

# Organizacja Kodu - Moduły i Pakiety

Gdzie jesteśmy? Opanowaliśmy zaawansowane funkcje i styl funkcyjny (**map**, **filter**).

**Cel na dziś:** Nauczyć się profesjonalnie organizować kod, aby był reużywalny i łatwy w utrzymaniu (zasada DRY).

## Agenda spotkania:

a. **Moduły:** Czym są i jak je tworzyć?

b. **Instrukcja import:** Różne sposoby importowania kodu.

c. **Ćwiczenie 1:** Tworzymy i używamy własny moduł z narzędziami.

d. **Magia if `__name__` == `"__main__"`:** Plik jako program i biblioteka.

e. **Ćwiczenie 2:** Zastosowanie if `__name__` == `"__main__"`.

f. **Pakiety:** Jak grupować moduły w foldery?

g. **Importy względne i absolutne.**

h. **Zadania do samodzielnej pracy.**

# Moduły: nasze własne biblioteki

**Problem:** Mamy 10 świetnych funkcji w jednym pliku. Jak użyć ich w innym projekcie bez kopiowania?

**Rozwiązanie: Moduł**

- **Moduł to po prostu plik z rozszerzeniem .py** zawierający kod Pythona (funkcje, zmienne, klasy).
- Każdy plik **.py** jest potencjalnym modulem.

**Jak używać modułów? Instrukcja import**

**Sposób 1 (Zalecany):** `import nazwa_modulu`

- Importuje cały moduł jako oddzielną "przestrzeń nazw".
- Dostęp do zawartości przez kropkę:

`nazwa_modulu.nazwa_funkcji`.

- **Zalety:** Bezpieczny, unika konfliktów nazw.

**Sposób 2 (Wygodny):** `from nazwa_modulu import element`

- Importuje konkretny element (funkcję, zmienną) bezpośrednio do bieżącej przestrzeni nazw.
- **Zalety:** Krótszy zapis. **Wady:** Ryzyko konfliktu nazw.

**Sposób 3 (Antywzorzec):** `from nazwa_modulu import *`

- Importuje **wszystko** z modułu.
- **Wady:** "Zaśmiecą" przestrzeń nazw, prowadzi do trudnych do znalezienia błędów.

# Ćwiczenie 1 - Tworzenie modułu "narzedzia"

**Cel:** Praktyczne utworzenie i zaimportowanie własnego modułu, aby zastosować zasadę DRY (Don't Repeat Yourself).

**Zadanie (Krok po kroku):**

## 1. Utwórz plik modułu:

- W swoim folderze projektu utwórz nowy plik o nazwie **narzedzia.py**.
- Do pliku **narzedzia.py** skopiuj kod uniwersalnego dekoratora **@mierz\_czas** z poprzednich zajęć (wraz z potrzebnymi importami, np. **import time**).

## 2. Utwórz plik główny:

- W tym samym folderze utwórz drugi plik o nazwie **main.py**.

## 3. Użyj modułu:

- W pliku **main.py**, zaimportuj swój nowy moduł, pisząc: **import narzedzia**.
- W **main.py** utwórz prostą funkcję, np. **wolna\_funkcja()**, która używa **time.sleep(2)**.
- Udekoruj ją za pomocą **@narzedzia.mierz\_czas**.
- Wywołaj **wolna\_funkcja()** i zaobserwuj wynik w konsoli.

```
/moj_projekt
|-- narzedzia.py  (zawiera dekorator @mierz_czas)
|-- main.py      (importuje i używa narzedzia)
```

# Omówienie Ćwiczenia 9: Tworzenie i używanie modułu

**Cel:** Analiza rozwiązania i utrwalenie wzorca tworzenia i importowania modułów.

**Kluczowa koncepcja:** Plik **narzedzia.py** stał się naszą własną, reużywalną biblioteką. **import narzedzia** tworzy przestrzeń nazw, dzięki czemu dostęp do dekoratora mamy przez **narzedzia.mierz\_czas**.

**Problem do rozwiązania:** Co, jeśli w **narzedzia.py** chcielibyśmy umieścić kod testujący nasz dekorator? Ten kod uruchomiłby się również w **main.py** podczas importu! Jak temu zapobiec?

# `if __name__ == "__main__":`

**Problem:** Jak sprawić, by plik był jednocześnie biblioteką i samodzielnym programem?

**Rozwiązanie:** Specjalna zmienna `__name__`

- Każdy moduł w Pythonie ma wbudowaną zmienną `__name__`.
- Gdy plik jest uruchamiany **bezpośrednio** (np. `python narzedzia.py`), Python ustawia `__name__ = "__main__"`.
- Gdy plik jest **importowany** (np. `import narzedzia`), Python ustawia `__name__ = "narzedzia"` (nazwa pliku).

**Wniosek:** Kod wewnątrz bloku `if __name__ == "__main__":` staje się "blokiem startowym" programu, który jest ignorowany podczas importu.

## Ćwiczenie 2 - Zastosowanie `if __name__ == "__main__"`

**Cel:** Uczynienie modułu **narzedzia.py** samodzielnym plikiem testowym, który nie wykonuje dodatkowego kodu podczas importu.

**Zadanie (Krok po kroku):**

- Otwórz plik **narzedzia.py** z poprzedniego ćwiczenia.
- Na końcu pliku dodaj blok `if __name__ == "__main__":`.
- Wewnątrz tego bloku umieść kod, który będzie testował Twój dekorator. Możesz stworzyć prostą funkcję testową i ją wywołać.
- Uruchom plik **narzedzia.py** bezpośrednio (np. `python narzedzia.py` w terminalu lub przyciskiem "Run" w IDE). Zaobserwuj, że kod testowy się wykonał.
- Uruchom plik **main.py**. Zaobserwuj, że program działa jak wcześniej, ale kod testowy z **narzedzia.py** nie został wykonany.

# Omówienie Ćwiczenia 2: Plik jako biblioteka i program

**Cel:** Analiza rozwiązania i utrwalenie idiomu

`if __name__ == "__main__":`

**Analiza działania:**

- Gdy uruchamiamy `python narzedzia.py`, `__name__` ma wartość `"__main__"`, więc blok `if` się wykonuje.
- Gdy uruchamiamy `python main.py`, który importuje `narzedzia`, `__name__` w module `narzedzia` ma wartość `"narzedzia"`, więc blok `if` jest ignorowany.

**Co dalej?** Opanowaliśmy moduły (pliki). A co, jeśli projekt rośnie i chcemy grupować moduły w foldery? Czas na **Pakiety**.

# Pakiety: Organizacja modułów w foldery

**Problem:** Mamy wiele modułów (.py). Trzymanie ich wszystkich w jednym folderze prowadzi do chaosu.

## Rozwiązanie: Pakiet (Package)

- Pakiet to po prostu folder zawierający moduły Pythona.
- Aby Python potraktował folder jako pakiet, musi on zawierać specjalny plik: `__init__.py`.
- Plik `__init__.py` może być pusty! Jego sama obecność jest sygnałem dla Pythona.

## Struktura Pakietu:

```
/moj_projekt  
|-- main.py  
|-- /narzedzia_firmowe  
    |-- __init__.py  
    |-- narzedzia.py  
    |-- raporty.py
```

## Jak importować z pakietu? (Składnia z kropką)

- Używamy notacji kropkowej, aby "wejść" do folderu.
- `import moj_pakiet.narzedzia`
- `from moj_pakiet import raporty`
- `from moj_pakiet.narzedzia import mierz_czas`

## Ćwiczenie 3: Tworzymy własny pakiet

**Cel:** Refaktoryzacja kodu do struktury pakietu, aby poprawić organizację i czytelność.

**Zadanie (Krok po kroku):**

**Stwórz folder pakietu:**

- W głównym folderze projektu stwórz nowy folder o nazwie **narzedzia\_firmowe**.

**Dodaj plik `__init__.py`:**

- Wewnątrz folderu **narzedzia\_firmowe** stwórz pusty plik **`__init__.py`**.

**Zorganizuj moduły:**

```
/moj_projekt
|-- main.py
|-- /narzedzia_firmowe
    |-- __init__.py
    |-- pomiar.py
    |-- formatowanie.py
```

- Przenieś swój stary plik **narzedzia.py** do folderu **narzedzia\_firmowe** i zmień jego nazwę na **pomiar.py**.
- W folderze **narzedzia\_firmowe** stwórz nowy plik **formatowanie.py** i wklej do niego prostą funkcję, np. **`def na_wielkie(tekst): return tekst.upper()`**.

**Zaktualizuj plik główny:**

- Otwórz plik **main.py** (który jest poza pakietem).
- Zmień importy, aby odwoływały się do nowej struktury:
  - a) **`from narzedzia_firmowe.pomiar import mierz_czas`**
  - b) **`from narzedzia_firmowe.formatowanie import na_wielkie`**

**Uruchom `main.py`** i sprawdź, czy wszystko działa jak poprzednio.

# Omówienie Ćwiczenia 3: Tworzenie pakietu

**Cel:** Analiza refaktoryzacji kodu do struktury pakietowej.

**Koncepcja:** Zamiast jednego modułu, mamy teraz zorganizowany pakiet. Dostęp do jego zawartości uzyskujemy przez notację kropkową, która odzwierciedla strukturę folderów.

**Finalna Struktura:**

```
/moj_projekt
|-- main.py
|-- /narzedzia_firmowe
    |-- __init__.py
    |-- pomiar.py      (zawiera mierz_czas)
    |-- formatowanie.py (zawiera na_wielkie)
```

# Importy wewnętrzne: Absolutne vs. Względne

**Problem:** Jak moduł `pomiar.py` ma zaimportować funkcję z modułu `formatowanie.py`, skoro oba są w tym samym pakiecie?

## Rozwiązanie 1: Import Absolutny (Zalecany)

- Ścieżka jest podawana od **głównego folderu projektu**.
- Składnia (w `pomiar.py`):  

```
from narzedzia_firmowe.formatowanie import  
na_wielkie
```
- **Zalety:** Jednoznaczny, czytelny, zawsze działa tak samo.
- **Wady:** Dłuższy zapis; jeśli zmienisz nazwę pakietu (`narzedzia_firmowe`), musisz poprawić wszystkie importy.

## Rozwiązanie 2: Import Względny

- Ścieżka jest podawana **względem bieżącego pliku**.
- `.` oznacza "ten sam folder".
- `..` oznacza "folder nadrzędny".
- Składnia (w `pomiar.py`):  

```
from .formatowanie import na_wielkie
```
- **Zalety:** Krótszy zapis; odporny na zmianę nazwy pakietu.
- **Wady:** Może być mylący w złożonych strukturach; nie zadziała, jeśli uruchomisz plik bezpośrednio.

**Złota zasada:** Używaj importów absolutnych dla przejrzystości. Importy względne są przydatne w bardzo dużych, samodzielnych pakietach.

## Ćwiczenie 4 – Komunikacja wewnątrz pakietu

**Cel:** Praktyczne zastosowanie importów (absolutnych) do komunikacji między modułami w tym samym pakiecie.

**Problem:** Chcemy, aby nasz dekorator `@mierz_czas` z modułu **pomiar** wyświetlał swój komunikat wielkimi literami, używając do tego funkcji `na_wielkie` z modułu **formatowanie**.

**Zadanie (Krok po kroku):**

- a. Otwórz plik `narzedzia_firmowe/pomiar.py`.
- b. Na górze pliku dodaj import absolutny, aby zaimportować funkcję `na_wielkie` z sąsiedniego

modułu.

c. **Zmodyfikuj dekorator `mierz_czas`:**

- Znajdź linię z instrukcją `print`, która wyświetla czas wykonania.
- "Owiń" cały f-string w wywołanie funkcji `na_wielkie()`, aby komunikat był wyświetlany wielkimi literami.

d. **Uruchom plik `main.py`** (bez żadnych zmian w nim!).

e. **Zaobserwuj wynik:** Komunikat z dekoratora powinien być teraz wyświetlony wielkimi literami.

## Omówienie Ćwiczenia 4 (Komunikacja wewnątrz pakietu)

**Cel:** Analiza komunikacji między modułami w pakiecie.

**Kluczowa koncepcja:** Moduły wewnątrz pakietu mogą się ze sobą komunikować. Import absolutny (**from nazwa\_pakietu...**) jest najbezpieczniejszym i najbardziej czytelnym sposobem.

## Ćwiczenie 5: Zaawansowane Sortowanie z kluczem

**Problem:** Jak posortować listę złożonych obiektów (np. słowników) według określonego atrybutu (np. wieku)?

**Rozwiązanie:** `sorted(sekwencja, key = funkcja)`

- Argument `key` przyjmuje funkcję, która jest wywoływana na każdym elemencie przed porównaniem.
- Sortowanie odbywa się na podstawie wyników tej funkcji.

**Cel:** Zastosowanie `sorted` i `lambda` do sortowania listy słowników.

**Zadanie:**

- a. Stwórz listę **produkty**, gdzie każdy element to słownik z

kluczami **"nazwa"** (str) i **"cena"** (float). Dodaj 3-4 produkty w dowolnej kolejności.

- b. Użyj funkcji `sorted` i `lambda`, aby posortować listę produkty rosnąco według ceny.
- c. Wynik zapisz w zmiennej **posortowane\_produkty** i wyświetl ją.

**Składnia (z lambda):**

```
studenci = [  
    {"imie": "Anna", "wiek": 22},  
    {"imie": "Piotr", "wiek": 20},  
    {"imie": "Zofia", "wiek": 21}  
]  
# Sortujemy po wartości klucza "wiek" w każdym słowniku  
posortowani = sorted(studenci, key=lambda s: s["wiek"])  
print(posortowani)
```

# Omówienie Ćwiczenia 5: Sortowanie z kluczem

**Cel:** Analiza rozwiązania i utrwalenie wzorca **sorted**  
+ **lambda**.

**Kluczowa koncepcja:** `key = lambda p: p["cena"]`

to "instrukcja" dla sortowania: "Dla każdego produktu `p`, użyj wartości jego klucza **cena** jako kryterium sortowania".

**Co dalej?** Umiemy transformować (**map**), filtrować (**filter**) i sortować (**sorted**). A co, jeśli potrzebujemy prostej odpowiedzi Tak/Nie na temat całej kolekcji?

## Ćwiczenie 6 - Sprawdzanie Warunków z `any` i `all`

**Problem:** Jak szybko sprawdzić, czy **jakikolwiek** lub **wszystkie** elementy w kolekcji spełniają dany warunek?

**Rozwiązanie z Wykładu (W2-2):** `any()` i `all()`

- **`any(sekwencja)`:** Zwraca **True**, jeśli **choć jeden** element w sekwencji jest "prawdziwy".
- **`all(sekwencja)`:** Zwraca **True**, jeśli **wszystkie** elementy w sekwencji są "prawdziwe".

**Cel:** Zastosowanie **`any`** i **`all`** z wyrażeniami generatorowymi do weryfikacji danych.

**Zadanie:**

- a. Stwórz listę oceny = `[2, 5, 3, 4, 2]`.
- b. Używając **`any()`**, sprawdź, czy na liście jest **jakakolwiek** ocena niedostateczna ( $< 3$ ). Wynik zapisz w zmiennej **`czy_jest_zagrozenie`**.
- c. Używając **`all()`**, sprawdź, czy wszystkie oceny są **pozytywne** ( $\geq 3$ ). Wynik zapisz w zmiennej **`czy_wszystkie_pozytywne`**.
- d. Wyświetl obie zmienne.

## Omówienie Ćwiczenia 6 (any i all)

**Cel:** Analiza rozwiązania i utrwalenie wzorca **any/all** z wyrażeniem generatorowym.

**Kluczowa koncepcja:** Wyrażenie generatorowe (**warunek for element in sekwencja**) jest "leniwe" i produkuje wartości **True/False** na żądanie. **any()** i **all()** również są "leniwe" i kończą pracę przy pierwszym rozstrzygającym wyniku.

**Co dalej?** Umiemy transformować, filtrować i sprawdzać. A co, jeśli chcemy "zwinąć" całą kolekcję do jednej wartości?

## Ćwiczenie 7 - Redukcja Danych z reduce

**Problem:** Jak "zwinąć" całą kolekcję do jednej wartości (np. obliczyć iloczyn, znaleźć maksimum)?

**Rozwiązanie:** `functools.reduce`

- **reduce(funkcja, sekwencja):** Kumulacyjnie stosuje **funkcję** do elementów sekwencji, redukując ją do pojedynczej wartości.
- Funkcja musi przyjmować **dwa argumenty: akumulator** (wynik z poprzedniego kroku) i **aktualny\_element**.
- Znajduje się w module **functools**, więc trzeba ją zaimportować: **from functools import reduce**.

**Jak to działa?**

**Cel:** Zastosowanie reduce do agregacji danych.

**Zadanie:**

- a. Zaimportuj **reduce** z modułu **functools**.
- b. Stwórz listę **liczby = [5, 2, 8, 1, 9]**.
- c. Użyj **reduce** i funkcji **lambda**, aby znaleźć **największą** liczbę na liście.
- d. *Podpowiedź:* Twoja lambda powinna przyjmować dwa argumenty (np. **a**, **b**) i zwracać ten, który jest większy.
- e. Wynik zapisz w zmiennej **najwieksza** i wyświetl ją.

```
from functools import reduce
liczby = [1, 2, 3, 4]
# reduce(lambda acc, x: acc + x, liczby)
# Krok 1: acc=1, x=2 -> 1 + 2 = 3
# Krok 2: acc=3, x=3 -> 3 + 3 = 6
# Krok 3: acc=6, x=4 -> 6 + 4 = 10
suma = reduce(lambda acc, x: acc + x, liczby)
print(suma) # -> 10
```

# Omówienie Ćwiczenia 7 (reduce)

**Cel:** Analiza rozwiązania i utrwalenie idei "zwijania" kolekcji.

# Środowiska wirtualne (venv)

**Problem:** Instalujemy pakiety (**pip install ...**)

globalnie. Projekt A potrzebuje **requests == 2.25**, a

Projekt B **requests == 2.28**. **Konflikt wersji!**

**Rozwiązanie:** Środowisko wirtualne (venv) -  
odizolowany "warsztat" dla każdego projektu z własną  
kopią Pythona i własnymi pakietami.

**Jak używać? (w TERMINALU)**

**1. Stwórz środowisko (raz na projekt):**

- **python -m venv nazwa\_srodowiska**  
(np. **python -m venv venv**)

**2. Aktywuj środowisko**

(ZAWSZE, gdy pracujesz nad projektem):

- Windows: **venv\Scripts\activate**
- macOS/Linux: **source venv/bin/activate**

(Znak zachęty w terminalu się zmieni, np. (venv) C:\...)

**3. Zainstaluj pakiety:**

- **pip install requests**  
(instaluje się **tylko** w aktywnym środowisku)

**4. Dezaktywuj środowisko (gdy kończysz pracę):**

- **deactivate**

**Złota zasada:** *Każdy nowy projekt zaczynaj od stworzenia i aktywacji wirtualnego środowiska.*

## Ćwiczenie 8: Cykl Życia Projektu

**Cel:** Praktyczne przećwiczenie cyklu pracy: izolacja (venv), instalacja (pip) i dokumentacja zależności (requirements.txt).

**Zadanie (do wykonania w TERMINALU):**

- **Stwórz nowy folder** dla projektu, np. **moj\_nowy\_projekt** i wejdź do niego.
- Stwórz środowisko wirtualne:  
**python -m venv venv**
- **Aktywuj środowisko:**
  - a. Windows: **venv\Scripts\activate**
  - b. macOS/Linux: **source venv/bin/activate**
- Zainstaluj zewnętrzny pakiet:  
**pip install requests == 2.31.0**
- Sprawdź zainstalowane pakiety: **pip list** (zobaczysz requests i

jego zależności).

- **Stwórz plik main.py** (w tym samym folderze) z prostym kodem używającym **requests**.

```
import requests

response = requests.get("https://google.com/")
print(f"Status odpowiedzi: {response.status_code}")
if response.status_code == 200:
    print("Połączono pomyślnie!")
else:
    print("Wystąpił błąd podczas połączenia.")
```

- **Uruchom program:** **python main.py**
- Zapisz zależności (stwórz "recepturę"):  
**pip freeze > requirements.txt**
- Zajrzyj do pliku **requirements.txt**.
- Zakończ pracę: **deactivate**

# Omówienie Ćwiczenia 8: Cykl Życia Projektu

**Cel:** Zrozumienie i utrwalenie cyklu

**venv -> pip install -> pip freeze.**

**Co osiągnęliśmy?**

- Stworzyliśmy **odizolowane środowisko**, które nie zaśmieca globalnej instalacji Pythona.
- Zainstalowaliśmy pakiet **requests** **tylko** w tym środowisku.
- Stworzyliśmy "**recepturę**" naszego projektu w pliku **requirements.txt**.

**Po co requirements.txt? (Druga strona medalu)**

- Ten plik pozwala **każdemu** odtworzyć Twoje

środowisko pracy.

- **Scenariusz:** Nowy programista pobiera Twój kod.

Co robi?

- a. Tworzy i aktywuje własne, puste venv.
- b. Uruchamia komendę: `pip install -r requirements.txt`
- c. `pip` czyta plik i instaluje wszystkie pakiety w dokładnie tych samych wersjach.

**Kluczowa koncepcja:** venv izoluje projekt.

**requirements.txt** czyni go przenośnym i odtwarzalnym.

# Podsumowanie: Od funkcji do profesjonalnego projektu

Co dzisiaj osiągnęliśmy?

**Organizacja Kodu:** Nauczyliśmy się dzielić kod na moduły i grupować je w pakiety, używając `import` i `if __name__ == "__main__":`.

**Styl Funkcyjny:** Opanowaliśmy narzędzia do zwięzłego przetwarzania danych: **map**, **filter**, **sorted(key = ...)**, **any**, **all**, **reduce**.

**Profesjonalny Workflow:** Przećwiczyliśmy absolutne podstawy: izolację projektów (**venv**),

instalację pakietów (**pip**) i zarządzanie zależnościami (**requirements.txt**).

**Gdzie jesteśmy?** Potrafisz nie tylko pisać kod, ale też budować i zarządzać małymi, kompletnymi projektami w Pythonie.

**Co dalej?**

Skoro już wiemy JAK budować, teraz nauczymy się, jak budować **DOBRCZE**.

**Następny krok:** Czysty Kod, PEP 8 i dobre praktyki.