...
```

В модуле `abc` так же есть и другие декораторы обозначающие методы класса и статические, но они избыточные, т.к. можно использовать несколько декораторов.

*Пример как объявлять абстрактный метод класса.*

```
class MyABC(abc.ABC):

    @classmethod
    @abc.abstractmethod
    def an_abstract_classmethod(cls, ...)
        pass
```

Порядок декораторов важен. Между `@abc.abstractmethod` и `def` не должно быть никаких других декораторов.

# Создание подклассов ABC Tombola

```
import random

from tombola import Tombola


class BingoCage(Tombola):

    def __init__(self, items):
        self._items = []
        self._randomizer = random.SystemRandom()
        self.load(items)

    def load(self, items):
        self._items.extend(items)
        self._randomizer.shuffle(self._items)

    def pick(self):
        try:
            return self._items.pop()
        except IndexError:
            raise LookupError('pick from empty BingoCage')

    def __call__(self, *args, **kwargs):
        self.pick()
```

```
import random

from tombola import Tombola


class LotteryBlower(Tombola):

    def __init__(self, iterable):
        self._balls = list(iterable)

    def load(self, iterable):
        self._balls.extend(iterable)

    def pick(self):
        try:
            position = random.randrange(len(self._balls))
        except ValueError:
            raise LookupError('pick from empty BingoCage')
        return self._balls.pop(position)

    def loaded(self):
        return bool(self._balls)

    def inspect(self):
        return tuple(sorted(self._balls))
```

```
from random import randrange
from tombola import Tombola


@Tombola.register           # Виртуальная регистрация как подкласс Tombola
class TomboList(list):
    """
    Пример реализации виртуального подкласса.
    Важно для корректности реализовать все методы из ABC.

    >>> from tombola import Tombola
    >>> t = TomboList(range(100))
    >>> isinstance(t, Tombola)
    True

    """
    def pick(self):
        if self:
            position = randrange(len(self))
            return self.pop(position)
        else:
            raise LookupError('pop from empty TomboList')

    load = list.extend          # Хитрое выражение >;)

    def loaded(self):
        return bool(self)

    def inspect(self):
        return tuple(sorted(self))


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

## Использование метода `register` на практике

Чаще всего `register` можно использовать как функцию для регистрации классов, определённых где-то в другом месте.

## Выводы

- **Интерфейсы** в духе протоколов не имеют ничего общего с наследованием; каждый класс реализует протокол независимо от остальных. Так выглядят интерфейсы в языках с динамической типизацией.
- Благодаря **гусиной типизации** абстрактные базовые классы (ABC) используются, чтобы

сделать интерфейсы явными, а классы могут реализовывать интерфейсы либо с помощью наследования ABC, либо регистрации, для которой не требуется сильная статическая связь, характерная для наследования.

*Гусиная типизация*

> What goose typing means is: `isinstance(obj, cls)` is now just fine… as long as cls is an Abstract Base Class.
>
> — Alex Martelli, https://en.wikipedia.org/wiki/Duck_typing

- Определение **абстрактного базового класса** позволяет зафиксировать общий API для множества подклассов. Эта возможность особенно полезна, когда человек, слабо знакомый с исходным кодом приложения, собирается написать для него подключаемый модуль. Why use Abstract Base Classes?
- `__subclasshook__` — позволяет ABC распознавать незарегистрированные классы в качестве своего подкласса при условии, что он проходит некоторую проверку, которая может быть простой или сложной, как нужно разработчику,- классы из стандартной библиотеки всего лишь проверяют имена методов.

> Однажды поняв, что такое последовательность, мы можем применять это знание в разных контекстах. Это и есть главная тема моей книги: выявление фундаментальных идиом языка, что позволяет сделать код кратким, эффективным и удобочитаемым - для мастера-питониста.
>
> — Luciano Ramalho

# Экстренное включение с асинхронных игр.

## По мотивам статьи "Как писать асинхронный код Python"

How to Write Asynchronous Python Code

*Простейший пример запуска в асинхронном режиме*

```
import asyncio


async def sample_coroutine():   # Образец программы
    print('5 start')
    await asyncio.sleep(5)
    print('5 end')

if __name__ == '__main__':
    loop = asyncio.new_event_loop()
    loop.run_until_complete(sample_coroutine())
```

```
import asyncio


async def gather_fxn1():
    print('6 start')
    await asyncio.sleep(6)
    print('6 end')


async def gather_fxn2():
    print('4 start')
    await asyncio.sleep(4)
    print('4 end')


async def sample_coroutine():
    await asyncio.gather(gather_fxn1(), gather_fxn2())

if __name__ == '__main__':
    loop = asyncio.new_event_loop()
    loop.run_until_complete(sample_coroutine())
```

# Асинхронные представления в Django >= 3.1

Async Views in Django 3.1

*Пример асинхронных представлений*

```
import asyncio
import random
from time import sleep
from typing import List

import httpx
from asgiref.sync import sync_to_async
from django.http import HttpResponse


# helpers

async def http_call_async():
    """
    Асинхронная функция которая ожидает 6 секунд,
    и создает асинхронный get запрос к URL.
    """

    for num in range(1, 6):
        await asyncio.sleep(1)
        print(num)

    async with httpx.AsyncClient() as client:
        r = await client.get("https://httpbin.org/")
        print(r)


def http_call_sync():
    """
    Функция выполняющая свою нагрузку в синхронном
    режиме
    """

    for num in range(1, 6):
        sleep(1)
        print(num)

    r = httpx.get("https://httpbin.org/")
    print(r)


# views
async def index(request):
    """
    Заглушка
```

```
    """
    return HttpResponse("Hello, async Django!")


async def async_view(request):
    """
    Пример асинхронного представления.
    Не блокирует страницу пока выполняется задача
    """
    loop = asyncio.get_event_loop()
    loop.create_task(http_call_async())
    return HttpResponse("Non-blocking HTTP request")


def sync_view(request):
    """
    Пример синхронного прдеставления.
    Пока http_call_sync() работает, браузер будет
    блокироваться.
    """
    http_call_sync()
    return HttpResponse("Blocking HTTP request")


async def smoke(smokables: List[str] = None, flavor: str = "Sweet Baby Ray's"):
    """ Smoke some meats and applies the Sweet Baby Ray's """

    for smokable in smokables:
        print(f"Smoking some {smokable}...")
        print(f"Applying the {flavor}...")
        print(f"{smokable.capitalize()} smoked.")

    return len(smokables)


async def get_smokables():
    print("Getting smokables...")

    await asyncio.sleep(2)
    async with httpx.AsyncClient() as client:
        await client.get("https://httpbin.org/")

        print('Returning smokable')
        return [
            "ribs",
            "brisket",
            "lemon chicken",
            "salmon",
            "bison sirloin",
            "sausage",
        ]
```

```
async def get_flavor():
    print("Getting flavor...")

    await asyncio.sleep(1)
    async with httpx.AsyncClient() as client:
        await client.get("https://httpbin.org/")

        print("Returning flavor")
        return random.choice(
            [
                "Sweet Baby Ray's",
                "Stubb's Original",
                "Famous Dave's"
            ]
        )


async def smoke_some_meats(request):
    """
    Пример асинхронного представления
    """
    results = await asyncio.gather(*[get_smokables(), get_flavor()])
    total = await asyncio.gather(*[smoke(results[0], results[1])])
    return HttpResponse(f"Smoked {total[0]} meats with {results[1]}!")


def oversmoke() -> None:
    """ If it's not dry, it must be uncooked """
    sleep(5)
    print("Who doesn't love burnt meats?")


async def burn_some_meats(request):
    """
    Пример асинхронного представления выполняющего соинхронную работу
    """
    oversmoke()
    return HttpResponse(f"Burned some meats.")


async def async_with_sync_view(request):
    """
    Применение sync_to_async для выполнения синхронной функции в асинхронном режиме
    """
    loop = asyncio.get_event_loop()
    async_function = sync_to_async(http_call_sync)
    loop.create_task(async_function())
    return HttpResponse("Non-blocking HTTP request (via sync_to_async)")
```

# Глава 12. Наследование: хорошо или плохо.

*The Early History Of Smalltalk*

> Мы начали продвигать наследование, что бы начинающие программисты могли пользоваться каркасами, спроектировать которые под силу только опытным специалистам.
>
> — Alan C. Kay

Мы начнём с вопроса о наследовании встроенным типам.

## Сложность наследования встроенным типам

> Прямое наследование таким встроенным типам, как `dict`, `list`, `str`, чревато ошибками, потом что встроенные методы, как правило, игнорируют написанные пользователем переопределенный метод. Вместо создания подклассов встроенных объектов наследуйте свои классы от классов в модуле `collections` — Container datatypes — `UserDict`, `UserList` и `UserString`, который специально предназначены для беспроблемного наследования.

## Жизнь с множественным наследованием

1. Отличайте наследование интерфейса от наследования реализации.
    - Наследование *интерфейса* создает подтип, подразумевая связь **"является"**;
    - Наследование *реализации* позволяет избежать **дублирования кода**;
2. Определяйте интерфейсы явно при помощи ABC
    - При помощи явного объявления наследования `abc.ABC`
3. Используйте примеси для повторного использования кода.
    - Примесь не определяет нового типа, просто создает контейнер для общеполезных методов.
4. Явно определяйте примеси с помощью наименования.
    - При помощи суффикса Mixin
5. ABC тоже может быть примесью, обратное неверно
6. Не наследуйте сразу нескольким конкретным классам
    - class MyClass(Alpha, Beta, Gamma):
    - Alpha — конкретный класс
    - Beta, Gamma — примеси или ABC
7. Предоставляйте пользователям агрегаторные класс.

- Это классы, которые строятся путём наследования примесями и не добавляет никакой структуры или поведения.

8. Предпочитайте композицию наследованию класса.

===

# Глава 13. Перезагрузка операторов как правильно?

## Унарные операторы

- `-`: `(__neg__)` — Унарный арифметический минус.
- `+`: `(__pos__)` — Унарный арифметический плюс.
- `~`: `(__invert__)` — Поразраядная инверсия целого числа, определяется как ~x == -(x+1)

Поддерживать унарные операторы легко, достаточно переопределить один из специальных методов, который принимает только один аргумент `self` и возвращающий **новый** объект.

Специальные методы, реализующие унарные операторы или инфиксные операторы не должны изменять свои операнды.

*vector_v7.py: добавлены методы оператора `*`*

```
import numbers
from vector_nd_2 import Vector2


class Vector(Vector2):
    def __mul__(self, other):
        """
        Методы для реализации умножения
        >>> v1 = Vector([1.0, 2.0, 3.0])
        >>> 14 * v1
        Vector([14.0, 28.0, 42.0])
        >>> v1 * True
        Vector([1.0, 2.0, 3.0])
        >>> from fractions import Fraction
        >>> v1 * Fraction(1, 3)
        Vector([0.3333333333333333, 0.6666666666666666, 1.0])
        """
        if isinstance(other, numbers.Real):
            return Vector(n * other for n in self)
        else:
            return NotImplemented

    def __rmul__(self, other):
        return self * other

    def __matmul__(self, other):
        """
        Метод для реализации матричного умножения
        Для Python >= 3.5
```

```
        >>> a = Vector([1,2,3])
        >>> b = Vector([4,5,6])
        >>> a @ b
        32.0
        >>> [10, 20, 30] @ b
        320.0
        >>> a @ 1
        Traceback (most recent call last):
        ...
            File "<input>", line 1, in <module>
        TypeError: unsupported operand type(s) for @: 'Vector' and 'int'
        """
        try:
            return sum(a * b for a, b in zip(self, other))
        except TypeError:
            return NotImplemented

    def __rmatmul__(self, other):
        return self @ other


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

*Table 2. Имена методов инфиксных операторов*

| Оператор | Прямой | Обратный | На месте | Описание |
|---|---|---|---|---|
| + | __add__ | __radd__ | __iadd__ | Сложение или конкатенация |
| - | __sub__ | __rsub__ | __isub__ | Вычитание |
| * | __mul__ | __rmul__ | __imul__ | Умножение, повторение |
| / | __truediv__ | __rtruediv__ | __itruediv__ | Истинное деление |
| // | __floordiv__ | __rfloordiv__ | __ifloordiv__ | Деление с округлением |
| % | __mod__ | __rmod__ | __imod__ | Деление по модулю |
| divmod() | __divmod__ | __rdivmod__ | __idivmod__ | Возвращает кортеж, содержащий частное и остаток |
| **, pow() | __pow__ | __rpow__ | __ipow__ | Возведение в степень |
| @ | __matmul__ | __rmatmul__ | __imatmul__ | Матричное умножение (Py>=3.5) PEP 465 — A dedicated infix operator for matrix multiplication |
| & | __and__ | __rand__ | __iand__ | Поразрядное **И** |
| \| | __or__ | __ror__ | __ior__ | Поразрядное **ИЛИ** |

| Оператор | Прямой | Обратный | На месте | Описание |
|---|---|---|---|---|
| ^ | __xor__ | __rxor__ | __ixor__ | Поразрядное **ИСКЛЮЧАЮЩЕЕ ИЛИ** |
| << | __lshift__ | __rlshift__ | __ilshift__ | Поразрядный сдвиг влево |
| >> | __rshift__ | __rrshift__ | __irshift__ | Поразрядный сдвиг вправо |

# Операторы сравнения

Обработка операторов сравнения ==, !=, >, <, >=, <= интерпретатором Python похожа на то, что было выше, но имеет два важных отличия:

- Для прямых и инверсных вызовов служит один и тот же набор методов.
- В случае сравнения ==, !=, если инверсный метод вызывает ошибку, то Python сравнивает идентификаторы объектов, а не возбуждает исключение TypeError

*Table 3. Операторы сравнения: инверсные методы вызываются, когда первый вызов вернул* NotImplemented

| Группа | Инфиксный оператор | Прямой вызов метода | Инверсный вызов метода | Запасной вариант |
|---|---|---|---|---|
| Равенство | a == b | a.__eq__(b) | b.__eq__(a) | return id(a) == id(b) |
| | a != b | a.__ne__(b) | b.__ne__(a) | return not (a == b) |
| Порядок | a > b | a.__gt__(b) | a.__lt__(b) | raise TypeError |
| | a < b | a.__lt__(b) | a.__gt__(b) | raise TypeError |
| | a >= b | a.__ge__(b) | a.__le__(b) | raise TypeError |
| | a <= b | a.__le__(b) | a.__ge__(b) | raise TypeError |

# Операторы составного присваивания

Если в классе не реализован метод "на месте", то операторы составного сравнения не более чем синтаксическая глазурь: a += b вычисляется так же как a = a + b. Это ожидаемое поведение для неизменяемых типов и, если реализовать метод __add__ то += будет работать без всякого дополнительного кода.

Однако, если всё-таки реализовать метод оператора "на месте", например __iadd__, то он будет вызван для вычисления выражения a += b. Как следует из названия, такие операторы изменяют сам левый операнд, а не создают новый объект—результат.

Специальны методы, вычисляемые на месте, никогда не следует реализовывать для неизменяемых типов и, в частности, нашего класса Vector. Это в общем-то, очевидно, но лишний раз подчеркнуть не помешает.

# Глава 14. Итерируемые объекты, итераторы и генераторы.

Итерирование — одна из важнейших операций обработки данных. А если просматривается набор данных, не помещающийся целиком в память, то нужен способ выполнять её *отложенно*, т.е. по одному элементу и по запросу. Именно в этом смысл паттерна **Итератор**.

Ключевое слово **yield** (*появилось только в Py >= 2.2*) позволяет конструировать генераторы, которые работают как итераторы.

Любой генератор является итератором: генераторы используют весь интерфейс итераторов. Но итератор — в том виде, как он определён в книге "Банды четырёх", — извлекает элементы из коллекции, тогда как генератор может порождать элементы "из воздуха". Типичным примером является генератор чисел Фибоначчи — бесконечно последовательности, которую нельзя сохранить в коллекции. Однако, надо иметь в виду, что в сообществе Python слова *итератор* и *генератор* обычно употребляется как синонимы.

## Класс Sentence, попытка  1: Последовательность слов.

В примере представлен класс `Sentence`, который умеет извлекать из текста слово с заданным индексом.

`sentence.py`: *объект* `Sentence` *как последовательность слов*

```
import re
import reprlib

RE_WORD = re.compile(r'\w+')


class Sentence:

    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text)
        # Возвращает список всех не пересекающихся
        # подстрок, соответствующих регулярному выражению RE_WORD.

    def __getitem__(self, item):
        """
        self.words содержит результат .fandall, поэтому мы просто
        возвращаем слово с заданным индексом
        """
        return self.words[item]
```

```
    def __len__(self):
        """
        Что бы выполнить требования протокола последовательности,
        реализуем данный метод,- но для получения итерируемого
        объекта он не нужен.
        """
        return len(self.words)

    def __repr__(self):
        """
        По умолчанию reprlib.repr ограничивает сгенерированную
        строку 30 символами.
        >>> s = Sentence('Время жизни на репит, просто что бы закрепить.')
        >>> s
        Sentence('Время жизни ...бы закрепить.')
        >>> for word in s:
        ...     print(word)
        ...
        Время
        жизни
        на
        репит
        просто
        что
        бы
        закрепить
        >>> list(s)
        ['Время', 'жизни', 'на', 'репит', 'просто', 'что', 'бы', 'закрепить']
        """
        return 'Sentence(%s)' % reprlib.repr(self.text)


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Всякий раз когда интерпретатору нужно обойти объект `x`, он автоматически вызывает функцию `iter(x)`.

Встроенная функция `iter(x)` выполняет следующие действия.

1. Смотрим, реализует ли объект метод `__iter__`, и если да, вызывает его, что бы получить итератор.
2. Если метод `__iter__` не реализован, но реализован метод `__getitem__`, то Python создает итератор, который пытается извлекать элементы по порядку, начиная с индекса 0.
3. Если и это не получается, то возбуждает исключение — обычно с сообщением `C object is not iterable`, где *C* — класс объекта.

sentence_iter.py: *класс* Sentence*, реализован с помощью паттерна Итератор*

```
import re
import reprlib

RE_WORD = re.compile(r'\w+')


class Sentence():
    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text)

    def __repr__(self):
        return f'Sentence({reprlib.repr(self.text)})'

    def __iter__(self):
        return SentenceIterator(self.words)


class SentenceIterator():

    def __init__(self, words):
        self.words = words
        self.index = 0

    def __next__(self):
        try:
            word = self.words[self.index]
        except IndexError:
            raise StopIteration()

        self.index += 1
        return word

    def __iter__(self):
        return self
```

sentence_gen.py: *класс* Sentence*, реализован с помощью паттерна Генератора*

```
import re
import reprlib

RE_WORD = re.compile(r'\w+')


class Sentence():
    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text)

    def __repr__(self):
        return f'Sentence({reprlib.repr(self.text)})'

    def __iter__(self):
        for word in self.words:
            yield word
        return
```

# Как работает генераторная функция

Любая функция Python, в теле которой встречается слово yield, называется генераторной функцией — при вызове она возвращает объект-генератор.

sentence_gen2.py: *класс* Sentence*, реализован с помощью паттерна Генератора*

```
import re
import reprlib

RE_WORD = re.compile(r'\w+')


class Sentence:
    def __init__(self, text):
        """
        Ленивая реализация класса Sentence.
        Хранить список слов не нужно.
        """
        self.text = text

    def __repr__(self):
        return f'Sentence({reprlib.repr(self.text)})'

    def __iter__(self):
        """
        finditer строит итератор, который обходит все
        соответствия текста self.text регулярному выражению
        RE_WORD, порождая объекты MatchObject.

        match.group() извлекает сопоставленный текст из
        объекта MatchObject
        """
        for match in RE_WORD.finditer(self.text):
            yield match.group()
```

Генераторные функции — замечательный способ сократить код, но генераторные выражения ещё круче.

Простые генераторные функции наподобие той, что мы использовали в предыдущем варианте класса Sentence, можно заменить *генераторным выражением*.

Можно считать что генераторное выражение — ленивая версия спискового включения: она не строит список энергично, а возвращает генератор, который лениво порождает элементы по запросу.

*Генераторная функция* `gen_AB` *используется сначала в списковом включении, а затем в генераторном выражении.*

```
>>> def gen_AB():
...     print('start')
...     yield 'A'
...     print('continue')
...     yield 'B'
...     print('end.')
...
>>> res1 = [x*3 for x in gen_AB()] # Списковое включение энергично обходит элементы,
порождаемые объектом-генератором, который был создан функцией gen_AB.
start
continue
end.
>>> for i in res1:
...     print('-->', i)
...
--> AAA
--> BBB
>>> res2 = (x*3 for x in gen_AB())
>>> res2
<generator object <genexpr> at 0x0000020BAC482C00>
>>> for i in res2:
...     print('-->', i)
...
start
--> AAA
continue
--> BBB
end.
```

Таким образом, генераторное выражение порождает генератор, и мы можем этим воспользоваться что бы ещё сократить размер класса Sentence.

`sentence_genexp.py`: *реализация класса* `Sentence` *с помощью генераторного выражения.*

```
import re
import reprlib

RE_WORD = re.compile(r'\w+')


class Sentence:
    def __init__(self, text):
        self.text = text

    def __repr__(self):
        return f'Sentence({reprlib.repr(self.text)})'

    def __iter__(self):
        return (match.group() for match in RE_WORD.finditer(self.text))
```

Генераторные выражения - это не более чем синтаксическая глазурь: их всегда можно заменить генераторными функциями.

# Построение арифметической прогрессии

```
def aritprog_gen(begin, step, end=None):
    """
    Реализация арифметической прогрессии
    с использованием генераторной функции.
    :param begin: начальное значение.
    :param step: шаг прогрессии.
    :param end: конечное значение
    (не обязательный аргумент)

    > Тест на проверку класса возвращаемеого значения
    >>> from fractions import Fraction
    >>> b = aritprog_gen(0, Fraction(1, 3), 1)
    >>> list(b)
    [Fraction(0, 1), Fraction(1, 3), Fraction(2, 3)]
    >>> from decimal import Decimal
    >>> c = aritprog_gen(0, Decimal('.01'), .03)
    >>> list(c)
    [Decimal('0'), Decimal('0.01'), Decimal('0.02')]
    """
    result = type(begin+step)(begin)
    # Возвращает значение того же класса,
    # что и начальное begin
    forever = end is None
    index = 0
    while forever or result < end:
        yield result
        index += 1
        result = begin + step * index
```

# Построение арифметической прогрессии с помощью `itertools`

Функция `itertools.count` возвращает генератор, порождающий числа. Без аргументов порождает ряд целых начиная с 0. А если задать аргументы `start` и `step`, то получится результат схожий с тем, что дает функция `aritprog_gen`

Однако `itertools.count` никогда не останавливается.

`itertools.takewhile` - порождает генератор, который потребляет другой генератор и останавливается, когда заданный предикат станет равен `False`.

```
>>> import itertools
>>> gen = itertools.takewhile(lambda n: n<3, itertools.count(1, .5))
>>> list (gen)
[1, 1.5, 2.0, 2.5]
```

`aritprog_v3.py`: *работает так же как и* `aritprog_gen.py`

```
import itertools


def aritprog_gen(begin, step, end=None):
    first = type(begin+step)(begin)
    ap_gen = itertools.count(first, step)
    if end is not None:
        ap_gen = itertools.takewhile(lambda n: n < end, ap_gen)
    return ap_gen
```

# Генераторные функции в стандартной библиотеке

*Table 4. Фильтрующие генераторные функции*

| Модуль | Функция | Описание |
|---|---|---|
| itertools | `compress(it, selector_it)` | Потребляет параллельно два итерируемых объекта; отдает элемент `it`, когда соответствующий `selector_it` принимает истинное значение. |
| | `dropwhile(predicate, it)` | Потребляет `it`, пропуская элементы, пока `predicate` принимает похожее на истину значение, а затем отдает все оставшиеся элементы (больше никаких проверок не делает) |
| встроенная | `filter(predicate, it)` | Применяет предикат к каждому элементу итерируемого объекта, отдавая элемент, если `predicate(item)` принимает похожее на истину значение; если `predicate` равен `None`, отдаются только элементы, принимающие похожее значение. |

| Модуль | Функция | Описание |
|---|---|---|
| itertools | `filterfalse(predicate, it)` | То же, что `filter`, но логика инвертирована: отдаются элементы, для которых предиката принимает похожее на ложь значение. |
| | `islice(it, stop)` или `islice(it, start, stop, step=1)` | Отдает элементы из среза `it` по аналогии с `s[:stop]` или `s[start:stop:step]`, только `it` может быть произвольным итерированным объектом, а операция ленивая. |
| | `takewhile(predicate, it)` | Отдает элементы, пока `predicate` принимает похожее на истину значение, затем останавливается, больше никаких проверок не делается. |

*Примеры фильтрующих генераторных функций*

```
>>> def vowel(c):
...     return c.lower() in 'aeiou'
>>> list(filter(vowel, 'Aardvark'))
['A', 'a', 'a']
>>> import itertools
>>> list(itertools.filterfalse(vowel, 'Aardvark'))
['r', 'd', 'v', 'r', 'k']
>>> list(itertools.dropwhile(vowel, 'Aardvark'))
['r', 'd', 'v', 'a', 'r', 'k']
>>> list(itertools.takewhile(vowel, 'Aardvark'))
['A', 'a']
>>> list(itertools.compress('Aardvark', (1,0,1,1,0,1)))
['A', 'r', 'd', 'a']
>>> list(itertools.islice('Aardvark', 4))
['A', 'a', 'r', 'd']
>>> list(itertools.islice('Aardvark', 4, 7))
['v', 'a', 'r']
>>> list(itertools.islice('Aardvark', 1, 4, 7))
['a']
>>> list(itertools.islice('Aardvark', 1, 7, 2))
['a', 'd', 'a']
```

*Table 5. Отображающие генераторные функции*

| Модуль | Функция | Описание |
|---|---|---|
| itertools | accumulate(it, [func]) | Отдает накопленные суммы; если задана функция func, то отдает результат, применяя её к первой паре элементов, затем к первому результату и следующему элементу и т.д. |
| встроенная | enumerate(iterable, start=0 | Отдает 2-кортежи вида (index, item), где index начинается со значения start, а item, извлекается из iterable. |
| | map(func, it1, [it2, ⋯, itN]) | Применяет func к каждому элементу it и отдает результат; если задано N итерируемых объектов, то func должна принимать N итерируемых объектов, и все итерируемые объекты обходятся параллельно. |
| itertools | starmap(func, it) | Применяет func к каждому элементу it и отдает результат; входной итерируемый объект должен отдавать итерируемые объекты iit, а func вызывается в виде func(*iit) |

*Примеры применения генераторной функции* `itertools.accumulate`

```
>>> sample = [5, 4, 2, 8, 7, 6, 3, 0, 9, 1]
>>> list(itertools.accumulate(sample)) # Частичные суммы.
[5, 9, 11, 19, 26, 32, 35, 35, 44, 45]
>>> list(itertools.accumulate(sample, min)) # Частичные минимумы.
[5, 4, 2, 2, 2, 2, 2, 0, 0, 0]
>>> list(itertools.accumulate(sample, max)) # Частичные максимумы.
[5, 5, 5, 8, 8, 8, 8, 8, 9, 9]
>>> import operator
>>> list(itertools.accumulate(sample, operator.mul)) # Частичные произведения.
[5, 20, 40, 320, 2240, 13440, 40320, 0, 0, 0]
>>> list(itertools.accumulate(range(1, 11), operator.mul)) # Факториалы от 1! до 10!.
[1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

*Примеры применения отображающих генераторных функций*

```
>>> list(enumerate('XOMRKOB', 1))
[(1, 'X'), (2, 'O'), (3, 'M'), (4, 'R'), (5, 'K'), (6, 'O'), (7, 'B')]
>>> list(map(operator.mul, range(11), range(11))) # Квадраты целых чисел от 0 до 10
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> list(map(operator.mul, range(11), [2, 4, 8])) # Перемножение целых чисел из двух
параллельных итерируемых объектов; операция заканчивается, когда будет достигнут конец
более короткого объекта.
[0, 4, 16]
>>> list(map(lambda a, b: (a, b), range(11), [2, 4, 8])) # То же самое делает
встроенная функция zip
[(0, 2), (1, 4), (2, 8)]
>>> list(itertools.starmap(operator.mul, enumerate('albatroz', 1))) # Повторение
каждой буквы слова, столько раз, каков номер ее позиции
['a', 'll', 'bbb', 'aaaa', 'ttttt', 'rrrrrr', 'ooooooo', 'zzzzzzzz']
>>> sample = [5, 4, 2, 8, 7, 6, 3, 0, 9, 1]
>>> list(itertools.starmap(lambda a, b: b/a, enumerate(itertools.accumulate(sample),
1))) # Частичное среднее 
[5.0, 4.5, 3.6666666666666665, 4.75, 5.2, 5.333333333333333, 5.0, 4.375,
4.888888888888889, 4.5]
```

| Модуль | Функция | Описание |
|---|---|---|
| itertools | `chain(it1, …, itN)` | Отдает все элементы из `it1`, затем из `it2` и т.д. |
| | `chain.from_iterable(it)` | Отдает все элементы из каждого итерируемого объекта, порождаемого `it`, перебирая их один за другим; `it` должен порождать итерируемые объекты, например это может быть список итерируемых объектов. |
| | `product(it1, …, itN, repeat=1)` | Декартово произведение: отдает N-кортежи, полученные путём комбинирования элементов из каждого входного итерируемого объекта,- так, как это делалось бы с помощью вложенных циклов `for`; аргумент `repeat` позволяет обходить входные итерируемые объекты больше одного раза. |

| Модуль | Функция | Описание |
|---|---|---|
| встроенная | `zip(it1, ⋯, itN)` | Отдает N-кортежи, построенные из элементов, которые берутся параллельно из входных итерируемых объектов; операция прекращается по исчерпанию самого короткого итерируемого элемента. |
| `itertools` | `zip_longest(it1, ⋯, itN, fillvalue=None)` | Отдает N-кортежи, построенные из элементов, которые берутся параллельно из входных итерируемых объектов; операция прекращается по исчерпанию самого длинного итерируемого элемента, а вместо недостающих элементов подставляется значение `fillvalue`. |

*Примеры применения объединяющих генераторных функций*

```
>>> list(itertools.chain('ABC', range(2))) # chain обычно вызывается с 2мя и более
итерируемых объектов
['A', 'B', 'C', 0, 1]
>>> list(itertools.chain(enumerate('ABC'))) # При вызове с одним элементом, chain не
делает ничего полезного
[(0, 'A'), (1, 'B'), (2, 'C')]
>>> list(itertools.chain.from_iterable(enumerate('ABC'))) # chain.from_iterable берёт
каждый элемент из итерируемого объекта и сцепляет их в последовательность, при
условии, что каждый элемент является итерируемым объектом
[0, 'A', 1, 'B', 2, 'C']
>>> list(zip('ABC', range(5)))
[('A', 0), ('B', 1), ('C', 2)]
>>> list(zip('ABC', range(5), [10, 20, 30, 40]))
[('A', 0, 10), ('B', 1, 20), ('C', 2, 30)]
>>> list(itertools.zip_longest('ABC', range(5), [10, 20, 30, 40]))
[('A', 0, 10), ('B', 1, 20), ('C', 2, 30), (None, 3, 40), (None, 4, None)]
```

*Примеры применения генераторной функции* `itertools.product`

```
>>> import itertools
>>> list(itertools.product('ABC', range(2)))
[('A', 0), ('A', 1), ('B', 0), ('B', 1), ('C', 0), ('C', 1)]
>>> suits = 'Черви Буби Кресты Пики'.split()
>>> list(itertools.product('AKQJ', suits))
[('A', 'Черви'), ('A', 'Буби'), ('A', 'Кресты'), ('A', 'Пики'), ('K', 'Черви'), ('K',
'Буби'), ('K', 'Кресты'), ('K', 'Пики'), ('Q', 'Черви'), ('Q', 'Буби'), ('Q',
'Кресты'), ('Q', 'Пики'), ('J', 'Черви'), ('J', 'Буби'), ('J', 'Кресты'), ('J',
'Пики')]
```

*Table 6. Генераторные функции, расширяющие каждый входной элемент в несколько выходных*

| Модуль | Функция | Описание |
|---|---|---|
| itertools | `combinations(in, out_len)` | Отдает комбинации `out_len` элементов из элементов, отдаваемых `it`. |
| | `combinations_with_replacement(it, out_len)` | Отдает комбинации `out_len` элементов из элементов, отдаваемых `it`, включая комбинации с повторяющимися элементами. |
| | `count(start=0, step=1)` | Отдает числа, начиная с `start` с шагом `step`. |
| | `cycle(it)` | Отдает элементы из `it`, запоминая копию каждого, после чего отдает всю последовательность еще раз — и так до бесконечности. |
| | `permutations(it, out_len=None)` | Отдает перестановки `out_len` элементов из элементов, отдаваемых `it`; по-умолчанию `out_len` равно `len(list(it))`. |
| | `repeat(item, [times])` | Повторно отдает заданный элемент - `times` раз или бесконечно, если аргумент не задан. |
| | `groupby(it, key=None)` | Порождает 2-кортежи вида (`key`, `group`), где `key` — критерий группировки, а `group` — генератор, отдающий элементы группы. |
| | `tee(it, n=2)` | Отдает кортеж `n` генераторов, каждый из которых независимо отдает элементы входного итерируемого объекта. |
| встроенная | `reversed(seq)` | Отдает элементы `seq` в обратном порядке, от последнего к первому; аргумент `seq` должен быть последовательностью или иметь реализованный метод `__reversed__`. |

*Пример использования* `itertools.groupby`

```
>>> list(itertools.groupby('AAAABBBBCCCCDDDD'))
[('A', <itertools._grouper object at 0x000001084DAE74F0>), ('B', <itertools._grouper
object at 0x000001084DAE7D30>), ('C', <itertools._grouper object at
0x000001084DAE7400>), ('D', <itertools._grouper object at 0x000001084DAE7CA0>)]
>>> for char, group in itertools.groupby('AAAABBBBCCCCDDDD'):
...     print(f'{char} --> {list(group)}')
...
A --> ['A', 'A', 'A', 'A']
B --> ['B', 'B', 'B', 'B']
C --> ['C', 'C', 'C', 'C']
D --> ['D', 'D', 'D', 'D']
>>> animals = ['утка', 'орёл', 'мышь', 'жираф', 'медведь', 'летучая мышь', 'дельфин',
'акула', 'лев']
>>> animals.sort(key=len)
>>> animals
['лев', 'утка', 'орёл', 'мышь', 'жираф', 'акула', 'медведь', 'дельфин', 'летучая
мышь']
>>> for length, group in itertools.groupby(reversed(animals), len):
...     print(f'{length} --> {list(group)}')
...
12 --> ['летучая мышь']
7 --> ['дельфин', 'медведь']
5 --> ['акула', 'жираф']
4 --> ['мышь', 'орёл', 'утка']
3 --> ['лев']
```

## `yield from` — новая конструкция в Python 3.3

Вложенные циклы `for` — традиционное решение в случае, когда генераторная функция должна отдавать значения, порождаемые другим генератором.

*Пример реализации сцепляющего генератора*

```
>>> def chain(*iterable):
...     for it in iterable:
...         for i in it:
...             yield i
...
>>> s = 'ABC'
>>> t = tuple(range(3))
>>> list(chain(s,t))
['A', 'B', 'C', 0, 1, 2]
```

Генераторная функция `chain` дает шанс поработать каждому полученному итерируемому объекту по очереди. В документе PEP 380 — Syntax for Delegating to a Subgenerator описан новый синтаксис для решения этой задачи, он показан ниже.

```
>>> def chain(*iterables):
...     for i in iterables:
...         yield from i

>>> s = 'ABC'
>>> t = tuple(range(3))
>>> list(chain(s, t))
['A', 'B', 'C', 0, 1, 2]
```

Как видим `yield from i` полностью заменяет внутренний цикл `for`. Данная конструкция выглядит лучше, но это всего лишь *синтаксическая глазурь*.

## Функции редуцирования итерируемого объекта

Все функции из таблицы ниже принимают в качестве аргумента итерируемый объект и возвращают единственный результат. Такие функции называются ***редуцирующими***, ***сворачивающими*** или ***аккумулирующими***.

| Модуль | Функция | Описание |
|---|---|---|
| встроенная | `all(it)` | Возвращает `True`, если все элементы `it` принимают истинное значение, в противном случае `False`; `all([])` возвращает `True` |
| | `any(it)` | Возвращает `True`, если хотя бы один элемент `it` принимает истинное значение, в противном случае `False`; `any([])` возвращает `False` |
| | `max(it, [key=], [default=])` | Возвращает максимальный элемент `it`; `key` — функция порядка, как в `sorted`; значение `default` возвращается, если итерируемый объект пуст. |
| | `min(it, [key=], [default=])` | Возвращает минимальный элемент `it`; `key` — функция порядка, как в `sorted`; значение `default` возвращается, если итерируемый объект пуст. |
| | `sum(it, start=0)` | Сумма всех элементов `it`, к которой может быть добавлено значение `start`, если оно задано (для получения большей точности при сложении чисел с плавающей запятой пользуйтесь функцией `math.fsum`). |
| `itertools` | `reduce(func, it, [initial])` | Возвращает результат выполнения следующей процедуры: функция `func` применяется к первым 2 элементам, затем к результату и третьему элементу и т.д. Если задан `initial`, то он образует начальную пару вместе с первым элементом. |

*Результаты применения `all` и `any` к некоторым последовательностям.*

```
>>> all([1,2,3])
True
>>> all([1,0,3])
False
>>> all([])
True
>>> any([1,2,3])
True
>>> any([1,0,3])
True
>>> any([0, 0.0])
False
>>> any([])
False
```

## Более пристальный взгляд на функцию `iter`

Функцию `iter` можно вызывать с двумя аргументами, для создания итератора из обычно функции. При таком использовании первый аргумент должен быть вызываемым объектом, который будет повторно вызываться (без аргументов), для порождения значений, а второй аргумент является ограничителем — если вызываемый объект возвращает такое значение, то итератор не отдает его, а возбуждает исключение `StopIteration`.

*Использование* `iter` *для бросания шестигранной кости до тех пор, пока не выпадет 1:*

```
>>> from random import randint
>>> def d6():
...     return randint(1, 6)
>>> d6_iter = iter(d6, 1)
>>> d6_iter
<callable_iterator object at 0x00000225B6517940>
>>> for roll in d6_iter:
...     print(roll)
...

6
3
5
5
3
4
6
3
2
4
3
```

Отметим что функция `iter` здесь возвращает вызываемый итератор (`callable_iterator`). Цикл `for` в этом примере может работать очень долго, но никогда не покажет единицы, поскольку это значение-ограничитель.

Одним из полезных приложений второй формы `iter()` является создание считывателя блоков. Например, чтение блоков фиксированной ширины из файла двоичной базы данных до тех пор, пока не будет достигнут конец файла:

*Полезный пример из документации по встроенной функции* `iter`

```
from functools import partial
with open('mydata.db', 'rb') as f:
    for block in iter(partial(f.read, 64), b''):
        process_block(block)
```

# Ссылочки

- PEP 255 — Simple Generators
- itertools — Functions creating iterators for efficient looping
- Itertools Recipes

# Глава 15. Контекстные менеджеры и блоки `else`

## Блоки `else` вне `if`

*for*

Блок `else` выполняется, только если цикл `for` дошел до конца (т.е. не было преждевременного выхода с помощью `break`)

*while*

Блок `else` выполняется, только если цикл `while` завершился вследствие того, что условие приняло ложное условие(а не в результате преждевременного выхода с помощью `break`).

*try*

Блок `else` выполняется, только если в блоке `try` не возникало исключение.

## Контекстные менеджеры и блоки `with`

Предложение `with` было задумано, для того что бы упростить конструкцию `try/finally`, гарантирующую, что некоторая операция будет выполнена после блока, даже если этот блок прерван в результате исключения, предложения `return` или вызова `sys.exit()`. Код внутри `finally` обычно освобождает критически важный ресурс или восстанавливает временно изменённое состояние.

*Пример реализации контекстного менеджера*

```
class LookingGlass:
    """
    >>> from mirror import LookingGlass
    >>> with LookingGlass() as what:
    ...     print('шалаш')
    ...     print(what)
    шалаш
    РАКОМ ДЕД ЕБЕТ КОБЫЛУ
    >>> what
    'УЛЫБОК ТЕБЕ ДЕД МОКАР'
    >>> print('Back to normal')
    Back to normal
    """

    def __enter__(self):
        import sys
        self.original_write = sys.stdout.write
        sys.stdout.write = self.reverse_write
        return 'УЛЫБОК ТЕБЕ ДЕД МОКАР'

    def reverse_write(self, text):
        self.original_write(text[::-1])

    def __exit__(self, exc_type, exc_val, exc_tb):  # Python вызывает метод __exit__
        # с аргументами None, None, None, если не было ошибок; если было исключение,
        # то в аргументах передаются данные об исключении
        import sys
        sys.stdout.write = self.original_write
        if exc_type is ZeroDivisionError:
            print('Division by zero')
            return True     # Посылаем True, что бы интерпретатор знал об обработке
            # исключения. Если метод __exit__ возвращает None или вообще что-нибудь,
            # кроме True, то исключение возникшее внутри блока with, распространяется
дальше.


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Реальные приложения, перехватывающие стандартный вывод, обычно хотят временно подменить `sys.stdout` похожим на файл объектом, а затем восстановить исходное состояние. Именно это и делает контекстный менеджер `contextlib.redirect_stdout` link: просто передайте ему похожий на файл объект.

### Примеры контекстных менеджеров из стандартной библиотеки

- Управление транзакциями в модуле `sqlite3` Using the connection as a context manager
- Хранение блокировок, условных переменных и семафоров в модуле `threading` Using locks, conditions, and semaphores in the with statement
- Настройка среды для арифметических операций с объектами `Decimal` decimal.localcontext(ctx=None)
- Внесение временных изменений в объекты для тестирования unittest.mock.patch(target

## Утилиты `contextlib`

Прежде чем писать свои собственные классы контекстных менеджеров, прочитайте contextlib — Utilities for with-statement contexts

`closing` — функция для построения контекстных менеджеров из объектов, которые предоставляют метод `close()`, но не реализуют протокол `__enter__/__exit__`.

`suppress` — контекстный менеджер для временного игнорирования заданных исключений.

`@contextmanager` — декоратор, который позволяет построить контекстный менеджер из простой генераторной функции, вместо того, что бы создавать класс и реализовывать протокол

`ContextDecorator` — базовый класс для определения контекстных менеджеров на основе классов, которые можно использовать так же как и декоратор функции, так что вся функция будет работать внутри управляемого контекста.

`ExitStack` — контекстный менеджер, который позволяет составить композицию из переменного числа контекстных менеджеров

## Использование `@contextmanager`

`mirror_gen.py`: *реализация контекстного менеджера с помощью генератора*

```
import contextlib


@contextlib.contextmanager
def looking_glass():

    import sys
    original_write = sys.stdout.write

    def reverse_write(text):
        """
        >>> from mirror_gen import looking_glass
        >>> with looking_glass() as what:
        ...     print('шалаш')
        ...     print(what)
        шалаш
        РАКОМ ДЕД ЕБЕТ КОБЫЛУ
        >>> what
        'УЛЫБОК ТЕБЕ ДЕД МОКАР'
        >>> print('Back to normal')
        Back to normal
        """
        original_write(text[::-1])
        sys.stdout.write = reverse_write
        yield 'УЛЫБОК ТЕБЕ ДЕД МОКАР'   # Отдаем значение, которое
        # будет связано с переменной в части as предложения with.
        # В этой точке функция приостанавливается на время
        # выполнения блока with.
        sys.stdout.write = original_write   # Всё что идет после
        # yield выполняется после блока with


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

# Глава 16. Сопрограммы

Строка вида `yield item` порождает(производит) значение, которое получает сторона, вызывающая `next(⋯)`, и кроме того, она уступает процессор, приостанавливая выполнение генератора, что бы *вызывающая* сторона могла продолжить работу до момента, когда ей понадобится следующее значение от `next()`. *Вызывающая* сторона "вытягивает" значения из генератора.

**Сопрограмма** синтаксически выглядит как генератор: просто функция в теле которой встречается ключевое слово `yield`. Однако в сопрограмме `yield` обычно находится в правой части выражения присваивания (пр.: `datum = yield`) и может порождать или не порождать значения — если после слова `yield` нет никакого выражения, генератор отдает `None`. **Сопрограмма** может получать данные от *вызывающей* стороны, если та вместо `next(⋯)` воспользуется методом `.send(datum)`. Обычно *вызывающая* сторона отправляет сопрограмме значения.

Сопрограмма может находиться в одном из четырех состояний. Узнать, в каком именно, позволяет функция `inspect.getgeneratorstate(⋯)`.

`GEN_CREATED` — *Ожидает начала выполнения.*

`GEN_RUNNING` — *Выполняется интерпретатором.*

`GEN_SUSPENDED` — *Приостановлена в выражении* `yield`.

`GEN_CLOSED` — *Исполнение завершилось.*

## Пример: сопрограмма для вычисления накопительного среднего

`coroaverager0.py`: *сопрограмма для вычисления накопительного среднего*

```
def averager():
    """
    >>> coro_avg = averager()
    >>> next(coro_avg)          # Init generator
    >>> coro_avg.send(10)
    10.0
    >>> coro_avg.send(30)
    20.0
    >>> coro_avg.send(5)
    15.0
    :return:
    """

    total = .0
    count = 0
    average = None
    while True:
        term = yield average
        total += term
        count += 1
        average = total/count
```

# Декоратор для инициализации сопрограммы

Пока сопрограмма не инициализирована, она почти бесполезна, нужно не забыть вызвать `next(my_coro)` до `my_coro.send(x)`. Что бы облегчить работу иногда используют инициализирующий декоратор.

`coroutil.py`: *декоратор для инициализации сопрограммы.*

```
from functools import wraps


def coroutine(func):
    """Decorator: primes `func` by advancing to first `yield`"""
    @wraps(func)
    # Декорированная функция подменяется этой функцией primer
    def primer(*args, **kwargs):
        # Вызываем декорируемую функцию, что бы получить инициализированный генератор
        gen = func(*args, **kwargs)
        # Инициализируем генератор
        next(gen)
        # Возвращаем его
        return gen
    return primer
```

coroaverager1.py: *код сопрограммы с использованием инициализирующего декоратора*

```
from coroutil import coroutine


@coroutine
def averager():
    """
    A coroutine to compute a running average
    Сопрограмма для вычисления текущего среднего значения

    >>> coro_avg = averager()   # вызываем `averager()`, она создает объект-генератор,
который
    >>> from inspect import getgeneratorstate   # инициализируется в функции `primer`
декоратора `coroutine`
    >>> getgeneratorstate(coro_avg) # возвращает ``'GEN_SUSPENDED'``, т.е. программа
готова к приему значений
    'GEN_SUSPENDED'
    >>> coro_avg.send(10)   # мы можем сразу отправлять значения, в этом и состоял
смысл генератора
    10.0
    >>> coro_avg.send(30)
    20.0
    >>> coro_avg.send(5)
    15.0
    """
    total = .0
    count = 0
    average = None
    while True:
        term = yield average
        total += term
        count += 1
        average = total/count


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

# Завершение сопрограммы и обработка исключений

Необработанное исключение распространяется в функцию, из которой был произведён вызов `next` или `send`, приведший к исключению.

*Как не обработанное исключение аварийно завершает сопрограмму*

```
>>> from source.coroaverager1 import averager
>>> coro_avg = averager()
>>> coro_avg.send(40)
40.0
>>> coro_avg.send(50)
45.0
>>> coro_avg.send('spam')
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +=: 'float' and 'str'
>>> coro_avg.send(60)
Traceback (most recent call last):
  File "C:\Users\User\AppData\Local\Programs\Python\Python310\lib\code.py", line 90,
in runcode
    exec(code, self.locals)
  File "<input>", line 1, in <module>
StopIteration
```

Отправка не числового значения приводит к исключению в сопрограмме. Поскольку исключение не обрабатывается сопрограммой, она завершается. Любая попытка вновь активировать сопрограмму вызовет исключение `StopIteration`.

Приведенный пример показывает возможный способ завершения сопрограммы: послать некоторое специально значение, которое сопрограмма интерпретирует как признак завершения. Удобным кандидатом на эту роль является константные встроенные значения *синглтоны*, например `None` и `Ellipsis`. У `Ellipsis` к тому же есть то достоинство, что в обычных потоках данных он практически не встречается. Так же встречается в качестве признака сам класс `StopIteration`, а не его экземпляр (и без возбуждения исключения такого типа), т.е. таким образом: `my_coro.send(StopIteration)`.

Начиная с версии Python 2.5, у объектов-генераторов есть два метода которые позволяют клиенту явно отправить сопрограмме исключение: `throw` и `close`.

`generator.throw(exc_type[, exc_value[, traceback]])` — приводит к тому, что выражение `yield`, в котором генератор приостановлен, возбуждается исключение. Если генератор обрабатывает исключение, то выполнение продолжится до следующего `yield`, а отданное значение станет значением вызова `generator.throw`. Если же исключение не обработано генератором, то оно распространяется в контекст вызывающей стороны.

`generator.close()` — выражение `yield`, в котором генератор приостановлен, возбуждает исключение `GeneratorExit`. Если генератор не обработает данное исключение или возбудит исключение `StopIteration` — обычно в результате выполнения до конца — вызывающая сторона не получит никакой ошибки. Получив исключение `GeneratorExit`, генератор **не должен** отдавать значения, иначе возникнет исключение `RuntimeError`. Если генератор возбудит другое исключение, то оно распространится в контексте вызывающей стороны.

Официальная документация по методам объекта-генератора https://docs.python.org/3/reference/expressions.html#generator-iterator-methods

`coro_exc_demo.py`: *тестовый код для изучения обработки исключений в сопрограммах*

```
class DemoException(Exception):
    """An exception type for the demonstration."""


def demo_exc_handling():
    print('--> coroutine started')
    while True:
        try:
            x = yield
        except DemoException:
            print('*** DemoException handled. Continuing...')
        else:
            print('--> coroutine received: {!r}'.format(x))
    raise RuntimeError('This line should never run.')
```

Последняя строка в примере не достижима, потому что из бесконечного цикла можно выйти только в результате необработанного исключения, а это приводит к немедленному завершению сопрограммы.

*Активация и завершени* `demo_exc_handling` *без исключения*

```
>>> exc_coro = demo_exc_handling()
>>> next(exc_coro)
--> coroutine started
>>> exc_coro.send(11)
--> coroutine received: 11
>>> exc_coro.send(22)
--> coroutine received: 22
>>> exc_coro.close()
>>> from inspect import getgeneratorstate
>>> getgeneratorstate(exc_coro)
'GEN_CLOSED'
```

Если в `demo_exc_handling` методом `throw` передано исключение `DemoException`, то оно обрабатывается, и сопрограмма продолжится, как показано в примере ниже.

*Возбуждение исключения* `DemoException` *не приводит к выходу из нее*

```
>>> from source.coro_exc_demo import demo_exc_handling, DemoException
>>> exc_coro = demo_exc_handling()
>>> next(exc_coro)
--> coroutine started
>>> exc_coro.send(11)
--> coroutine received: 11
>>> exc_coro.throw(DemoException)
*** DemoException handled. Continuing...
>>> from inspect import getgeneratorstate
>>> getgeneratorstate(exc_coro)
'GEN_SUSPENDED'
```

С другой стороны, если возбуждённое в сопрограмме исключение не обработано, то она останавливается и переходит в состояние `GEN_CLOSED`.

*Сопрограмма завершается, если не может обработать возбуждённое в ней исключение*

```
>>> exc_coro = demo_exc_handling()
>>> next(exc_coro)
--> coroutine started
>>> exc_coro.throw(ZeroDivisionError)
Traceback (most recent call last):
...
    x = yield
ZeroDivisionError
>>> getgeneratorstate(exc_coro)
'GEN_CLOSED'
```

Если необходимо, чтобы вне зависимости от способа завершения сопрограммы был выполнен какой-то код очистки, то соответсвующую часть тела сопрограммы нужно обернуть блоком `try\finally`, как показано в примере ниже.

`coro_finally_demo.py`: *использование* `try\finally` *для выполнения некоторых действий по завершении сопрограммы*

```
class DemoException(Exception):
    """An exception type for the demonstration."""


def demo_exc_handling():
    print('--> coroutine started')
    try:
        while True:
            try:
                x = yield
            except DemoException:
                print('*** DemoException handled. Continuing...')
            else:
                print('--> coroutine received: {!r}'.format(x))
    finally:
        print('--> coroutine ending')
```

Одна из основных причин добавления конструкции `yield from` в **Py >= 3.3** имеет отношение к возбуждению исключений во вложенных сопрограммах. Другая причина — обеспечить более удобный возврат значений их сопрограмм.

## Возврат значений из сопрограммы

Некоторые сопрограммы не отдают ничего интересного, а написаны с целью вернуть значение в конце — зачастую некий аккумулированный результат.

coroaverager2.py: *сопрограмма* averager, *возвращающая результат*

```
from collections import namedtuple

Result = namedtuple('Result', 'count average')


def averager():
    total = 0.0
    count = 0
    average = None
    while True:
        term = yield
        if term is None:
            # Что бы вернуть значение, сопрограмма должна
            # завершиться успешно, поэтому проверяется
            # условие выхода из цикла подсчета среднего
            break
        total += term
        count += 1
        average = total/count
    return Result(count, average)
```

```
>>> from source.coroaverager2 import averager
>>> coro_avg = averager()
>>> next(coro_avg)
>>> coro_avg.send(10)   # Эта версия не отдает значений
>>> coro_avg.send(20)
>>> coro_avg.send(30)
>>> coro_avg.send(6.5)
>>> coro_avg.send(None) # Отправка None приводит к выводу из цикла и завершению
сопрограммы  возвратом результата.
Traceback (most recent call last):
...
StopIteration: Result(count=4, average=16.625)
```

Как обычно, генератор возбуждает исключение StopIteration. Возвращаемое значение можно прочитать из атрибута исключения value.

*Перехват* `StopIteration` *позволяет получить значение, возвращаемое* `averager`

```
>>> from source.coroaverager2 import averager
>>> coro_avg = averager()
>>> next(coro_avg)
>>> coro_avg.send(10)
>>> coro_avg.send(30)
>>> coro_avg.send(6.5)
>>> try:
...     coro_avg.send(None)
... except StopIteration as exc:
...     result = exc.value
...
>>> result
Result(count=3, average=15.5)
```

# Использование `yield from`

`yield from` — совершенно новая конструкция. Она уместна настолько больше `yield`, что использование одного и того же слова только вводит в заблуждение. Аналогичные конструкции в других языках называют `await`.

Основное применение `yield from` — открытие двустороннего канала между внешней вызывающей программой и внутренним субгенератором, так что бы значения можно было отправлять и отдавать напрямую, а исключения возбуждать и обрабатывать без написания громоздкого стереотипного кода и промежуточных сопрограмм. Это открывает новую возможность — делегирование сопрограмме.

`coroaverager3.py`: *использование* `yield from` *для управления сопрограммой* `averager` *и печати соответствующего отчёта.*

```
from collections import namedtuple

Result = namedtuple('Result', 'count average')


def averager():
    """
    Субгенератор
    """
    total = .0
    count = 0
    average = None
    while True:
        term = yield
        # каждое значение, отправленное клиентским
        # кодом, здесь связывается с переменной term.
        if term is None:
```

```
            # Условие окончания. Без него выражение
            # yield from, вызвавшее эту сопрограмму,
            # оказалось бы навечно заблокированным.
            break
        total += term
        count += 1
        average = total/count
    return Result(count, average)
    # Возвращённое значение Result является значением
    # выражения yield from в grouper


def grouper(results, key):
    """
    Делегирующий генератор
    """
    while True:
        # На каждой итерации этого цикла создается
        # новый экземпляр averager; каждый из них
        # является объектом-генератором, работающим
        # как сопрограмма.
        # Значение, отправленное генератору grouper,
        # помещается выражением yield from в канал,
        # открытый с объектом averager.
        # grouper остается остановленным, пока averager
        # потребляет значения, отправляемые клиентом.
        # Когда выполнение averager завершится,
        # возвращённое им значение будет связано с
        # result[key].
        # После этого в цикле white создается очередной
        # экземпляр averager для потребления последующих
        # значений.
        results[key] = yield from averager()


def main(data):
    """
    Клиентский код или вызывающая сторона в терминологии
    PEP 380. Эта функция управляет всеми остальными
    """
    results = {}
    for key, values in data.items():
        group = grouper(results, key)   # <- Делегирующий генератор
        # group -- объект-генератор, получающийся в результате
        # вызова grouper с аргументом results -- словарём, в
        # котором будут собираться результаты, - и key - конкретным
        # ключом этого словаря. Этот объект будет работать как
        # сопрограмма.
        next(group)
        # Инициализируем сопрограмму
        for value in values:
```

```
            group.send(value)
            # Отправляем каждое значение value объекту grouper.
            # Оно будет получено в строке term = yield кода averager;
            # grouper его никогда не увидит.
        group.send(None)
        # Отправка значения None объекту grouper приводит к завершению
        # текущего экземпляра averager и дает возможность grouper
        # возобновить выполнение и создать очередной объект averager
        # для обработки следующей группы значений.
    # print(results)
    report(results)


def report(results):
    for key, result in sorted(results.items()):
        group, unit = key.split(';')
        print(f'{result.count:2} {group:5} averaging {result.average:.2f}{unit}')


data = {
    'girls;kg': [40.9, 44.4, 40.9, 44.4, 40.9, 44.4, 40.9, 44.4, ],
    'girls;m': [1.6, 1.72, 1.5, 1.6, 1.72, 1.5, 1.6, 1.72, 1.5, ],
    'boys;kg': [40.9, 44.4, 40.9, 44.4, 40.9, 44.4, 40.9, 44.4, ],
    'boys;m': [1.6, 1.72, 1.5, 1.6, 1.72, 1.5, 1.6, 1.72, 1.5, ],
}

if __name__ == '__main__':
    main(data)
```

*По поводу* `send(None)`

Если субгенератор не остановить, при помощи `None`, то инициализирующий генератор будет навечно заблокирован в `yield from`.

Данный пример показывает простейшую конфигурацию `yield from`, когда имеется только один делегирующий генератор и один субгенератор. Поскольку делегирующий генератор работает как канал, мы можем соединить любое их количество, сформировав конвейер.

# Пример: применение сопрограмм для моделирования дискретных событий.

> Сопрограммы дают естественный способ выразить многие алгоритмы, в том числе моделирование, игры, асинхронный ввод-вывод и другие формы событийно-управляемого программирования или невытесняющей многозадачности[1].
>
> — Гвидо ван Россум и Филипп Дж. Эби. PEP 342 - Coroutines via Enhanced Generators

Мотивация приведённого ниже примера - не только академический интерес. Сопрограммы — это фундаментальный структурный элемент пакета `asyncio`. Моделирование показывает, как реализовать параллельные операции, используя сопрограммы вместо потоков, и это очень пригодится, когда в главе 18 мы займёмся асинхронным вводом-выводом.

## О моделировании дискретных событий

Моделирование дискретных событий (discrete event simulation - DFS) — методика, предполагающая, что система моделирования в виде хронологической последовательности событий. В DES часы модельного времени сдвигаются не на одинаковое приращение, а сразу к модельному времени следующего моделируемого события. Например, если моделируется работа такси на верхнем уровне, то первое событие - это посадка пассажира, а следующее - высадка. Не важно сколько времени заняла поездка - 5 или 50 минут: когда наступает событие высадки, часы сдвигаются к времени окончания поездки за одну операцию. В DES работу такси в течении целого года можно смоделировать за секунду. Этим оно отличается от непрерывного моделирования, когда часы сдвигаются за фиксированный - и обычно небольшой - интервал.

Интуитивно понятно, что игры со сменой хода — примеры моделирования дискретных событий: состояние игры изменяется только после хода игрока, а пока игрок обдумывает следующий ход, часы модельного времени стоят.

SimPy[2] — написанный на Python пакет DES, в котором каждый моделируемый процесс представлен одной сопрограммой.

В моделировании *процессом* называют действия модельной сущности, а не процесс в смысле ОС. Моделируемый процесс можно реализовать в виде процесса ОС, но обычно для этой цели применяют сопрограмму или поток.

## Моделирование работы таксопарка

В нашей программе моделирования `taxi_sim.py` создается несколько экземпляров такси. Каждое такси совершает фиксированное количество поездок и возвращается в гараж. Такси выезжает из гаража и начинает "рыскать" — искать пассажира. Это продолжается, пока пассажир не сядет в такси, в этот момент начинается поездка. Когда пассажир выходит, такси возвращается в режим поиска.

Время поиска и поездок имеет экспоненциальное распределение. Для простоты отображения время измеряется в минутах, но для моделирования можно применять и интервалы типа `float`. Всякое изменение состояния любого такси выводится как событие.

*Реализация действий каждого такси в сопрограмме*

```
# BEGIN TAXI_PROCESS
def taxi_process(ident, trips, start_time=0):
    """
    Отдает модели события при каждом изменении состояния
    * taxi_process -- вызывается один раз для каждого такси и
      создаёт объект-генератор, представляющий его действия.
    * ident -- это номер такси.
    * trips -- сколько поездок должен совершить такси перед
      тем как вернуться в гараж.
    * start_time -- когда такси выезжает из гаража.
    """
    time = yield Event(start_time, ident, 'leave garage')
    """
    Первый отданный объект Event -- 'leave garage'
    Выполнение сопрограммы приостанавливается, так что главный
    цикл моделирования может перейти к следующему запланированному
    событию. Когда настанет время возобновить процесс, главный цикл
    отправит (методом send) текущее модельное время, которое будет
    присвоено переменной time.
    """
    for i in range(trips):
        # этот блок повторяется по одному разу для каждой поездки.
        time = yield Event(time, ident, 'pick up passenger')
        """
        Отдает событие посадки пассажира. Здесь сопрограмма
        приостанавливается. Когда настанет время возобновить этот
        процесс, главный цикл снова отправит текущее время.
        """
        time = yield Event(time, ident, 'drop off passenger')
        """
        Отдает событие высадки пассажира. Сопрограмма снова
        приостанавливается и ждёт, когда главный цикл отправит
        ее время возобновления.
        """

    yield Event(time, ident, 'going home')
    """
    Цикл for заканчивается после заданного числа поездок и отдает
    последнее событие 'going home'. Сопрограмма приостанавливается
    в последний раз. При возобновлении она получит от главного цикла
    модельное время, но я не присваиваю его никакой переменной,
    потому что оно не будет использоваться.
    """
# END TAXI_PROCESS
```

*Управление сопрограммой* `taxi_process`

```
>>> from source.taxi_sim import taxi_process
>>> taxi = taxi_process(ident=13, trips=2, start_time=0)
>>> next(taxi)
Event(time=0, proc=13, action='leave garage')
>>> taxi.send(_.time + 7) # Теперь можно отправить текущее время. В оболочке
переменная _ связана с последним результатом; здесь я добавлю 7 к текущему времени,
т.е. такси потратит на поиск первого пассажира 7 минут
Event(time=7, proc=13, action='pick up passenger')
>>> taxi.send(_.time + 23) # Отправка _.time + 23 означает, что поездка с первым
пассажиром займёт 23 минуты.
Event(time=30, proc=13, action='drop off passenger')
>>> taxi.send(_.time + 5) # Затем такси будет 5 минут искать пассажира.
Event(time=35, proc=13, action='pick up passenger')
>>> taxi.send(_.time + 48) # Последняя поездка займёт 48 минут.
Event(time=83, proc=13, action='drop off passenger')
>>> taxi.send(_.time + 1) # После завершения 2х поездок цикл заканчивается и отдается
событие `going home`.
Event(time=84, proc=13, action='going home')
>>> taxi.send(_.time + 10) # Следующая попытка послать что-то сопрограмме приводит к
естественному возврату из нее. В этот момент интерпретатор возбуждает исключение
StopIteration.
Traceback (most recent call last):
  File "C:\Users\User\AppData\Local\Programs\Python\Python310\lib\code.py", line 90,
in runcode
    exec(code, self.locals)
  File "<input>", line 1, in <module>
StopIteration
```

*Что бы создать экземпляр класса* `Simulator`*, функция* `main` *из скрипта* `taxi_sim.py` *строит словарь* `taxis`.

```
    taxis = {i: taxi_process(i, (i+1)*2, i*DEPARTURE_INTERVAL) for i in range
(num_taxis)}

    sim = Simulator(taxis)
    sim.run(end_time)
```

taxi_sim.py: *Инициализация класса* Simulator.

```
    def __init__(self, proc_map):
        """
        Очередь PriorityQueue для хранения запланированных событий,
        упорядоченная по возрастанию времени.

        :param proc_map: словарь, для построения локальной копии,
        потому что по ходу моделирования каждое такси, возвращающееся
        в гараж, удаляется из self.procs, а мы не хотим изменять объект,
        переданный пользователем
        """
        self.events = queue.PriorityQueue()
        self.procs = dict(proc_map)
```

Очередь с приоритетами — важнейшая структура данных в моделировании дискретных событий: события создаются в произвольном порядке, помещаются в очередь, а впоследствии извлекаются в порядке запланированного времени события.

taxi_sim.py: Simulator, *простейший класс моделирования дискретных событий, наиболее интересен метод* run.

```
class Simulator:

    def __init__(self, proc_map):
        """
        Очередь PriorityQueue для хранения запланированных событий,
        упорядоченная по возрастанию времени.

        :param proc_map: словарь, для построения локальной копии,
        потому что по ходу моделирования каждое такси, возвращающееся
        в гараж, удаляется из self.procs, а мы не хотим изменять объект,
        переданный пользователем
        """
        self.events = queue.PriorityQueue()
        self.procs = dict(proc_map)

    def run(self, end_time):
        """
        Планирует и отображает события, пока не истечёт время.
        :param: end_time: окончание модельного времени -- единственный
        обязательный аргумент run.
        """
        # планируем первое событие для каждой машины
        for _, proc in sorted(self.procs.items()):
            """
            Используя функцию sorted для выборки элементов self.proc,
            упорядоченных по ключу; сам ключ нам не важен, поэтому
            присваиваем его переменной _.
            """
```

```
            first_event = next(proc)
            """
            Данный вызов инициализирует каждую сопрограмму, заставляя
            её дойти до первого предложения yield, после чего ей можно
            посылать данные. Отдаётся объект event.
            """
            self.events.put(first_event)
            """
            Помещаем каждое событие в очередь с приоритетами self.events.
            Первым событием для каждого такси является `leave garage`.
            """


        sim_time = 0                # обнуляем часы модельного времени
        while sim_time < end_time:
            # Главный цикл моделирования: выполнять, пока sim_time меньше end_time.
            if self.events.empty():
                print('*** end of events ***')
                # Выход из цикла по окончанию событий в очереди.
                break

            current_event = self.events.get()
            # Получаем из очереди объект Event с наименьшим значением time.

            sim_time, proc_id, previous_action = current_event
            # Распаковываем кортеж Event.

            print('taxi: ', proc_id, proc_id * ' ', current_event)
            # Распечатываем объект Event.

            active_proc = self.procs[proc_id]
            # Извлекаем сопрограмму для активного такси из словаря self.procs.

            next_time = sim_time + compute_duration(previous_action)
            # Вычисляем время следующего возобновления, складывая sim_time и
            # результат вызова функции compute_duration(...) для предыдущего действия.

            try:
                next_event = active_proc.send(next_time)
                # Отправляем time сопрограмме такси. Сопрограмма отдаст
                # next_event или возбудит исключение
            except StopIteration:
                del self.procs[proc_id]
                # Удаляем событие из словаря, если возникло исключение.
            else:
                self.events.put(next_event)
                # В противном случае помещаем next_event в очередь.
        else:
            msg = '*** end of simulation time: {} events pending ***'
            print(msg.format(self.events.qsize()))
            # Если произошел выход из цикла в связи с истечением времени,
            # печатаем количество оставшихся в очереди событий.
```

```
# END TAXI_SIMULATOR
```

[1] Первая фраза в разделе 'Мотивация' документа PEP 342 (https://www.python.org/dev/peps/pep-0342/)

[2] См. официальную документацию по Simpy (https://simpy.readthedocs.io/en/latest/) - не путайте с хорошо известным пакетом SymPy (https://www.sympy.org/en/index.html) для символьных вычислений.

# Глава 18. Применение пакета `asyncio` для организации конкурентной работы

> Предмет конкурентности — как управляться со многими вещами одновременно. Предмет параллелизма — как сделать много вещей одномвременно. Не одно и то же, но близко. Первое касается структуры, второе — выполнения. Конкурентность предполагает способ решения задач, которая возможно (но не обязательно) поддается распараллеливанию.
>
> — Роб Пайк, соавтор языка GO.

## Сравнение потока и сопрограммы

Для сравнения будем писать CLI-спиннер в двух реализациях: потоки и сопрограммы.

`spinner_thread.py`: *анимация текстового индикатора с помощью потока*

```
import itertools
import sys
import threading
import time


class Signal:
    go = True


def spin(msg, signal):
    write, flush = sys.stdout.write, sys.stdout.flush
    for char in itertools.cycle('|/-\\'):
        status = char + ' ' + msg
        write(status)
        flush()
        write('\x08' * len(status))
        # \x08 -- символ забоя, возвращает курсор назад
        time.sleep(.1)
        if not signal.go:
            break
    write(' ' * len(status) + '\x08' * len(status))


def slow_function():
    # имитируем длительную операцию ввода-вывода
    time.sleep(5)
    """
    Данный вызов блокирует главный поток, но GIL
```

```
    освобождается,  так  что  второй поток может
    работать далее.
    """
    return 42


def supervisor():
    signal = Signal()

    spinner = threading.Thread(
        target=spin,
        args=('Thinking!', signal)
    )
    print('spinner object:', spinner)
    spinner.start()     # Запускаем второй поток
    result = slow_function()
    signal.go = False   # Сигналим для остановки
    spinner.join()      # Ожидаем завершения 2-го потока
    return result


def main():
    result = supervisor()
    print('Answer:', result)


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

`spinner_asyncio.py`: *анимация текстового индикатора с помощью сопрограммы*

```
import asyncio
import itertools
import sys


async def spin(msg):
    write, flush = sys.stdout.write, sys.stdout.flush
    status = ''
    for char in itertools.cycle(('...', 'o..', 'Oo.', 'oOo', '.oO', '..o')):
        status = char + ' ' + msg
        write(status)
        flush()
        write('\x08' * len(status))
        try:
            await asyncio.sleep(.1)
        except asyncio.CancelledError:
            break
    write(' ' * len(status) + '\x08' * len(status))


async def slow_function():
    await asyncio.sleep(10)
    return 42


async def supervisor():
    spinner = asyncio.create_task(spin('thinking!'))
    print('spinner object:', spinner)
    result = await slow_function()
    spinner.cancel()
    return result


async def main():
    result = await supervisor()
    print('Answer:  ', result)


if __name__ == '__main__':
    asyncio.run(main())
```

Пришлось немного переделать пример. Книга издана до выпуска Py 3.10. Ранее сопрограммы модуля `asyncio` задавались декоратором `asyncio.croutines`. В свежих версия перешли на синтаксии `async`/`await`. → (asyncio — Asynchronous I/O)

Сопрограмма по умолчанию защищена от прерывания. Мы должны явно уступить управление, что бы другие части программы могли продолжить работу. Вместо удержания

блокировок для синхронизации нескольких потоков мы имеем сопрограммы, которые "синхронизированы" по определению: в каждый момент может работать только одна сопрограмма. А когда мы захотим уступить работу планировщику, то воспользуемся выражением `yield` или `yield from` (или `await` [3]). Именно поэтому прерывания сопрограммы безопасно: по определению сопрограмму можно прервать, только когда она приостановлена в точке `yield`, а, значит, мы сможем выполнить необходимую очистку, перехватив исключение `CancelledError`.

[3] Pycharm с Python 3.10 сразу начал говорить об устаревшем методе `@asyncio.coroutine` в пользу использования синтаксиса `async` / `await`