


# Общие принципы оптимизации кода Python



☆ trade-pioneer.ru

РЕКЛАМА

## Купить поливинилхлорид оптом

4,5

★ Рейтинг организации

ⓘ

[Узнать больше](#)

[Справочник по языку Python3.](#) / Общие принципы оптимизации кода Python

### Цитата:

"Если необходимо, чтобы Ваш код работал быстрее, то вероятно, следует просто [использовать PyPy](#)"

Гвидо ван Россум (создатель Python).

Любой код, в первую очередь, должен быть написан обдуманно и быть рабочим и только потом можно приступать к его дополнительной оптимизации. При этом оптимизация должна проходить прямо в процессе разработки, так как потом сложно будет изменить множество мелких моментов.

Оптимизация производительности в любом языке программирования не имеет четкого алгоритма действий. Иногда даже незначительные изменения в "нужном месте" могут ускорить работу кода Python в несколько раз. Оптимизация кода - это придание важности мелочам.

## Избегайте применение глобальных переменных.

Python очень медленно обрабатывает доступ к глобальным переменным, особенно внутри циклов. Также множественное применение глобальных переменных очень часто приводит к спагетти-коду, что вызывает дополнительные трудности и проблемы.

## Использование сторонних модулей и библиотек.

Обращайте внимание на каком языке написан сторонний модуль или библиотека. Если есть возможность, то используйте библиотеки, написанные на C, так как они работают быстрее, а это значит, что и программа на Python будет работать немного быстрее.

## Активно используйте встроенные инструменты Python.

Их применение ускоряет код за счет того, что они предварительно оптимизированы, скомпилированы и следовательно выполняются быстрее.

Примером таких инструментов может быть:

- встроенная функция [map\(\)](#);
- встроенная функция [str.join\(\)](#);
- встроенный модуль [itertools](#);
- и многое другое.

## Работа над кодом.

- пишите код обдуманно и максимально лаконично;
- по возможности, внедряйте кэширование объектов;
- не создавать лишние экземпляры объектов (помните, объекты потребляют дополнительную память);

[Перевод статьи словацкого разработчика Мартина Хайнца](#), в которой он описывает девять практических советов о том, как [сделать разработку на Python лучше](#).

## Обрабатывайте входные данные.

Чем больше размер программы, тем выше шансы пропустить уязвимость в коде. Один из способов обезопасить себя от возможных ошибок - очистка входных данных перед выполнением программы (*input sanitization*). В большинстве случаев при таком подходе достаточно поменять регистр символов или использовать регулярные выражения. Но для сложных случаев есть и более эффективный способ:

```
user_input = "This\nstring has\tsome whitespaces...\r\n"

character_map = {
    ord('\n') : ' ',
    ord('\t') : ' ',
    ord('\r') : None
}

user_input.translate(character_map)
```

Это пример заменяет "пробельные" символы `\n` и `\t` обычным пробелом и удаляет `\r` (все перечисленные конструкции обозначают разные виды пробелов). В зависимости от задач, можно генерировать таблицы соответствий разного размера. Задачу облегчает [встроенный модуль unicodedata](#) и функция `combining()` для генерации и отображения. Их можно использовать для удаления всех акцентов из строки.

## Используйте итераторы со срезами.

[Итератор](#) - это инструмент для поточной обработки данных. Он отвечает за упрощение навигации по элементам: списку, словарю и так далее. Это такой объект-перечислитель, который выдаёт следующий элемент. В основном его используют в [цикле for/in](#).

Но использовать итератор на полную мощность нужно не всегда. И тут задача: если попытаться использовать [срез](#) итератора, то получим [ошибку TypeError](#). Это произойдёт из-за того, что объект итератора не является подписываемым. К счастью, на такой случай есть простое решение:

```
import itertools

s = itertools.islice(range(50), 10, 20)
for val in s:
    ...
```

Используя [itertools.islice\(\)](#), можно создать объект `islice`, который сам по себе является итератором, производящим нужные значения. Важно отметить, что этот объект использует все элементы генератора вплоть до начала среза, что делает `itertools.islice()` мощным инструментом.

## Пропускайте начало итерируемого объекта.

Иногда приходится работать с файлами, которые начинаются с неизвестного количества бесполезных строк, например, с комментариев. И тут [модуль itertools](#) снова предлагает простое решение:

```
string_from_file = """ // Автор: ...
// Лицензия: ...
//
// Дата: ...

Содержание...
"""

import itertools

for line in itertools.dropwhile(lambda line: line.strip().startswith('//'),
                               string_from_file.splitlines()):
    print(line)
```

Этот фрагмент кода создаёт строки, пропуская начальные комментарии. Такой подход может быть полезен, если нужно отбросить элементы (в нашем случае строки) в начале итерируемого объекта.

## Используйте kwargs.

Если при разработке программы нужно выполнить несколько похожих действий, то лучшее решение - определить функции для многократного использования в коде. Для этого создается функция с аргументами. Но что делать, если аргументы функции определены, а для ее многократного использования (унификации) нужно передавать разное количество аргументов? Для этого можно использовать [ключевые аргументы функции kwargs](#).

Этот инструмент очень пригодится для создания функции только с именованными аргументами. Это даст возможность (или, скорее, обяжет) [использовать такие функции более прозрачно](#):

```
def test(*, a, b):
    pass

test("value for a", "value for b")
# Traceback (most recent call last):
#   File "ad.py", line 4, in <module>
#     test("value for a", "value for b")
# TypeError: test() takes 0 positional arguments but 2 were given

test(a="value1", b="value2")
# Работает...
```

Как видно из примера, задачу легко решить, если поместить аргумент \* перед ключевыми словами. И, конечно, можно использовать позиционные аргументы, если вставить их до аргумента \*.

## Используйте объекты, которые поддерживают оператор with.

Открыть файл и заблокировать фрагмент кода можно с помощью оператора with, но можно ли сделать это, пользуясь собственным методом? Да, можно реализовать [протокол context manager](#), используя методы `__enter__` и `__exit__`:

```
class Connection:
    def __init__(self):
        ...

    def __enter__(self):
        # инициализация соединения

    def __exit__(self, type, value, traceback):
        # закрытие соединения

# использование
with Connection() as conn:
    # __enter__() executes
    '''
    # conn.__exit__() executes
```

Это распространённый вариант управления контекстом в Python, но есть и более простой способ:

```
from contextlib import contextmanager

@contextmanager
def tag(name):
    print (f"<{name}>")
    yield
    print (f"</{name}>")

with tag("h1"):
    print("This is Title.")
```

Этот фрагмент кода реализует протокол управления контекстом, используя декоратор менеджера [contextmanager](#). Первая часть функции tag() (до [yield](#)) выполняется при входе в блок with, затем исполняется блок, а после него и остальная часть функции tag().

## Сохраните всё с помощью `__slots__`.

Если программа создаёт большое количество экземпляров какого-либо класса, то она может потребовать больше памяти. Это связано с тем, что Python использует словари для представления атрибутов экземпляров классов. Это делает язык быстрым, но не очень эффективным с точки зрения оптимизации памяти. Если это становится проблемой, то поможет магический [атрибут `\_\_slots\_\_`](#):

```
class Person:
    __slots__ = ["first_name", "last_name", "phone"]
    def __init__(self, first_name, last_name, phone):
        self.first_name = first_name
        self.last_name = last_name
        self.phone = phone
```

При определении атрибута `__slots__` Python использует небольшой массив фиксированного размера для атрибутов вместо словаря. Это значительно сокращает объём памяти, необходимый для каждого экземпляра.

Следует учесть, что в этом случае есть и недостатки: нельзя объявлять какие-либо новые атрибуты помимо используемых в `__slots__`, а классы со `__slots__` не могут использовать множественное наследование.

## Ограничьте использование процессора и памяти.

Если лень оптимизировать память программы или корректировать работу процессора, то можно просто установить лимиты. К счастью, в Python для этого есть специальный [модуль `resource`](#):

```
import signal
import resource

def time_exceeded(signo, frame):
    print("CPU exceeded...")
    raise SystemExit(1)

def set_max_runtime(seconds):
    soft, hard = resource.getrlimit(resource.RLIMIT_CPU)
    resource.setrlimit(resource.RLIMIT_CPU, (seconds, hard) )
    signal.signal(signal.SIGXCPU, time_exceeded)

def set_max_memory(size):
    soft, hard = resource.getrlimit(resource.RLIMIT_AS)
    resource.setrlimit(resource.RLIMIT_AS, (size, hard))
```

Здесь можно увидеть две опции: установку на максимальное процессорное время и максимальный предел используемой памяти.

При ограничении работы процессора необходимо извлечь мягкий и жёсткий лимиты для конкретного ресурса (`resource.RLIMIT_CPU`), а затем установить его значение. Для этого используется количество секунд, указанное в аргументе, и ранее полученное жёсткое ограничение. В конце нужно зарегистрировать сигнал, который будет отвечать за выход из системы, если процессорное время превышено.

Что касается памяти, то, как и в случае с процессором, устанавливаем мягкий и жёсткий лимиты. Для этого используется функция [resource.setrlimit\(\)](#) с аргументом `size` и жёсткое ограничение, которое было получено.

## Управляйте экспортом элементов.

Такие языки программирования, как Go, имеют механизм экспорта только для элементов (переменных, методов, интерфейсов) начинающихся с заглавной буквы. В Python подобного можно добиться с помощью [переменной `\_\_all\_\_`](#):

```
def foo():
    pass

def bar():
    pass

__all__ = ["bar"]
```

В данном случае, благодаря `__all__` экспортирован будет не весь код, а только функция `bar()`. Кроме того, можно оставить переменную пустой, то при попытке импорта из этого модуля ничего не попадёт в экспорт, что приведёт к ошибке `AttributeError`.

## Упростите использование операторов сравнения.

Использовать все операторы сравнения для одного класса может быть довольно сложно, учитывая, что их немало: `__lt__`, `__le__`, `__gt__` или `__ge__`. Но есть ли более простой способ сделать это? Здесь поможет [functools.total\\_ordering](#):

```
from functools import total_ordering

@total_ordering
class Number:
    def __init__(self, value):
        self.value = value

    def __lt__(self, other):
        return self.value < other.value

    def __eq__(self, other):
        return self.value == other.value

print(Number(20) > Number(3))
print(Number(1) < Number(5))
print(Number(15) >= Number(15))
print(Number(10) <= Number(2))
```

Как это работает? Декоратор `@total_ordering` автоматически добавляет все остальные методы. В этом случае нужно только определить `__lt__` и `__eq__`, а все остальные пробелы за нас заполнит декоратор.

ХОЧУ ПОМОЧЬ  
ПРОЕКТУ

РЕКЛАМА • 18+

practicum.yandex.ru

**Бесплатное  
занятие  
английским  
в Яндекс  
Практикуме**

Полноценное занятие  
с преподавателем, а не  
презентация курсов

Устный тест на уровень  
языка >

Практика английского >

Узнать больше