Laboratorium 7 – Elementy programowania funkcyjnego w Python.

Cele dydaktyczne

1. Zapoznanie z elementami programowania funkcyjnego w Python.
2. Zapoznanie z tworzeniem iteratorów i generatorów.
3. Zapoznanie z tworzeniem dekoratorów.

Zadania

1. Nie wykorzystując imperatywnych instrukcji **for**, **while**, **if**[1], opracuj implementację każdej z poniższych funkcji:
   a. Funkcja **acronym**, która przyjmuje na wejściu listę ciągów znaków zwraca akronim zbudowany z tych ciągów znaków. Przykład:

   ```
   >>>  acronym(["Zakład", "Ubezpieczeń", "Społecznych"])

   ZUS
   ```

   b. Funkcja **median,** która przyjmuje na wejściu listę liczb i zwraca ich medianę. Funkcja **nie może korzystać** z modułu **statistics** ani żadnego innego modułu do obliczeń statystycznych. Przykład:

   ```
   >>> median([1,1,19,2,3,4,4,5,1])

   3
   ```

   c. Funkcja obliczająca pierwiastek kwadratowy metodą Netwona. Funkcja przyjmuje na wejściu pierwiastkowaną liczbę x oraz *epsilon* i zwraca taki y, że $y \geq 0$ i $|y^2 - x| < epsilon$.

   Przykład:

   ```
   >>> pierwiastek(3, epsilon = 0.1)

   1.75
   ```

[1] O ile nie zostaną użyte w ramach tzw. *list/dict comprehensions* lub operatora ternarnego .

d. Funkcja **make_alpha_dict**, która przyjmuje na wejściu ciąg znaków, a zwraca na wyjściu słownik, w którym kluczami są znaki występujące alfabetyczne występujące ciągu, a wartościami listy słów zawierających te znaki. Przykład:

```
>>> make_alpha_dict(“on i ona”)

{'o': ['on', 'ona'], 'n': ['on', 'ona'], 'i': ['i'], 'a': ['ona']}
```

e. Funkcja **flatten** spłaszczająca listy. Funkcja powinna przyjmować listę, której elementami mogą być elementy skalarne lub sekwencje. Spłaszczenie polega na zmianie zagnieżdżonej struktury na jednowymiarową listę zawierającą wszystkie elementy wewnętrznych sekwencji. Spłaszczenie powinno działać na wszystkich poziomach zagnieżdżeń, tzn. wynikowa lista powinna zawierać tylko elementy skalarne. Na potrzeby zadania należy przyjąć, że elementy skalarne to takie, które nie są listami ani krotkami. Przykład:

```
>>> flatten([1, [2, 3], [[4, 5], 6]])

[1, 2, 3, 4, 5, 6]
```

`Punkty:2`

2. Zaproponuj implementacje następujących funkcji wyższego rzędu przyjmujących predykat *pred* (unarną funkcję zwracającą wartość logiczną), `iterable` (obiekt, po którym można iterować) i potencjalnie *n* - dodatnią liczbę całkowitą.
   a. `forall(`*pred*`, iterable)` - funkcja zwraca True, jeśli każdy element iterable spełnia predykat *pred*, w przeciwnym przypadku False,
   b. `exists(`*pred*`, iterable)` – funkcja zwraca True, jeśli co najmniej jeden element iterable spełnia predykat pred, w przeciwnym przypadku False,
   c. `atleast(n,` *pred*`, iterable)` – funkcja zwraca True, jeśli co najmniej n elementów iterable spełnia predykat pred, w przeciwnym przypadku False.
   d. `atmost(n,` *pred*`, iterable)` – funkcja zwraca True, jeśli co najwyżej n elementów iterable spełnia predykat pred, w przeciwnym przypadku False.

`Punkty:2`

3. Skonstruuj klasę `PasswordGenerator`, która będzie obsługiwać protokół iteratora. Iterator powinien zwracać kolejne losowo generowane hasła. Iterator powinien mieć następujące metody:
   a. `__init__(self, length, charset, count)`: funkcja inicjująca iterator z parametrami:
      i. długością hasła oraz
      ii. zestawem znaków, z których losowo będą tworzone hasła (domyślnie

wszystkie litery alfabetu oraz cyfry),
   iii. maksymalną liczbą haseł do wygenerowania.
b. `__iter__(self)`: metoda zwracająca iterator
c. `__next__(self)`: metoda zwracająca kolejne losowo wygenerowane hasło.

   Po wygenerowaniu `self.count` haseł, podnieś wyjątek `StopIteration`. Przetestuj iterator wywołując jawnie wbudowaną funkcję `next()` oraz w ramach pętli `for`.

`Punkty:1,5`

4. Wykorzystując domknięcia, skonstruuj funkcję **`make_generator`** zwracającą generator. Niech funkcja **`make_generator`** przyjmuje jako parametr jednoargumentową funkcję *f* i zwraca generator. Generator powinien leniwie obliczać wartości funkcji *f* dla kolejnych argumentów, rozpoczynając od 1.
   a. Przetestuj funkcję **`make_generator`** przekazując jako funkcję *f* samodzielnie zaimplementowaną funkcję reprezentującą znany ciągi liczbowy, np. Fibonacciego, ciąg Catalana, etc.
   b. Przetestuj funkcję **`make_generator`** przekazując jako funkcję *f* wybrane ciągi liczbowe zaimplementowane jako wyrażenia lambda, np. ciągi arytmetyczne, ciągi geometryczne, ciągi potęgowe.

   `Punkty:1,5`

5. Korzystając z modułu functools, utwórz funkcję `make_generator_mem`, która działa tak, jak `make_generator`, ale memoizuje funkcję *f*. Implementację wykonaj w taki sposób, aby uniknąć duplikowania kodu.

   `Punkty:1`

6. Skonstruuj dekorator **`log`**, który będzie służył do dekorowania funkcji lub klas. Dekoracja ma polegać na logowaniu danych o wywołaniu funkcji z wykorzystaniem modułu logging.
   a. Udekorowana funkcja powinna logować informację o czasie wywołania, czasie trwania, nazwie funkcji oraz jej argumentach i wartości zwracanej. Dekorator powinien przyjmować poziom logowania jako argument (np. `DEBUG`, `INFO` itp.).
   b. W przypadku udekorowania klasy, logowany powinien być fakt jej zainstancjonowania, tzn. utworzenia obiektu.

   `Punkty:2`