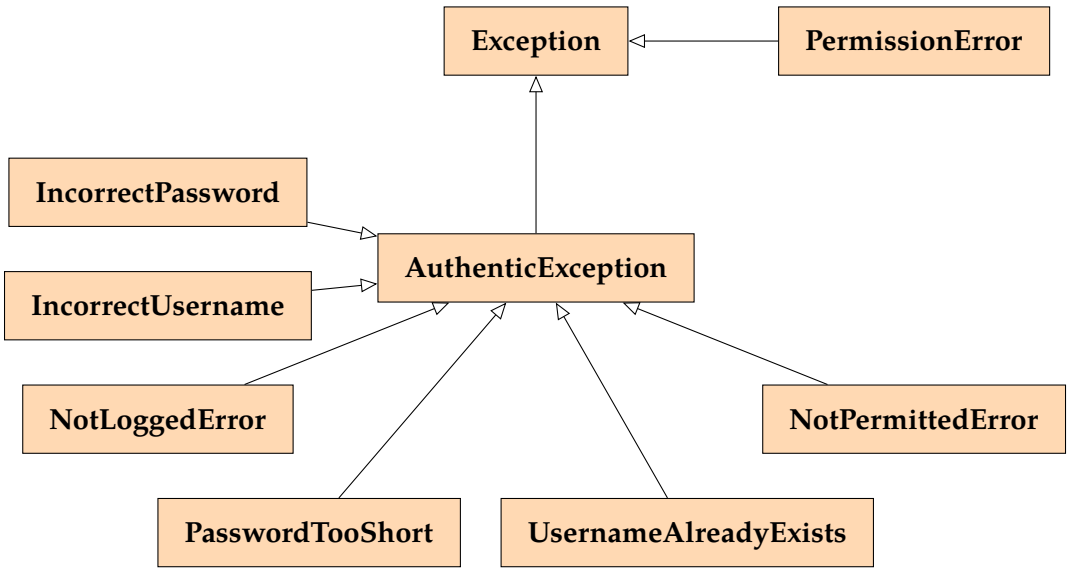
# Lista zadań nr 3

**Zadanie 1** Napisz program, który symuluje prosty system uwierzytelniania i nadawania uprawnień użytkownikom. W module `authorization_system` utwórz następujące klasy:

- `User` - klasa przechowująca nazwę użytkownika i zaszyfrowane hasło. Klasa powinna zawierać metody:
  - `__init__()` - inicjalizuje atrybuty:
    - `username` - nazwę użytkownika podaną podczas tworzenia obiektu,
    - `password` - zaszyfrowane hasło (z wykorzystaniem metody prywatnej `_encrypt_password()`),
    - `is_logged` - początkowo ustawione na `False`.
  - `_encrypt_password()` - metoda zaszyfrowująca nazwę użytkownika i hasło, zwracająca ich skrót (np. z użyciem funkcji `SHA-256` z modułu `hashlib`); można np. połączyć nazwę i hasło w jeden ciąg.
  - `check_password()` - sprawdza, czy podane hasło zgadza się z przechowywanym w atrybucie; zwraca `True` lub `False`.
- Hierarchię klas wyjątków (bez dodatkowych metod), zgodną z poniższym diagramem UML:




- Klasa `Authenticator` - kontrolująca użytkowników. Zawiera metody:
  - `__init__()` - tworzy pusty słownik `users`, przechowujący pary `{username: user_object}`.
  - `add_user()` - dodaje nowego użytkownika, jeśli:

- użytkownik o tej nazwie nie istnieje (w przeciwnym wypadku zgłasza wyjątek `UsernameAlreadyExists`),
- hasło ma więcej niż 7 znaków (w przeciwnym wypadku zgłasza wyjątek `PasswordTooShort`).

◇ `login()` - loguje użytkownika (ustawia `is_logged` na `True`); zgłasza wyjątek `IncorrectUsername`, jeśli użytkownik nie istnieje, oraz wyjątek `IncorrectPassword`, jeśli hasło jest niepoprawne. Zwraca `True`, gdy logowanie zakończy się sukcesem.

◇ `is_logged_in()` - zwraca `True` lub `False`, w zależności od tego, czy użytkownik jest zalogowany.

- Klasa `Authorizor` - zarządza uprawnieniami użytkowników. Zawiera metody:

◇ `__init__()` - tworzy pusty słownik `permissions` (zawierający pary `{permission: set_of_usernames}`) oraz atrybut `authenticator`.

◇ `add_permission()` - dodaje nowe uprawnienie do słownika jako klucz z pustym zbiorem; zgłasza wyjątek `PermissionError`, jeśli uprawnienie już istnieje.

◇ `permit_user()` - przypisuje użytkownikowi dane uprawnienie; zgłasza wyjątki `PermissionError` lub `IncorrectUsername`.

◇ `check_permission()` - sprawdza, czy użytkownik ma podane uprawnienie; zgłasza wyjątki:

- `NotLoggedError` - gdy użytkownik nie jest zalogowany,
- `PermissionError` - gdy uprawnienie nie istnieje,
- `NotPermittedError` - gdy użytkownik nie ma danego uprawnienia.

Zwraca `True`, jeśli użytkownik ma dane uprawnienie.

Na końcu modułu utwórz instancje klas `Authenticator` oraz `Authorizor` (druga instancja przyjmuje jako argument pierwszą).

W głównym programie:

- Utwórz kilku użytkowników i przypisz im różne uprawnienia (np. testowania i/lub edytowania programów).
- Zdefiniuj klasę `Editor` z podstawowym interfejsem menu, zawierającą metody:

◇ `__init__()` - inicjalizuje atrybuty: `username` (wartość `None`) oraz `options` (słownik, który ma postać `{"a": self.login, "b": self.test, "c": self.change, "d": self.quit}`).

- ◇ `login()` - pobiera nazwę i hasło użytkownika, wywołując metodę instancji `Authenticator` z obsługą wyjątków.
- ◇ `is_permitted()` - sprawdza, czy użytkownik jest zalogowany i posiada wymagane uprawnienia.
- ◇ `test()` - imituje testowanie programu (korzysta z `is_permitted()`).
- ◇ `change()` - imituje edytowanie programu (korzysta z `is_permitted()`).
- ◇ `quit()` - kończy działanie programu.
- ◇ `run()` - działa w pętli, pobiera opcję od użytkownika i wywołuje odpowiadającą jej metodę (należy użyć funkcji `input()`).

Główny program powinien utworzyć instancję klasy `Editor` i wywołać metodę `run()`. Przykład inicjalizacji:

```
auth = Authenticator()
authorizor = Authorizor(auth)
editor = Editor()
editor.run()
```

**Zadanie 2** Napisz nieskończony generator produkujący kolejne dodatnie liczby całkowite (od 1 wzwyż), a następnie generator kolejnych kwadratów dodatnich liczb całkowitych, wykorzystujący poprzedni generator.

Zaprojektuj i napisz funkcję `select()`, która zwraca listę pierwszych $n$ wartości dowolnego obiektu iterowalnego (np. generatora). W implementacji użyj funkcji `iter()` oraz `next()`. Przetestuj funkcję na wcześniej zdefiniowanych nieskończonych generatorach.

Zdefiniuj generator produkujący trójelementowe krotki zawierające tzw. *trójki pitagorejskie*, tj. krotki postaci $(a, b, c)$, dla których $a^2 + b^2 = c^2$ oraz $a < b < c$. Wyświetl 15 pierwszych takich trójek, korzystając z funkcji `select()`.

**Zadanie 3** Napisz klasę implementującą iterator ciągu Fibonacciego, zwracający kolejne wyrazy ciągu mniejsze od $n > 0$. Wykonaj to samo zadanie, definiując odpowiedni generator (funkcję generatora) wykorzystującą instrukcję `yield`.

Następnie, na podstawie nieskończonego generatora ciągu Fibonacciego, utwórz iterator zwracający liczby od $F_{100000}$ do $F_{100020}$ i zapisz te liczby do pliku tekstowego (każda liczba w osobnym wierszu).

Ile cyfr ma liczba $F_{100000}$?

**Zadanie 4** Napisz generator `gen_primes()`, który produkuje kolejne liczby pierwsze. Zapisz 10 000 początkowych liczb pierwszych do pliku tekstowego (każda liczba w osobnym wierszu).

Jak dużą liczbę pierwszą jesteś w stanie wygenerować w rozsądnym czasie?

**Zadanie 5** Napisz rekurencyjną funkcję generatora, która przekształca zagnieżdżoną sekwencję na postać jednowymiarowej listy wartości, nie rozwijając ciągów tekstowych (tj. traktując napisy jako pojedyncze elementy), np.:

`([1, 'kot'], 3, (4, 5, [7, 8, 9]))` → `[1, 'kot', 3, 4, 5, 7, 8, 9]`.

**Zadanie 6** Napisz generator `gen_time()`, który produkuje kolejne sekwencje czasu w postaci krotek `(godziny, minuty, sekundy)`. Generator powinien przyjmować trzy argumenty: czasy `start`, `stop` i `step` (każdy w postaci krotki `(godziny, minuty, sekundy)`). Zamiast zwykłych krotek można wykorzystać krotki nazwane.

Przykładowe działanie programu:

```
>>> for time in gen_time((8, 10, 00), (10, 50, 15), (0, 15, 12) ):
    print(time)


(8, 10, 0)
(8, 25, 12)
(8, 40, 24)
(8, 55, 36)
(9, 10, 48)
(9, 26, 0)
(9, 41, 12)
(9, 56, 24)
(10, 11, 36)
(10, 26, 48)
(10, 42, 0)
>>>
```

**Zadanie 7** Napisz generator `permute(items)`, który dla listy `items` generuje wszystkie jej permutacje (jako krotki). Nie korzystaj z modułu `itertools`. Kolejność generowania może być dowolna, byle każda permutacja pojawiła się dokładnie raz.
Przykładowe działanie:

```
for p in permute([1, 2, 3]):
    print(p)
```

Oczekiwany wynik:

```
(1, 2, 3)
(1, 3, 2)
(2, 1, 3)
(2, 3, 1)
(3, 1, 2)
(3, 2, 1)
```

Zaprojektuj wersję `permute_unique(items)` zwracającą *unikalne* permutacje bez duplikatów, jeśli w `items` występują powtórzenia, (np. `[1,1,2]`).

**Zadanie 8** Napisz generator `collatz(n)`, który dla danej liczby całkowitej $n > 0$ generuje kolejne elementy sekwencji Collatza aż do osiągnięcia wartości $1$ (włącznie).
Reguły sekwencji Collatza:

- jeśli $n$ jest parzyste, następny element to $n//2$;
- jeśli $n$ jest nieparzyste, następny element to $3n + 1$.

Przykład:

```
list(collatz(12))  # -> [12, 6, 3, 10, 5, 16, 8, 4, 2, 1]
```

Zaimplementuj klasę `CollatzSequence`, której instancja jest *iteratorem* zwracającym kolejne elementy sekwencji Collatza (jak powyżej). Zaimplementuj zgodnie ze sztuką metody `__iter__()` oraz `__next__()`.
Zdefiniuj funkcje pomocnicze:

- ◇ `stopping_time(n)` - zwraca długość drogi do `1` (liczbę elementów *bez* zliczania wartości początkowej lub z, ale konsekwentnie; opisz w docstringu),
- ◇ `max_value(n)` - największa wartość w ciągu Collatza startując od $n$.

Przetestuj dla $n = 1, \ldots, 100$ i wypisz $n$ o największym `stopping_time`.

**Zadanie 9** Napisz generator `estimate_pi(seed = None, batch = 1)`, który w każdej iteracji zwraca kolejne przybliżenie liczby $\pi$ metodą Monte Carlo. W każdej iteracji wykonaj `batch` nowych losowań i zaktualizuj estymator.
Reguły:

- w każdym kroku dolosuj `batch` punktów `(x,y)` równomiernie z kwadratu $[-1, 1] \times [-1, 1]$;
- zliczaj punkty w kole jednostkowym $x^2 + y^2 \leqslant 1$;
- po k całkowitych losowaniach zwróć estymator $4 \cdot \frac{\#\{\text{w kole}\}}{k}$.

Przykładowe użycie:

```
gen = estimate_pi(seed=123, batch=1000)
for _ in range(5):
    print(next(gen))
```

Napisz dodatkowo funkcję `run_until_pi(eps, seed = None, batch = 1000)`, która korzysta z `estimate_pi` i zatrzymuje się, gdy `abs(pi_hat - math.pi) < eps`. Zwróć parę: `(pi_hat, total_samples)`.

**Zadanie 10** Napisz generator `estimate_e()`, który przybliża wartość liczby $e$ na podstawie szeregu Taylora:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}.$$

Każde wywołanie `next()` ma zwracać coraz dokładniejsze przybliżenie `e`, aktualizując inkrementalnie $n!$ i sumę częściową (bez przeliczania od zera).
Przykład:

```
gen = estimate_e()
for _ in range(5):
    print(next(gen))
```

Zaimplementuj funkcję `approx_e(eps)`, która korzysta z `estimate_e()` i przerywa działanie, gdy różnica między dwiema kolejnymi sumami jest mniejsza niż `eps`. Funkcja powinna zwracać parę: `(e_hat, terms_used)`, gdzie:

- `e_hat` – ostatnie (najdokładniejsze) przybliżenie liczby `e`,
- `terms_used` – liczba wyrazów szeregu (iteracji generatora) wykorzystanych do osiągnięcia tego przybliżenia.

**Zadanie 11** Napisz generator `random_walk(start=0, p=0.5, max_steps=None)`, który symuluje jednowymiarowy losowy spacer zaczynający się od pozycji `start`. W każdej iteracji pozycja zmienia się o `+1` (z prawdopodobieństwem `p`) lub `-1` (z prawdopodobieństwem `1 - p`). Jeśli `max_steps` nie jest `None`, generator kończy działanie po `max_steps` krokach.
Przykład:

```
walk = random_walk(start=0, p=0.5)
for _ in range(10):
    print(next(walk))
```

Dodaj parametry barier `left`, `right` (liczby całkowite). Generator kończy działanie, gdy pozycja osiągnie którąś z barier.