Dobre praktyki programowania w Pythonie

Paweł Gliwny

Środowisko wirtualne (virtual environment)

Czym jest środowisko wirtualne?

- Izolowane środowisko dla projektów Python, które pozwala uniknąć konfliktów między wersjami bibliotek.
- Każdy projekt może mieć niezależne zestawy bibliotek.

Windows:

```
$ python -m venv moj_env
$ moj_env\Scripts\activate
```

Mac/Linux:

```
$ python3 -m venv moj_env
$ source moj_env/bin/activate
```

Po aktywacji środowiska instalowane pakiety będą odizolowane od globalnego Pythona.

Alternatywy dla venv

- **Conda**
  - Zarządza środowiskami oraz instalacją bibliotek i ich zależności.
- **Miniconda**
  - Minimalna wersja Condy bez zbędnych bibliotek.
  - Zajmuje mniej miejsca na dysku, idealna do lekkich zastosowań.
- **Mamba**
  - Szybsza alternatywa dla Condy (napisana w C++).
  - Kompatybilna z poleceniami Condy, ale dużo bardziej wydajna.
- **Poetry**
  - Narzędzie do kompleksowego zarządzania zależnościami oraz środowiskami Python.
  - Automatycznie generuje i zarządza plikami `pyproject.toml` i `poetry.lock`.

Narzędzie pip – instalacja pakietów

- Domyślny menedżer pakietów dla języka Python.
- Pozwala na łatwe instalowanie bibliotek oraz zarządzanie ich wersjami.

- Instalacja pakietu:
  - `$ pip install numpy`
- Instalacja konkretnej wersji pakietu:
  - `$ pip install numpy==1.26.4`
- Instalacja z pliku z wymaganiami (`requirements.txt`):
  - `$ pip install -r requirements.txt`

Czym jest plik **requirements.txt**?

- Plik tekstowy **służący do zarządzania zależnościami** (pakietami) w projektach Python.
- Zawiera **listę wszystkich bibliotek i ich wersji**, potrzebnych do działania aplikacji.
- Ułatwia współpracę – inni mogą szybko odtworzyć identyczne środowisko.

- Jak utworzyć plik requirements.txt?
  - `$ pip freeze > requirements.txt`
- Jak instalować pakiety z requirements.txt?
  - `$ pip install -r requirements.txt`

requirements.txt:

```
numpy==1.26.4
pandas==2.2.1
requests==2.31.0
matplotlib<3.9
```

Zadanie 0

- Utwórz środowisko wirtualne Pythona oraz spróbować uruchomić test_app_with_flask.py

Odpowiednie formatowanie kodu

- **Dlaczego to ważne?**
  - Python słynie ze swojej czytelności.
  - Czytelny kod = łatwiejsza praca zespołowa i utrzymanie projektu.
  - Przejrzysty kod minimalizuje błędy i zwiększa efektywność debugowania.

Czytelne nazewnictwo

- **Używaj jasnych, opisowych nazw zmiennych i funkcji.**

Źle:

```
a = 10
b = 5
c = a * b
```

Dobrze:

```
szerokosc = 10
wysokosc = 5
pole_prostokata = szerokosc * wysokosc
```

Spacje w wyrażeniach

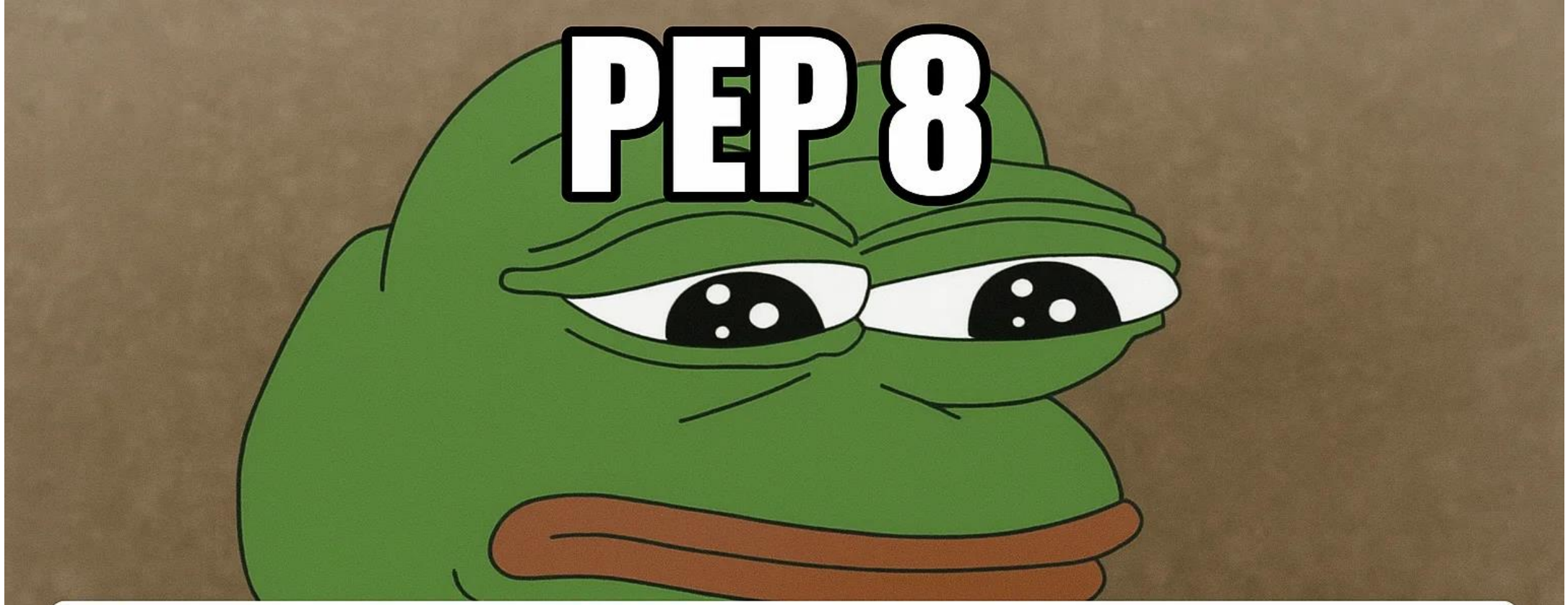
- Stosuj spacje wokół operatorów i po przecinkach.

Źle:

- wynik=(a+b)*(c-d)

Dobrze:

- wynik = (a + b) * (c - d)



Rules:

1. Use 4 spaces per indentation level
2. Limit all lines to a maximum of 79 characters
3. Separate top-level function and class definitions with two blank lines
4. Surround top-level function and class definitions

PEP 8

- **PEP 8** (ang. *Python Enhancement Proposal*) – oficjalny dokument określający zasady formatowania kodu w Pythonie.
- Zawiera wskazówki dotyczące stylu pisania kodu, aby był przejrzysty i spójny.
- Przykładowe reguły PEP 8:
  - 4 spacje na wcięcie kodu
  - Ograniczenie długości linii do 79 znaków
  - Zalecenia dotyczące nazywania zmiennych, klas, funkcji

Zalecenia PEP 8 dotyczące funkcji

- **Nazewnictwo**:
- Stosuj małe litery i _ do rozdzielenia słów (*snake_case*).
- Nazwy powinny jasno opisywać działanie funkcji.

```
# dobrze
def oblicz_pole_kwadratu(bok):
    return bok ** 2

# źle
def ObliczPoleKwadratu(bok):
    return bok ** 2
```

Funkcje rozmiar i argumenty

Funkcje powinny być możliwie krótkie i robić tylko jedną rzecz.

Argumenty:

- Nie stosuj spacji przed nawiasem otwierającym.
- Po przecinku zawsze dodawaj spację.

```
# dobrze
def funkcja(arg1, arg2='domyslny'):
    pass
```

Zalecenia PEP 8 dotyczące nazw w OOP

Nazwy klas zapisujemy w konwencji **CamelCase** (każde słowo z dużej litery).

```
# dobrze
class MojKalkulator:
    pass

# źle
class moj_kalkulator:
    pass
```

Metody powinny być pisane małymi literami, w formacie *snake_case*.

```
class Samochod:
    def uruchom_silnik(self):
        pass
```

Struktura i artybuty klasy

Atrybuty klas:

- Atrybuty powinny być zapisywane małymi literami (*snake_case*).
- Nie nadużywaj prefiksów typu `get_`, `set_`.

```
class KontoBankowe:
    def __init__(self, saldo_poczatkowe):
        self.saldo = saldo_poczatkowe
```

Struktura klas:

- Zaleca się zachowanie czytelności, definiowanie metod specjalnych (np. `__str__`) przed zwykłymi metodami.
- Klasa powinna realizować jedną odpowiedzialność (ang. *Single Responsibility Principle*).

Zalecane narzędzia do formatowania automatycznego:

- **flake8** lub **pylint** – kontrola jakości kodu
- **Black** – automatyczny formatter kodu

- **Czytelność = efektywność** – dobry kod jest czytelny i łatwo utrzymywalny.

Pamiętaj, że kod częściej czytasz, niż piszesz!

Narzędzie Flake8

Flake8 to narzędzie służące do analizy jakości i poprawności stylu kodu Python.

Co robi Flake8?

- Szybko wykrywa **błędy składniowe** oraz **naruszenia standardu PEP 8**.
- Zwraca informacje o liniach kodu, które wymagają poprawy.

Kiedy używać Flake8?

- Podczas pisania i przed zatwierdzeniem kodu (commit).
- W procesach automatyzacji (Continuous Integration – CI).

```
$ pip install flake8
```

```
$ flake8 nazwa_pliku.py
$ flake8 folder_z_kodem/
```

Narzędzie pylint

- **pylint** to zaawansowane narzędzie analizy statycznej, służące do szczegółowej oceny jakości kodu Python.

```
$ pip install pylint
```

  - Wykrywa **potencjalne błędy logiczne.**
  - Sprawdza zgodność kodu ze **standardami (PEP 8)**.
  - Wskazuje nieużywane zmienne, zbyt długie funkcje, problemy projektowe, itp.
  - Wystawia **ocenę punktową** (score) jakości kodu.

Kiedy używać `pylint`?

- Regularnie podczas rozwoju dużych projektów.
- W celu poprawy jakości i utrzymania długoterminowego projektu.

```
$ pylint nazwa_pliku.py
$ pylint folder_z_kodem/
```

Narzędzie: black

- **black** to automatyczny formater kodu Python.
- Zapewnia jednolity styl formatowania w całym projekcie.
- **Główne zalety**:
  - Automatycznie stosuje zasady PEP 8.
  - Eliminuje dyskusje na temat stylu kodowania (styl jest narzucony przez narzędzie).

$ black nazwa_pliku.py

Zadanie 1

- Popraw kod biblioteka.py

Budowanie pakietów

Czym jest moduł w Pythonie?

- **Moduł** to pojedynczy plik w Pythonie zawierający definicje funkcji, klas lub zmiennych.
- Pozwala na logiczne organizowanie kodu.
- Ułatwia ponowne używanie kodu.

```
# moj_modul.py
def powitanie(imie):
    print(f"Witaj, {imie}!")
```

Czym jest pakiet w Pythonie?

- **Pakiet** (ang. *package*) to folder zawierający wiele modułów Pythonowych.
- Pozwala na logiczne grupowanie modułów związanych tematycznie.
- Pakiet zawiera specjalny plik `__init__.py`, który mówi Pythonowi, że folder jest pakietem.

```
moja_biblioteka/
│
├── __init__.py
├── matematyka.py
└── teksty.py
```

Do czego służy `__init__.py`?

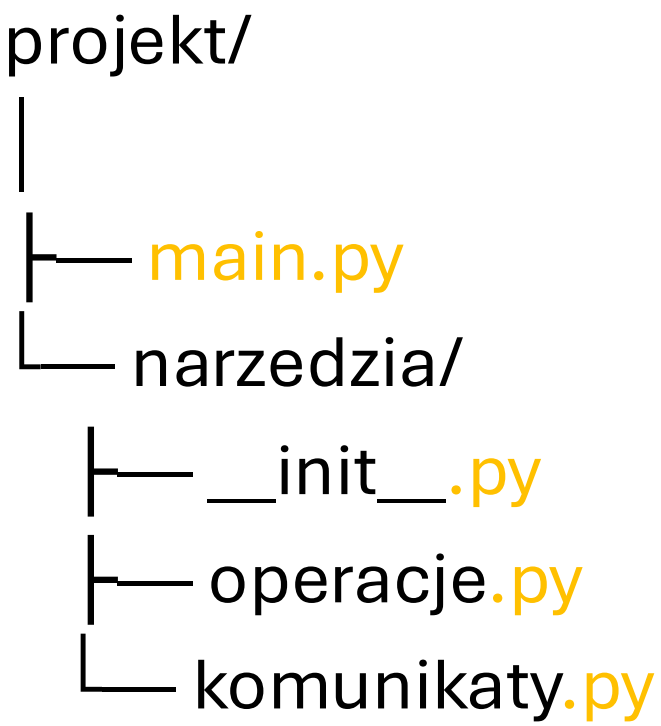
- Plik `__init__.py` informuje interpreter Pythona, że katalog jest pakietem.
- Może być **pusty** lub zawierać inicjalizację pakietu (np. importy, ustawienia).
- Umożliwia wygodne importowanie modułów.

`__init__.py`:

```
# moja_biblioteka/__init__.py
from .matematyka import dodaj, odejmij
```

Jak budować moduły i pakiety?

- Tworzysz logicznie powiązane pliki **`.py`** (moduły).
- Grupujesz je w folderach zawierających plik `__init__.py`.
- Zadbaj o czytelne nazwy modułów i funkcji.



Jak importować moduły i pakiety?

- Import całego modułu:

```
import narzedzia.operacje


narzedzia.operacje.dodaj(2, 3)
```

- Import wybranej funkcji:

```
from narzedzia.operacje import dodah

dodaj(2, 3)
```

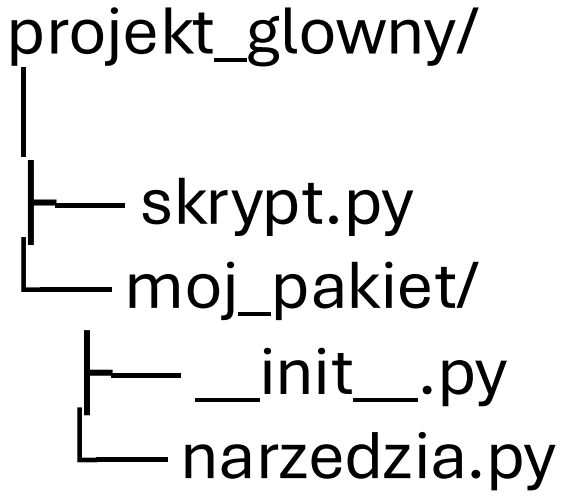
- Import z aliasem:

```
import narzedzia.operacje as op

op.dodaj(2, 3)
```

Pisanie kodu modułu do użytku w innych folderach/projektach

- Aby moduł mógł być używany w innym folderze, najlepiej umieścić go w osobnym pakiecie.
- Twój pakiet powinien znajdować się w katalogu dostępnym dla Python (np. w tym samym folderze co skrypt główny lub instalowany przez pip).




```
# skrypt.py
from moj_pakiet.narzedzia import powitaj

powitaj("Ania")
```

Zobacz github:moj_kalkulator

Zadanie 2

- Napisz pakiet ***geompy***, który będzie zawierał moduły umożliwiające obliczanie podstawowych własności figur geometrycznych

Struktura pakietu Python do globalnej instalacji

- Aby pakiet mógł być instalowany globalnie (np. `pip install .`), musi mieć **odpowiednią strukturę.**
- Najważniejsze elementy:
  - **`setup.py`** – plik zawierający konfigurację pakietu (nazwa, wersja, autor, zależności).
  - **`__init__.py`** – wymagany plik w folderze pakietu, informujący, że folder jest pakietem Python.

  - **Dodatkowe pliki** (opcjonalnie):
    - `README.md` – opis i dokumentacja
    - `LICENSE` – informacje licencyjne

```
moj_pakiet/
├── setup.py          # konfiguracja pakietu
├── README.md         # opis pakietu
├── LICENSE           # licencja
└── moj_pakiet/
    ├── __init__.py   # deklaracja pakietu
    └── modul.py      # kod pakietu
```

Czym jest plik `setup.py`?

- **`setup.py`** to specjalny skrypt w Pythonie, który zawiera konfigurację pakietu przeznaczonego do instalacji za pomocą `pip`.
- Określa szczegóły dotyczące pakietu, takie jak:
  - Nazwa pakietu
  - Wersja pakietu
  - Autor, licencja, opis
  - Wymagane zależności (biblioteki)
  - Pakiety i moduły do zainstalowania

```
from setuptools import setup, find_packages


setup(
    name="moja_biblioteka",
    version="0.1.0",
    author="Jan Kowalski",
    packages=find_packages(),
    install_requires=[
        "numpy>=1.26",
        "requests"
    ],
)
```

Instalacja lokalnych pakietów Python

Czym jest `pip install -e .`?

- Polecenie instaluje lokalny pakiet w trybie edytowalnym (*editable mode*).

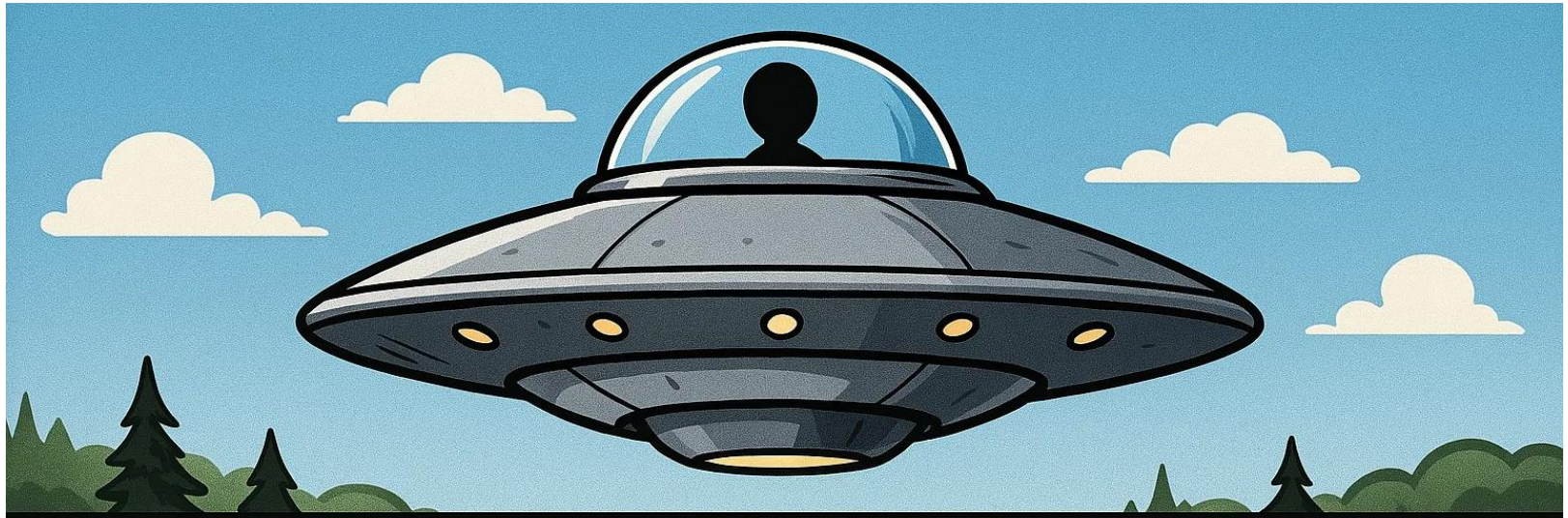
```
$ pip install -e .
```

- Wszelkie zmiany w kodzie pakietu są od razu dostępne bez ponownej instalacji.

Jak zainstalować pakiet globalnie w systemie?

- Aby pakiet był szeroko dostępny (globalnie), można użyć polecenia:

```
$ pip install .
```



```
import ufo_area
h = 4.0     # wysokość w metrach
w = 10.0    # szerokość w metrach
r = 3.0     # promień podstawy w metrach
area = ufo_area.get_area(h,w,r)
print("Powierzchnia statku UFO wynosi: {area:`
                                        .2f) m^2)
```

Przykład: ufo_area

Zadanie 3

- Dodaj plik setup.py do geompy

Poza katologiem pakietu:

```
Import geompy
>> geompy.figury2d.pole_kwadratu(5)
25
```