# Podstawy programowania w Pythonie – laboratorium

Anna Kelm

25 października 2023

# 1 Laboratorium 5

# 2 Klasy abstrakcyjne (3 pkt)

## 2.1 Dekoratory

Klasy, dziedziczenie itd. działają w Pythonie w większości przypadków bardzo podobnie do C++. Jedną z różnic jest "częste i gęste" używanie dektoratorów, które będą na laboratorium za ok. miesiąc. W dużym skrócie, dekorator to funkcja/metoda, która jako argument przyjmuje inną funkcję/metodę i jej argument i coś robi z argumentami/wynikiem. Skrótowo funkcję udekorowaną zapisuje się jako:

```
@my_awsome_decorator
def my_awsome_decorated_function(arg_1, arg_2):
    ...
```

Wywołując funkcję

```
my_awsome_decorated_function("def", "abc")
```

jeśli została ona udekorowana, tak na prawdę wykonuje się:

```
my_awsome_decorator(my_awsome_decorated_function, "def", "abc")
```

## 2.2 Materiały

### 2.2.1 Tematyka zadania

Proszę zapoznać się z rodz. 3.1.3 Wybrane bazy wielomianowe skryptu D. Dąbrowskiej "Wprowadzenie do metod numerycznych".

### 2.2.2 Klasy

Przekazywanie argumentów do konstruktora w przypadku wielodziedziczenia:
https://stackoverflow.com/questions/29311504/multiple-inheritance-with-arguments
Klasy abstrakcyjne:
https://docs.python.org/3.10/library/abc.html

## 2.3 Instrukcje

### 2.3.1 Klasy bazowe

Twoim zadaniem jest zaimplementowanie dwóch klas w języku Python zgodnie z opisem klasy `BasePolynomial`.

- Stwórz abstrakcyjną metodę instancji `eval_basis(self, x)`.
- Stwórz abstrakcyjną metodę klasową `from_natural(cls, natural, *args)`. Argumenty `*args` są to dodatkowe argumenty konstruktora klasy.
- Dodaj (zwykłe, nie abstrakcyjne) pola `degree` i `poly_coefficients` zabezpieczone przed moźliwością modyfikacji ich po utworzeniu instancji (analogicznie zostało to zrobione w klasie `ParameterBasis`).

### 2.3.2 Klasy pochodne

`NaturalPolynomial`

Zaimplementuj klasę `NaturalPolynomial`, która dziedziczy po klasie `BasePolynomial`. Najlepiej w osobnym pliku.

`NaturalPolynomial` ma reprezentować wielomiany w bazie naturalnej.

Zaimplementuj metody z klasy bazowej.

Zaimplementuj metodę `eval_basis(self, x)`. Metoda `eval_basis(self, x)` zwraca listę wartości elementów bazy wielomianu w punkcie `x`. Dla bazy naturalnej indeksy listy odpowiadają potędze `x`.

Zaimplementuj metodę klasową `from_natural(cls, natural, *args)`, która będzie tworzyć nowy obiekt typu `cls` (czyli tutaj: `NaturalPolynomial`) z innego obiektu `NaturalPolynomial`.

`NewtonPolynomial`

Klasa `NewtonPolynomial` dziedziczy po klasach `ParameterBasis` i `BasePolynomial`. Klasa `NewtonPolynomial` ma zawierać następujące metody i konstruktor:

- `eval_basis(self, x)`.
- `__init_(self, poly_coefficients, basis_coefficients)`: Konstruktor klasy `NewtonPolynomial` który przyjmuje dwie listy współczynników - `poly_coefficients` i `basis_coefficients`. Te parametry należy przekazać do konstruktorów klas bazowych.
- `from_natural(cls, natural, *args)`: Metoda klasowa `from_natural` ma być zaimplementowana w klasie `NewtonPolynomial`. Funkcja ta przyjmuje jako argumenty instancję klasy `NaturalPolynomial` i dodatkowe argumenty `*args`. W tym przypadku, ma ona rzucać wyjątek `TooLazyToImplementError`.

`ChebyshevPolynomial`

Ta klasa została już zaimplementowana, niezależnie od `BasePolynomial`. Prawdę mówiąc, posiada one dokładnie te same pola i metody, jak pochodne klasy `BasePolynomial`, więc równie dobrze na potrzeby dalszego ewentualnego użycia wygodnie byłoby sprawić, aby ta klasa "przedstawiała się" jako pochodna klasy `BasePolynomial`. W tym celu w pliku `chebyshev_polynomial.py`, nie modyfikując ciała klasy `ChebyshevPolynomial`, dodaj kod, który sprawi, że klasa `ChebyshevPolynomial` będzie identyfikowana jako pochodna `BasePolynomial`.

### 2.3.3 Testy

Najlepiej napisać je przed implementacją `ChebyshevPolynomial`, `NaturalPolynomial`, `NewtonPolynomial`. Wówczas resztę zadania można wykonać używając klasycznego TDD.

Argumenty do testowania, dotyczą jednego wielomianu zapisanego w różnych bazach:

```
coeff_natural = [1, 4, -2] # wspolczynniki w bazie natrualnej

node_newton = [-1, 0, 1] # węzły dla bazy Newtona
coeff_newton = [-5, 6, -2, 0] # wspolczynniki w bazie Newtona

coeff_chebyshev = [0, 4, -1, 0] # wspolczynniki w bazie Czebyszewa

# punkty do ewaluacji
val_points = list(range(5))
res_points = [1, 3, 1, -5, -15]
```

Napisz testy jednostkowe, które sprawdzają, czy każda z klas `ChebyshevPolynomial`, `NaturalPolynom` `NewtonPolynomial` poprawnie oblicza wartość wielomianu.

Dodatkowo napisz test sprawdzający, czy `ChebyshevPolynomial` istotnie jest pochodną `BasePolynomial`.

# 3 Konteksty (1 pkt)

## 3.1 Materiały

https://docs.python.org/3/reference/datamodel.html#context-managers
https://book.pythontips.com/en/latest/context_managers.html
https://realpython.com/python-with-statement/

## 3.2 Instrukcja

Celem tego zadania jest stworzenie kodu, który będzie mierzył czas wykonywania operacji wyszukiwania elementów w różnych rodzajach kontenerów przy użyciu różnych typów danych. W tym celu użyjemy dwóch klas: `Timer` i `SearchMeasurement`.

Klasa Timer jest jednym z sensowniejszych przykładów użycia własnej klasy kontekstu (zaraz za połączeniami do bazy danych, połączeniami HTTP, pisaniem do plików, użyciem przetwarzania równoległego...). Natomiast klasa `SearchMeasurement`, w której zaimplementowane są metody kontekstu ma jedynie (chyba) dwie zalety:

- czas życia instancji klasy SearchMeasurement kończy się wraz z końcem bloku `with ...`, zasoby są od razu zwalniane, nie trzeba używać `del` ani garbage collectora (po polsku: "śmieciarz"?)
- kod wygląda bardzo "pythonicznie".
- Napisz klasę `Timer`, która będzie realizowała menedżer kontekstu. Obiekt klasy `Timer` użyty w następujący sposób:

```
with Timer():
    <<jakiś kod>>
<<jakiś inny kod>>
```

ma wypisywać czas wykonania kodu `<<jakiś kod>>`.

- Napisz klasę `SearchMeasurement`, która ma inicjować się z następującymi parametrami:
  - `num_elements` - liczba elementów do umieszczenia w kontenerze.
  - `num_search` - liczba operacji wyszukiwania.
  - `dtype` - typ danych, który ma być używany do wypełnienia kontenera (np. `str` lub `int`).
  - `container_type` - rodzaj kontenera (np. `list`, `set` lub `collections.Counter`).
- W konstruktorze klasy `SearchMeasurement`, stwórz kontener zawierający `num_elements` różnych elementów typu `dtype`. Wybierz losowych `num_search` elementów z kontenera i zapisz je jako `random_container`.
- Napisz metodę `execute`, która będzie wyszukiwała każdy element z `random_container` w kontenerze.
- Klasa `SearchMeasurement` ma również być menedżerem kontekstu.
- Stałe do użycia w programie:

```
N = 2**18
K = 10000
DTYPES = str, int
CONTAINER_DTYPES = list, set, Counter
```

- W głównym bloku programu przetestuj czasy wykonania metody `SearchMeasurement.execute` dla różnych kombinacje typów danych i rodzajów kontenerów.

Ostatecznym wynikiem powinno być wypisanie czasu trwania operacji wyszukiwania dla różnych typów danych i rodzajów kontenerów. Upewnij się, że implementacja jest zgodna z podanymi wyżej instrukcjami.

**Podpowiedź**: Możesz wykorzystać bibliotekę `random` do losowego wybierania elementów z kontenera.