PRZEGLĄD PODSTAWOWYCH BIBLIOTEK PYTHONA

ĆWICZENIA DO PREZENTACJI

MODUŁ 8

Altkom Akademia S.A., materiały własne

1 PRZYDATNE MODUŁY

ĆWICZENIE 1.1:

Implementacja iteratora

UMIEJĘTNOŚCI:

- Po wykonaniu ćwiczenia zdobędziesz umiejętności:
  - implementacji własnych iteratorów
  - ich wykorzystania

CELE I ZADANIA:

- Utwórz klasę implementującą protokół iteratora
- Tworząc iterator należy poprzez argumenty przekazać przedział liczbowy
  - wartość początkowa jest opcjonalna – domyślna wartość to zero
  - wartość końcowa jest obowiązkowa
- Zadaniem iteratora jest zwracanie kolejnych liczb z zadanego przedziału – każdą dwukrotnie

ALGORYTM WYKONANIA:

- Utwórz klasę o nazwie *DoubleIterator*
- Zadaniem iteratora jest zwracanie kolejnych liczb z podanego przedziału, powtarzając każdą dwukrotnie
- Zdefiniuj metodę *__init__* i przekaż do niej wartości definiujące przedział liczbowy:
  - wartość początkowa jest opcjonalna z domyślną wartością zero
  - wartość końcowa jest obowiązkowa
- Metoda powinna zdefiniować atrybuty instancyjne przechowujące:
  - zakres przedziału
  - wartość bieżącą
  - flagę kontrolującą, czy wartość bieżąca jest zwracana pierwszy, czy drugi raz
- Utwórz metodę *__iter__* zwracającą obiekt iteratora (może to być bieżąca instancja klasy)
- Utwórz metodę *__next__* odpowiedzialną za zwrócenie kolejnej wartości lub powtórzenie bieżącej
- Pamiętaj, aby zakończyć działanie iteratora wyrzuceniem wyjątku *StopIteration*
- Przetestuj działanie iteratora dla różnych przedziałów

ĆWICZENIE 1.2:

Implementacja generatora

UMIEJĘTNOŚCI:

- Po wykonaniu ćwiczenia zdobędziesz umiejętności:
  - tworzenia funkcji generatorów
  - wykorzystania generatorów do implementacji iteratorów

CELE I ZADANIA:

- Wzorując się na rozwiązaniu poprzedniego ćwiczenia 1.1, utwórz iterator dublujący każdy element za pomocą funkcji generatora

ALGORYTM WYKONANIA:

- Utwórz funkcję *double_generator* z dwoma parametrami definującymi przedział liczbowy
  - wartość początkowa jest opcjonalna – domyślna wartość to zero
  - wartość końcowa jest obowiązkowa
- Funkcja powinna zwracać kolejne wartości z zadanego przedziału, każdą dwukrotnie
- Do zwrócenia wartości z funkcji generatora użyj polecenia `yield`
- Przetestuj działanie generatora dla różnych przedziałów

ĆWICZENIE 1.3:

Wieczny kalendarz – typowanie statyczne

UMIEJĘTNOŚCI:

- Po wykonaniu ćwiczenia zdobędziesz umiejętności tworzenia kodu typowanego statycznie

CELE I ZADANIA:

- Korzystając z rozwiązania ćwiczenia 1.1 z modułu 4, uzupełnij kod o deklaracje typów
- Sprawdź, czy niezgodności typów mogą być wykryte przed uruchomieniem programu

ALGORYTM WYKONANIA:

- Skopiuj rozwiązanie ćwiczenia 1.1 z modułu 4
- Dodaj deklaracje typów:
  - zmiennych
  - parametrów funkcji
  - wartości zwracanych z funkcji
- Opisz typy tak szczegółowo, jak się da (czy można zadeklarować typy elementów w krotce?)
- Sprawdź, czy po tych zmianach aplikacja działa
- Dokonaj wywołania funkcji z błędnymi typami argumentów
- Czy można wykryć taką niezgodność przed uruchomieniem aplikacji?

ĆWICZENIE 1.4:

Użycie cache'a

UMIEJĘTNOŚCI:

- Po wykonaniu ćwiczenia:
  - zdobędziesz umiejętność wykorzystania cache'owania wyników
  - przekonasz się, jaki wpływ ma ono na wydajność aplikacji

CELE I ZADANIA:

- Napisz dwie funkcje obliczające kolejne wyrazy ciągu Fibonacciego: z cache'owaniem wyników i bez
- Porównaj czasy wykonania obu funkcji

ALGORYTM WYKONANIA:

- Napisz funkcję obliczającą kolejne wyrazy ciągu Fibonacciego – wykorzystaj rekurencję
  - pierwsze dwa wyrazy ciągu to jedynki
  - każdy kolejny jest sumą dwóch poprzednich, czyli:
    $F_1 = 1$
    $F_2 = 1$
    $F_n = F_{n-2} + F_{n-1}$ dla $n \geqslant 3$
- Napisz drugą funkcję obliczającą to samo, ale wykorzystującą cache'owanie wyników
  - funkcja bedzie miała taką samą treść co poprzednia
  - należy ją tylko opisać odpowiednim dekoratorem z modułu *functools* (jakim?)
  - zastanów się nad wielkością rozmiaru cache'a
- Zmierz czas wykonania obu funkcji dla tego samego wyrazu ciągu
  - możesz do tego wykorzystać moduł *time* i jego funkcję *time()*
  - zczytaj czas przed i po wykonaniu funkcji, a nastepnie oblicz różnicę

ĆWICZENIE 1.5:

Pomiar czasu

UMIEJĘTNOŚCI:

- Po wykonaniu ćwiczenia zdobędziesz umiejętność wykorzystania modułu *timeit* do pomiaru czasu wykonania kodu

CELE I ZADANIA:

- Zrealizuj to samo zadanie, co w poprzednim ćwiczeniu
- Tym razem do pomiaru czasu wykonania funkcji wykorzystaj możliwości modułu *timeit*

ALGORYTM WYKONANIA:

- Skopiuj rozwiązanie poprzedniego ćwiczenia
- Zmodyfikuj funkcję pomiaru czasu wykonania funkcji obliczających wyrazy ciągu Fibonacciego
- Użyj funkcji *timeit* z modułu *timeit*
- Zastanów się, jakich zmian dokonać w funkcjach, których czas mierzymy

2 KOLEKCJE

ĆWICZENIE 2.1:

Porównanie słowników i nazwanych krotek

UMIEJĘTNOŚCI:

- Po wykonaniu ćwiczenia zdobędziesz umiejętność:
  - definiowania nazwanych krotek
  - porównywania wielkości obiektów

CELE I ZADANIA:

- Napisz funkcję umożliwiającą konwersję słownika do krotki nazwanej
- Porównaj wielkości obu obiektów – wykorzystaj moduł *pympler.asizeof*

ALGORYTM WYKONANIA:

- Utwórz krotkę nazwaną zawierającą te same dane, co słownik
  - utwórz krotkę nazwaną i podaj nazwy pól (pobierz nazwy kluczy ze słownika)
  - zapełnij krotkę danymi (pobierz wartości ze słownika)
- Porównaj wielkość krotki nazwanej i słownika
- W tym celu:
  - zainstaluj pakiet *pympler* i zaimportuj funkcję *asizeof* z modułu *pympler.asizeof*
  - użyj funkcji do odczytu wielkości obu obiektów
  - porównaj ich rozmiary

ĆWICZENIE 2.2:

Implementacja polecenia 'tail'

UMIEJĘTNOŚCI:

- Po wykonaniu ćwiczenia zdobędziesz umiejętność:
  - pracy z plikami tekstowymi
  - użycia kolekcji

CELE I ZADANIA:

- Napisz program, który pozwoli wypisać zadaną liczbę linii z końca podanego pliku tekstowego
- Do określenia liczby linii powinny posłużyć opcje wiersza poleceń: $-n$ lub $--lines$ lub w razie ich pominięcia – wartość domyślna 10

ALGORYTM WYKONANIA:

- Napisz moduł o nazwie *tail*, który pozwoli wypisać zadaną liczbę linii z końca podanego pliku tekstowego
- Składnia wywołania programu:

```
tail /path/to/file
tail -n <number_of_lines> /path/to/file
tail --lines <number_of_lines> /path/to/file
```

- W razie pominięcia opcji $-n$ lub $--lines$, domyślną wartością liczby linii powinno być 10
- Nazwa wraz z położeniem pliku powinna być ostatnim parametrem wywołania
- Do odczytu i wypisania zadanej liczby ostatnich linii z pliku wykorzystaj kolekcję *deque* z modułu *collections* (dlaczego właśnie tę?)
- Sprawdź, czy program działa poprawnie