

# Работа с кодом

Борисенко Глеб, 14.09.2023

# Для кого/чего пишется код?

**Код пишется для решения задачи разработчиками сейчас и в будущем.**

# Пример итеративного решения задачи

- Рассмотрим простую задачу — проверку наличия элемента в отсортированном массиве
- Начнём с такого кода:

```
def f(a, x):  
    if len(a) in [0]:  
        return False  
    i = len(a) // 2  
    y = a[i]  
    if y <= x:  
        return f(a[i:], x)  
    else:  
        return f(a[:i], x)
```

Во-первых, код должен быть хотя бы минимально понятным

```
def search(arr, val):  
    if not arr:  
        return False  
    mid = len(arr) // 2  
    if arr[mid] <= val:  
        return search(arr[mid:], val)  
    else:  
        return search(arr[:mid], val)
```

Во-вторых, код должен быть рабочим :)

```
def search(arr, val):  
    if len(arr) <= 1:  
        return val in arr  
    mid = len(arr) // 2  
    if arr[mid] < val:  
        return search(arr[(mid+1):], val)  
    else:  
        return search(arr[:mid], val)
```

В третьих, код должен быть достаточно эффективным, **если того требует задача**

```
def search(arr, val):  
    def _search(left, right):  
        if (left + 1 >= right):  
            return left != right and arr[left] == val  
        mid = (left + right) // 2  
        if arr[mid] < val:  
            return _search(mid + 1, right)  
        else:  
            return _search(left, mid)  
    return _search(0, len(arr))
```

В четвёртых, код должен быть достаточно оптимизирован, **если того требует задача**

```
def search(arr, val):  
    left, right = 0, len(arr)  
    while (left + 1 < right):  
        mid = (left + right) // 2  
        if arr[mid] < val:  
            left = mid + 1  
        else:  
            right = mid  
    return left != right and arr[left] == val
```

# Приоритет при написании кода

1. Понятно что происходит
2. Происходит то, что нужно
3. Код достаточно оптимален при ограничениях задачи (алгоритм, оптимизация)

**Примечание 0.** Первые два пункта могут меняться местами

**Примечание 1.** Обратите внимание, что часто лучше сначала сделать работающую версию, а только потом заниматься оптимизацией.

**Примечание 2.** Если вы за дополнительные 10 минут можете написать более эффективную версию кода, которая не противоречит предыдущим двум правилам, то имеет смысл это сделать. Иначе лучше дождаться необходимости.



# Как улучшить свой код?

## **Знание языка**

Понятность +

Функциональность +++

Эффективность +++

Поддержка +

## **Паттерны/Шаблоны проектирования**

Понятность ++

Функциональность +

Эффективность +

Поддержка ++

## **Оптимизация / Алгоритмистика**

Понятность -

Функциональность ++

Эффективность +++

Поддержка -

# Знание языка



# Нужно знать особенности в поведении языка

- Что выведет этот код? Что можно было бы ожидать без учёта подвоха?

```
list_of_functions = list()
fst_value = 0
for snd_value in range(10):
    list_of_functions.append(
        lambda: (fst_value, snd_value)
    )
    fst_value += 1

for function in list_of_functions:
    print(function())
```

# Нужно знать особенности в поведении языка

```
list_of_functions = list()
fst_value = 0
for snd_value in range(10):
    list_of_functions.append(
        lambda: (fst_value, snd_value)
    )
    fst_value += 1

for function in list_of_functions:
    print(function())
```

(10, 9)  
(10, 9)  
(10, 9)  
(10, 9)  
(10, 9)  
(10, 9)  
(10, 9)  
(10, 9)  
(10, 9)  
(10, 9)

## Второй пример

- Что будет выводиться при многократных вызовах команды `python3 script.py` ?

```
for _ in range(3):  
    print(hash('10'), hash(10))
```

## Второй пример

- **Ответ** : будут выводиться 3 одинаковых пары чисел, но от запуска к запуску скрипта первое число будет меняться, а второе будет равно 10

# Немножко про классы

Типы полей в классах:

- Public – видны всем
- Protected – видны только внутри класса и в его наследниках
- Private – видны только внутри класса

Как сделать private или protected поля в Python?

А как всё равно потом к ним обратиться?

# Немножко про классы

## Ответ:

- для `protected` полей принято называть их, начиная с `_`, то есть `_field_name`;
- для `private` полей принято называть их, начиная с `__`, то есть `__field_name`, а интерпретатор переводит названия таких полей в `__class_name_field_name`, по которому можно обратиться к этим полям.



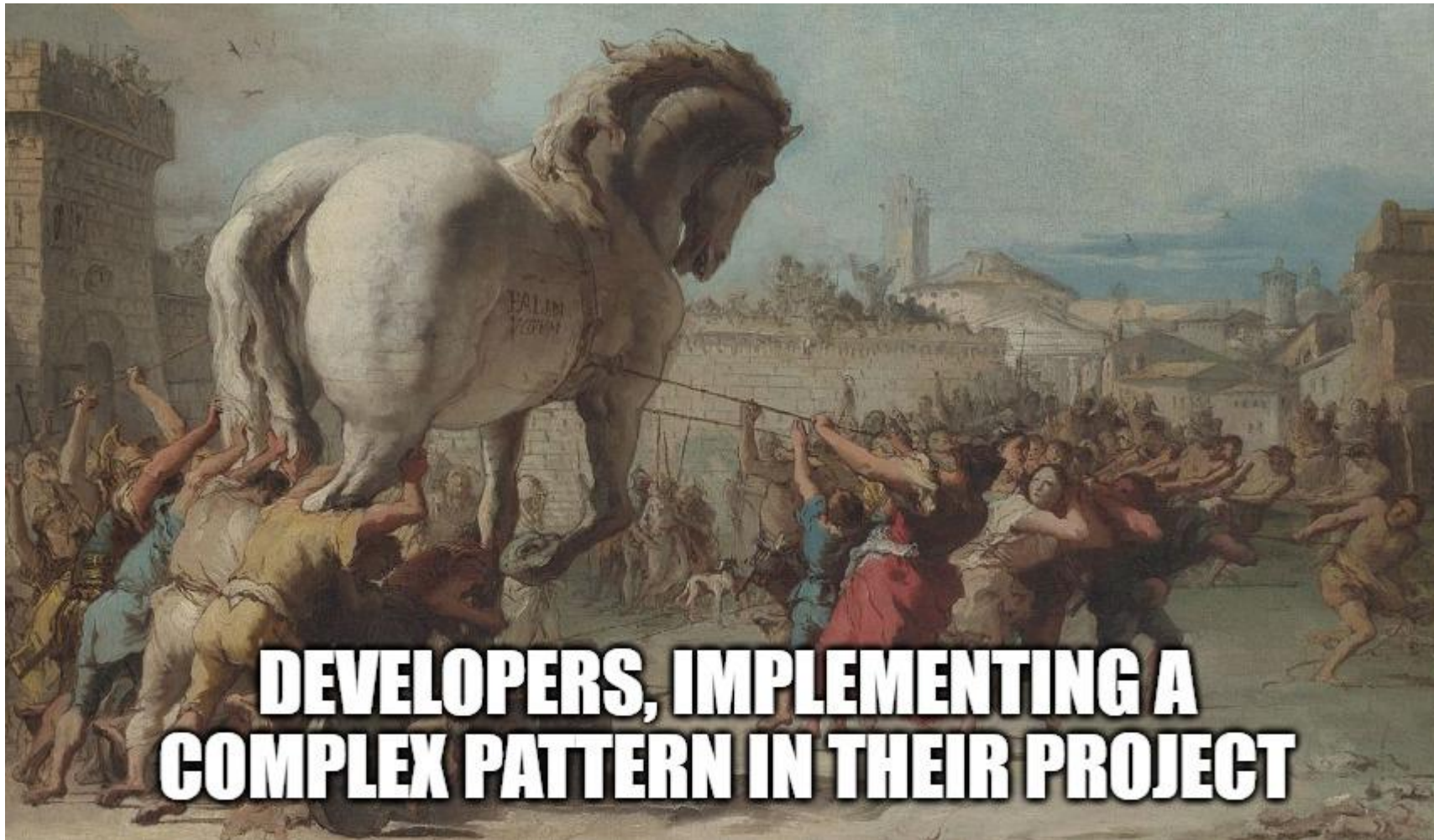
# Процессы, потоки, GIL

- Это все тоже сюда, в особенности языка, но мы рассмотрим это чууууууть чуть позже.

# Итого

- Лучше знать особенности в поведении языка
- Лучше знать ограничения языка
- Лучше знать возможности и библиотеки языка

# Паттерны/Шаблоны проектирования



# Паттерны/Шаблоны проектирования

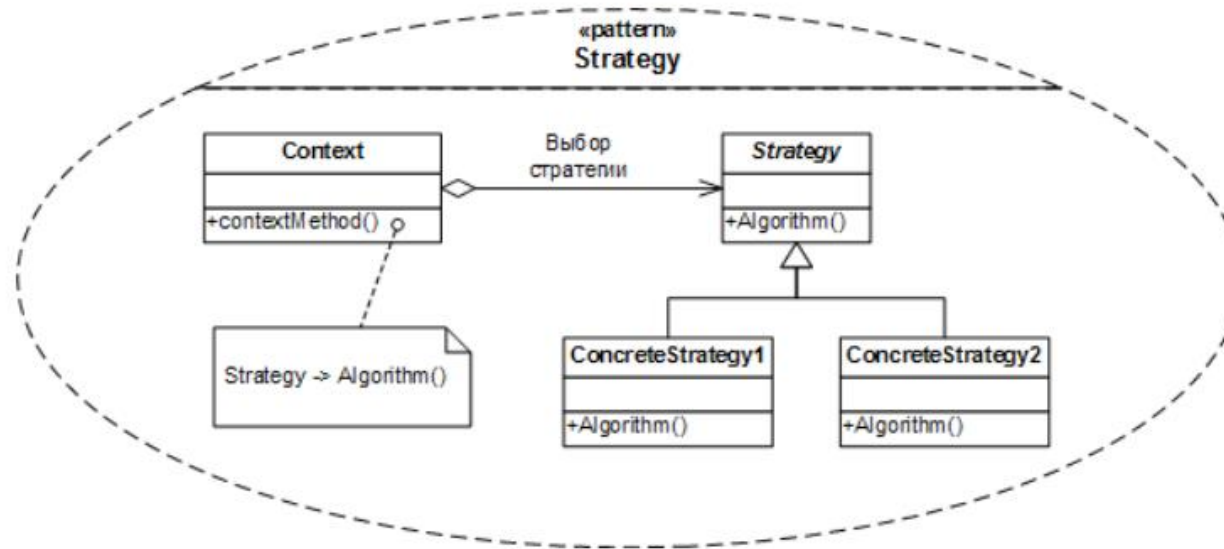
Есть две основные задачи, которые решаются паттернами проектирования:

- систематизация эффективных схем проектирования систем
- введение единой системы понятий для этих схем

Использование паттернов должно упрощать жизнь, но важно не переборщить.

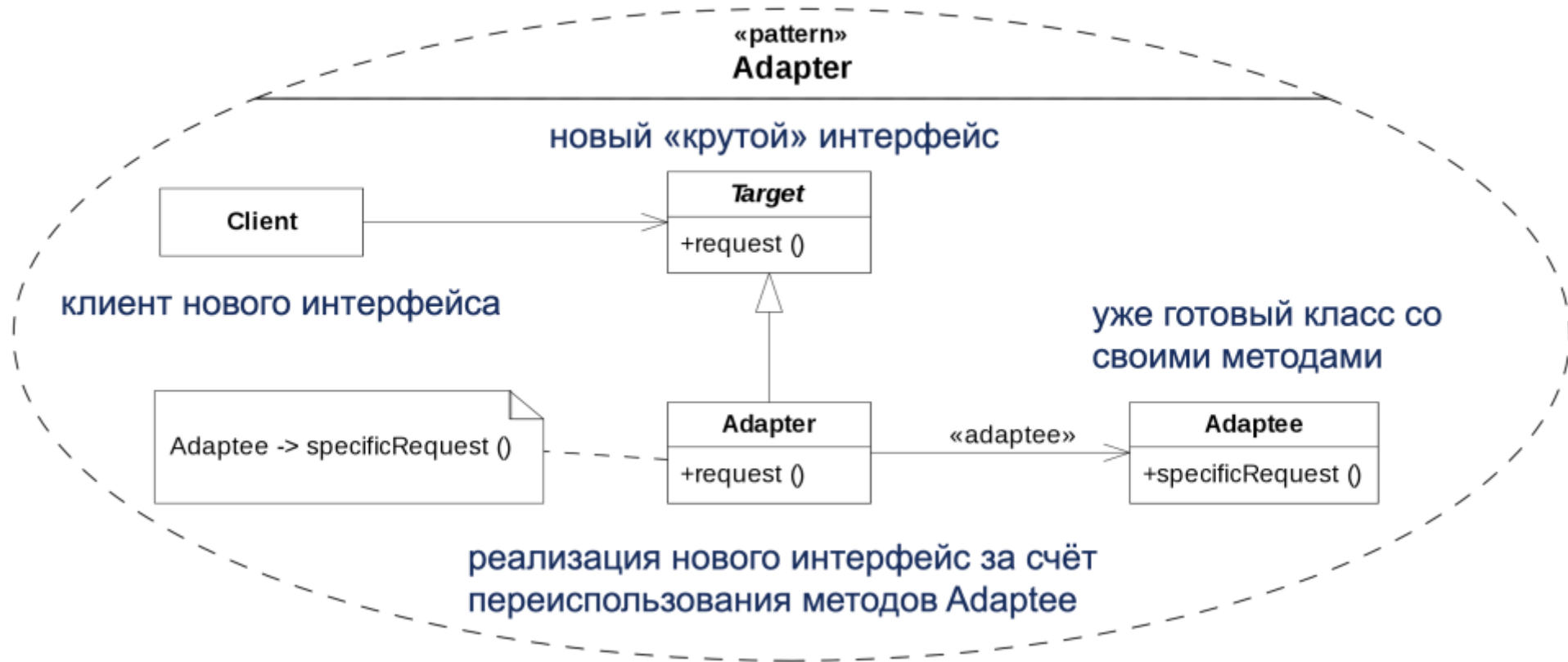
Сейчас просто рассмотрим примеры нескольких шаблонов (в сети есть все куда лучше меня объясненное).

# Стратегия



- Отделяем «важную» часть логики в интерфейс (стратегию) со своим “контрактом”
- На этом «контракте» основывается взаимодействие с объектом
- Есть реализации стратегии и они взаимозаменяемы
- Типичный пример — sklearn.base

# Адаптер



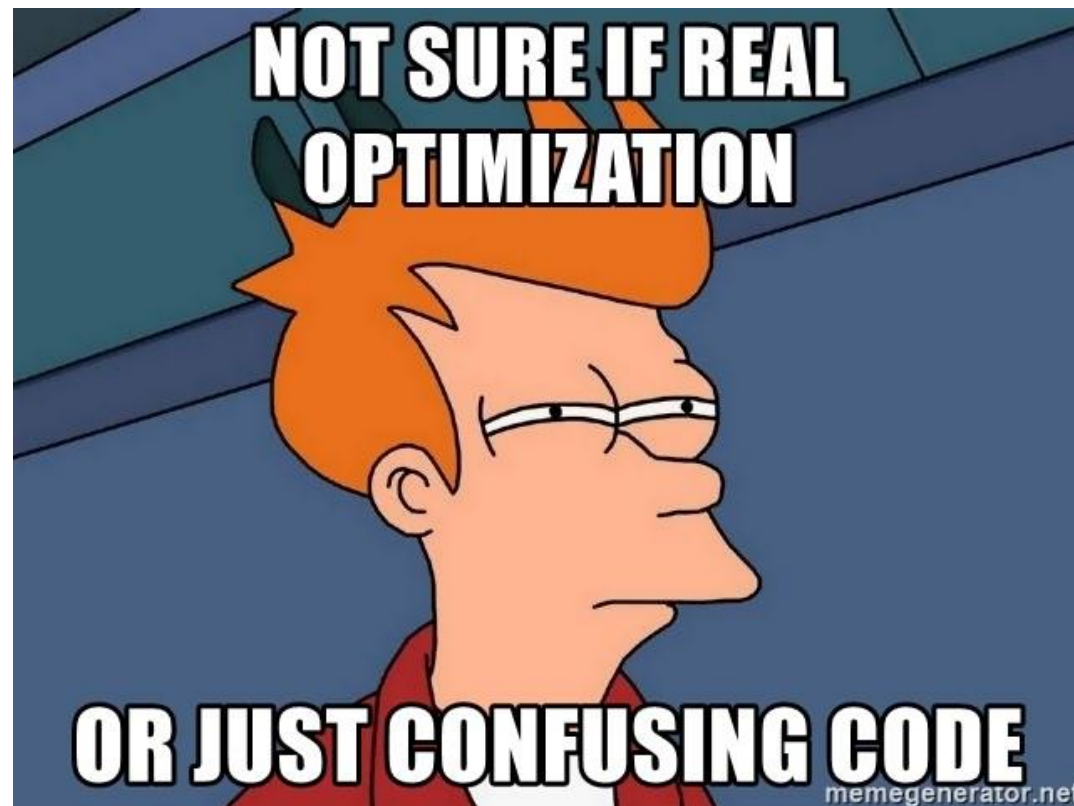
- Типичный пример — XGBRegressor

# Итого

- паттерны проектирования это результат рефлексии на тему как уменьшить проблемы от поддержки кода
- они широко применяются на практике и окружающие нас библиотеки их массово используют

# Оптимизация

- Первое правило оптимизации — убедиться, что нужна оптимизация.
- Второе правило оптимизации — код должен остаться понятным и выполнять свою задачу и выполнять свою задачу.





# Основные моменты:

- чтобы оценить и локализовать места, где нужна оптимизация, стоит использовать профилирование по времени (вручную или, например, используя cProfile)
- два основных подхода к оптимизации: ускорение отдельных команд и изменение алгоритма
- для первого важно знать особенности языка и общепринятые практики
- для второго важно знать какие вообще алгоритмы существуют и условия их применимости

- В Python-е лучшая оптимизация происходит не в Python-е :)
- По максимум использовать возможности библиотек, поменьше конструкций самого питона
- Если же приходится – тогда уж прибегать к оптимизации в питоне
- Знание алгоритмов реально помогает

# Общие советы и хорошие практики

Вещи, которые всегда советуют делать, но каждый начинает делать только после того как испытает острую боль на себе

- писать документацию (type hints, docstrings)
- писать тесты (pytest)

# Используйте существующие инструменты

- IDE (PyCharm, VSCode + plugins, Sublime Text + plugins, Vim + plugins и многие другие)
- linters (flake8, mypy, pylint)
- formatters (black)
- security checkers (bandit)

# Linters

```
sublime_linter.py — SublimeLinter

class sublime_linter_lint(sublime_plugin.TextCommand):
    """A command that lints the current view if it has a linter."""
    def __want_event(self):
        flake8: E271 – multiple spaces after keyword
        Copy | Click ⌘ to trigger a quick action
    def __run_lint(self, event=None, kwargs=None):
        return (
            util.is_lintable(self.view)
            and any(
                info["settings"].get("lint_mode") != "background"
                for info in elect.runnable_linters_for_view(self.view, "on_u
            )
        ) if event else True

sublime_linter.py:
1:80 warning flake8:E501 line too long (81 > 79 characters)
29:80 warning flake8:E501 line too long (88 > 79 characters)
166:80 warning flake8:E501 line too long (100 > 79 characters)
237:80 warning flake8:E501 line too long (93 > 79 characters)
289:8 warning flake8:E271 multiple spaces after keyword
292:1 warning flake8:E303 too many blank lines (2)
298:80 warning flake8:E501 line too long (89 > 79 characters)
334:80 warning flake8:E501 line too long (82 > 79 characters)

flake8(w:13), flake8: too many blank lines (2), Line 292, Column 1 dev 1 Spaces: 4
```

- Не все проблемы реально таковыми являются (но большинство!)
- Можно настроить на прекоммит и т.п.
- В IDE есть возможность настроить автоматический линтер
- Сам пользуюсь flake8

# Formatter

## Black v18.6b4

The uncompromising Python code formatter.

Playground built by [José Padilla](#) 

```
1 from seven_dwarfs import Grumpy, Happy, Sleepy, Bashful, Sneezzy, Dopey, Doc
2 x = { 'a':37,'b':42,
3
4 'c':927}
5
6 if very_long_variable_name is not None and \
7     very_long_variable_name.field > 0 or \
8     very_long_variable_name.is_debug:
9     z = 'hello '+'world'
10 else:
11     world = 'world'
12     a = 'hello {}'.format(world)
13     f = rf'hello {world}'
14 if (this
15 and that): y = 'hello 'world'#FIXME: https://github.com/ambv/black/issues/2
16 class Foo ( object ):
17     def f (self ):
18         return 37*-2
19     def g(self, x,y=42):
20         return y
21 def f ( a: List[ int ]):
22     return 37-a[42-u : y**3]
23 def very_important_function(template: str,*variables,file: os.PathLike,debug
24     """Applies `variables` to the `template` and writes to `file`."""
25     with open(file, "w") as f:
26         ...
27 # fmt: off
28 custom_formatting = [
29     0, 1, 2,
```

```
1 from seven_dwarfs import (
2     Grumpy,
3     Happy,
4     Sleepy,
5     Bashful,
6     Sneezzy,
7     Dopey,
8     Doc,
9 )
10
11 x = {"a": 37, "b": 42, "c": 927}
12
13 if (
14     very_long_variable_name is not None
15     and very_long_variable_name.field > 0
16     or very_long_variable_name.is_debug
17 ):
18     z = "hello " + "world"
19 else:
20     world = "world"
21     a = "hello {}".format(world)
22     f = rf"hello {world}"
23 if this and that:
24     y = "hello " "world" # FIXME: https://github.com/ambv/black/issues/26
25
26
27 class Foo(object):
28     def f(self):
29         return 37 * -2
```

[Report issue](#)



Submit



# Сесуриту чекер

```
bandit -- -bash
[ > bandit examples/yaml_load.py
[main] INFO    profile include tests: None
[main] INFO    profile exclude tests: None
[main] INFO    cli include tests: None
[main] INFO    cli exclude tests: None
[main] INFO    running on Python 3.8.2
Run started:2022-02-15 19:18:52.689821

Test results:
>> Issue: [B506:yaml_load] Use of unsafe yaml load. Allows instantiation of arbitrary objects. Consider yaml.safe_load().
Severity: Medium Confidence: High
Location: examples/yaml_load.py:7:8
More Info: https://bandit.readthedocs.io/en/latest/plugins/b506\_yaml\_load.html
6       ystr = yaml.dump({'a': 1, 'b': 2, 'c': 3})
7       y = yaml.load(ystr)
8       yaml.dump(y)

-----

Code scanned:
  Total lines of code: 12
  Total lines skipped (#nosec): 0

Run metrics:
  Total issues (by severity):
    Undefined: 0.0
    Low: 0.0
    Medium: 1.0
    High: 0.0
  Total issues (by confidence):
    Undefined: 0.0
    Low: 0.0
    Medium: 0.0
    High: 1.0
Files skipped (0):
> 
```

# Точечные неструктурированные советы

- перед оптимизацией, рефакторингом или изменением архитектуры стоит всегда задаться вопросом какую проблему это должно решить, порой оказывается, что проблемы нет
- думайте о чувствах человека, который будет читать то, что вы написали
- старайтесь мыслить контрактами и гарантиями при проектировании

# Общее резюме по работе с кодом

Код пишется для решения задачи разработчиками сейчас и в будущем.

Код должен быть понятен, функционален и эффективен по отношению к задаче.

Чтобы улучшить свой код можно прокачаться в:

- знании языка и других инструментов
- навыках оптимизации
- алгоритмистике
- проектировании архитектуры

# Зависимости и окружение

# Пакет, модуль, окружение

- Пакет в Python – это каталог, включающий в себя другие каталоги и модули, но при этом дополнительно содержащий файл `__init__.py`.
- Модуль - файл с расширением `.py`. Предназначены для того, чтобы в них хранить часто используемые функции, классы, константы и т.п.
- Виртуальное окружение - это изолированные настройки среды Python которые позволяют нам использовать определенные, нужные нам, библиотеки и их версии в нашем приложении.

# Что дает окружение

- Изолированность - работе изолированного решения нельзя случайно помешать
- Воспроизводимость/Переносимость - на другом компьютере решение должно работать так же (без ошибок и выдавать те же результаты)
- Фиксирование версий библиотек

# Проблемы с зависимостями

- У зависимостей есть уровни. Вам, например, нужен только пандас, но он подтягивает другие либы, а те другие, и так далее
- Библиотеки обновляются, поэтому версии надо фиксировать
- У зависимостей зависимостей (здесь нет опечатки) могут быть не зафиксированы версии пакетов

# Стадии управления зависимостями

1. Pip install
2. Requirements.txt
3. Pip freeze > pip\_freeze.txt
4. Pipenv
5. Poetry (the best choice)



# Чем хорош poetry

- Есть lock файл
- Есть деление на dev зависимости и не dev
- Единый файл управления проектом (pyproject.toml) и окружением
- Быстро резолвит зависимости (быстрее чем пипэнв)
- Ультрамегахорош
- Добавление, удаление пакетов не сложнее, чем в пипе

# Poetry.lock

```
[[package]]
name = "CacheControl"
version = "0.12.11"
description = "httplib2 caching for requests"
category = "main"
optional = false
python-versions = ">=3.6"

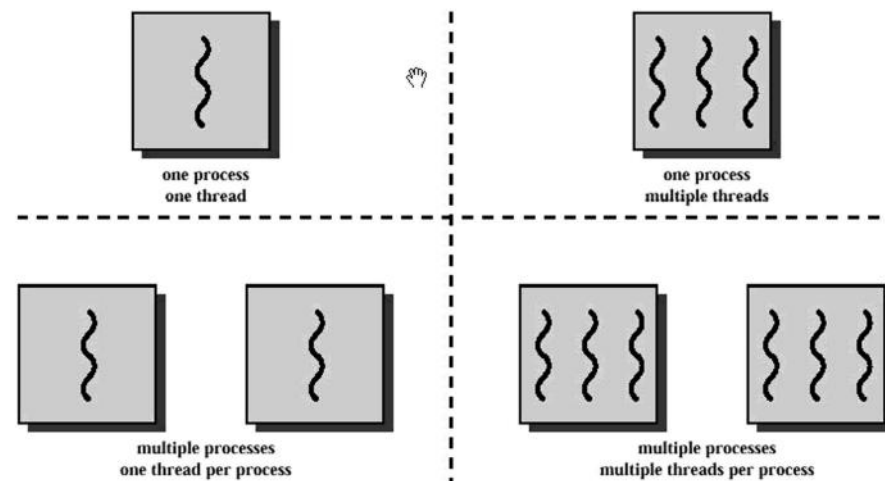
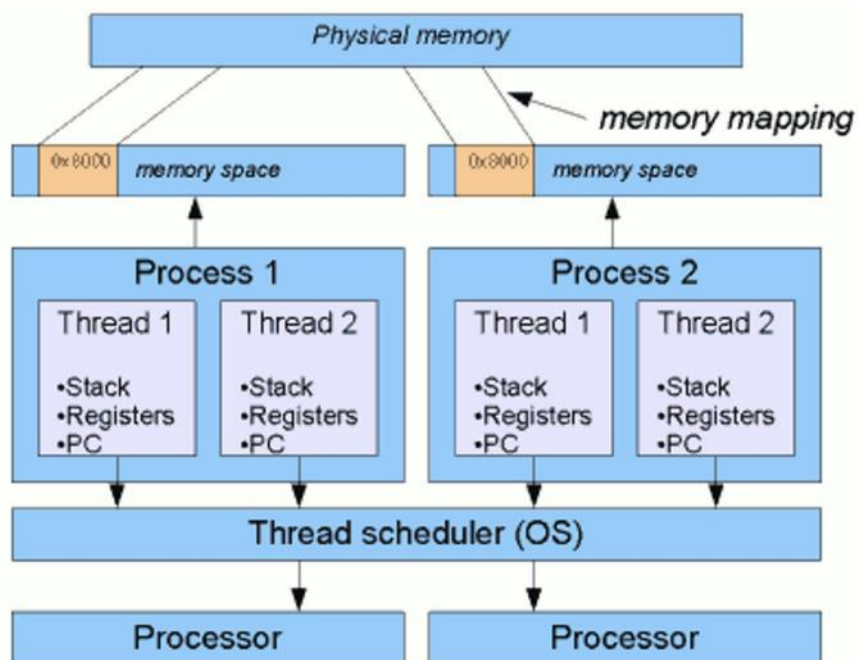
[package.dependencies]
lockfile = {version = ">=0.9", optional = true, markers = "extra == \\"filecache\\""}
msgpack = ">=0.5.2"
requests = "*"

[package.extras]
filecache = ["lockfile (>=0.9)"]
redis = ["redis (>=2.10.5)"]
```

# Параллелизм и конкурентность



# Процессы и потоки



Эффективно ли работают multiprocessing в python?

Эффективно ли работают multithreading в python?

# GIL

- Блокирует все потоки, пока работает текущий
- Не позволяет писать параллельный код в одном процессе

# В чем разница

- Цель **конкурентности** – предотвратить взаимоблокировки задач путем переключения между ними, когда одна из задач вынуждена ждать внешнего ресурса. Типичный пример – обработка нескольких сетевых запросов.
- **Параллелизм** – это история о максимальном использовании этих ресурсов путем запуска процессов или потоков, использующих все ядра процессора, которыми располагает компьютер.

# И что же есть в питоне

Для конкурентности используется **многопоточность** и **асинхронность**, для параллелизма используется **многопроцессорность**.

- Многопоточность: *multithreading*
- Асинхронность: *asyncio*
- Многопроцессорность: *multiprocessing*

# Когда и что

- С параллелизмом все понятно – хотим использовать многоядерность, используем multiprocessing
- А в чем разница между asyncio и multithreading?
- В то время, как многопоточность берет и запускает функции в отдельных потоках, asyncio работает в одном потоке и разрешает циклу обработки событий программы взаимодействовать с несколькими задачами, чтобы каждая из них выполнялась по очереди в оптимальное время. Отдельное выполнение такой задачи – это, можно сказать, **корутина**.



# Когда и что

- При использовании многопоточности операционная система знает о наличии различных потоков и может в любое время прерывать их работу и переключать на другую задачу. Сама программа это не контролирует. Может случиться так, что два потока трогают одни и те же данные. Небезопасно, короч.
- При использовании модуля `asyncio` программа сама принимает решение о том, когда ей нужно переключиться между задачами.
- Корутины не связаны архитектурными ограничениями как потоки и требуют меньше памяти из-за того, что выполняются в одном потоке. Ну и они быстрее потоков в питоне.

# Таки все

- Посмотрели, как правильно писать код
- Посмотрели, как управлять зависимостями и окружением
- Посмотрели, что такое конкурентность и параллелизм