Дата последней сборки документа:
24 октября 2022 г. в 03:42

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

Содержание

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 интсрукция `a = 42`, интерпретатор создает целочисленный объект со значением 42. Можно рассматривать идентичность объекта как указатель на область памяти, где находится объект, а индентификатор `a` – как имя, которое ссылается на эту область памяти.

Тип объекта сам по себе является *объектом*, который называется *классом объекта*. Все объекты в яызке 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. Объкт `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. Передача параметров и возвращаемые значения

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 передачи изменяемого объекта (такого как список или словарь), который модифицируется функцией, эти изменения будут отражаться на исходном объекте [1, стр. 133].

3. 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 присвоины 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 встроенном 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 *встроенном 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 инструкция 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].

4. 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 соответсвтует 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.

5. Модули, пакеты и дистрибутивы

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 `*.pyc`. При всех последующих обращениях к импортированию этого модуля интепретатор будет загружать скомпилированный байт-код, если только с момента создания байт-кода в файл `.py` не вносились изменения (в этом случае файл `.pyc` будет создан заново).

Автоматическая компиляция программного кода в файл с расширением `.pyc` производиться 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 содержаться 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 `primiitves` 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 помешает. 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 ситуациюю 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.

5.1. 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.

6. Некоторые приемы

6.1. 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 обратить 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 мнгожество 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"]
```

6.2. Удаление дубликатов из последовательности

Вы хотите ислкючить дублирующиеся значения из последовательности, но при этом сохранить порядрк следования оставшихся элементов.

Если значения в последовательности являютеся 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
```

6.3. 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 itergetter

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=itergetter("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=itemtegger("lname", "fname"))
```

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

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

6.4. 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 взаимозаменям с кортежем и поддерживает все обычные операции кортежей, такие как индексирование и распаковка

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

Возможное использование именнованного 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)
```

7. Строки и текст

7.1. 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]\s*", line)
```

8. 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 нагруженность 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, **kwrags):
        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 contexlib import contexmanager

@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
```

9. Итераторы и генераторы

В большинстве случаев для прохода по итериуемому 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 вовзвращает 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`.

10. 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`, используемые 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 случиться 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
```

11. 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
```

12. 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.

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

Эффективный путь определинея 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 вызывать 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 экзмпляра* 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` можн 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_cacha
    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...
```

13. Определение более одного конструктора в классе

Вы пишите 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 {"raidus": self.radius, "color": self.color}
```

Одно из главных применений *методов класса* – это определение *альтренативных 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()
    ...
```

14. Параметрические декораторы

Требуется создать функцию-декторатор, 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`.

15. 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 фукнциям, 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]

16. Параллельное программирование

Библиотека `concurent.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 *сериализоватъся*,
- 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 клонирует состояние,
- Нужно с великой осторожностью объединять пулы процессов с программами, которые используют потоки.

16.1. Процессы, потоки и GIL в Python

Выдержка из книги Л. Рамальо [3, стр. 650]:

- Каждый *экземпляр интрепретатора 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, сетеовой 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 вполнен пригодной моделью, если требуется одновременно выполнять несколько задач с большим объемом ввода-вывода [3, 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.

16.2. 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.

17. Приемы работы с библиотекой Pandas

17.1. 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 страндартных функций, он дает значительное улучшение, по сравнению с базовым итерированием [4, стр. 328]

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

Более эффктивным способом является использование метода `.apply()`, который применяет функцию вдоль определеной 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 [4, 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 `langitude` 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
    downcase="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 [4, 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 бо́льший объем памяти. Это обусловлено тем, что в столбце, помимо целочисленных кодов, представляющих категории, храняться все исходные строковые значения.

17.2. Рецепты

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").apply(lambda group: group["col_name"])
"""
   color  col_name
0   blue  73.0
1  green  52.0
2  green   NaN
3  green  77.0
4   blue   NaN
5  green  86.0
6    red  73.0
7    red   NaN
8  green  77.0
9    red  81.0
"""

# вывести индексы строк, по соответсвующим группам можно с помощью атрибута groups
df.groupby("color").groups
# {'blue': [0, 4], 'green': [1, 2, 3, 5, 8], 'red': [6, 7, 9]}
```

Заполнить пропущенные значения средним на сгруппированном по некоторому полю кадре данных

```
df.groupby("color").apply(
    lambda group: group.fillna(group.mean())
)
```

Найти среднее и страндартное 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 [4, 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* [4, 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) [4, 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 *внутренее 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 аггрегации 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  Heigth  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   Heigth    6.1
1  Mikael   Heigth    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)
```

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. *Рамальо Л.* Python – к вершинам мастерства: Лаконичное и эффективное программирование. – М.: МК Пресс, 2022. – 898 с.
4. *Хейдт М., Груздев А.* Изучаем pandas. – М.: ДМК Пресс, 2019. – 682 с.