

# Некоторые вопросы программирования на языке Python и приемы работы со специализированными библиотеками

## Содержание

<b>1</b>	<b>Терминология</b>	<b>3</b>
<b>2</b>	<b>Соглашения по именованию классов, функций и переменных</b>	<b>4</b>
<b>3</b>	<b>Приемы работы с пакетом Nox</b>	<b>4</b>
3.1	Общий шаблон . . . . .	4
3.2	Запуск тестов в мультисредах Python . . . . .	6
3.3	Nox как утилита командной строки . . . . .	6
<b>4</b>	<b>Приемы работы с pip</b>	<b>8</b>
<b>5</b>	<b>Аннотация типов</b>	<b>8</b>
5.1	Вариантность в типах Callable . . . . .	10
5.2	Аннотирование чисто позиционных и вариadicеских параметров . . . . .	10
<b>6</b>	<b>Приемы работы с pytest</b>	<b>10</b>
6.1	Особенности импорта . . . . .	10
<b>7</b>	<b>Ошибка ValueError: generator already executing в многопоточных приложениях с генераторами</b>	<b>11</b>
<b>8</b>	<b>Раскраска ячеек в Jupyterlab</b>	<b>12</b>
<b>9</b>	<b>Разреженные матрицы LIL и CSC</b>	<b>12</b>
<b>10</b>	<b>Метод __repr__ и модуль inspect</b>	<b>13</b>
<b>11</b>	<b>Интерфейсы, протоколы и ABC</b>	<b>14</b>
11.1	Гусиная типизация . . . . .	15
11.2	Статические протоколы . . . . .	17
<b>12</b>	<b>Итераторы, генераторы и классические сопрограммы</b>	<b>18</b>
<b>13</b>	<b>Замечание о хвостовой рекурсии в Python</b>	<b>23</b>
<b>14</b>	<b>Замечание о пользовательских пакетах</b>	<b>23</b>

<b>15 Инвариантность, ковариантность и контрвариантность</b>	<b>25</b>
15.1 Обзор инвариантности . . . . .	26
15.1.1 Инвариантные типы . . . . .	26
15.1.2 Ковариантные типы . . . . .	26
15.2 Контравариантные типы . . . . .	27
15.2.1 Эвристические правила вариантности . . . . .	27
<b>16 Передача параметров и возвращаемые значения</b>	<b>27</b>
<b>17 Значения по умолчанию изменяемого типа: неудачная мысль</b>	<b>28</b>
<b>18 Сопоставление с последовательностями-образцами</b>	<b>28</b>
<b>19 Правила видимости в функциях</b>	<b>29</b>
<b>20 Функции как объекты и замыкания</b>	<b>30</b>
<b>21 Типизация</b>	<b>31</b>
<b>22 Модули, пакеты и дистрибутивы</b>	<b>32</b>
22.1 Создание отдельных каталогов с кодом для импорта под общим пространством имен	36
<b>23 Некоторые приемы</b>	<b>37</b>
23.1 Вычисления со словарями . . . . .	37
23.2 Удаление дубликатов из последовательности . . . . .	39
23.3 Сортировка списка словарей по общему ключу . . . . .	39
23.4 Отображение имен на последовательность элементов . . . . .	40
<b>24 Строки и текст</b>	<b>40</b>
24.1 Разрезание строк различными разделителями . . . . .	40
<b>25 Профилирование и замеры времени выполнения</b>	<b>41</b>
<b>26 Итераторы и генераторы</b>	<b>42</b>
<b>27 Захват переменных в анонимных функциях</b>	<b>44</b>
<b>28 Передача дополнительного состояния с функциями обратного вызова</b>	<b>44</b>
<b>29 Использование лениво вычисляемых свойств</b>	<b>45</b>
<b>30 Определение более одного конструктора в классе</b>	<b>47</b>
<b>31 Класс загрузчик данных</b>	<b>49</b>
<b>32 Параметрические декораторы</b>	<b>49</b>
<b>33 Пользовательские исключения</b>	<b>50</b>
<b>34 Определение декоратора, принимающего необязательный аргумент</b>	<b>50</b>

<b>35 Параллельное программирование</b>	<b>52</b>
35.1 Пример использования пула потоков . . . . .	52
35.2 Процессы, потоки и GIL в Python . . . . .	54
35.3 Глобальная блокировка интерпретатора . . . . .	55
<b>36 Проверка существования путей в dataclass</b>	<b>55</b>
<b>37 Приемы работы с библиотекой SPyQL</b>	<b>56</b>
<b>38 Приемы работы с библиотекой Pandas</b>	<b>57</b>
38.1 Общие замечания . . . . .	57
38.2 Советы по оптимизации вычислений . . . . .	57
38.3 Рецепты . . . . .	59
38.3.1 Приемы работы с кадрами данных . . . . .	59
38.3.2 Изменение настроек отдельной линии графика на базе кадра данных . . . .	63
38.3.3 Использование регулярных выражений и обращений по имени группы при обработке строк . . . . .	63
<b>Список литературы</b>	<b>64</b>

## 1. Терминология

Любой элемент данных, используемый в программе на Python, является *объектом* [1, стр. 57].

Каждый объект имеет свою:

- идентичность,
- тип (или класс),
- значение.

Например, когда в программе встречается инструкция `a = 42`, интерпретатор создает целочисленный объект со значением 42. Можно рассматривать идентичность объекта как указатель на область памяти, где находится объект, а идентификатор `a` – как имя, которое ссылается на эту область памяти.

*Тип объекта* сам по себе является *объектом*, который называется *классом объекта*. Все объекты в языке Python могут быть отнесены к *объектам первого класса* [1, стр. 61]. Это означает, что все объекты, имеющие идентификатор, можно интерпретировать как *данные*.

Тип `None` используется для представления пустых объектов (т.е. объектов, не имеющих значений). Этот объект возвращается функциями, которые не имеют явно возвращаемого значения. Объект `None` часто используется как значение по умолчанию для необязательных аргументов. Объект `None` не имеет атрибутов и в логическом контексте оценивается как значение `False`.

*Функции, классы и модули* в языке Python являются объектами, которыми можно манипулировать как обычными данными.

*Свободные переменные* – переменные, которые были определены в объемлющих функциях, а используются вложенными функциями [1, стр. 81].

Все функциональные возможности языка, включая присваивание значений переменным, определение функций и классов, импортирование модулей, реализованы в виде инструкций, обладающих равным положением со всеми остальными инструкциями.

## 2. Соглашения по именованию классов, функций и переменных

Шаблон именования функции (P)A/HC/LC

префикс? (P) + действие (A) + высокоуровневый контекст (HC) + низкоуровневый контекст (LC)

## 3. Приемы работы с пакетом Nox

### 3.1. Общий шаблон

Nox <https://nox.thea.codes/en/stable/index.html> – библиотека и утилита командной строки для автоматизации различных процедур в мульти-средах Python – от простого запуска тестов с помощью, например, `pytest`, линтеров или сборщиков Docker-образов и до запуска цепочек выполнения произвольной сложности.

Если говорить о Python-сценариях, то в файле `noxfile.py` описывается только процедура запуска сценария (вызов сценария из оболочки), а не сам сценарий.

Для запуска утилиты `nox` требуется подготовить файл `noxfile.py` и положить его в корень проекта

noxfile.py

```
import nox

nox.needs_version = ">=2019.5.30"
nox.options.default_venv_backend = "conda"

@nox.session(python=False)
def docker(session):
    session.run(
        "sudo", "docker", "build",
        "--build-arg", "USER_ID=1000",
        "--build-arg", "GROUP_ID=1000",
        "--build-arg", "STRATEGY_NAME=fix_bins_ints_in_relax_sol",
        "--build-arg", "PATH_TO_STRATEGIES_DIR=./data/strategies",
        "-t", "tthec-fix_bins_ints_in_relax_sol",
        "."
    )

@nox.session(
    python=["3.8", "3.9", "3.10"], # тесты выполняются для 3-х версий Python
    venv_backend="conda",
    reuse_venv=True,
)
def test(session):
    # conda ставит только PySCIP0pt==4.3.0 с канала conda-forge
    session.conda_install("pyscipopt==4.3.0", channel="conda-forge")
    # --no-deps, чтобы не сломать окружение conda
    session.install("--no-deps", "-r", "requirements.txt")
    session.run(
        "pytest",
        "-v",
        "-k", "solver", # запускает только те тесты, в имени которых есть подстрока 'solver'
        env = { # здесь описываются переменные окружения
            "PYTHONPATH": "./src", # как если бы запускали $ PYTHONPATH=./src pytest
        }
    )
```

)

Теперь для запуска сессии сборки образа нужно просто запустить утилиту с указанием имени сессии

```
$ nox -s docker
```

То есть в файле `noxfile.py` можно описывать любые сессии, которые автоматизируют различные задачи (запуск тестов, сборку Docker-образов и пр.) и доступ к этим сессиям будет, так сказать, с одной точки.

Можно запускать цепочки

```
import nox
import pathlib2

# NOX OPTIONS
nox.needs_version = ">=2022"
nox.options.default_venv_backend = "conda"

# PROJECT PARAMS
STRATEGY_NAME = "fix_bins_ints_in_relax_sol_with_perturbation"
PROBLEM_FILE_NAME = "model_MNPZ_march_no_plecho_no_CDO_only_BRN.mps"

PATH_TO_DATA_DIR = Path().joinpath("data/").absolute()
PATH_TO_PROBLEMS_DIR = PATH_TO_DATA_DIR.joinpath("problems/")
PATH_TO_MAKE_STRATEGY_FILE = Path("./src/strategy_templates/make_strategy_file.py")
PATH_TO_SETTINGS_DIR = PATH_TO_DATA_DIR.joinpath("settings/")
PATH_TO_RELAX_SET_FILE = PATH_TO_SETTINGS_DIR.joinpath("scip_relax.set")
PATH_TO_MILP_SET_FILE = PATH_TO_SETTINGS_DIR.joinpath("scip_milp.set")
PATH_TO_STRATEGIES_DIR_HOST = PATH_TO_DATA_DIR.joinpath("strategies/")
PATH_TO_STRATEGIES_DIR_CONTAINER = "./data/strategies"

DOCKER_MEMORY = 8000 # Mb
DOCKER_MEMORY_SWAP = 8000 # Mb

DEFAULT_INTERPRETER = "3.8"
TARGET_INTERPRETERS = ("3.8", "3.9", "3.10")

# ENVs
env = {"PYTHONPATH": "./src"}

@nox.session(python=False)
def run_app_with_docker(session):
    session.run(
        "sudo", "docker", "build",
        "--build-arg", "USER_ID=1000",
        "--build-arg", "GROUP_ID=1000",
        "--build-arg", "STRATEGY_NAME=fix_bins_ints_in_relax_sol",
        "--build-arg", "PATH_TO_STRATEGIES_DIR=./data/strategies",
        "-t", "tthec-fix_bins_ints_in_relax_sol",
        ".", # контекст
    )
    # вызов следующего вспомогательного сценария
    session.notify("make_strategy_file")

@nox.session(python=DEFAULT_INTERPRETER)
def make_strategy_file(session):
    session.install("pathlib2>=2.3.7")
    session.run(
```

```

    "python", PATH_TO_MAKE_STRATEGY_FILE,
    "--strategy-name", STRATEGY_NAME,
    "--path-to-relax-set-file", PATH_TO_RELAX_SET_FILE,
    "--path-to-milp-set-file", PATH_TO_MILP_SET_FILE,
    "--path-to-test-problem-file", PATH_TO_PROBLEMS_DIR.joinpath(PROBLEM_FILE_NAME),
    "--path-to-strategies-dir", PATH_TO_STRATEGIES_DIR_HOST,
    env=env,
)
# вызов следующего вспомогательного сценария
session.notify("docker_run")

@nox.session(python=False)
def docker_run(session):
    session.run(
        "sudo", "docker", "run",
        "--rm",
        "-v", f"{PATH_TO_DATA_DIR}:/data",
        "-m", f"{DOCKER_MEMORY}m",
        "--memory-swap", f"{DOCKER_MEMORY_SWAP}m",
        f"tthea-{STRATEGY_NAME}"
    )
...

```

Запуск цепочки

```
$ nox -s run_app_with_docker
```

Чтобы захватить вывод команды оболочки, нужно у метода `run` выставить `silent=True`

```

@nox.session(python=False)
def f(session):
    USER_ID = session.run("bash", "-c", "echo $(id -u)", silent=True)
    GROUP_ID = session.run("bash", "-c", "echo $(id -g)", silent=True)
    ...

```

### 3.2. Запуск тестов в мультисредах Python

Для того чтобы запустить тесты сразу для нескольких версий интерпретатора достаточно просто передать список нужных версий декоратору `@nox.session(python=["3.8", "3.9", ...])`

```

@nox.session(
    python=["3.8", "3.9"],
    venv_backend="conda",
    reuse_venv=False,
)
def test(session):
    session.conda_install("pysciptopt==4.3.0", channel="conda-forge")
    session.install("--no-deps", "-r", "requirements.txt")

    session.run(
        "pytest",
        "-v",
        env={"PYTHONPATH": "./src"}
    )

```

### 3.3. Nox как утилита командной строки

<https://nox.thea.codes/en/stable/usage.html>

Вывести список сессий

```
$ nox -l
* test-3.8
* test-3.9
* test-3.10
```

Запустить только тестирование для Python 3.10

```
$ nox --session test-3.10 # или с коротким флагом '-s'
```

После запуска сессий по умолчанию в текущей директории создается скрытая директория `.nox`, в которую записывается сводка по запускам. Чтобы создать эту сводку в указанном пользователем месте, нужно использовать флаг `--envdir`

```
$ nox --envdir /tmp/envs
```

Утилите `nox` можно передать позиционные аргументы

```
...
@nox.session
def test(session):
    if session.posargs:
        test_files = session.posargs
    else:
        test_files = ["test_a.py", "test_b.py"]

    session.run("pytest", *test_files)

$ nox -- test_c.py
```

Еще один важный момент заключается в том, что если требуется управлять цепочкой выполнения по условию, то можно пробросить значения аргументов командной строки через аргумент `posargs` функции `notify`

```
@nox.session(python=DEFAULT_INTERPRETER)
def fake1(session):
    args: t.List[str] = session.posargs

    if args and ("docker" in args):
        use_docker = True
    else:
        use_docker = False

    session.notify("fake2", posargs=[use_docker]) # пробрасываем значение аргумента

@nox.session(python=DEFAULT_INTERPRETER)
def fake2(session):
    print(session.posargs)
```

Теперь можно вызвать сессию так

```
$ nox -s fake1 -- docker
```

**ВАЖНО:** Не обязательно пробрасывать значения аргумента через все элементы цепочки. Значение аргумента, переданное в «головной» элемент цепочки, можно прочитать в любом другом элементе как `session.posargs`

## 4. Приемы работы с pip

С одной стороны в виртуальное окружение conda можно устанавливать пакеты с помощью менеджера pip, но все-таки лучше с pip использовать флаг `--no-deps`. Это поможет не сломать окружение conda. В противном случае пакеты устанавливаемые с помощью pip могут получить несовместимые версии с пакетами уже установленными в окружении conda <https://nox.thea.codes/en/stable/tutorial.html>

```
$ pip install --no-deps -r req.txt
```

## 5. Аннотация типов

Аннотации обрабатываются интерпретатором *на этапе импорта*, тогда же, когда значения по умолчанию [4, стр. 508].

Расширяющееся использование аннотации типов подняло две проблемы [4]:

1. импорт модулей занимает больше времени и потребляет больше памяти, если используется много аннотаций,
2. ссылка на еще не определенные типы требует использования строк, а не фактических типов.

Хранение аннотации в виде строк иногда необходимо из-за проблемы «опережающей ссылки»: когда аннотация типа должна сослаться на класс, определенный в том же модуле ниже. Поскольку объект класса не определен, пока Python не закончит вычисление тела класса, в аннотациях типов необходимо указывать *имя класса в виде строки* [4, стр. 508]. Пример

```
class Rectangle:
    # ...
    def stretch(self, factor: float) -> "Rectangle":
        return Rectangle(width=self.width * factor)
```

Запись еще не определенных типов в виде строки в аннотациях типов – стандартная и обязательная практика в версии Python 3.10.

В PEP 563 «Postponed Evaluation of Annotations» сделал необязательной запись аннотации в виде строк и уменьшил время, необходимое для обработки аннотаций типов во время выполнения. В этом PEP предлагается изменить аннотации функций и переменных, так чтобы они больше не вычислялись в момент определения функции. Вместо этого они сохраняются в аннотациях в строковой форме.

Начиная с версии Python 3.7 именно так и обрабатываются аннотации в любом модуле, который начинается следующим предложением

```
from __future__ import annotations
```

Вызов `typing.get_type_hints` дает нам реальные типы – даже в тех случаях, когда в исходной аннотации тип был записан в виде заковыченной строки. Это рекомендуемый способ читать аннотации типов во время выполнения.

Значением функции всегда является конкретный объект, поэтому в аннотации для возвращаемого значения должен быть указан конкретный тип [4, стр. 277]. В разделе документации, посвященном `typing.List` говорится, что обобщенная версия `list` полезна для аннотирования типов возвращаемых значений, а для аннотирования аргументов лучше использовать абстрактные коллекции, например `Sequence` или `Iterable`.



То есть функция *всегда* принимает объекты «широких» типов, то есть подтипов или абстрактных типов (чтобы можно было, скажем вместо `dict` передать `OrderedDict` или `UserDict`), а возвращает объекты «узких» конкретных типов.

Начиная с версии Python 3.9 большинство ABC из модуля `collections.abc` и другие конкретные классы из модуля `collections`, а также встроенные коллекции поддерживают нотацию аннотации обобщенных классов вида `collections.deque[str]`. Соответствующие коллекции из модуля `typing` нужны только для поддержки кода, написанного для версии Python 3.8 или более ранней [4, стр. 278].

То есть

```
from collections.abc import Iterable

def f(seq: Iterable[str]) -> list[str]:
    return sorted(seq, key=len)
```

В документе PEP 613 «Explicit Type Aliases» введен специальный тип `TypeAlias`, идея которого в том, чтобы сделать создаваемые псевдонимы типов хорошо видимыми и упростить для них проверку типов

```
from typing import TypeAlias

FromTo: TypeAlias = tuple[str, str]
```

Тип `Sequence` можно использовать тогда, когда по логике нужно знать длину последовательности. Как и `Sequence`, объект `Iterable` лучше использовать в качестве *типа параметра*. В качестве типа возвращаемого значения он не позволяет составить представление о том, что же будет на выходе. Функция должна более ясно говорить о том, какой конкретный тип она возвращает.

Если требуется указать *верхнюю границу допустимых типов* (параметр-тип может быть `Hashable` или любым его *подтипом*), то можно сделать так

```
from collections.abc import Iterable, Hashable
from typing import TypeVar

HashableT = TypeVar("T", bound=Hashable)

def mode(data: Iterable[HashableT]) -> HashableT:
    pairs = Counter(data).most_common(1)
    if len(pairs) == 0:
        raise ValueError("no mode for empty data")

    return pairs[0][0]
```

А ограничить `TypeVar` можно так

```
NumberT = TypeVar("NumberT", float, Decimal, Fraction)
```

С помощью `Iterator` можно зааннотировать генераторное выражение, например

```
from collections.abc import Iterator

# генераторное выражение
series: Iterator[tuple[int, str]] = (len(s), s) for s in ...)
```

Если нужна аннотация типа для функций с *гибкой сигнатурой*, нужно заменить весь список параметров многоточием [4, стр. 289]

```
Callable[..., ReturnType]
```

Если даны тип T1 и подтип T2, то T2 *совместим* с типом T1 (подстановка Лисков) [4, стр. 268].

## 5.1. Вариантность в типах Callable

Пусть есть функция высшего порядка, которая принимает два вызываемых объекта

```
from collections.abc import Callable

def update(
    probe: Callable[[], float], # "-" у аргумента 'probe' КОНТРАВариантная позиция
    display: Callable[[float], None], # "-" у аргумента 'display' КОНТРАВариантная позиция
) -> None: # "+" у выхода функции КОВАРИантная позиция
    ...
```

Чтобы сохранить правильную вариантность по отношению к аргументам функции `update` в `Callable` вариантность *инвертируется*, то есть `Callable[[], float]`.

Например, аргументу `probe` можно вместо объекта типа `Callable[[], float]` передать объект типа `Callable[[], int]`, так как по отношению к `Callable float` стоит в контравариантной позиции (-) и потому допускает замену на свой *подтип*.

Напротив в `Callable[[float], None]` `float` стоит в ковариантной позиции (+) и допускает замену на свой *супертип*.

В общем случае аргументы функции стоят в контравариантной позиции (-), так как функция ожидает объекты более общих типов (подтипов или абстрактных типов), а выход функций стоит в ковариантной позиции, чтобы можно было возвращать более конкретный тип.

## 5.2. Аннотирование чисто позиционных и вариадических параметров

Пример

```
import typing as t

def tag(
    name: str,
    /, # все что слева от слеша это чисто позиционные аргументы
    *content: str,
    class_: t.Optional[str] = None,
    **attrs: str,
) -> None:
    ...
```

Внутри функции `content` будет иметь тип `tuple[str, ...]`, а `attrs` – тип `dict[str, str]`. Если бы аннотация имела вид `**attrs: float`, то аргумент `attr` имел бы тип `dict[str, float]`.

## 6. Приемы работы с pytest

### 6.1. Особенности импорта

Пусковой сценарий проекта обычно располагается в директории `./src`. В этом случае сканирование окружения на предмет поиска пользовательских пакетов и модулей начинается с той директории, в которой *лежит* этот пусковой сценарий, то есть с директории `./src`

```

project_root/
  src/
    common/  # пакет
      __init__.py
      logger.py
      exceptoins.py
    units/  # пакет
      __init__.py
      fix_vars.py
      base_unit.py
      solver.py
      strategy_manager.py
      run.py

```

Тогда импорт в самом сценарии может выглядеть так

run.py

```

# путь отсчитывается от той директории, в которой лежит run.py
from common.logger import make_logger
from strategy_manager import StrategyManager
...

```

В модулях пути тоже отсчитываются от той директории, в которой *лежит* пусковой сценарий

solver.py

```

from common.logger import make_logger
from units.base_unit import Unit
...

```

В тестах можно указывать пути от той же директории `./src`, то есть

test\_solver.py

```

import pytest
# путь отсчитывается от директории ./src
from common.exceptions import HiGHS
from units.solver import Solver

@pytest.mark.unit
def test_solver_unit_highs_with_unsupported_solver_name():
    ...

```

Но запускать тесты нужно будет так

```

# требуется включить директорию ./src в список путей поиска
$ PYTHONPATH=./src pytest -v
$ PYTHONPATH=./src pytest -v --cov=. --cov-report=html

```

## 7. Ошибка `ValueError: generator already executing` в многопоточных приложениях с генераторами

Ошибка «`ValueError: generator already executing`» возникает когда потоки пытаются одновременно обратиться к генератору. Можно просто добавить блокировку на вызов следующего метода

```

import threading

```

```

class ThreadSafeGenerator:
    def __init__(self, gen):
        self.gen = gen
        self.lock = threading.Lock()

    def __iter__(self):
        return self._next()

    def _next(self):
        with self.lock:
            return self.gen

```

Затем нужно просто «пропустить» генератор через этот класс и пользоваться генератором как раньше

```

conss = ThreadSafeGenerator(self._make_generator(model))

for cons in conss:
    # что-то делаем

```

## 8. Раскраска ячеек в Jupyterlab

Чтобы покрасить ячейку в заданный нужно добавить следующие функции в блокнот

```

from IPython.core.magic import register_cell_magic
from IPython.display import HTML, display

def set_background(color):
    script = (
        "var cell = this.closest('.jp-CodeCell');"
        "var editor = cell.querySelector('.jp-Editor');"
        "editor.style.background='{color}';"
        "this.parentNode.removeChild(this)"
    ).format(color)

    display(HTML('<img src onerror="{color}" style="display:none">'.format(script)))

@register_cell_magic
def background(color, cell):
    set_background(color)
    return eval(cell)

```

Затем нужно просто запустить ячейку с магической командой `%%background _color_`

```

%%background red
# здесь какой-то код

```

## 9. Разреженные матрицы LIL и CSC

Для представления больших матриц (тысячи строк на тысячи столбцов), которые ограничиваются небольшим числом операций, удобно использовать `scipy.sparse.lil_matrix`, то есть матрицы в формате списка списков разреженных матриц. Такие матрицы эффективны с точки зрения заполнения. Пример

```

from scipy.sparse import lil_matrix, csc_matrix

_matrix = lil_matrix((n_conss, n_vars), dtype=np.int8)

conss_gen = ((cons_name, cons) for cons_name, cons in model.all_conss.items())
for cons_name, cons in tqdm(conss_gen, total=n_conss, desc="Building sparse matrix"):
    cons_idx = cons_name_to_cons_idx.get(cons_name)
    _var_names_context = list(cons.keys())
    var_idxes = var_name_to_var_idx.loc[_var_names_context].values.tolist()
    _matrix[cons_idx, var_idxes] = 1

```

Но с точки зрения доступа по столбцам эффективнее использовать разреженные матрицы в формате сжатого разреженного столбца – так как LIL-матрицы строко-ориентированные

```
_matrix.tocsc()
```

Теперь можно эффективно получить множество индексов столбцов, содержащих ненулевые элементы

```

_set_context_var_idxes: t.Set[int] = set(
    self._matrix[self._matrix.getcol(var_idx).nonzero()[0], :].nonzero()[1]
)

```

## 10. Метод `__repr__` и модуль `inspect`

Метод `__repr__` предназначен для вывода полезной информации на шаге отладки, а метод `__str__` – вывода полезной информации для пользователей. При этом принято, чтобы метод `__repr__` возвращал такую строку, обернув которую функцией `eval()`, можно было получить экземпляр класса.

Для того чтобы специальный метод `__repr__` мог аккуратно выводить сигнатуру класса удобно воспользоваться модулем `inspect`

```

import inspect

class MyClass:
    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age

    def __repr__(self):
        _args: t.List[str] = []
        # аргументы класса: name и age
        _class_args = tuple(inspect.signature(type(self)).parameters.keys())
        _obj_attrs = self.__dict__

        for key, value in _obj_attrs.items():
            if key in _class_args:
                # обязательно использовать сырое форматирование !r
                _args.append(f"{key}={value!r}")
        args = ", ".join(_args)

        return f"{type(self).__name__}({args})

```

Чтобы получить имеющиеся функции в модуле (пусть называется `promotions`) можно сделать так [4, стр. 343]

```
import promotions
import inspect

promos = [func for _, func in inspect.getmembers(promotions, inspect.isfunction)]
```

## 11. Интерфейсы, протоколы и ABC

Объектно-ориентированное программирование – это об интерфейсах. Хотите понять, что делает тип в Python, – узнайте, какие методы он предоставляет.

Начиная с Python 3.8 существует 4 способа определения и использования интерфейсов [4, стр. 413]:

- *Утиная типизация*: подход к типизации, по умолчанию принятый в Python с момента его возникновения. Утиная типизация это игнорирование фактического типа объекта и акцент на том, чтобы объект реализовывал методы с именами, сигнатурами и семантикой, требуемыми для конкретного применения. В Python это сводится в основном к тому, чтобы избегать использование функции `isinstance()` для проверки типа объекта [4, стр. 422]. Очень часто бывает, что во время выполнения утиная типизация – лучший подход к проверке типа: вместо того чтобы вызывать `isinstance` или `hasattr`, просто попробуйте выполнить над объектом нужную операцию и обработайте исключения [4, стр. 446]. Яркий пример *утиной типизации*

```
class Vector:
    ...
    def __mul__(self, scalar):
        try:
            factor = float(scalar)
        except TypeError:
            return NotImplemented

        return Vector(n * factor for n in self)

    def __rmul__(self, scalar):
        return self * scalar
```

В этом примере метод `__mul__` не проверяет тип `scalar` явно, а пытается преобразовать его в тип `float` и возвращает `NotImplemented`, если эта попытка завершается неудачно. Метод `__rmul__` просто вычисляет произведение `self * scalar`, делегируя всю работу методу `__mul__`.

- *Гусиная типизация*: этот подход поддерживается абстрактными базовыми классами; в его основе лежит сравнение объектов с ABC, выполняемое на этапе выполнения. Гусиная типизация означает следующее: вызов `isinstance(obj, cls)` теперь считается приемлемым, но при условии, что `cls` – абстрактный базовый класс, то есть метаклассом `cls` является `abc.ABCMeta`. Пусть имеется класс `FrenchDeck`, и требуется проверить его тип следующим образом: `issubclass(FrenchDeck, Sequence)`. Для этого можно сделать его виртуальным подклассом ABC `Sequence`

```
from collections.abc import Sequence
Sequence.register(FrenchDeck)
```

Пример *гусиной типизации*

```
class Vector:
    ...
    def __matmul__(self, other):
        if (isinstance(other, abc.Sized) and isinstance(other, abc.Iterable)):
            if len(self) == len(other):
                return sum(a * b for a, b in zip(self, other))
            else:
                raise ValueError("@ requires vectors of equal length")
        else:
            return NotImplemented

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

Здесь оба операнда должны реализовывать методы `__len__` (`abc.Sized`) и `__iter__` (`abc.Iterable`).

- *Статическая типизация*: поддерживается с помощью модуля `typing`.
- *Статическая утиная типизация*: поддерживается подклассами класса `typing.Protocol`, и также проверяется внешними программами.

Реализации метода `__getitem__()` достаточно для [4, стр. 414]:

- получения элементов по индексу,
- поддержки итерирования,
- поддержки оператора `in`.

Специальный метод `__getitem__()` – ключ к протоколу последовательности. PEP 544 позволяет создавать подклассы `typing.Protocol` с целью определить, какие методы должен реализовывать (или унаследовать) класс, чтобы не раздражать программу статической проверки типов.

Между динамическим и статическим протоколами есть два основных различия [4, стр. 415]:

- объект может реализовать только часть *динамического* протокола и при этом быть полезным; но чтобы удовлетворить *статическому* протоколу, объект должен предоставить *все методы*, объявленные в классе протокола, даже если некоторые из них программе не нужны,
- статические протоколы можно проверить с помощью программ статической проверки типов, динамические – нельзя.

Помимо статических протоколов, Python предлагает еще один способ программно определить явный интерфейс: абстрактный базовый класс.

Правильно написанный подкласс абстрактного базового класса `Sequence` должен реализовывать методы `__getitem__()` и `__len__()` (унаследованный от `Sized`) [4, стр. 416].

Даже если метода `__iter__()` у объекта нет, но есть метод `__getitem__()`, Python будет считать *объект итерируемым*. Поскольку если Python находит метод `__getitem__()` и не имеет ничего лучше, то он пытается обходить объект, вызывая этот метод с целочисленными индексами, начиная с 0 [4, стр. 416].

Короче говоря, осознавая важность структур данных, обладающих свойствами последовательностей, Python ухитряется заставить *итерирование* и *оператор in* работать, вызывая метод `__getitem__()` в случае, когда методы `__iter__()` и `__contains__()` отсутствуют.

### 11.1. Гусиная типизация

В Python используются абстрактные базовые классы, чтобы определить интерфейсы для явной проверки типов *во время выполнения*. Они также поддерживаются программами статической проверки типов.

Гусиная типизация – это *подход к проверке типов* во время выполнения, основанный на применении абстрактных базовых классов.

Однако и при использовании абстрактных базовых классов нужно помнить, что злоупотребление функцией `isinstance` может быть признаком «дурно пахнущего кода» – плохо спроектированной объектно-ориентированной программы.

Обычно *НЕ* должно быть цепочек предложений `if/elif/else`, в которых с помощью `isinstance` определяется тип объекта и в зависимости от него выполняются те или иные действия; для этой цели следует использовать *полиморфизм*, то есть проектировать классы так, чтобы интерпретатор сам вызывал правильные методы, а не «зашивать» логику диспетчеризации в блоки `if/elif/else` [4, стр. 425].

Важнейшая характеристика *гусиной типизации* – возможность регистрировать класс как *виртуальный подкласс* абстрактного базового класса, даже без наследования. При этом мы обещаем, что класс честно реализует интерфейс, определенный в абстрактном базовом классе, а Python верит нам на слово, *не производя проверку*. Если мы со временем будем наказаны исключением во время выполнения.

Это делается путем вызова метода `register` абстрактного базового класса. В результате зарегистрированный класс становится виртуальным подклассом ABC и распознается в качестве такового функцией `issubclass`, *однако не наследует ни методы, ни атрибуты ABC*.

Виртуальные подклассы *не наследуют ABC*, для которых зарегистрированы. Их согласованность с интерфейсом ABC *не проверяется никогда*, даже в момент создания экземпляра. Кроме того, программы статической проверки типов не могут обрабатывать виртуальные классы.

И наследование ABC, и регистрация в качестве виртуального подкласса ABC – явные способы сделать так, чтобы проходили проверки с помощью функции `issubclass`, а равно и проверки с помощью функции `isinstance`, которая опирается на `issubclass`. Но некоторые ABC поддерживают также структурную типизацию [4, стр. 440].

ABC чаще всего используются в сочетании с *номинальной типизацией*. Когда класс `Sub` явно наследует `AnABC` или регистрируется в качестве виртуального подкласса `AnABC`, имя `AnABC` связывается с классом `Sub`, именно поэтому во время выполнения вызов `issubclass(AnABC, Sub)` возвращает `True`.

Напротив, *структурная типизация* подразумевает изучение структуры открытого интерфейса объекта с целью определить его тип: *объект совместим с типом, если он реализует все методы, определенные типом* (то есть другими словами, является *подтипом*). Динамическая и статическая утиные типизации – два подхода к *структурной типизации* [4, стр. 441].

Класс может быть распознан как *виртуальный подкласс ABC даже без регистрации*

```
class Struggle:
    def __len__(self): return 42

from collections.abc import Sized

isinstance(Struggle(), Sized) # True
issubclass(Struggle, Sized)  # True
```

Функция `issubclass` (а значит, и `isinstance`) считает класс `Struggle` подклассом `abc.Sized`, потому что `abc.Sized` реализует специальный метод класса `__subclasshook__`.

Метод `__subclasshook__` в классе `Sized` проверяет, имеет ли переданный в аргументе класс атрибут с именем `__len__`. Если да, то класс считается виртуальным подклассом `Sized` [4, стр. 441].



Вот так метод `__subclasshook__` позволяет ABC поддержать структурную типизацию. Несмотря на наличие формального определения интерфейса в ABC и скрупулезных проверок, осуществляемых функцией `isinstance`, в определенных контекстах вполне можно использовать никак не связанный с ABC класс просто потому, что в нем реализован определенный метод (или потому, что он постарался убедить `__subclasshook__`, что за него можно поручиться) [4, стр. 442].

Лично я не готов поверить, что класс с именем `Spam`, который реализует или наследует методы `load`, `pick`, `inspect` и `loaded`, гарантировано ведет себя как `Tombola`. Пусть уж лучше программист явно подтвердит это, сделав `Spam` подклассом `Tombola` или хотя бы зарегистрировав его: `Tombola.register(Spam)`.

## 11.2. Статические протоколы

Пример

```
from typing import TypeVar, Protocol

T = TypeVar("T")

class Repeatable(Protocol):
    def __mul__(self: T, repeat_count: int) -> T: ...

RT = TypeVar("RT", bound=Repeatable)

def double(x: RT) -> RT:
    return x * 2
```

Параметр `self` обычно не аннотируется – предполагается, что его тип – сам класс. Здесь мы используем `T`, чтобы тип результата гарантированно совпадал с типом `self`.

Номинальный тип фактического аргумента `x`, переданного `double`, не играет роли, коль скоро он умеет квакать – т.е. реализует метод `__mul__`.

На карте типизации `typing.Protocol` располагается в области статической проверки. Но при определении подкласса `typing.Protocol` мы можем использовать декоратор `@runtime_checkable`, чтобы протокол поддерживал проверки с помощью функции `isinstance/issubclass` во время выполнения.

Исходный код протокола `typing.SupportsComplex`

```
@runtime_checkable
class SupportComplex(Protocol):
    __slots__ = ()

    @abstractmethod
    def __complex__(self) -> complex:
        pass
```

В процессе *статической проверки типов* объект будет считаться *совместимым* с протоколом `SupportsComplex`, если он реализует метод `__complex__`, принимающий только аргумент `self` и возвращающий `complex`.

Благодаря применению декоратора класса `@runtime_checkable` к `SupportsComplex` этот протокол теперь можно использовать в сочетании с функцией `isinstance`.

Если требуется проверить, верно ли, что объект `x` имеет тип `complex` или `SupportsComplex`, то можем предоставить кортеж типов в качестве второго аргумента `isinstance`

```
isinstance(c, (complex, SupportsComplex))
```

Встроенный тип `complex`, а также типы NumPy `complex64` и `complex128` зарегистрированы как виртуальные подклассы `numbers.Complex` и потому проходят проверку с помощью `isinstance`.

Очень часто бывает, что во время выполнения утиная типизация – лучший подход к проверке типа: вместо того чтобы вызывать `isinstance` или `hasattr`, просто попробуйте выполнить над объектом нужную операцию и обработайте исключения.

Пример. Дан объект `o`, который требуется использовать как комплексное число, – можно подойти к решению следующим образом:

```
if isinstance(o, (complex, SupportsComplex)):
    # to something
else:
    raise TypeError("o must be convertible to complex")
```

Гусиная типизация подразумевала бы использование ABC `numbers.Complex`

```
if isinstance(o, numbers.Complex):
    # do something
else:
    raise TypeError("...")
```

Л. Ромальо предпочитает использовать утиную типизацию и принцип EAFP (it's easier to ask for forgiveness than permission) – проще попросить прощения, чем испрашивать разрешения

```
try:
    c = complex(o)
except TypeError as exc:
    raise TypeError("...") from exc
```

Общая проблема: функции `isinstance`/`issubclass` смотрят только на наличие или отсутствие методов, не проверяя их сигнатуры, а уж тем более аннотации типов [4, стр. 448].

В классе `Vector2d` можно разместить метод следующего вида

```
from __future__ import annotations # NB

class Vector2d:
    ...

    @classmethod
    def fromcomplex(cls, datnum: SupportsComplex) -> Vector2d: # NB
        c = complex(datnum)
        return cls(c.real, c.imag)
```

Тип возвращаемого `fromcomplex` значения может быть `Vector2d`, если в начале модуля находится предложение `from __future__ import annotations`. Этот импорт приводит к тому, что аннотации типов сохраняются в виде строк, а не вычисляются на этапе импорта, когда интерпретатор обрабатывает определения функций. Если бы `annotations` не импортировался, то ссылка `Vector2d` была бы в этой точке недопустима (класс еще не полностью определен) и ее следовало бы записать в виде строки `'Vector2d'`, как если бы это была опережающая ссылка [4, стр. 449].

## 12. Итераторы, генераторы и классические сопрограммы

Если просматривается набор данных, не помещающийся целиком в память, то нужен способ выполнять ее *лениво*, то есть по одному элементу и по запросу. Именно это и делает Итератор.

Любая стандартная коллекция в Python является *итерируемым объектом*, то есть предоставляет *итератор*, который используется для поддержки следующих операций [4, стр. 556]:

- циклов `for`,
- списковых, словарных и множественных включений,
- распаковки операций присваивания,
- конструирования экземпляров коллекций.

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

Встроенная функция `iter` выполняет следующие действия [4, стр. 559]:

- Смотрит, реализует ли объект метод `__iter__`, и, если да, вызывает его, чтобы получить итератор,
- Если метод `__iter__` не реализован, но реализован метод `__getitem__`, то Python создает итератор, который пытается извлекать элементы по порядку, начиная с индекса 0,
- Если и это не получается, то возбуждается исключение – обычно с сообщением «'C' object is not iterable»

Именно поэтому любая последовательность в Python является итерируемой: все они реализуют метод `__getitem__`. На самом деле стандартные последовательности реализуют и метод `__iter__`, а специальная обработка метода `__getitem__` оставлена *только ради обратной совместимости* [4, стр. 559].

Пример

```
class Spam:
    def __getitem__(self, i):
        print(">", i)
        raise IndexError()

spam = Spam()
iter(spam) # <iterator object ...>
list(spam)
# -> 0
# []
from collections import abc
isinstance(spam, abc.Iterable) # False
```

Если класс предоставляет метод `__getitem__`, то встроенная функция `iter()` принимает *экземпляр* этого класса в качестве *итерируемого объекта* и строит по нему *итератор*. Механизм итерирования Python будет вызывать `__getitem__` с индексами, начинающимися с 0, и воспринимать исключение `IndexError` как сигнал о том, что элементы кончились.

Итерируемый объект – любой объект, от которого встроенная функция `iter` может получить *итератор*. Объекты, которые реализуют метод `__iter__`, возвращающий *итератор*, являются итерируемыми. *Последовательности* всегда *итерируемы*, поскольку это объекты, реализующие метод `__getitem__`, который принимает индексы, начинающиеся с нуля [4, стр. 561].

Итераторы в Python следует считать не типом, а *протоколом*. Лучше не проверять тип итератора, а использовать функцию `hasattr` для проверки наличия атрибутов «`__iter__`» и «`__next__`» [4, стр. 563].

Объект считается *итерируемым*, если реализует метод `__iter__` [4, стр. 559].

Лучший способ узнать, является ли объект итератором, – вызвать функцию `isinstance(x, collections.abc.Iterator)`. Благодаря методу `Iterator.__subclasshook__` эта

проверка работает даже тогда, когда класс не является ни настоящим, ни виртуальным подклассом `Iterator`.

Любая функция в Python, в теле которой встречается ключевое слово `yield`, называется *генераторной функцией* – при вызове она возвращает *объект-генератор*. Иными словами, генераторная функция – фабрика генераторов.

*Объекты-генераторы* реализуют интерфейс `Iterator`, поэтому являются также *итерируемыми объектами* [4, стр. 569].

Генераторная функция не возбуждает исключение `StopIteration`, когда значений не остается она просто выходит. Другими словами, генераторная функция просто возвращает управление, а исключение `StopIteration` возбуждает объект-генератор (в полном соответствии с протоколом `Iterator`).

Вызов генераторной функции возвращает генератор. А генератор отдает значения.

*Генераторное выражение* возвращает *объект-генератор*. Пример

```
RE_WORD = re.compile(r"\w+")

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

    def __repr__(self):
        return f"... "

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

Конечный результат не изменился: код, вызывающий `__iter__`, получает объект-генератор.

Генераторные выражения – не более чем синтаксический сахар: их всегда можно заменить генераторными функциями, но иногда выражение удобнее.

Если генераторное выражение занимает больших двух строк, то рекомендуется использовать генераторную функцию.

В официальной документации и кодовой базе Python терминология, относящаяся к итераторам и генераторам, противоречива и постоянно изменяется. Рамальо предлагает следующие определения [4, стр. 574]:

- *Итератор* – общий термин, обозначающий любой объект, который реализует метод `__next__`. Итераторы предназначены для порождения данных, потребляемых клиентским кодом, т.е. кодом, который управляет итератором посредством цикла `for` или другой итеративной конструкции либо путем явного вызова функции `next(it)` для итератора. На практике большинство итераторов, встречающихся в Python, являются генераторами.
- *Генератор* – итератор, построенный компилятором Python. Для создания генератора мы не реализуем метод `__next__`. Вместо этого используется ключевое слово `yield`, в результате чего получается *генераторная функция*, то есть фабрика *объектов-генераторов*. Генераторное выражение – это еще один способ построить объект-генератор. Объекты-генераторы предоставляют метод `__next__`, то есть являются генераторами.

---

### Замечание

В глоссарии Python недавно появился термин *генераторный итератор*, так называют объекты, построенные генераторными функциями, тогда как в статье о генераторных выражениях говорится, что они возвращают «итератор». Но в обоих случаях, если верить Python, возвращаются *объекты-генераторы*.

---

Выражение `yield from` позволяет генератору делегировать работу субгенератору.

Пример

```
def tree(cls):
    yield cls.__name__, 0
    yield from sub_tree(cls, 1)

def sub_tree(cls, level):
    for sub_cls in cls.__subclasses__():
        yield sub_cls.__name__, level
        yield from sub_tree(sub_cls, level + 1)

def display(cls):
    for cls_name, level in tree(cls):
        indent = " " * 4 * level
        print(f"{indent}{cls_name}")

if __name__ == "__main__":
    display(BaseException)
```

В любом нормальном пособии по рекурсии подчеркивается важность базы, позволяющей избежать бесконечной рекурсии. Неявное условие есть в цикле `for`: если `cls.__subclasses__()` возвращает пустой список, то тело цикла не выполняется, так что рекурсивного вызова не будет. Базовым является случай, когда класс `cls` не имеет подклассов. Тогда `sub_tree` ничего не отдает, а просто возвращает управление [4, стр. 595].

Функции, принимающие итерируемые объекты в качестве аргументов, можно аннотировать с помощью `collections.abc.Iterable` (или `typing.Iterable`, если нужно обязательно поддерживать версию Python 3.8 или более ранние)

```
from collections.abc import Iterable

FromTo = tuple[str, str]

def zip_replace(text: str, changes: Iterable[FromTo]) -> str:
    for from_, to in changes:
        text = text.replace(from_, to)
    return text
```

Начиная с версии Python 3.10 `FromTo` должна иметь аннотацию типа `typing.TypeAlias`, чтобы прояснить назначение этой строки

```
FromTo: TypeAlias = tuple[str, str]
```

Типы `Iterator` встречаются не так часто, как `Iterable`

```
from collections.abc import Iterator

def fibonacci() -> Iterator[int]:
    a, b = 0, 1
    while True:
```

```
yield a
a, b = b, a + b
```

Тип `Iterator` используется для генераторов, оформленных в виде функции с `yield`, а также итераторов, написанных «вручную» как классы с методом `__next__`. Существует также тип `collections.abc.Generator` (и соответствующий объявленный **нерекомендуемым тип** `typing.Generator`), который можно использовать для аннотирования объекто-генераторов, но он слишком многословен для генераторов в роли итераторов [4, стр. 597].

Пример

```
# Генераторное выражение, отдающее строки
long_kw: Iterator[str] = (k for k in kwlist if len(k) >= 4)
```

`abc.Iterator[str]` совместим с `abc.Generator[str, None, None]`. `Iterator[T]` – это краткое обозначение `Generator[T, None, None]`. Обе аннотации означают «генератор, который отдает объект типа `T`, но *не потребляет* и *не возвращает* значений». Генераторы, способные потреблять и возвращать значения, называются сопрограммами.

Генераторы чаще используются в роли простых итераторов.

Единственное разумное действие с генератором, используемым в роли итератора, – вызов метода `next(it)` прямо или косвенно посредством цикла `for` и других форм итерирования.

Есть еще тип `typing.Coroutine` (объявленный **нерекомендуемым**) и `collections.abc.Coroutine` (обобщенный начиная с версии Python 3.9) предназначены для аннотирования *только платформенных*, но не классических сопрограмм.

Если требуется использовать аннотации типов в сочетании с классическими сопрограммами, то придется испытывать замешательство, аннотируя их как `Generator[YieldType, SendType, ReturnType]`.

Из презентаций Бизли [4, стр. 599]:

- генераторы порождают данные для итерирования,
- сопрограммы являются потребителями данных,
- сопрограммы не имеют никакого отношения к итерированию.

Пример сопрограммы

```
def averager() -> Generator[float, float, None]:
    total = 0.0
    count = 0
    average = 0.0
    while True:
        term = yield average
        total += term
        count += 1
        average = total / count

coro_avg = averager()
next(coro_avg) # 0.0
coro_avg.send(10) # 10
coro_avg.send(30) # 20
```

Здесь не нужны ни атрибуты экземпляра, ни замыкания. Потому-то сопрограммы и являются привлекательной альтернативой обратным вызовам при асинхронном программировании – они сохраняют локальное состояние между активациями.

В этом тесте вызов `next(coro_avg)` заставляет сопрограмму дойти до `yield`, при этом будет отдано начальное значение `average`. Запустить сопрограмму можно также, вызвав `coro_avg.send(None)`,

– именно так и поступает встроенная функция `next()`. Но отправить какое-то значение, кроме `None`, нельзя, потому что сопрограмма может принимать отправленные значения, только когда приостановлена в точке `yield`. Вызов `next()` или `.send(None)`, чтобы продвинуть выполнение к первому предложению `yield`, называется «инициализацией сопрограммы».

## 13. Замечание о хвостовой рекурсии в Python

В Python нет РТС (Proper Tail Calls, *чисто хвостовой рекурсии*), поэтому от написания хвостово-рекурсивных функций мы никакого наваара не получим. Хвостовая рекурсия имеет место, когда функция возвращает результат некоторого вызова функции – самой себя или какой-то другой [4, стр. 643].

Проблема в том, что даже в языках, где РТС реализована, дивиденды получают не все рекурсивные функции, а только специально написанные, так чтобы имел место хвостовой вызов. Если РТС поддерживается языком, то *интерпретатор, видя хвостовой вызов*, переходит прямо в тело вызываемой функции, *не создавая новый кадр стека, что экономит память*. Есть также компилируемые языки, в которых реализована РТС, иногда в качестве оптимизации, включаемой по желанию.

Если в языке нет никакого механизма итераций, кроме рекурсии, то РТС необходима из практических соображений.

В CPython РТС не реализована и, скорее всего, никогда не будет. РТС усложняет отладку для всех, а преимущества получают только те немногие, кто предпочитает использовать *рекурсию вместо итераций* [4, стр. 644].

## 14. Замечание о пользовательских пакетах

При написании пользовательских пакетов файл зависимостей должен быть как можно менее ограничительным

requirements.txt

```
# Data
numpy >= 1.16.0, !=1.24.0
pandas >= 1.1.0, < 1.3.0; python_version == '3.7'
pandas >= 1.3.0; python_version >= '3.8'
scipy >= 1.9.3; python_version >= '3.8'

# Parallelization
joblib >= 1.2.0; python_version >= '3.8'

# Models and frameworks
scikit-learn >= 1.0.0; python_version >= '3.8'
pyod >= 1.0.7; python_version >= '3.8'

# Optimization and solvers
# pyomo >= 6.4.2; python_version >= '3.8'
# PySCIPOpt installed using environment.yaml file of conda package manager
pyscipopt >= 4.3.0; python_version == '3.8'

# Plotting
matplotlib >= 3.3.1; python_version >= '3.8'
```

```

# Misc
pathlib2 >= 2.3.7
python-dotenv >= 0.21.0
pyyaml >= 6.0
tqdm
psutil >= 5.7.3

# Tests
pytest >= 6.2.0

```

Файл `setup.py` может выглядеть так

```

from pathlib import Path
from typing import List

import setuptools

# The directory containing this file
HERE = Path(__file__).parent.resolve()

# The text of the README file
NAME = "zyopt"
VERSION = "0.0.1"
AUTHOR = "Digital Industrial Platform"
SHORT_DESCRIPTION = (
    "Add-in for the SCIP solver with support for heuristics, "
    "classical machine learning and deep learning methods"
)
README = Path(HERE, "README.md").read_text(encoding="utf-8")
URL = ""
REQUIRES_PYTHON = ">=3.8"
LICENSE = "BSD 3-Clause"

def _readlines(*names: str, **kwargs) -> List[str]:
    encoding = kwargs.get("encoding", "utf-8")
    lines = Path(__file__).parent.joinpath(*names).read_text(encoding=encoding).splitlines()
    return list(map(str.strip, lines))

def _extract_requirements(file_name: str):
    return [line for line in _readlines(file_name) if line and not line.startswith("#")]

def _get_requirements(req_name: str):
    requirements = _extract_requirements(req_name)
    return requirements

setuptools.setup(
    name=NAME,
    version=VERSION,
    author=AUTHOR,
    author_email="itmo.nss.team@gmail.com",
    description=SHORT_DESCRIPTION,
    long_description=README,
    long_description_content_type="text/x-rst",
    url=URL,
    python_requires=REQUIRES_PYTHON,
    license=LICENSE,

```



```

packages=setuptools.find_packages(exclude=["test*"]),
include_package_data=True,
install_requires=get_requirements("requirements.txt"),
classifiers=[
    "License :: OSI Approved :: BSD License",
    "Programming Language :: Python :: 3.8",
    "Programming Language :: Python :: 3.9",
    "Programming Language :: Python :: 3.10",
],
)

```

Сборка выполняется в корне проекта

```
$ python setup.py sdist bdist_wheel
```

Если все прошло успешно, то теперь можно опубликовать пакет на TestPyPI с помощью утилиты twine

```
$ twine upload -r testpypi dist/* --verbose
```

Посмотреть, что получилось можно на [https://test.pypi.org/project/my\\_prjoect\\_name/](https://test.pypi.org/project/my_prjoect_name/). Для проверки работоспособности пакета нужно его поставить на локальную машину

```
$ pip install --index-url https://test.pypi.org/simple/ \
--extra-index-url https://pypi.org/simple my_package_name
```

ВАЖНО! Флаг `--extra-index-url` нужен, чтобы pip мог при установке извлекать зависимости с PyPI.

И, наконец, если все устраивает, то можно опубликовать пакет на PyPI

```
$ twine upload dist/*
```

## 15. Инвариантность, ковариантность и контрвариантность

Обобщенный класс с ковариантным параметром типа

```

T_co = TypeVar("T_co", covariant=True)

class BeverageDispenser(Generic[T_co]):
    def __init__(self, beverage: T_co) -> None:
        self.beverage = beverage

    def dispense(self) -> T_co:
        return self.beverage

def install(dispenser: BeverageDispenser[Juice]) -> None:
    """..."""

```

По соглашению суффикс `_co` в `typed` обозначает ковариантные параметры-типы.

*Ковариантность*: связь тип-подтип между параметризованными классами изменяется в том же направлении, что и связь тип-подтип между параметрами-типами [4, стр. 516].

Пример на контрвариантность

```

from typing import TypeVar, Generic

class Refuse:
    """..."""

```

```

class Biodegradable(Refuse):
    """..."""

class Compostable(Biodegradable):
    """..."""

T_contra = TypeVar("T_contra", contravariant=True)

class TrashCan(Generic[T_contra]):
    def put(self, refuse: T_contra) -> None:
        """..."""

def deploy(trash_can: TrashCan[Biodegradable]):
    """..."""

```

`T_contra` – принятое по соглашению имя котравариантной переменной-типа.

## 15.1. Обзор инвариантности

### 15.1.1. Инвариантные типы

Обобщенный тип *L* *инвариантен*, если между двумя параметризованными типами нет отношения тип-подтип, даже если такое отношение существует между фактическими параметрами. Иными словами, если *L* инвариантен, то *L*[*A*] не является ни подтипом, ни супертипом *L*[*B*]. Они *несовместимы* в обоих направлениях.

*Изменяемые* типы в Python по умолчанию инвариантны. Например, `list[int]` не совместим с `list[float]`, и наоборот.

В общем случае, если формальный параметр-тип встречается в аннотациях типов аргументов метода и тот же параметр встречается в типе возвращаемого методом значения, то параметр должен быть *инвариантен*, чтобы гарантировать *типобезопасность* при *обновлении* коллекции и *чтении* из нее [4, стр. 518].

### 15.1.2. Ковариантные типы

Некоторые авторы используют символы `<:` и `>:`, чтобы обозначить следующие отношения: *A* `>:` *B* означает, что *A* является супертипом *B*.

Если *A* `>:` *B*, то обобщенный тип *C* ковариантен, когда *C*[*A*] `>:` *C*[*B*]. Направление символа `>:` одинаково в обоих случаях, когда *A* встречается слева от *B*. Ковариантные обобщенные типы повторяют отношение тип-подтип между фактическими параметрами-типами.

Например

```

float >: int
frozenset[float] >: frozenset[int]

```

Любой код, ожидающий итератор `abc.Iterator[float]`, который отдает числа с плавающей точкой, может безопасно использовать итератор `abc.Iterator[int]`, отдающий целые числа. По той же причине типы `Callable` ковариантны относительно типа возвращаемого значения [4, стр. 519].

## 15.2. Контравариантные типы

Если  $A \rightarrow B$ , то обобщенный тип  $K$  контравариантен, если  $K[A] \sqsubseteq K[B]$ . Контравариантные обобщенные типы обращают связь тип-подтип между фактическими параметрами-типами.

Пример может служить класс `TrashCan`

```
Refuse :> Biodegradable
TrashCan[Refuse] <: TrashCan[Biodegradable]
```

Контравариантный контейнер обычно представляет собой структуру данных, предназначенную только для записи и называемую «стоком». В стандартной библиотеке нет таких коллекций, но есть несколько типов с контравариантным параметрами-типами.

Тип `Callable[[ParamType, ...], ReturnType]` контравариантен относительно параметров-типов, но ковариантен относительно `ReturnType` [4, стр. 519]. Главное то, что *контравариантные* формальные параметры определяют *типы аргументов*, используемых для вызова или отправки данных объекту, тогда как *ковариантные* формальные параметры определяют *типы выходов*, порожденных объектов, – тип отдаваемого или возвращаемого значения, в зависимости от объекта.

### 15.2.1. Эвристические правила вариантности

Несколько эвристических правил [4, стр. 520]:

- Если формальный параметр-тип определяет тип данных, *исходящих* из объекта, то он может быть *ковариантным*.
- Если формальный параметр-тип определяет тип данных, *входящих* в объект после его начального конструирования, то он может быть *контравариантным*.
- Если формальный параметр-тип определяет тип данных, *исходящих* из объекта, и тот же параметр определяет тип данных, *входящих* в объект, то он должен быть *инвариантным*.
- Чтобы ненароком не допустить ошибку, делайте формальные параметры инвариантными.

Тип `Callable[[ParamType, ...], ReturnType]: ReturnType` *ковариантный*, а каждый `ParamType` *контравариантный* [4, стр. 520]. По умолчанию `TypeVar` создает инвариантные формальные параметры, и именно так аннотированы изменяемые коллекции в стандартной библиотеке.

Пример обобщенного типа с ковариантным формальным параметром-типом

```
from typing import Protocol, runtime_checkable, TypeVar

T_co = TypeVar("T_co", covariant=True)

@runtime_checkable
class RandomPicker(Protocol[T_co]):
    def pick(self) -> T_co:
        ...
```

Обобщенный протокол `RandomPicker` может быть ковариантным, потому что его единственный формальный параметр встречается в *типе возвращаемого значения*.

## 16. Передача параметров и возвращаемые значения

В книге Ромальо [4, стр. 219] говорится, что в Python единственный способ передачи параметров – *вызов по соиспользованию* (call by sharing). Вызов по соиспользованию означает, что каж-

дый формальный параметр функции получает *копию ссылки* на фактический аргумент. Иначе говоря, внутри функции *параметры* становятся *псевдонимами* фактических аргументов.

Параметры функции, которые передаются ей при вызове, являются *обычными именами*, ссылающимися на *входные объекты*. Семантика передачи параметров в языке Python не имеет точного соответствия какому-либо одному способу, такому как «передача по значению» или «передача по ссылке». Например, если функции передается *неизменяемое* значение, это выглядит, как передача аргумента *по значению*. Однако при передаче *изменяемого* объекта (такого как список или словарь), который модифицируется функцией, эти изменения *будут отражаться на исходном объекте* [1, стр. 133].

## 17. Значения по умолчанию изменяемого типа: неудачная мысль

Не следует использовать в качестве значений по умолчанию изменяемые объекты. Проблема в том, что все экземпляры `HauntBus`, конструктору которых не был явно передан список пассажиров, разделяют один и тот же список по умолчанию.

Беда в том, что любое значение по умолчанию вычисляется один раз в момент определения функции, то есть обычно на этапе загрузки модуля, после чего значения по умолчанию становятся атрибутами объекта-функции. Так что если значение по умолчанию – изменяемый объект и вы его изменили, то изменение отразится и на всех последующих вызовах.

## 18. Сопоставление с последовательностями-образцами

Пример

```
metro_areas: t.List[t.Tuple[str, str, float, t.Tuple[float, float]]] = [
    ("Tokyo", "JP", 36.933, (35.689, 139.693)),
    ("Delhi NCR", "IN", 21.935, (28.61, 77.21)),
    ...
]

def main():
    for record in metro_areas:
        match record: # record это субъект
            case [name, _, _, (lat, lon)] if lon <= 0:
                print(...)
```

Образцы можно сделать более специфичными, добавив информацию о типе

```
case [str(name), _, _, (float(lat), float(lon))]:
    ...
```

С другой стороны, если мы хотим произвести сопоставление произвольной последовательности-субъекта, начинающейся с `str` и заканчивающейся вложенной последовательностью из двух `float`, то можно написать

```
case [str(name), *_ , (float(lat), float(lon))]:
    ...
```

---

---

#### Замечание

Важное соглашение в Python API: функции и методы, изменяющие объект на месте, должны возвращать `None`, давая вызывающей стороне понять, что изменился сам объект в противовес созданию нового [4, стр. 81]

---

---

#### Замечание

Кратная конкатенация *неизменяемых последовательностей* выполняется **неэффективно**, потому что вместо добавления элементов интерпретатор вынужден **копировать всю конечную последовательность**, чтобы создать новую с добавленными элементами. Тип `str` – исключение из этого правила. Поскольку построение строки с помощью оператора `+=` в цикле – весьма распространенная операция, в CPython этот случай *оптимизирован*. Экземпляры `str` создаются с *запасом памяти*, чтобы при конкатенации не приходилось каждый раз копировать всю строку [4, стр. 79]

---

---

#### Замечание

Большинство функций `numpy` и `scipy` написаны на C или C++ и могут задействовать все доступные ядра процессора, так как освобождают глобальную блокировку интерпретатора [4, стр. 90]

---

---

## 19. Правила видимости в функциях

При каждом вызове функции создается новое локальное пространство имен. Это пространство имен представляет локальное окружение, содержащее имена параметров функции, а также имена переменных, которым были присвоены значения в теле функции. Когда возникает необходимость отыскать имя, интерпретатор в первую очередь просматривает локальное пространство имен. Если искомое имя не было найдено, поиск продолжается в глобальном пространстве имен. Глобальным пространством имен для функций всегда является пространство имен модуля, в котором эта функция была определена. Если интерпретатор не найдет искомое имя в глобальном пространстве имен, поиск будет продолжен во встроеном пространстве имен. Если и эта попытка окажется неудачной, будет возбуждено исключение `NameError`.

В языке Python поддерживается возможность определять вложенные функции. Переменные во вложенных функциях привязаны к лексической области видимости. То есть поиск имени переменной начинается в *локальной области видимости* и затем последовательно продолжается во всех *объемлющих областях видимости* внешних функций, в направлении от внутренних к внешним. Если и в этих пространствах имен искомое имя не будет найдено, поиск будет продолжен в *глобальном*, а затем во *встроеном пространстве имен*, как и прежде.

При обращении к локальной переменной до того, как ей будет присвоено значение, возбуждается исключение `UnboundLocalError`

```
i = 0

def foo():
    i = i + 1
    print(i) # UnboundLocalError
```

В функции `foo` переменная `i` определяется как локальная переменная, потому что внутри функции ей присваивается некоторое значение и отсутствует инструкция `global`). При этом ин-

струкция присваивания `i = i + 1` пытается прочитать значение переменной `i` еще до того, как ей будет присвоено значение.

Хотя в этом примере существует глобальная переменная `i`, она не используется для получения значения. Переменные в функциях могут быть *либо локальными, либо глобальными* и не могут произвольно изменять область видимости в середине функции. Например, нельзя считать, что переменная `i` в выражении `i = i + 1` в предыдущем фрагменте обращается к глобальной переменной `i`; при этом переменная `i` в вызове `print(i)` подразумевает локальную переменную `i`, созданную в предыдущей инструкции [1, стр. 136].

## 20. Функции как объекты и замыкания

*Функции* в языке Python – *объекты первого класса*. Это означает, что они могут передаваться другим функциям в виде аргументов, сохраняться в структурах данных и возвращаться функциями в виде результата [1, стр. 136].

Когда инструкции, составляющие функцию, упаковываются вместе с окружением, в котором они выполняются, получившийся объект называют *замыканием*. Такое поведение объясняется наличием у каждой функции атрибута `__globals__`, ссылающегося на глобальное пространство имен, в котором функция была определена. Это пространство имен всегда соответствует модулю, в котором функция была объявлена [1, стр. 137].

Когда функция используется как *вложенная*, в *замыкание* включается все ее окружение, необходимое для работы внутренней функции.

*Замыкание* – это функция, назовем ее `f`, с расширенной областью видимости, которая охватывает переменные, на которые есть ссылки в теле `f`, но которые не являются ни глобальными, ни локальными переменными `f`. Такие переменные должны происходить из *локальной* области видимости *внешней* функции, *объемлющей* `f`. Не имеет значения, является функция анонимной или нет; важно лишь, что она может обращаться к *неглобальным* переменным, определенным *вне ее тела* [4, стр. 307].

Пример

```
def make_averager():
    series = [] # свободная переменная

    def averager(new_value): # функция-замыкание
        series.append(new_value)
        total = sum(series)
        return total / len(series)

    return averager
```

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

*Замыкание* `averager` (вложенная функция) расширяет область видимости функции, включая в нее привязку *свободной переменной* `series`.

Замыкание – функция, которая запоминает привязки 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.

Отметим, что единственная ситуация, когда функции может понадобиться доступ к внешним неглобальным переменным, – это когда она вложена в другую функцию и эти переменные являются частью локальной области видимости внешней функции [4, стр. 310].

## 21. Типизация

От типов модуля `typing` можно наследоваться

```
import typing as t
from collection import namedtuple

# Наследуемся от именованного кортежа
class Coordinates(t.NamedTuple):
    latit: float
    long: float

# Или так
# Но тип поля теперь не указывать
# Coordinates = namedtuple("Coordinates", ["latit", "long"])

# Доступ к полям через точечную нотацию
coord = Coordinates(latit=0.45, long=1.45)
coord.latit # 0.45
coord.long # 1.45
```

Функционально тоже что и дата-класс

```
from dataclasses import dataclass

@dataclass(frozen=False)
class Coordinates:
    latit: float
    long: float
```

Именованные кортежи от дата-классов отличаются тем, что именованные кортежи относятся к объектам неизменяемого типа данных. Дата-классы вообще говоря тоже можно сделать неизменяемыми после создания с помощью параметра `frozen=True`.

Именованные кортежи эффективнее с точки зрения хранения. С помощью библиотеки `pympler` <https://github.com/pympler/pympler>

```
import typing as t
from pympler import asizeof

class Coordinates(t.NamedTuple):
    latit: float
    long: float

print(asideof.asized(coord).size) # 104 Bytes
```

Иногда бывает полезно воспользоваться *типизированным словарем* `TypedDict`

```
import typing as t

# Доступ к полям будет как у словаря
class Coordinates(t.TypedDict):
    latit: float
    long: float
```

```
coord = Coordinates(latit=0.45, long=0.15)
coord["latit"] # 0.45
coord["long"] # 0.15
```

Еще бывает удобно воспользоваться *перечислением* Enum. Модуль enum это стандартная часть библиотеки Python, но если по какой-то причине интерпретатор не может его найти, то модуль можно установить так `pip install enum`

```
from enum import Enum

# Перечисление
class FileState(Enum):
    OPENED = "opened"
    CLOSE = "close"

FileState.OPENED.value # opened
```

В принципе поведение перечисления можно симитировать с помощью именованного кортежа

```
import typing as t

class FileState(t.NamedTuple):
    OPENED = "opened"
    CLOSE = "close"

FileState.OPENED # "opened"
```

Для неименованных кортежей можно создавать псевдонимы

```
# Кортеж с произвольным количеством целых чисел
int_tuple = t.Tuple[int, ...]

def f(*args: int_tuple) -> int:
    return sum(args)

print(f(10, 20, 30)) # 60

two_ints = t.Tuple[int, int]
# etc.
```

Generic (обобщенные типы)

```
import typing as t
T = t.TypeVar("T") # обобщенный тип

def first(iterable: t.Iterable[T]) -> t.Optional[T]:
    for item in iterable:
        return item
```

## 22. Модули, пакеты и дистрибутивы

ВАЖНО: *текущим каталогом* (`os.path.curdir`) будет тот, из-под которого запускается сценарий, но сканирование «окружающего пространства» в поисках нужных пользовательских модулей и пр. начинается с той директории, в которой *расположен* сценарий (см. `sys.path`). Если требуется какие-то подмодули сделать доступными через пространство имен пакета с помощью `__init__.py`, то лучше воспользоваться относительным импортом (он более четко указывает о намерениях).



Можно указывать относительный путь, а можно абсолютный, но от той директории, в которой лежит пусковой сценарий (например, `./src/run.py`). То есть, если

```
./ # корень проекта
src/
  config/ # пакет
    __init__.py
    config.py # модуль
    ...
```

то

`./src/config/__init__.py`

```
# поиск начнется со сканирования src/ (потому что здесь лежит пакет config/)
from config.config import Config
# или относительно директории пакета
from .config import Config
```

ВАЖНО: в общем случае абсолютный путь в модулях `__init__.py` отсчитывается от директории родительского пакета, то есть от той директории, в которой лежит пусковой сценарий. Этот сценарий указывает от какой директории теперь отсчитываться (не включая эту директорию в пути).

Для сценариев командной оболочки можно явно указать директорию, которая должна просматриваться первой в поисках модулей и пакетов с помощью переменной окружения `PYTHONPATH`

`./src/strategy_templates/make_strategy_file.py`

```
from strategy_templates.templates import *
...
```

```
# Сканироваться будет директория ./src
PYTHONPATH=./src python ./src/strategy_templates/make_strategy_file.py ...
```

Пусковой сценарий удобно располагать в поддиректории проекта `./src`. Если запускать сценарий так `python ./src/run.py`, то сканирование начнется с директории `src` и технически все будет верно, но PyCharm будет подсвечивать пути красным. Чтобы убрать эту красноту, нужно просто объявить `./src` как «Sources Root», кликнув правой кнопкой мыши на директории в дереве проекта и выбрав соответствующую метку.

Когда инструкция `import` впервые загружает модуль, она выполняет следующие три операции [1, стр. 189]:

1. Создает новое пространство имен, которое будет служить контейнером для всех объектов, определенных в соответствующем файле.
2. Выполняет программный код в модуле внутри вновь созданного пространства имен.
3. Создает в вызывающей программе имя, ссылающееся на пространство имен модуля. Это имя совпадает с именем модуля.

Когда модуль импортируется впервые, он компилируется в байт-код и сохраняется на диске в файле с расширением `*.рус`. При всех последующих обращениях к импортированию этого модуля интерпретатор будет загружать скомпилированный байт-код, если только с момента создания байт-кода в файл `.py` не вносились изменения (в этом случае файл `.рус` будет создан заново).

Автоматическая компиляция программного кода в файл с расширением `.рус` производится только при использовании инструкции `import`. При запуске программ из командной строки этот файл не создается.

*Модули* в языке Python – это *объекты первого класса* [1, стр. 190]. То есть они могут присваиваться переменным, помещаться в структуры данных, такие как списки, и передаваться между частями программы в виде элемента данных. Например

```
import pandas as pd
```

просто создает переменную `pd`, которая ссылается на объект модуля `pandas`.

Важно подчеркнуть, что инструкция `import` выполнит все инструкции в загруженном файле. Если в дополнение к объявлению переменных, функций и классов в модуле содержатся некоторые вычисления и вывод результатов, то результаты будут выведены на экран в момент загрузки модуля.

Инструкция `import` может появляться в любом месте программы. Однако программный код любого модуля *загружается и выполняется* только один раз, независимо от количества инструкций `import`.

*Глобальным пространством имен* для функции всегда будет *модуль*, в котором она была объявлена, а не пространство имен, в которое эта функция была импортирована и откуда была вызвана [1, стр. 192].

Пакеты позволяют сгруппировать коллекцию модулей под общим именем пакета. Пакет создается как каталог с тем же именем, в котором создается файл с именем `__init__.py`.

Например, пакет может иметь такую структуру

```
graphics/  
  __init__.py  
  primitives/  
    __init__.py  
    lines.py  
    fill.py  
    text.py  
    ...  
  graph2d/  
    __init__.py  
    plot2d.py  
    ...  
  graph3d/  
    plot3d.py  
    ...  
  formats/  
    __init__.py  
    gif.py  
    png.py  
    tiff.py  
    ...
```

Всякий раз когда какая-либо *часть пакета импортируется впервые*, выполняется программный код в файле `__init__.py` [1, стр. 198]. Этот файл может быть пустым, но может также содержать программный код, выполняющий инициализацию пакета. Выполнены будут все файлы `__init__.py`, которые встретятся инструкции `import` в процессе ее выполнения.

То есть инструкция

```
import graphics.primitives.fill
```

сначала выполнит файл `__init__.py` в каталоге `graphics`, а затем файл `__init__.py` в каталоге `primitives`.

При импортировании модулей из пакета следует быть особенно внимательными и не использовать инструкцию вида `import module`, так как в Python 3, инструкция `import` предполагает, что указан абсолютный путь, и будет пытаться загрузить модуль из стандартной библиотеки. Использование инструкции импортирования по относительному пути более четко говорит о ваших намерениях.

Возможность импортирования по относительному пути можно также использовать для загрузки модулей, находящихся в других каталогах того же пакета. Например, если в модуле `Graphics.Graph2d.plot2d` потребуется импортировать модуль `Graphics.Primitives.lines`, инструкция импорта будет иметь следующий вид

```
from ..primitives import lines # так можно!
```

В этом примере символы `..` перемещают точку начала поиска на уровень выше в дереве каталогов, а имя `primitives` перемещает ее вниз, в другой каталог пакета.

Импорт по относительному пути может выполняться только при использовании инструкции импортирования вида

```
from module import symbol
```

То есть такие конструкции, как

```
import ..primitives.lines # Ошибка!
import .lines # Ошибка!
```

будут рассматриваться как синтаксическая ошибка.

Кроме того, имя `symbol` должно быть допустимым идентификатором. Поэтому такая инструкция, как

```
from .. import primitives.lines # Ошибка!
```

также считается ошибочной.

Наконец, импортирование по относительному пути может выполняться только для модулей в пакете; не допускается использовать эту возможность для ссылки на модули, которые просто находятся в другом каталоге файловой системы.

Импортирование по одному только имени пакета не приводит к импортированию всех модулей, содержащихся в этом пакете [1, стр. 199], однако, так как инструкция `import graphics` выполнит файл `__init__.py` в каталоге `graphics`, в него можно добавить инструкции импортирования по относительному пути, которые автоматически загрузят все модули, как показано ниже

```
# graphics/__init__.py
from . import primitives, graph2d, graph3d

# graphics/primitives/__init__.py
from . import lines, fill, text
...
```

Для того чтобы сделать функции модулей подпакетов доступными из-под имени подпакетов (без обращения к модулям, в которых были объявлены эти функции), можно относительный импорт организовать следующим образом

```
# graphics/primitives/__init__.py
from .fill import make_fill
from .lines import make_lines
...
```

Теперь вызвать, например, функцию `make_fill` модуля `fill` подпакета `primitives` можно так

```
from graphics.primitives import make_fill
# вместо
from graphics.primitives.fill import make_fill
```

Грубо говоря, можно считать, что элементы расположенные справа от инструкции `import` в файле `__init__.py` будут как бы замещать имя модуля `__init__.py` в пути до этого файла, т.е.

```
# graphics/formats/__init__.py
from .png import print_png
from .jpg import print_jpg

# В сессии
>>> import graphics.formats.print_png
```

Переменная `__all__` управляет логикой работы инструкции `import *` и проявляется только если пользователь модуля/пакета использует прием «импортировать все». Если известен путь до нужного модуля, то переменная `__all__` не мешает. Если определить `__all__` как пустой список, ничего экспортироваться не будет [2, стр. 395].

Важное замечание: относительное импортирование работает только для модулей, которые размещены внутри подходящего пакета. В частности, оно не работает внутри простых модулей, размещенных на верхнем уровне скриптов. Оно также не работает, если *части пакета* исполняются напрямую, как *скрипты*, например [2, стр. 396]

```
$ python mypackage/A/spam.py # Относительное импортирование не работает!!!
```

С другой стороны, если вы выполните предыдущий скрипт, передав Python опцию `-m`, относительное импортирование будет работать правильно

```
$ python -m mypackage/A/spam # Относительное импортирование работает!
```

---

#### Замечание

Относительный импорт не работает, если части пакета исполняются напрямую, как скрипты. Но ситуацию можно исправить, если воспользоваться опцией `-m`

Наконец, когда интерпретатор импортирует пакет, он объявляет специальную переменную `__path__`, содержащую список каталогов, в которых выполняется поиск модулей пакета (`__path__` представляет собой аналог списка `sys.path` для пакета). Переменная `__path__` доступна для программного кода в файлах `__init__.py` и изначально содержит единственный элемент с именем каталога пакета.

При необходимости пакет может добавлять в список `__path__` дополнительные каталоги, чтобы изменить путь поиска модулей. Это может потребоваться в случае сложной организации дерева каталогов пакета в файловой системе, которая не совпадает с иерархией пакета.

## 22.1. Создание отдельных каталогов с кодом для импорта под общим пространством имен

Требуется определить пакет Python высшего уровня, который будет служить пространством имен для большой коллекции отдельно поддерживаемых подпакетов.

Нужно организовать код так же, как и в обычном пакете Python, но опустить файлы `__init__.py` в каталогах, где компоненты будут объединяться. Пример [2, стр. 399]

```
foo-package/  
  spam/  
    blah.py  
  
bar-package/  
  spam/  
    grok.py
```

В этих каталогах имя `spam` используется в качестве общего пространства имен. Обратите внимание, что файл `__init__.py` отсутствует в обоих каталогах.

Теперь, если добавить оба пакета `foo-package` и `bar-package` к пути поиска модулей Python и попробуете импортировать

```
import sys  
sys.path.extend(["foo-package", "bar-package"])  
import spam.blah  
import spam.grok
```

Для разных каталога пакетов слились вместе. Механизм, который здесь работает, известен под названием «пакет пространства имен». По сути, пакет пространства имен – это специальный пакет, разработанный для слияния различных каталогов с кодом под общим пространством имен.

Ключ к созданию пакета пространства имен – отсутствие файлов `__init__.py` в каталоге высшего уровня, который служит общим пространством имен. Вместо того чтобы выкинуть ошибку, интерпретатор начинает создавать список всех каталогов, которые содержит совпадающее имя пакета. Затем создается специальный модуль-пакет пространства имен, и в его переменной `__path__` сохраняется доступная только для чтения копия списка каталогов.

## 23. Некоторые приемы

### 23.1. Вычисления со словарями

Рассмотрим словарь, который отображает тикеры на цены

```
d = {  
    "ACME": 45.23,  
    "AAPL": 612.78,  
    "IBM": 205.55,  
    "HPQ": 37.20,  
    "FB": 10.75,  
}
```

Чтобы найти наименьшую/наибольшую цены с тикером можно обратиться к ключи и значения, а затем воспользоваться функций `zip()`

```
min(zip(d.values(), d.keys())) # (10.75, "FB")  
max(zip(d.values(), d.keys())) # (612.78, "AAPL")
```

Важно иметь в виду, что функция `zip()` создает итератор, по которому можно пройти только один раз.

Использование функции `zip()` решает задачу путем «обращения» словаря в последовательность пар (value, key).

Однако, вариант с функцией `zip()` требует большего времени, чем вариант на цикле

```

%%timeit -n 1_000_000
# 639 ns +/- 3.04 ns per loop (mean +/- std. dev. of 7 runs, 1,000,000 loops each)
min(zip(d.values(), d.values()))

%%timeit -n 1_000_000
# 576 ns +/- 1.4 ns per loop (mean +/- std. dev. of 7 runs, 1,000,000 loops each)
def find_min_pair(d: t.Dict[str, float]) -> t.Tuple[float, str]:
    min_value = float("inf")
    for key, value in d.items():
        if value < min_value:
            min_value = d[key]
            min_key = key
    return (min_value, min_key)

```

Пусть есть два словаря. Требуется выяснить, что у них общего

```

d1 = {"x": 1, "y": 2, "z": 3}
d2 = {"w": 10, "x": 11, "y": 2}

# Найти общие ключи
d1.keys() & d2.keys()

# Находим ключи, которые есть в d1, но которых нет в d2
d1.keys() - d2.keys()

# Находим общие пары (key, value)
d1.items() & d2.items() # {"y", 2}

```

Словарь – это отображение множества ключей на множество значений. Метод словаря `keys()` возвращает *объект ключей словаря* `dict_keys`. Малоизвестная особенность этих объектов заключается в том, что они поддерживают набор операций над *множествами*: объединение, пересечение и разность. Так что, если требуется выполнить этот набор операций над ключами словаря, то можно использовать объект ключей словаря напрямую, без предварительного конвертирования во множество [2, стр. 35], т.е.

```

d1.keys() & d2.keys() # {"x", "y"}
# вместо
set(d1.keys()) & set(d2.keys()) # {"x", "y"}
# или
set(d1.keys()).intersection(set(d2.keys())) # {"x", "y"}

```

Найти пересечение индексов двух серий можно было бы так

```

ser1 = pd.Series(d1, name="ser1")
ser2 = pd.Series(d2, name="ser2")

pd.merge(
    ser1,
    ser2,
    left_index=True,
    right_index=True,
    how="inner"
).index.to_list() # ["x", "y"]

```

Ремарка: в *контейнерных последовательностях* (`list`, `tuple` etc.) хранятся *ссылки* на объекты *любого типа*, тогда как в *плоских последовательностях* (`str`, `bytes` etc.) – сами значения прямо в памяти, занятой последовательностью, а не как отдельные объекты Python [4, стр. 49].

## 23.2. Удаление дубликатов из последовательности

Вы хотите исключить дублирующиеся значения из последовательности, но при этом сохранить порядок следования оставшихся элементов.

Если значения в последовательности являются хешируемыми, задача может быть легко решена с использованием множества и генератора

```
%%timeit -n 100_000
# 984 ns +/- 17.6 ns per loop (mean +/- std. dev. of 7 runs, 100,000 loops each)
def dedupe(items: t.Iterable[int]) -> t.Iterable[int]:
    seen: t.Set[int] = set()
    for item in items:
        if item not in seen:
            yield item # отдать элемент
            seen.add(item) # обновить множество

lst = [1, 5, 2, 1, 9, 1, 5, 10]
list(dedupe(lst)) # [1, 5, 2, 9, 10]
```

Или так

```
%%timeit -n 100_000
# 663 ns +/- 26.2 ns per loop (mean +/- std. dev. of 7 runs, 100,000 loops each)
def dedupe_list(items: t.Iterable[int]) -> t.Iterable[int]:
    seen: t.Iterable[int] = []
    for item in items:
        if item not in seen:
            seen.append(item)
    return seen
```

## 23.3. Сортировка списка словарей по общему ключу

У вас есть список словарей, и вы хотите отсортировать записи согласно одному или более полям. Сортировка структур этого типа легко выполняется с помощью функции `operator.itemgetter`. Именованный аргумент `key` должен быть *вызываемым объектом* (т.е. объектом, в котором реализован метод `__call__`). Функция `itemgetter()` создает такой вызываемый объект

```
from operator import itemgetter

records: t.Iterable[dict] = [
    {"fname": "Brian", "lname": "Jones", "uid": 1003},
    {"fname": "David", "lname": "Beazley", "uid": 1002},
    {"fname": "John", "lname": "Cleese", "uid": 1004},
]

# аргумент key ожидает получить вызываемый объект
sorted(records, key=itemgetter("fname"))
sorted(records, key=itemgetter("uid"))
# то же, что и
sorted(records, key=lambda record: record["fname"])
sorted(records, key=lambda record: record["uid"])
```

Функция `itemgetter()` может принимать несколько полей

```
sorted(records, key=itemgetter("lname", "fname"))
```

Эту технику можно применять и к функциям `min`, `max`

```
# найти строку с наименьшим значением идентификационного номера
min(records, key=itemgetter("uid"))
```

## 23.4. Отображение имен на последовательность элементов

У вас есть код, который осуществляет доступ к элементам в списке или кортеже по позиции. Однако такой подход часто программу нечитабельной.

`collections.namedtuple()` – фабричный метод, который возвращает подкласс стандартного типа Python – `tuple`. Метод возвращает класс, который может порождать экземпляры

```
Person = namedtuple("Person", ["name", "age", "job"])
leor = Person(name="Leor", age=36, job="DS")
```

Хотя экземпляр `namedtuple` выглядит так же, как и обычный экземпляр класса, он взаимозаменяем с кортежем и поддерживает все обычные операции кортежей, такие как индексирование и распаковка

```
name, age, job = leor
```

Возможное использование именованного кортежа – замена словаря, который требует больше места для хранения. Так что, если создаете крупные структуры данных с использованием словарей, применение именованных кортежей будет более эффективным. Однако, именованные кортежи неизменяемы в отличие от словарей.

Если вам нужно изменить любой из атрибутов, это может быть сделано с помощью метода `_replace()`, которым обладают экземпляры именованных кортежей.

Тонкость использования метода `_replace()` заключается в том, что он может стать удобным способом наполнить значениями именованный кортеж, у которого есть опциональные или отсутствующие поля. Чтобы сделать это, создайте прототип кортежа, содержащий значения по умолчанию, а затем применяйте `_replace()` для создания новых экземпляров с замененными значениями

```
from collection import namedtuple

Stock = namedtuple("Stock", ["name", "shares", "price", "date", "time"])
stock_prototype = Stock("", 0, 0.0, None, None)

def dict_to_stock(s):
    return stock_prototype._replace(**s)
```

## 24. Строки и текст

### 24.1. Разрезание строк различными разделителями

Нужно разделить строку на поля, но разделители (и пробелы вокруг них) внутри строки разные

```
import re
line = "asdf fjdk; afed, fjek,asdf,      foo"
re.split(r"[;,\s]*", line)
```



## 25. Профилирование и замеры времени выполнения

При проведении измерений производительности нужно помнить, что любые результаты будут приблизительными. Функция `time.perf_counter()` предоставляет наиболее точный таймер из доступных. Однако она все-таки измеряет *внешнее время*, и *на результаты влияют различные факторы, такие как нагрузка компьютера*.

Если вы хотите получить время обработки, а не внешнее время, используйте `time.process_time()` [2, стр. 574]

```
from functools import wraps

def timethis(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.process_time() # <- NB
        r = func(*args, **kwargs)
        stop = time.process_time() # <- NB
        print(f"{func.__module__}.{func.__name__} : {end - start}")
        return r
    return wrapper

@timethis
def countdown(n):
    while n > 0:
        n -= 1

countdown(100000)
```

Чтобы подсчитать время выполнения блока инструкций, можно определить менеджер контекста

```
from contextlib import contextmanager

@contextmanager
def timeblock(label):
    start = time.process_time()
    try:
        yield
    finally:
        end = time.process_time()
        print(f"{label} : {end - start}")

with timeblock("counting"):
    n = 100000
    while n > 0:
        n -= 1
    # counting: 1.55555
```

Запустить профилировщик для веб-приложения и перенаправить вывод профилировщика в файл

```
# В основном терминале
$ python -m cProfile flask_app.py > profile.log
* Serving Flask app 'solverapi' (lazy loading)
* Environment: production
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
* Debug mode: on
* Running on http://127.0.0.1:5000 (Press CTRL+C to quit)
```

```
* Restarting with stat
* Debugger is active!
* Debugger PIN: 158-204-808

# В параллельном терминале
$ curl -H "Content-Type: application/json" -X POST --data "@file_name.json" "localhost:5000/api/solver/balance"
# После завершения расчета можно прервать сессию в основном терминале
$ vim profile.log
```

Граф цепочки выполнения программы можно построить следующим образом

```
$ pip install gprof2dot
$ python -m cProfile -o profile.pstat app.py
$ gprof2dot -f pstats profile.pstat | dot -Tpng -o output.png
```

Потребление памяти приложением можно оценить с помощью библиотеки `memory_profiler` <https://pypi.org/project/memory-profiler/>. После установки библиотеки будет доступна утилита командной строки `mprof`.

Запустить приложение в режиме замера потребления памяти для основного (родительского) процесса и его дочерних процессов (если они существуют) можно следующим образом

```
$ mprof run --include-children --multiprocess script.py
```

После остановки приложения в рабочей директории будет создан `dat`-файл с результатами измерений потребления памяти. Построить график потребления можно так

```
# -s: угол наклона, по которому можно судить об утечке памяти
# -t: заголовок графика
$ mprof plot -s -t "496.lp"
```

Перечень поддерживаемых флагов, связанных с конкретной подкомандой `mprof`, можно посмотреть так

```
$ mprof <subcommand> --help
...
```

Для измерения потребления памяти какой-то конкретной функции можно воспользоваться декоратором `@memory_profiler.profile`

```
from memory_profiler import profile

@profile
def my_func():
    a = [1] * (10 ** 6)
    b = [2] * (2 * 10 ** 7)
    del b
    return a
```

Затем остается только запустить интерпретатор с флагом `-m memory_profiler` и проанализировать ответ `memory_profiler`.

## 26. Итераторы и генераторы

Ремарка: Инициализацию кортежей, массивов и других последовательностей можно начинать с использования спискового включения, но *генераторное выражение* экономит память, так как

отдает элементы по одному, применяя протокол итератора, вместо того чтобы сразу строить целиком список для передачи другому конструктору [4, стр. 55].

В большинстве случаев для прохода по итерируемому объекту используется цикл `for`. Однако иногда задачи требуют более точного контроля лежащего в основе механизма итераций.

Следующий код иллюстрирует базовые механизмы того, что происходит во время итерирования

```
items = [1, 2, 3] # Итерируемый объект
# Получаем объект итератора
# Функция iter(items) вызывает метод итерируемого объекта items.__iter__()
it = iter(items) # Итератор
# Запускаем итератор
next(it) # Вызывается it.__next__() -> 1
next(it) # -> 2
next(it) # -> 3
next(it) # Возбуждается исключение StopIteration
```

Список `items` как *итерируемый объект* имеет метод `__iter__()`, который должен возвращать *объект-итератора* (`it`). У объекта-итератора должен быть метод `__next__()` для перебора элементов. Вот функция `next(it)` и вызывает метод `__next__()` объекта-итератора для получения следующего элемента. Когда список исчерпывается, возбуждается исключение `StopIteration`.

*Протокол итераций* Python требует, чтобы метод `__iter__()` возвращал специальный объект-итератор, в котором реализован метод `__next__()`, который выполняет итерацию [2, стр. 128]. Функция `iter()` просто возвращает внутренний итератор, вызывая `s.__iter__()`.

*Протокол итератора* Python требует `__iter__()`, чтобы вернуть специальный *объект итератора*, в котором реализован метод `__next__()`, а исключение `StopIteration` используется для подачи сигнала о завершении [2, стр. 131].

Когда поток управления покидает тело генераторной функции, возбуждается исключение `StopIteration`.

Метод `__iter__()` *итерируемого объекта* может быть реализован как обычная *генераторная функция* [2, стр. 133]

```
class linehistory:
    ...
    def __iter__(self):
        for lineno, line in enumerate(self.lines, 1):
            self.history.append((lineno, line))
            yield line
```

Для того чтобы пропустить первые несколько элементов по какому-то условию, можно воспользоваться функцией `itertools.dropwhile`

```
from itertools import dropwhile

def read_wo_header(file_name: str):
    with open(file_name, mode="r") as f:
        for line in dropwhile(lambda line: line.startswith("#"), f):
            print(line.rstrip())
```

Возвращаемый итератор отбрасывает первые элементы в последовательности до тех пор, пока предоставленная функция возвращает `True`.

Если нужно просто пропустить первые несколько строк файла (не по условию), то будет полезна функция `itertools.islice`

```

with open(file_name, mode="r", encoding="utf-8") as f:
    for line in islice(f, 7, None): # пропустить первые 7 строк файла
        if line.startswith("# rows".lower()):
            break
    ...

```

## 27. Захват переменных в анонимных функциях

Рассмотрим поведение следующей программы:

```

>>> x = 10
>>> a = lambda y: x + y
>>> x = 20
>>> b = lambda y: x + y
>>> a(10) # 30
>>> b(10) # 30

```

Проблема в том, что значение `x`, используемое `lambda`-выражением, является *свободной переменной*, которая связывается во время *выполнения*, а не во время *определения* [2, стр. 233]. Так что значение `x` будет таким, каким ему случится быть во время выполнения.

---

*Замечание*

*Свободные переменные связываются во время выполнения, а не во время определения*

---

Другими словами у замыканий позднее связывание. Замыкания – это функции с расширенной областью видимости, которая включает все неглобальные переменные. То есть замыкания умеют запоминать привязки свободных переменных.

Например,

```

funcs = [
    lambda x: x + n
    for n in range(3)
]
for f in funcs:
    print(f(0))
# 2
# 2
# 2

```

## 28. Передача дополнительного состояния с функциями обратного вызова

```

import typing as t

def apply_async(
    func: t.Callable,
    args: t.Tuple[t.Union[str, int]],
    *,
    callback: t.Callable]
) -> t.NoReturn:
    result: t.Union[str, int] = func(*args)
    callback(result)

```

```
def add(x: int, y: int) -> int:
    return x + y
```

Для хранения состояния можно использовать *замыкание* [2, стр. 238]

```
def make_handler():
    count = 0
    def handler(result: t.Union[str, int]) -> t.NoReturn:
        nonlocal count
        count += 1
        print(f"[{count}] Got: {result}")
    return handler

handler = make_handler()
apply_async(add, (2, 3), callback=handler) # [1] Got: 5
apply_async(add, ("hello", "world"), callback=handler) # [2] Got: hello world
```

## 29. Использование лениво вычисляемых свойств

Вы хотите определить доступный только для чтения атрибут как свойство, которое вычисляется при доступе к нему. Однако после того, как доступ произойдет, значение должно кешироваться и не пересчитываться при следующих запросах.

Дескриптор – класс, который реализует три ключевые операции доступа к атрибутам (получения, присваивания и удаления) в форме специальных методов `__get__()`, `__set__()` и `__delete__()`.

Эффективный путь определения ленивых атрибутов – это использование *класса-дескриптора* [2, стр. 271]

```
# дескрипторный класс
class lazyproperty:
    def __init__(self, f: t.Callable):
        self.f = f

    def __get__(self, instance, cls):
        if instance is None:
            # Если дескриптор вызывается через объект управляющего класса,
            # например как Circle.area, то instance=None и будет возвращена
            # ссылка на объект экземпляра дескриптора
            return self
        else:
            value = self.f(instance)
            setattr(instance, self.f.__name__, value)
            return value
```

Чтобы использовать этот код, вы можете применить его в классе

```
class Circle:
    def __init__(self, radius: float):
        self.radius = radius

    @lazyproperty
    def area(self):
        print("Computing area")
        return math.pi * self.radius ** 2

    @lazyproperty
    def perimeter(self):
        print("Computing perimeter")
```

```
return 2 * math.pi * self.radius
```

Вот пример использования

```
>>> c = Circle(radius=4.0)
>>> c.area
# Computing area
# 50.26...
>>> c.area # 50.26...
```

Во многих случаях цель применения лениво вычисляемых атрибутов заключается в увеличении производительности. Например, вы можете избежать вычисления значений, если только они действительно где-то не нужны.

Когда дескриптор помещается в определение класса, его методы `__get__()`, `__set__()` и `__delete__()` задействуются при доступе к атрибуту. Но если дескриптор определяет только метод `__get__()`, то у него намного более слабое связывание, нежели обычно. В частности, метод `__get__()` срабатывает, *только если атрибут*, к которому осуществляется доступ, *отсутствует в словаре экземпляра* управляющего класса (в данном случае класса `Circle`) [2, стр. 272].

Класс `lazyproperty` использует это так: он заставляет метод `__get__()` сохранять вычисленное значение в экземпляре, используя то же имя, что и само свойство. С помощью этого значение сохраняется в словаре экземпляра и отключает будущие вычисления свойства.

Возможный недостаток этого рецепта в том, что вычисленное значение становится изменяемым после создания. То есть значение, например, свойства `area` можно затереть.

Если это проблема, вы можете использовать немного менее эффективное решение [2, стр. 273]

```
def lazyproperty(func):
    name = "_lazy_" + func.__name__
    @property
    def lazy(self):
        if hasattr(self, name):
            return getattr(self, name)
        else:
            value = func(self)
            setattr(self, name, value)
            return value
    return lazy
```

В этом случае операции присваивания недоступны

```
>>> c = Circle(4.0)
>>> c.area
Computing area
50.26...
>>> c.area
50.26...
>>> c.area = 25 # Поднимется исключение AttributeError
```

В этом случае все операции получения значения проводятся через функцию-геттер свойства. Это менее эффективно, чем простой поиск значения в словаре экземпляра.

Еще можно просто задекорировать свойство декоратором `lru_cache`

```
from functools import lru_cache

class Circle:
    def __init__(self, radius: float):
        self.radius = radius
```

```

@property
@lru_cache
def area(self):
    print("Computing area")
    return math.pi * self.radius ** 2

@property
@lru_cache
def perimeter(self):
    print("Computing perimeter")
    return 2 * math.pi * self.radius

>>> circle = Circle(4.0)
>>> circle.area
# Computing area
# 50.26...
>>> circle.area # 50.26...

```

## 30. Определение более одного конструктора в классе

Вы пишете класс и хотите, чтобы пользователи могли создавать экземпляры не только лишь единственным способом, предоставленным `__init__()`.

Чтобы определить класс с более чем одним конструктором, вы должны использовать метод класса

```

class Circle:
    def __init__(self, radius: float, color: str = "black"):
        """
        Первичный конструктор
        """
        self.radius = radius
        self.color = color

    @classmethod
    def make_default_circle(cls):
        """
        Альтернативный конструктор. Конструктор тривиального класса
        """
        return cls(radius=1.0, color="red")

    @property
    @lru_cache
    def area(self):
        print("Computing area")
        return math.pi * self.radius ** 2

    @property
    @lru_cache
    def perimeter(self):
        print("Computing perimeter")
        return 2 * math.pi * self.radius

    def __repr__(self):
        return f"{type(self).__name__}(radius={self.radius}, color={self.color})"

```

```
def get_params(self) -> dict:
    return {"radius": self.radius, "color": self.color}
```

Одно из главных применений *методов класса* – это определение *альтернативных конструкторов* [2, стр. 294].

При определении класса с множественными конструкторами необходимо делать функцию `__init__()` максимально простой – она должна просто присваивать атрибутам значения. А вот уже альтернативные конструкторы будут вызываться при необходимости выполнения продвинутых операций.

Если требуется вызывать методы по имени, то можно воспользоваться `operator.methodcaller()`

```
import operator

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"Point({self.x}, {self.y})"

    def distance(self, x, y):
        return math.hypot(self.x - x, self.y - y)

p = Point(2, 3)
operator.methodcaller("distance", 0, 0)(p)
```

Функция `methodcaller()` может быть полезна, например, в следующем случае

```
class Person:
    def __init__(self, name: str, job: str):
        self.name = name
        self.job = job

    def action_1(self):
        return "Action-1"

    def action_2(self):
        return "Action-2"

    def action_N(self):
        return "Action-N"
```

Вызвать действие теперь можно так

```
def make(*, obj, action: str):
    if hasattr(obj, action):
        return methodcaller(action)(obj)
    else:
        raise ValueError(f"Object '{type(obj).__name__}' has't action '{action}' ...")

leor = Person(name="Leor", job="ML")
make(obj=leor, action="action_1") # Action-1
make(obj=leor, action="action_2") # Action-2
make(obj=leor, actin="action_10") # ValueError
```

Без `methodcaller()` пришлось бы писать что-то вроде

```
def bad_make(*, obj, action: str):
```



```

if action == "action_1":
    obj.action_1()
elif action == "action_2":
    obj.action_2()
...

```

## 31. Класс загрузчик данных

Иногда бывает удобно использовать свой загрузчик. Например, когда нужно работать с большими npz-файлами временных рядов

```

import pathlib2
import typing as t

class DataLoader:
    def __init__(self, data_dir):
        self.files = list(pathlib2.Path(data_dir).glob("*.npz"))

    def __getitem__(self, key):
        return self.read(self.files[key])

    def __iter__(self):
        yield from map(lambda file: self.read(file), self.files)

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

    def read(self, filepath):
        loader = np.load(filepath, allow_pickle=True)

        X = loader["X"]
        index = loader["index"]
        columns = loader["columns"]
        y = loader["y"]

data = DataLoader("./data")

```

## 32. Параметрические декораторы

Требуется создать функцию-декоратор, которая принимала бы аргументы

```

from functools import wraps
import logging

# level, name и message -- это параметры декоратора
def logged(level, name=None, message=None):
    # это обычный декоратор, аргумент func которого ссылается на декорируемую функцию
    def decorate(func: t.Callable):
        logname = name if name else func.__module__
        log = logging.getLogger(logname)
        logmsg = message if message else func.__name__

        @wraps(func)
        # args и kwargs -- это аргументы задекорированной функции
        def wrapper(*args, **kwargs):
            log.log(level, logmsg)

```

```

        return func(*args, **kwargs)
    return wrapper
return decorate

# Пример использования
@logged(logging.DEBUG) # -> @decorate: add = deocrate(add) -> wrapper || add -> wrapper
def add(x, y):
    return x + y

@logged(logging.CRITICAL, "example")
def spam():
    print("Spam!")

```

Можно считать, что после объявления функции `add` вместо выражения `@logged(logging.DEBUG)` стоит `@decorate`, но при этом еще доступна переменная `level` со значением `@logging.DEBUG`, а также переменные `name` и `message` со значением `None`. Аргумент функции `decorate` получает ссылку на декорируемую функцию `add`. Затем локальные переменные `logname`, `log` и `logmsg` получают значения, после чего возвращается ссылка на вложенную функцию `wrapper`. Таким образом, при вызове функции `add` будет вызываться функция `wrapper`.

### 33. Пользовательские исключения

Можно не просто наследовать пользовательский класс исключения от класса `Exception`, задавать сообщения по умолчанию и пр.

```

class PathToProblemError(Exception):
    """
    Incorrect path to problem
    """

    def __init__(
        self,
        message="Error! Incorrect path to problem: {}",
        *,
        incorrect_path_to_problem="",
    ):
        super().__init__(message.format(incorrect_path_to_problem))

```

### 34. Определение декоратора, принимающего необязательный аргумент

Вы хотели бы написать один декоратор, который можно было бы использовать и без аргументов – `@decorator`, и с необязательными аргументами `@decorator(x, y, z)` [2, стр. 339].

```

from functools import wraps, partial
import logging

def logged(func=None, *, level=logging.DEBUG, name=None, message=None):
    if func is None:
        return partial(logged, level=level, name=name, message=message)

    logname = name if name else func.__module__
    log = logging.getLogger(logname)
    logmsg = message if message else func.__name__

```

```

@wraps(func)
def wrapper(*args, **kwargs):
    log.log(level, logmsg)
    return func(*args, **kwargs)
return wrapper

# Пример использования
@logged
def add(x, y):
    return x + y

@logged(level=logging.CRITICAL, name="example")
def spam():
    print("Spam")

```

Этот рецепт просто заставляет декоратор одинаково работать и с дополнительными скобками, и без.

Чтобы понять принцип работы кода, вы должны четко понимать то, как декораторы применяются к функциям, а также условия их вызова. Для простого декоратора, такого как этот

```

@logged # logged(func=add, ...)
def add(x, y):
    return x + y

```

последовательность вызова будет такой

```

def add(x, y):
    return x + y

add = logged(add)

```

В этом случае обертываемая функция просто передается в `logged` первым аргументом. Поэтому в решении первый аргумент `logged()` – это обертываемая функция. Все остальные аргументы должны иметь значения по умолчанию.

Для декоратора, принимающего аргументы, такого как этот

```

@logged(level=logging.CRITICAL, name="example") # logged(func=None, ...)
def spam():
    print("Spam")

```

последовательность вызова будет такой

```

def spam():
    print("Spam")

spam = logged(level=logging.CRITICAL, name="example")(spam)

```

При первичном вызове `logged()` обертываемая функция не передается. Так что в декораторе она должна быть необязательной. Это, в свою очередь, заставляет другие аргументы быть именованными. Более того, когда аргументы переданы, декоратор должен вернуть функцию, которая принимает функцию и оборачивает ее. Чтобы сделать это, в решении используется хитрый трюк с `functools.partial`. Если точнее, он просто возвращает частично примененную версию себя, где все аргументы зафиксированы, за исключением обертываемой функции.

Таким образом, при повторном вызове функции `logged` через `partial` вызов будет выглядеть следующим образом

```
spam = logged(func=spam, level=logging.CRITICAL, name="example", message=None)
```

Одна из особенностей декораторов в том, что они применяются только один раз, во время *определения* функции [2, стр. 342]

## 35. Параллельное программирование

Библиотека `concurrent.futures` предоставляет класс `ProcessPoolExecutor`, который может быть использован для выполнения тяжелых вычислительных задач в *отдельно запущенных экземплярах интерпретатора Python* [2, стр. 498].

«Под капотом» `ProcessPoolExecutor` создает  $N$  независимо работающих интерпретаторов Python, где  $N$  – это количество доступных обнаруженных в системе CPU. Пул работает до тех пор, пока не будет выполнена последняя инструкция в блоке `with`, после чего пул процессов завершается. Однако программа будет ждать, пока вся отправленная работа не будет сделана.

Чтобы получить результат от экземпляра `Future`, нужно вызвать метод `result()`. Это вызовет *блокировку* на время, пока результат не посчитается и не будет возвращен пулом.

Несколько вопросов, связанных с пулами процессов:

- Этот прием распараллеливания работает только для задач, которые легко раскладываются на независимые части,
- Работа должна отправляться в форме простых функций,
- Аргументы функций и возвращаемые значения должны быть совместимы с `pickle`. Работа выполняется в отдельном интерпретаторе при использовании межпроцессной коммуникации. Так что данные, которыми обмениваются интерпретаторы, должны *сериализоваться*,
- Пулы процессов в Unix создаются с помощью системного вызова `fork()`. Он создает клон интерпретатора Python, включая все состояние программы на момент копирования. В Windows запускается независимая копия интерпретатора, которая не копирует состояние,
- Нужно с великой осторожностью объединять пулы процессов с программами, которые используют потоки.

### 35.1. Пример использования пула потоков

Требуется для каждой переменной в MILP-задаче описать контекст переменной через типы переменных, которые встречаются в тех ограничениях, в которые входит рассматриваемая переменная. Для примера пусть переменная входит в 3 ограничения. В первом ограничении кроме рассматриваемой переменной есть еще две: одна, скажем, вещественная, а другая целочисленная. Во втором ограничении кроме рассматриваемой переменной есть еще 3 вещественные. А в третьем ограничении кроме рассматриваемой есть еще одна бинарная. Тогда для рассматриваемой переменной мы должны получить такой контекст: `{"CONTINUOUS": 4, "BINARY": 1, "INTEGER": 1}`. Затем полученные контексты собираются в список словарей. На этом списке требуется построить кадр данных. В данном случае это можно сделать так

```
pd.DataFrame.from_dict(ChainMap(*results), orient="index") # ОЧЕНЬ МЕДЛЕННО!
```

Построение кадра данных на 26 000 контекстов занимает около 2-х минут. Однако, если список `results` разбить на пакеты, для каждого пакета построить кадр данных, собрать в список, а затем склеить с помощью `pd.concat()`, то время построения снижается до 6 секунд

```

dfs = []
for batch_idx in range(math.ceil(len(results) / batch_size)):
    _part = results[batch_idx * batch_size : (batch_idx + 1) * batch_size]
    dfs.append(pd.DataFrame.from_dict(ChainMap(*_part), orient="index"))

_features = pd.concat(dfs, axis=0)

```

Каждое обращение к `executor.submit` *планирует выполнение* одного вызываемого объекта и возвращает экземпляр `Future`. Первый аргумент – сам вызываемый объект, остальные – передаваемые ему аргументы.

Функция `as_completed()` возвращает итератор, который отдает будущие объекты по мере их завершения: `as_completed()` возвращает *только уже завершенные* будущие объекты.

```

def _get_var_context_types(
    self,
    cons: t.Iterable[t.Tuple[str, dict]],
    var_name: str,
) -> dict:
    """
    Gets var types in context current var. For example,
    "y_var_1" -> {"CONTINUOUS": 8, "INTEGER": 4, "BINARY": 0}
    """
    cons_name: str
    cons: dict
    _var_types: t.List[str] = []
    var_context_types: t.Dict[str, int]

    for cons_name, cons in cons:
        if var_name in cons:
            _var_types.extend(
                [
                    self._var_name_to_var_type.get(_var_name)
                    for _var_name in cons.keys()
                    if var_name != _var_name
                ]
            )

    if not _var_types:
        var_context_types = {VAR_TYPE_CONTINUOUS: 0, VAR_TYPE_BINARY: 0, VAR_TYPE_INTEGER: 0}
    else:
        var_context_types = Counter(_var_types)

    not_represented_var_types: t.Set[str] = {
        VAR_TYPE_CONTINUOUS,
        VAR_TYPE_BINARY,
        VAR_TYPE_INTEGER,
    }.difference(set(_var_types))

    if not_represented_var_types:
        for var_type in not_represented_var_types:
            var_context_types.update({var_type: 0})

    return {var_name: var_context_types}

def build_var_context_types(
    self,
    var_names: t.List[str],
    cons: t.Iterable[t.Tuple[str, dict]],

```

```

batch_size: int = 2_000,
max_n_threads: int = 100,
) -> pd.DataFrame:
    """
    Builds features for var context types in parallel mode
    """
    with ThreadPoolExecutor(max_workers=max_n_threads) as executor:
        to_do: t.List[Future] = []

        for var_name in tqdm(var_names):
            future: Future = executor.submit(self._get_var_context_types, conss, var_name)
            to_do.append(future)

        results: t.List[dict] = []
        for future in as_completed(to_do):
            result = future.result()
            results.append(result)

    dfs: t.List[pd.DataFrame] = []
    for batch_idx in tqdm(range(math.ceil(len(results) / batch_size))):
        _part = results[batch_idx * batch_size : (batch_idx + 1) * batch_size]
        dfs.append(pd.DataFrame.from_dict(ChainMap(*_part), orient="index"))

    _features = pd.concat(dfs, axis=0)
    _features.columns = [f"{col_name.lower()}_type_context" for col_name in _features.columns]

    return _features

```

## 35.2. Процессы, потоки и GIL в Python

Выдержка из книги Л. Рамальо [4, стр. 650]:

- Каждый экземпляр интерпретатора Python является процессом. Дополнительные процессы Python можно запускать с помощью библиотек `multiprocessing` или `concurrent.futures`.
- Интерпретатор Python использует единственный поток, в котором выполняется и пользовательская программа, и сборщик мусора. Для запуска дополнительных потоков предназначены библиотеки `threading` и `concurrent.futures`.
- Только один поток может выполнять Python-код, и от числа процессорных ядер это не зависит.
- Любая стандартная библиотечная функция Python, делающая системный вызов, освобождает GIL. Сюда относятся все функции, выполняющие дисковый ввод-вывод, сетевой ввод-вывод, а также `time.sleep()`. Многие счетные функции в библиотеках `numpy/scipy`, а также функции сжатия и распаковки из модулей `zlib` и `bz2` также освобождают GIL.
- Влияние GIL на сетевое программирование с помощью потоков Python сравнительно невелико, потому что функции ввода-вывода освобождают GIL, а чтение или запись в сеть всегда подразумевает высокую задержку по сравнению с чтением-записью в память. Следовательно, каждый отдельный поток все равно тратит много времени на ожидание, так что их выполнение можно чередовать без заметного снижения общей пропускной способности.
- Состязание за GIL замедляет работу счетных потоков в Python. В таких случаях последовательный однопоточный код проще и быстрее.
- Для выполнения счетного Python-кода на нескольких ядрах нужно использовать несколько процессов Python.

Деталь реализации CPython. В CPython, из-за глобальной блокировки интерпретатора, в каждый момент времени Python-код может выполняться только одним потоком (хотя некоторые высокопроизводительные библиотеки умеют обходить это ограничение). Если вы хотите, чтобы приложение более эффективно использовало вычислительные ресурсы многоядерных машин, то пользуйтесь модулем `multiprocessing` или классом `concurrent.futures.ProcessPoolExecutor`. Однако многопоточное выполнение все же является вполне пригодной моделью, если требуется одновременно выполнять несколько задач с большим объемом ввода-вывода [4, стр. 652].

По умолчанию *сопрограммы* вместе с *управляющим циклом событий*, который предоставляется каркасом асинхронного программирования, работают в *одном потоке*, поэтому GIL не оказывает на них никакого влияния. Можно использовать несколько потоков в асинхронной программе, но рекомендуется, чтобы и цикл событий, и все сопрограммы исполнялись в одном потоке, а дополнительные потоки выделялись для специальных задач.

### 35.3. Глобальная блокировка интерпретатора

Интерпретатор защищен так называемой глобальной блокировкой интерпретатора (GIL), которая позволяет *только одному потоку* Python выполняться в любой конкретный момент времени [2, стр. 503].

Наиболее заметный эффект GIL в том, что многопоточные программы Python не могут полностью воспользоваться преимуществами многоядерных процессоров (тяжелые вычислительные задачи, использующие больше одного потока, работают только на одном ядре процессора) [2, стр. 503].

GIL влияет только на программы, сильно нагружающие CPU (то есть те, в которых вычисления доминируют). Если ваша программа в основном занимается вводом-выводом, что типично для сетевых коммуникаций, потоки часто являются разумным выбором, потому что они проводят большую часть времени в ожидании.

## 36. Проверка существования путей в dataclass

Для того чтобы при чтении конфигурационного файла проекта, выполнялась проверка существования путей, следует задекорировать класс-схему следующим образом <https://harrisonmorgan.dev/2020/04/27/advanced-python-data-classes-custom-tools/>

```
def validated_dataclass(cls):
    """
    Class decorator for validating fields
    """
    cls = dataclass(cls)

    def _set_attribute(self, attr, value):
        for field in fields(self):
            if field.name == attr and "validator" in field.metadata:
                value = field.metadata["validator"](value)
                break

        object.__setattr__(self, attr, value)
        cls.__setattr__ = _set_attribute

    return cls
```

```

@validated_dataclass
class Paths:
    path_to_test_lp_file: str = field(metadata={"validator": check_existence_path})
    path_to_set_file: str = field(metadata={"validator": check_existence_path})
    path_to_output_dir: str = field(metadata={"validator": check_existence_path})

def check_existence_path(path: str):
    path = pathlib2.Path(path)
    if not path.exists():
        raise FileNotFoundError(f"Path {path} not found ...")

    return path

```

## 37. Приемы работы с библиотекой SPyQL

SPyQL <https://github.com/dcmoura/spyql> – это утилита командной строки, позволяющая писать SQL-подобные запросы к csv-, json-файлам, с использованием выразительных средств Python.

Прочитать csv-файл и вывести первые две записи в json-формате

```

$ spyql "SELECT * FROM csv LIMIT 2 TO json(indent=2)" < features_a78cbead_bin.csv
{
  "var": "alpha_tu_0_1_12_1",
  "scenario": "a78cbead_bin",
  "varBinaryOriginal": 1,
  "varTypeTrans": 0,
  "varStatus": 1,
  "varMayRoundUp": 0,
  "varMayRoundDown": 0,
  "varMayIsActive": 1,
  "varIsDeletable": 0,
  "varIsRemovable": 0,
  "varObj": 0.0,
  "varPseudoSol": -0.0,
  "NLocksDown": 1,
  "NLocksUp": 1,
  "IsTransformed": 1,
  "multaggrConstant": 0,
  "varAggrScalar": 0,
  "varAggrConstant": 0,
  "varMultaggrNVars": 0,
  "varBestBound": -0.0,
  "varWorstBound": 1.0,
  "varBranchFactor": 1,
  "varBranchPriority": 0,
  "varBranchDirection": 3,
  "varNImpls0": 0,
  "varNImpls1": 0,
  "varGetNCliques0": 0,
  "varGetNCliques1": 0,
  "varConflictScore": 1e-12,
  "varAvgInferenceScore": 87.0136,
  "relaxSolVal": 0.458516,
  "varImplRedcost0": 0.0,
  "varImplRedcost1": 0.0,
  "varPseudocostScore": 0.248279,

```



```

    "equalToLb": 0,
    "equalToUb": 0,
    "target": 1
}
...

```

Прочитать csv-файл, сгруппировать по полю `varStatus`, а затем из результата выбрать строки, в которых `varStatus > 2`

```

$ cat features_a78cbead_bin.csv \
  | spyql "SELECT varStatus AS status, count_agg(*) AS count FROM csv GROUP BY 1 TO spy" \
  | spyql "SELECT * FROM spy WHERE status > 2 ORDER BY 2 DESC TO pretty"

```

status	count
3	8361
4	552

## 38. Приемы работы с библиотекой Pandas

### 38.1. Общие замечания

Как отмечается в библиотеке `pandarallel` <https://nalepae.github.io/pandarallel/> основной недостаток библиотеки `pandas` заключается в том, что она может *утилизировать только одно ядро процессора*, даже если доступно несколько ядер.

Библиотека `pandarallel` может использовать все доступные ядра процессора, однако ей требуется в два раза больше памяти, чем `pandas`. Не рекомендуется использовать `pandarallel`, если `pandas`-данные не помещаются в память. В этом случае лучше подойдет Spark.

Библиотека Spark позволяет работать с данными, которые *значительно превышают доступную память* (Handle data much bigger than your memory) и может распределять вычисления по нескольким узлам кластера.

### 38.2. Советы по оптимизации вычислений

В ситуации, когда необходимо итерирование, более быстрым способом итерирования строк будет использование метода `.iterrows()`. Метод `.iterrows()` оптимизирован для работы с кадрами данных, и хотя это наименее эффективный способ большинства стандартных функций, он дает значительное улучшение, по сравнению с базовым итерированием [5, стр. 328]

```

haversine_series = []
for index, row in df.iterrows():
    haversine_series.append(haversine(...))
df["distance"] = haversin_series

```

Однако метод `.iterrows()` не сохраняет типы по строкам. Если требуется сохранять типы атрибутов строки, то лучше воспользоваться методом `.itertuples()`, который поддерживает итерирование по строкам в виде именованных кортежей. Кроме того часто `.itertuples()` оказывается быстрее `.iterrows()`.

Более эффективным способом является использование метода `.apply()`, который применяет функцию вдоль определенной оси (вдоль строк или вдоль столбцов) кадра данных.

Хотя метод `.apply()` также по своей сути *перебирает строки* (!), он делает это намного эффективнее, чем метод `.iterrows()`, используя ряд внутренних оптимизаций, например, применяя итераторы, написанные на Cython [5, стр. 328]

```
df["distance"] = df.apply(lambda row: haversine(..., ..., row["latitude"], row["longitude"]),
                           axis=1)
```

Но гораздо эффективнее задействовать *векторизацию* и передать не скаляры, а столбцы

```
df["distance"] = haversine(..., ..., df["latitude"], df["longitude"])
```

Если скорость имеет наивысший приоритет, можно вместо серий использовать *numpy-массивы*. Как и `pandas`, `numpy` работает с массивами. Однако она освобождена от дополнительных вычислительных затрат, связанных с операциями в `pandas`, такими как индексирование, проверка типов данных и т.д. В результате операции над массивами `numpy` могут выполняться значительно быстрее, чем операции над объектами `Series`.

Массивы `numpy` можно использовать вместо объектов `Series`, когда дополнительная функциональность, предлагаемая объектами `Series`, не является критичной. Например, векторизованная реализация функции `haversine` фактически не использует индексы в сериях `longitude` и `latitude`, и поэтому отсутствие этих индексов не приведет к нарушению работы функции

```
df["distance"] = haversine(..., ..., df["latitude"].values, df["longitude"].values)
```

Оптимизацию числовых столбцов можно выполнить с помощью *понижающего преобразования*, используя функцию `pd.to_numeric`

```
df.select_dtypes(np.dtype("int64")).apply(
    pd.to_numeric, # функция, которая применяется к int-столбцам
    downcast="unsigned" # аргумент функции pd.to_numeric
)
```

В значительной степени снижение потребления памяти будет зависеть от оптимизации столбцов типа `object`. Тип `object` представляет значения, использующие питоновские объекты-строки, отчасти это обусловлено отсутствием поддержки пропущенных строковых значений в `numpy`. Python не предполагает точной настройки способа хранения значений в памяти. Это ограничение приводит к тому, что строки хранятся фрагментированно, это потребляет больше памяти и замедляет доступ. Каждый элемент в столбце типа `object` является, по сути, указателем, который содержит «адрес» фактического значения в памяти [5, стр. 347].

Преобразовать столбец типа `object` в столбец типа `category` можно так

```
df["object_col_name"].astype("category")
```

Хотя каждый указатель занимает 1 байт памяти, каждое фактическое строковое значение использует такой объем памяти, какой строка использовала бы, если бы отдельно хранилась в Python.

Тип `category` под капотом для представления строковых значений в столбце вместо исходных использует целочисленные значения. Для этого создается отдельный словарь, в котором исходным значениям сопоставлены целочисленные значения. Это сопоставление будет полезно для столбцов с небольшим числом уникальных значений.

Рекомендуется придерживаться типа `category` при работе с такими столбцами `object`, в которых менее 50% значений являются уникальными. Если все значения в столбце являются уникальными, тип `category` будет использовать больший объем памяти. Это обусловлено тем, что

в столбце, помимо целочисленных кодов, представляющих категории, хранятся все исходные строковые значения.

### 38.3. Рецепты

#### 38.3.1. Приемы работы с кадрами данных

Построить кадр данных заполненный NaN

```
df = pd.DataFrame(np.nan, index=range(10), columns=["col1", "col2", "col3"])
```

Ремарка: с помощью `scipy.sparse.scr_matrix` можно создавать огромные разреженные матрицы

```
from scipy.sparse import csr_matrix

mtx = csr_matrix((300_000, 30_000), dtype=np.int8)
```

Еще для создания разреженных матриц можно воспользоваться функцией `scipy.sparse.lil_matrix`, которая создает разреженные матрицы инкрементно (поэтапно) и представляет список списков разреженных матриц.

Например, индексы столбцов в строке номер 100, элементы которых равны единице можно получить так

```
from scipy.sparse import lil_matrix

mtx = lil_matrix((300_000, 30_000), dtype=np.int8)
mtx[100:300, 200:250] = 1
(mtx[100, :] == 1).indices
```

Применить регулярное выражение к строковому атрибуту кадра данных, а затем сделать его вещественным можно так

```
df["col_name"].str.extract(r"^\.*?(\d+[.]?(\d+)s$)").astype(np.float32)
```

Вывести точную информацию об использовании памяти

```
df.info(memory_usage="deep")
"""
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 3 columns):
#   Column  Non-Null Count  Dtype
---  -
0   key1    1000 non-null    float64
1   key2    1000 non-null    int64
2   color   1000 non-null    object
dtypes: float64(1), int64(1), object(1)
memory usage: 75.3 KB
"""
```

Посмотреть какие строки значений (а не индексы) кадра данных попали в ассоциированные группы

```
df.groupby("color").groups.keys()
```

Заполнить пропущенные значения групповым средним по столбцу. Метод `apply` в случае сгруппированных объектов применяет переданную функцию (в данном случае анонимную) к каждой группе, а внутри группы операции применяются вдоль указанных осей

```
# Нужно отобрать поля, к которым будет применяться функция
df["year"] = df.groupby("color")["year"].apply( # .loc[:, "year"] НЕ РАБОТАЕТ!
    lambda group: group.fillna(group.mean())
)
```

Для того чтобы метод `apply` корректно работал на объекте групп нужно указать с какими полями мы будем работать

```
# Указываем поля "year" и "mark"
df.groupby("model_car_id")["year", "mark"].apply(lambda gr: gr.fillna(gr.mean()))
# или
df.groupby("model_car_id")["year", "mark"].transform(lambda gr: gr.fillna(gr.mean()))
```

Метод `transform` объекта `GroupBy` применяет указанную функцию к каждой группе, а затем помещает результаты в нужные места [3, стр. 291].

В самом простом случае метод `transform` применяет переданную функцию вдоль указанного направления и для каждого элемента возвращает результат преобразования, а в случае если метод `transform` вызывается на `GroupBy`-объекте, то метод применяет указанную функцию для каждой группы и «заменяет» каждый элемент своей группы групповым агрегатом или результатом преобразования (причем для каждого столбца вычисляется свой агрегат)

```
# каждый элемент групп будет заменен количеством элементов в группе
df.groupby("color")["elems"].transform(len)
```

Другими словами, метод `transform` на сгруппированном объекте в том подкадре данных, который возвращается методом, каждый элемент группы «заменяет» групповым агрегатом (или результатом преобразования), а метод `apply` просто применяет указанную функцию к каждой группе [3, стр. 292] и склеивает результаты, т.е. возвращает результат для каждой группы

```
$ df.groupby("color")["a", "e"].transform(lambda gr: gr.mean())
# В столбце 'a' для элементов, попавших в группу, среднее было 49.377..., поэтому эти элементы з
# аменены на соответствующее групповое среднее
```

	a	e
0	49.377209	49.611246
1	49.950178	49.839233
2	49.730188	48.043373
3	49.730188	48.043373
4	49.950178	49.839233
...	...	...
9995	49.377209	49.611246
9996	49.377209	49.611246
9997	49.377209	49.611246
9998	49.950178	49.839233
9999	49.950178	49.839233

```
$ df.groupby("color")["a", "e"].apply(lambda gr: gr.mean())
```

	a	e
color		
blue	49.730188	48.043373
green	49.950178	49.839233
red	49.377209	49.611246

Получается, что ключевое отличие метода `transform` от метода `apply` на `GroupBy`-объектах заключается в том, что `transform` преобразует элементы группы, а метод `apply` просто разбивает кадр данных на группы, применяет указанную функцию к каждой группе, а затем пытается склеить результаты, то есть это что-то вроде концепции `map-reduce`.

Например, если требуется создать новый столбец, элементы которого помечаются меткой "old", если элемент меньше группового среднего и – меткой "new", если элемент больше группового среднего, то можно решить эту задачу с помощью метода `transform`

```
df["avg_a"] = df.groupby("color")["a"].transform(np.mean)
df["age"] = np.where(df["a"] < df["avg_a"], "old", "new")
```

То есть еще раз, метод `transform` применяет указанную функцию (`np.mean`) к каждой группе, а затем возвращает подкадр данных, в котором каждый элемент заменяется групповым агрегатом.

Найти среднее и стандартное отклонение по группам для вещественных столбцов кадра данных

```
df.groupby("label") [
    df.select_dtypes(np.dtype("float64")).columns
].agg([np.mean, np.std]).stack()
```

При проведении разведочного анализа данных лучше всего сначала загрузить данные и исследовать их с помощью запросов/логического отбора. Затем создайте индекс, если ваши данные поддерживают его или если вам требуется повышенная производительность [5, стр. 115]. Операции поиска с использованием индекса обычно выполняются быстрее. В силу лучшей производительности выполнение поиска по индексу (в тех случаях, когда это возможно) обычно является оптимальным решением. Недостаток использования индекса заключается в том, что потребуется время на его создание, кроме того, он занимает больше памяти.

Выполнить слияние кадров данных можно с помощью функции `pd.merge` или метода `.merge`. По умолчанию слияние выполняется по *общим меткам столбцов*, однако сливать кадры данных можно и *по строкам с общими индексами* [5, стр. 230]

```
# Слияние по строкам
# Нужно задать оба параметра!
left.merge(right, left_index=True, right_index=True)
```

Кроме того, библиотека `pandas` предлагает метод `.join()`, который можно использовать для выполнения соединения с помощью *индексных меток* двух объектов `DataFrame` (вместо значений столбцов) [5, стр. 232]

```
# Слияние по строкам
# Здесь предполагается, что кадры данных имеют
# дублирующиеся имена столбцов, поэтому мы задаем lsuffix и rsuffix
left.join(right, lsuffix="_left", rsuffix="_right")
```

---

#### Замечание

Метод `.join()` по умолчанию используется *внешнее соединение*, в отличие от метода `.merge()`, в котором по умолчанию применяется *внутреннее соединение*.

---

*Состыковка (stack)* помещает уровень индекса столбцов в новый уровень индекса строк

```
df = pd.DataFrame({
    "a": [1, 2],
    "b": [100, 200]
})
"""
   a    b
one 1  100
```

```
two 2 200
"""
df.stack()
"""
one a 1
   b 100
two a 2
   b 200
"""
df.loc[("one", "b")] # 100
```

Состыковку удобно применять к результатам агрегации на группах

```
df.groupby("color")[["key1", "key4"]].agg([np.mean, np.std])
"""
           key1          key4
        mean      std mean      std
color
blue  0.904027  0.508690  73.5  21.920310
green -0.493756  1.025554  65.0   9.899495
red   -0.399363      NaN  55.0      NaN
"""
# Состыковка
res = df.groupby("color")[["key1", "key4"]].agg([np.mean, np.std]).stack()
"""
           key1          key4
color
blue mean  0.904027  73.500000
      std  0.508690  21.920310
green mean -0.493756  65.000000
      std  1.025554   9.899495
red   mean -0.399363  55.000000
"""
res.loc[("blue", "mean")]
"""
key1    0.904027
key4    73.500000
Name: (blue, mean), dtype: float64
"""
```

При построении агрегатов со сложным именем можно воспользоваться псевдонимами

```
df.groupby("color")[["key1", "key2"]].agg([("MEAN", np.nanmean), ("STD", np.nanstd)]).stack()
"""
           key1          key2
color
blue MEAN  0.544329  0.731969
green MEAN  0.231420  1.272040
      STD      NaN  1.255945
red   MEAN -0.399363  0.483054
"""
```

*Расстыковка* (unstack) помещает самый внутренний уровень индекса строк в новый уровень индекса столбцов.

*Расплавление* – это тип организации данных, который часто называют преобразованием объекта DataFrame из «широкого» формата в «длинный» формат.

```
data = pd.DataFrame({
    "Name": ["Mike", "Mikal"],
    "Height": [6.1, 6.0],
```

```

    "Weight": [220, 185],
})
data
"""
    Name  Height  Weight
0   Mike     6.1    220
1 Mikael     6.0    185
"""

```

Расплавляем кадр данных

```

pd.melt(
    data,
    id_vars=["Name"],
    value_vars=["Height", "Weight"]
)
"""
    Name variable  value
0   Mike   Height    6.1
1 Mikael   Height    6.0
2   Mike   Weight   220.0
3 Mikael   Weight   185.0
"""

```

Получить данные по группе

```
df.groupby("color").get_group("blue")
```

Отфильтровать группы по условию. Если функция возвращает True, то группа включается в результат

```
df.groupby("color").filter(lambda group: group.col_name.count() > 1)
```

### 38.3.2. Изменение настроек отдельной линии графика на базе кадра данных

Чтобы изменить, например, толщину линии для какого-то заданного столбца кадра данных нужно получить доступ к перечню линий `ax.get_lines()`

```

fig, ax = plt.subplots(figsize=(15, 5))

df.plot(ax=ax, marker="o", style=["b--", "k-", "r-"])

for line in ax.get_lines():
    if line.get_label() == "col1":
        line.set_linewidth(3.5)
        line.set_alpha(0.8)
        line.set_marker("x")

```

### 38.3.3. Использование регулярных выражений и обращений по имени группы при обработке строк

Привести столбец строкового типа к числовому типу с предварительной подготовкой строки по регулярному выражению можно так

```

pd.to_numeric(
    logs.loc[:, "time"].replace( # HE .str.replace!
        to_replace=r"^\.*?(\d+)\.*?$",
        value=r"\1", # обращение к первой группе

```

```
)  
    regex=True,  
)
```

## Список литературы

1. Бизли Д. Python. Подробный справочник. – СПб.: Символ-Плюс, 2010. – 864 с.
2. Бизли Д. Python. Книга рецептов. – М.: ДМК Пресс., 2019. – 648 с.
3. Маккинли У. Python и анализ данных, 2015. – 482 с.
4. Рамальо Л. Python – к вершинам мастерства: Лаконичное и эффективное программирование. – М.: МК Пресс, 2022. – 898 с.
5. Хейдт М., Груздев А. Изучаем pandas. – М.: ДМК Пресс, 2019. – 682 с.
6. Хостманн К. Scala для нетерпеливых. – М.: ДМК Пресс, 2013. – 408 с.