30 sposobów aby Twój kod był bardziej pythonik

Mariusz Perkowski

Grape Up Sp. z o.o. | Kraków, Poland | info@grapeup.com

Mariusz Perkowski

Python developer (6 years of experience)

Consultant at Grape Up

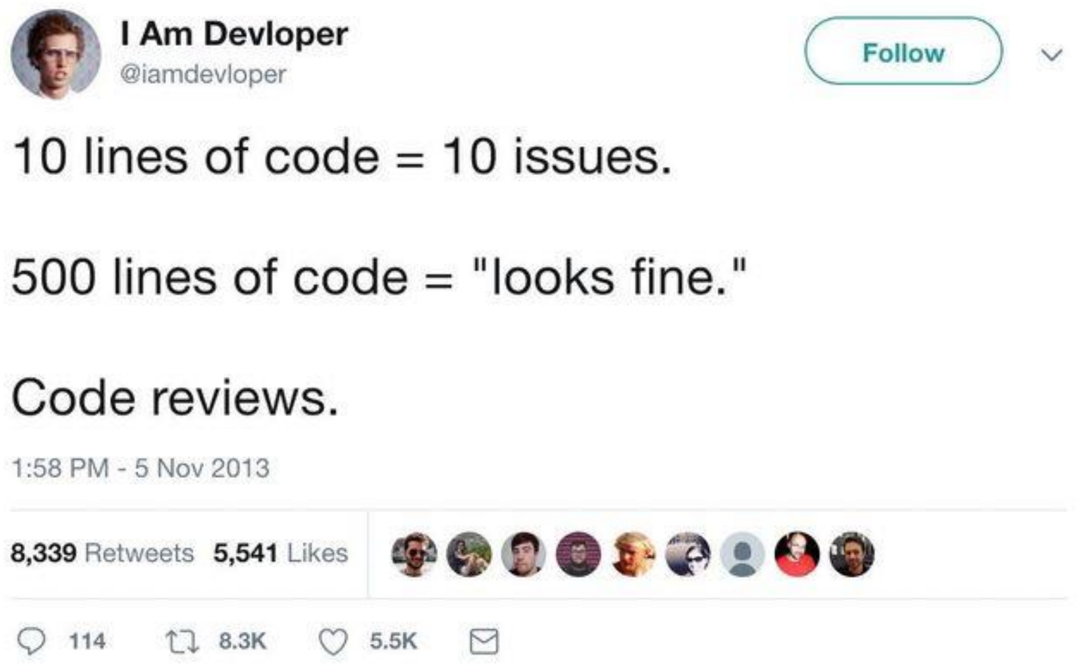
Britenet

NetStation

grape up®

AGENDA

- 30 sposobów aby Twój kod był bardziej pythonik


I Am Devloper
@iamdevloper

Follow

10 lines of code = 10 issues.

500 lines of code = "looks fine."

Code reviews.

1:58 PM - 5 Nov 2013

8,339 Retweets 5,541 Likes

114 8.3K 5.5K

If statemant vs If expression

```
condition = True

if condition:
    x = 1
else:
    x = 2
```

```
x = 1 if condition else 2
```

If statemant vs If expression

```
score = 57
if score > 90:
    grade = "A*"
elif score > 50:
    grade = "pass"
else:
    grade = "fail"
```

```
score = 57
grade = "A*" if score > 90 else "pass" if score > 50 else "fail"
```

grape up®

Any

```
def use_any_instead_of_loop():
    numbers = [-1, -2, -4, 0, 3, 7]
    has_positives = False

    for n in numbers:
        if n > 0:
            has_positives = True
            break
```

```
def use_any_instead_of_loop():
    numbers = [-1, -2, -4, 0, 3, 7]
    has_positives = any(n > 0 for n in numbers)

    return has_positives
```

Is numeric

```
example = []
data = "1"
```

```
def loop_over_1():
    for i in range(0, 10):
        try:
            data = int(data)
        except:
            pass
    example.append(data)
```

```
def loop_over_2():
    for i in range(0, 10):
        data = int(data) if data.isnumeric() else data
        example.append(data)
```

Enum

```python
class Employee:
    def __init__(self, name: str, surname: str, role: str):
        self.name = name
        self.surname = surname
        self.role = role


john = Employee("John", "Doe", "manager")
kris = Employee("Kris", "Foo", "developer")
tom = Employee("Tom", "Barr", "tester")

print(john.role)
print(kris.role)
print(tom.role)

# manager
# developer
# tester
```

```python
from enum import Enum

class Role(Enum):
    MANAGER = "manager"
    DEVELOPER = "developer"
    TESTER = "tester"

john = Employee("John", "Doe", Role.MANAGER.value)
kris = Employee("Kris", "Foo", Role.DEVELOPER.value)
tom = Employee("Tom", "Barr", Role.TESTER.value)

print(john.role)
print(kris.role)
print(tom.role)

# manager
# developer
# tester
```

grape up®

Exceptions

```
def data_fetch_from_my_api(params):
    if "date" not in params.keys():
        raise KeyError("Params are required")
    return requests.get("https://my-api.com").raise_for_status()
```

```
# BAD
try:
    data_fetch_from_my_api()
except Exception as e:
    logger(e)


# BETTER
try:
    data_fetch_from_my_api()
except (HTTPError, KeyError) as e:
    logger(e)


# GOOD
try:
    data_fetch_from_my_api()
except HTTPError as e:
    logger("HTTPError api leży i kwiczy")
    logger(e)
except KeyError as e:
    logger("KeyError no weś się ogarnij")
    logger(e)
```

grape up®

Exceptions

```
class ParamValidationException(Exception):
    pass


def data_fetch_from_my_api(params):
    if "date" not in params.keys():
        raise ParamValidationException("Params are required")
    return requests.get("https://my-api.com").raise_for_status()
```

```
try:
    data_fetch_from_my_api()
except HTTPError as e:
    logger("HTTPError Api leży i kwiczy")
    logger(e)
except ParamValidationException as e:
    logger("ParamValidationException  no weś się ogarnij")
    logger(e)
```

grape up®

List comprehension

```
[i for i in range(50) if i%2 == 0]

[i  if i%2 == 0 else "buba" for i in range(50)]

[i for i in range(50) if i%2==0 and i%3==0 and i%3==0]
```

```
result = (
    (x, y, z)
    for x in range(5)
    for y in range(5)
    if x != y
    for z in range(5)
    if y != z
)
```

grape up®

Guard clause

```
def should_i_wear_this_hat(self, hat):
    if isinstance(hat, Hat):
        jacket_color = self.get_jacket_color()
        current_weather = self.get_current_weather()
        is_stylish = self.is_stylish(hat, jacket_color)
        if current_weather.is_raining():
            print("Oh no, it's raining! I can't wear this hat!")
            return True
        else:
            print("Nice")
            return is_stylish
    else:
        print("This is not a hat!")
        return False
```

```
def should_i_wear_this_hat(self, hat):
    if not isinstance(hat, Hat):
        return False
    jacket_color = self.get_jacket_color()
    current_weather = self.get_current_weather()
    is_stylish = self.is_stylish(hat, jacket_color)
    if current_weather.is_raining():
        print("Oh no, it's raining! I can't wear this hat!")
        return True
    else:
        print("Nice")
        return is_stylish
```

grape up®

Przypisania bliżej użycia

```
def should_i_wear_this_hat(self, hat):
    if not isinstance(hat, Hat):
        return False
    jacket_color = self.get_jacket_color()
    current_weather = self.get_current_weather()
    is_stylish = self.is_stylish(hat, jacket_color)
    if current_weather.is_raining():
        print("Oh no, it's raining! I can't wear this hat!")
        return True
    else:
        print("Nice")
        return is_stylish
```

```
def should_i_wear_this_hat(self, hat):
    if not isinstance(hat, Hat):
        return False
    jacket_color = self.get_jacket_color()
    current_weather = self.get_current_weather()
    if current_weather.is_raining():
        print("Oh no, it's raining! I can't wear this hat!")
        return True
    else:
        print("Nice")
        return self.is_stylish(hat, jacket_color)
```

Result

```
def should_i_wear_this_hat(self, hat):
    if isinstance(hat, Hat):
        jacket_color = self.get_jacket_color()
        current_weather = self.get_current_weather()
        is_stylish = self.is_stylish(hat, jacket_color)
        if current_weather.is_raining():
            print("Oh no, it's raining! I can't wear this hat!")
            return True
        else:
            print("Nice")
            return is_stylish
    else:
        print("This is not a hat!")
        return False
```

```
def should_i_wear_this_hat(self, hat):
    if not isinstance(hat, Hat):
        return False
    jacket_color = self.get_jacket_color()
    current_weather = self.get_current_weather()
    if current_weather.is_raining():
        print("Oh no, it's raining! I can't wear this hat!")
        return True
    else:
        print("Nice")
        return self.is_stylish(hat, jacket_color)
```

Walrus operator

```python
def func_to_get_author(author: str = None):
    return "Adam Mickiewicz" if author == "Adaś" else None

author = func_to_get_author("Adaś")
if author:
    print(f"The author is {author}.")
```

```python
if author := func_to_get_author("Adaś"):
    print(f"The author is {author}.")
```

Walrus operator

```
nums = [1, 2, 3, 4, 5]

def func(x):
    return x * 2


results = [(x, y) for x in nums if (y := func(x)) > 4]
print(results)
# [(3, 6), (4, 8), (5, 10)]
```

Contex manager with requests

```
import requests
s = requests.Session()

s.get('https://httpbin.org/cookies/set/sessioncookie/123456789')
r = s.get('https://httpbin.org/cookies')
print(r.text)
# '{"cookies": {"sessioncookie": "123456789"}}'
```

```
with requests.Session() as session:
    session.request(method="get", url='https://httpbin.org/cookies')
```

grape up®

Itertools.groupby

```
from itertools import groupby

for key, group in groupby("YAaANNGGG"):
    lg = list(group)
    print(key, len(lg), lg)
```

```
# Y 1 ['Y']
# A 1 ['A']
# a 1 ['a']
# A 1 ['A']
# N 2 ['N', 'N']
# G 3 ['G', 'G', 'G']
```

grape up®

Itertools.groupby

```
class Plumber:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    def __repr__(self):
        return f"{self.name} {self.surname}"
```

```
p1 = Plumber("John", "Smith")
p2 = Plumber("John", "Doe")
p3 = Plumber("Mario", "Pe")
p4 = Plumber("Kazik", "Kowalsky")

plumbers = [p1, p2, p3, p4]
```

```
for key, group in groupby(plumbers, lambda x: x.name):
    lg = list(group)
    print(key, len(lg), lg)
```

```
# John 2 [John Smith, John Doe]
# Mario 1 [Mario Pe]
# Kazik 1 [Kazik Kowalsky]
```

Itertools chain

```
from itertools import chain

x = [0, 1, 2, 3, 4]
y = tuple(("a", "b", "c"))
z = set((9, 7))


w = x + y

# Traceback (most recent call last):
#   File
"C:\Users\mariusz.perkowski\warsztat\PyStok\warsztat.py",
line 7, in <module>
#     w = x + y
#         ~~^~~
# TypeError: can only concatenate list (not "tuple") to list

w = x + list(y) + list(z)
# [0, 1, 2, 3, 4, 'a', 'b', 'c', 9, 7]


list(chain(x, y, z))
# [0, 1, 2, 3, 4, 'a', 'b', 'c', 9, 7]
```

Itertools chain

```
from itertools import chain

x = [0, 1, 2, 3, 4]
y = tuple(("a", "b", "c"))
nested = [x, y]

print(list(chain(nested)))

# [[0, 1, 2, 3, 4], ('a', 'b', 'c')]

print(list(chain.from_iterable(nested)))

# [0, 1, 2, 3, 4, 'a', 'b', 'c']
```

Itertools.zip_longest

```python
x = [0, 1, 2, 3, 4]
y = tuple(("a", "b", "c"))

print(list(zip(x, y)))
# [(0, 'a'), (1, 'b'), (2, 'c')]
```

```python
from itertools import zip_longest


print(list(zip_longest(x, y)))
# [(0, 'a'), (1, 'b'), (2, 'c'), (3, None), (4, None)]


print(list(zip_longest(x, y, fillvalue="*")))
# [(0, 'a'), (1, 'b'), (2, 'c'), (3, '*'), (4, '*')]
```

Itertools.islice

```
# slice(start, stop, step)
nums = [1, 2, 3, 4]

# reversed list
print(nums[::-1])
```

```
from itertools import islice


@timer
def slice():
    ite = range(1000)[1:]
    for item in ite:
        print(item)


@timer
def islice():
    ite = islice(range(1000), 1, None, 1)
    for item in ite:
        print(item)
```

Merge two dicts

```
cities_us = {"New York City": "US", "Los Angeles": "US"}
cities_uk = {"London": "UK", "Birmingham": "UK"}
cities_jp = {"Tokyo": "JP"}

cities = {}


for city_dict in [cities_us, cities_uk, cities_jp,]:
    for city, country in city_dict.items():
        cities[city] = country

print(cities)
# {'New York City': 'US', 'Los Angeles': 'US', 'London': 'UK', 'Birmingham': 'UK', 'Tokyo': 'JP'}
```

Merge two dicts

```
cities = {**cities_us, **cities_uk, **cities_jp}

print({**cities_us, **cities_uk, **cities_jp})
# {'New York City': 'US', 'Los Angeles': 'US', 'London': 'UK', 'Birmingham': 'UK', 'Tokyo': 'JP'}
```

```
cities = cities_us | cities_uk | cities_jp

print(cities)
# {'New York City': 'US', 'Los Angeles': 'US', 'London': 'UK', 'Birmingham': 'UK', 'Tokyo': 'JP'}
```

Pyton dict setdefault

```
plumbers: dict = {"mario": "present", "luigi": "present"}

print(f"get        plumber mario  {plumbers.get('mario', 'not present')}")
print(f"setdefault plumber mario  {plumbers.setdefault('mario', 'not present')}")
print(f"all        plumbers  {plumbers}")


# get                plumber          mario  present
# setdefault         plumber          mario  present
# all                plumbers  {'mario': 'present', 'luigi': 'present'}
```

grape up®

Pyton dict setdefault

```
plumbers: dict = {"mario": "present", "luigi": "present"}

print(f"get        plumber kazik       {plumbers.get('kazik', 'not present')}")
print(f"setdefault plumber kazik  {plumbers.setdefault('kazik', 'not present')}")

# get        plumber kazik  not present
# setdefault plumber kazik  not present


print(f"all        plumbers                {plumbers}")
print(f"setdefault plumbers          {plumbers}")

# all        plumbers          {'mario': 'present', 'luigi': 'present'}
# setdefault plumbers     {'mario': 'present', 'luigi': 'present', 'kazik': 'not present'}
```

grape up®

Collections defaultdict

```
from collections import defaultdict


def default_value():
    return "not present"

plumbers = defaultdict(def_value)
plumbers["mario"] = "present"
plumbers["luigi"] = "present"

print(f"defaultdict plumber mario  {plumbers['mario']}")
print(f"defaultdict plumber luigi      {plumbers['luigi']}")
print(f"defaultdict plumber kazik     {plumbers['kazik']}")
print(f"defaultdict plumbers              {plumbers}")

# defaultdict plumber mario   present
# defaultdict plumber luigi     present
# defaultdict plumber kazik     not present
# defaultdict plumbers           {'mario': 'present', 'luigi': 'present', 'kazik': 'not present'}
```

grape up®

Collections .Counter

```
from collections import Counter

sentence = "This is a simple sentence for demonstration purposes. With a few repetitions in this sentence"


count_letters = Counter(sentence)
count_words = Counter(sentence.split())
most_common_words = Counter(sentence.split()).most_common(3)

print(count_letters)
print(count_words)
print(most_common_words)

# Counter({' ': 14, 'e': 12, 's': 10, 'i': 9, 'n': 8, 't': 8, 'o': 5, 'p': 4, 'r': 4, 'h': 3, 'a': 3, 'm': 2, … 'W': 1, 'w': 1})
# Counter({'a': 2, 'sentence': 2, 'This': 1, 'is': 1, 'simple': 1, 'for': 1, 'demonstration': 1,…, 'repetitions': 1, 'in': 1, 'this': 1})
# [('a', 2), ('sentence', 2), ('This', 1)]
```

Collections. namedtuple

```
point = (2, 4)
point
# (2, 4)

point[0]
# 2

point[1]
# 1

point[0] = 3
# Traceback (most recent call last):
#     point[0] = 3
#     ~~~~~^^^
# TypeError: 'tuple' object does
   not support item assignment
```

```
from collections import namedtuple

Person = namedtuple("Person", "name children")
john = Person("John Doe", ["Timmy", "Jimmy"])
john
# Person(name='John Doe', children=['Timmy', 'Jimmy’])

id(john.children)  # 139695902374144
john.children.append("Tina")
john
# Person(name='John Doe', children=['Timmy', 'Jimmy', 'Tina’])

id(john.children) # 139695902374144

john.name = "Frank"
# Traceback (most recent call last):
#     john.name = "Frank"
#     ^^^^^^^^^
# AttributeError: can't set attribute
```

Structural Pattern Matching

```
def lets_if(input: str):
    splited_input = input.split()
    if splited_input[0] == "load" and len(splited_input) == 2 and splited_input[1]:
        print(f"Loading file: {splited_input[1]}")
    elif splited_input[0] == "save" and len(splited_input) == 2 and splited_input[1]:
        print(f"Saving file: {splited_input[1]}")
    elif splited_input[0] in ["quit", "exit", "bye"]:
        print("Quitting")
    else:
        print(f"Unknown input: {splited_input[0]}")
```

```
def lets_match(input: str):
    match input.split():
        case ["load", filename]:
            print(f"Loading file: {filename}")
        case ["save", filename]:
            print(f"Saving file: {filename}")
        case ["quit" | "exit" | "bye"]:
            print("Quitting")
        case other:
            print(f"Unknown input: {other}")
```

grape up®

lru cache memoization

```
def get_a_lot_of_data(query_params: dict) -> dict:
    return query_params

def complicated_alghorytm_replace_a_b(x: str):
    return x.replace("a", "b")


def slow_endpoint(query_params):
    very_complicated_query = get_a_lot_of_data(query_params)
    return [complicated_alghorytm_replace_a_b(x) for x in very_complicated_query]



from functools import lru_cache

@lru_cache()
def fast_endpoint(query_params):
    very_complicated_query = get_a_lot_of_data(query_params)
    return [complicated_alghorytm_replace_a_b(x) for x in very_complicated_query]
```

Decorator

```
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"{func.__name__} took {end_time - start_time} seconds to execute.")
        return result
    return wrapper


@timer
def slow_function():
    time.sleep(2)


slow_function()
# slow_function took 2.0004498958587646 seconds to execute.
```

Regex

```
import re


text = "--- a0 ---"

def check_patern(text: str) -> bool:
    REGEX = re.compile("[0-9a-fA-f]")
    return re.match(REGEX, text)
```

```
import re

text = "--- a0 ---"
REGEX = re.compile("[0-9a-fA-f]")


def check_patern(text: str) -> bool:
    return re.match(REGEX, text)
```

grape up®

Abstract vs Protocol

```
class Device(ABC):
    @abstractmethod
    def connect(self) -> None:
        pass

    @abstractmethod
    def turn_on(self) -> None:
        pass

    @abstractmethod
    def turn_off(self) -> None:
        pass

    @abstractmethod
    def get_status(self) -> bool:
        pass
```

```
class Service:
    def __init__(self):
        self.devices = []

    def register_device(self, device: Device) -> None:
        device.connect()
        self.devices.append(device)
```

grape up®

Abstract vs Protocol

```
class TV(Device):
    def __init__(self):
        self._status = False

    def connect(self) -> str:
        return "TV"

    def turn_on(self) -> None:
        self._status = True

    def turn_off(self) -> None:
        self._status = False

    def get_status(self) -> bool:
        return self._status
```

```
class Radio(Device):
    def __init__(self):
        self._status = False

    def connect(self) -> str:
        return "Radio"

    def turn_on(self) -> None:
        self._status = True

    def turn_off(self) -> None:
        self._status = False

    def get_status(self) -> bool:
        return self._status
```

```
service = Service()

tv = TV()
radio = Radio()

service.register_device(tv)
service.register_device(radio)


print(radio.turn_on())
print(radio.get_status())
# True
print(tv.turn_on())
print(tv.get_status())
# True
```

grape up®

Abstract vs Protocol

```
from typing import Protocol


class DeviceProtocol(Protocol):
    def connect(self) -> None:
        ...

    def turn_on(self) -> None:
        ...

    def turn_off(self) -> None:
        ...


class Diagnostic(Protocol):
    def get_status(self) -> bool:
        ...
```

```
class ServiceProtocol:
    def __init__(self):
        self.devices = []

    def register_device(self, device: DeviceProtocol) -> None:
        device.connect()
        self.devices.append(device)
```

Abstract vs Protocol

```
class TV:
    def connect(self) -> str:
        return "TV"

    def turn_on(self) -> None:
        self._status = True

    def turn_off(self) -> None:
        self._status = False

    def get_status(self) -> bool:
        return self._status
```

```
class Radio:
    def connect(self) -> str:
        return "Radio"

    def turn_on(self) -> None:
        self._status = True

    def turn_off(self) -> None:
        self._status = False

    def get_status(self) -> bool:
        return self._status
```

```
tv = TV()
radio = Radio()

service_protocol = ServiceProtocol()
service_protocol.register_device(tv)
service_protocol.register_device(radio)



print(radio.turn_on())
print(radio.get_status())
# True
print(tv.turn_on())
print(tv.get_status())
# True
```

grape up®

Abstract vs Protocol

```
class TV:
    def __init__(self):
        self._status = False

    def connect(self) -> str:
        return "TV"

    def turn_on(self) -> None:
        self._status = True

    def turn_off(self) -> None:
        self._status = False

    def get_status(self) -> bool:
        return self._status
```

```
class TV(Device):
    def __init__(self):
        self._status = False

    def connect(self) -> str:
        return "TV"

    def turn_on(self) -> None:
        self._status = True

    def turn_off(self) -> None:
        self._status = False

    def get_status(self) -> bool:
        return self._status
```

```
tv = TV()
radio = Radio()

service_protocol = ServiceProtocol()
service_protocol.register_device(tv)
service_protocol.register_device(radio)



print(radio.turn_on())
print(radio.get_status())
# True
print(tv.turn_on())
print(tv.get_status())
# True
```

Abstract vs Protocol

```
from typing import Protocol


class DeviceProtocol(Protocol):
    def connect(self) -> None:
        ...

    def turn_on(self) -> None:
        ...

    def turn_off(self) -> None:
        ...

class Diagnostic(Protocol):
    def get_status(self) -> bool:
        ...

class ScheduleProtocol(Protocol):
    def set_schedule(self , schedule: dict):
        ...
```

```
class TV:
    def __init__(self):
        self._status = False
        self.schedule = None

    def connect(self) -> str:
        return "TV"

    def turn_on(self) -> None:
        self._status = True

    def turn_off(self) -> None:
        self._status = False

    def get_status(self) -> bool:
        return self._status

    def set_schedule(self , schedule: dict):
        self.schedule = schedule
```

grape up®

Dziękuje za uwagę