Obsługa wyjątków, pliki oraz metaprogramowanie

Obsługa wyjątków

Do obsługi wyjątków służy instrukcja try-except:

```
try:
    <blok chroniony>

except <TypWyjątku> [ as <zmienna> ]:
    <blok wyjątku>

except:
    <blok wyjątku>

else:
    <blok else>

finally:
    <blok finally>
```

Sekcja try

Blok sekcji try to instrukcje chronione – wystąpienie wyjątku podczas ich wykonywania powoduje przerwanie wykonywania tego bloku, a następnie obsłużenie wyjątku. Po sekcji try musi być chociaż jedna sekcja except albo sekcja finally

```
try:
    y = 2 / x

print('y =', y)  # błąd, brak except/finally
```

Sekcja except

Każda sekcja except ze wskazaniem typu i opcjonalnie przypisaniem wyjątku do zmiennej obsługuje wyjątki zgodnego typu. Poprzez zmienną wyjątku można uzyskać dodatkowe informacje o nim, przy czym zakres tych informacji zależy od typu wyjątku;

```
try:
    y = 2 / x

except ZeroDivisionError as e:
    print('dzielenie przez zero!', e.args)

except FloatingPointError:
    print('błąd operacji zmiennoprzecinkowych')
```

Sekcja except

Sekcja except bez wskazania typu wyjątku może wystąpić tylko raz i musi być ostatnia; zostanie wykonana, jeżeli wyjątek nie należy do żadnego typu spośród wskazanych w poprzednich sekcjach except:

```
try:
    y = 2 / x
```

```
except ZeroDivisionError:
    print('dzielenie przez zero!')

except:
    print('inny błąd')
```

Podobny efekt można uzyskać tworząc sekcję dla wyjątku typu Exception (wspólny typ bazowy dla wszystkich wyjątków), ale dodatkowo można uzyskać dodatkowe informacje o wyjątku

```
try:
    y = 2 / x

except ZeroDivisionError:
    print('dzielenie przez zero!')

except Exception as e:
    print('inny błąd,', e.args)
```

Sekcja else

Sekcja else jest opcjonalna: może wystąpić wyłącznie tuż po ostatniej sekcji except i zostanie wykonana tylko wtedy, kiedy w sekcji try nie wystąpi żaden wyjątek

```
try:
    y = 2 / x

except:
    print('wystąpił błąd')

else:
    print('nie było błędu')
```

Sekcja finally

Sekcja finally może wystąpić jako ostatnia w instrukcji try-except Jest wykonywana zawsze, niezależnie od wystąpienia wyjątku, zwykle służy do zwalniania zasobów wykorzystywanych w try. Wykonanie samej sekcji finally (jeżeli wyjątek nie został dopasowany do żadnej sekcji except albo jeżeli w ogóle nie ma sekcji except) NIE oznacza obsłużenia wyjątku – program zostanie zatrzymany

```
try:
    y = 2 / x
    print('y =', y)

finally:
    print('tak czy inaczej koniec')
```

Rzucanie wyjątków

Do rzucenia wyjątku służy instrukcja raise:

```
raise [ <wyrażenie> [ from <wyrażenie>] ]
```

Wyjątek musi być obiektem klasy BaseException lub klasy potomnej albo nazwą takiej klasy (wówczas Python sam utworzy obiekt). Można pominąć <wyrażenie>, wówczas zostanie powtórzone ostatnio rzucony wyjątek. Klauzula from służy do tworzenia łańcuchów

wyjątków (wskazuje wyjątek, będący przyczyną powstania drugiego wyjątku); Jeżeli pierwszy nie zostanie obsłużony, środowisko wyświetli oba:

```
raise
raise Exception
raise Exception('nie jest dobrze')
raise Exception('nie jest dobrze') from ValueException
```

Można wykorzystać jedną z wbudowanych klas wyjątków, (np. Exception, ValueError, ArithmeticError, UserWarning, …), ale nie w kodzie produkcyjnym – wbudowane wyjątki mają ściśle określony powód wystąpienia, ich użycie może być mylące. Powinno się zdefiniować własne klasy wyjątków

```
try:
    if gamma < 2:
        raise ModelValueError('wsp. gamma', gamma)
    pathLoss = distance**gamma

except ModelValueError as e:
    print('Nieprawidłowa wartość:', e.Name, e.Value)
    sys.exit(0)
```

Typy wyjątków

Python ma kilkadziesiąt wbudowanych klas wyjątków, a dodatkowe moduły mogą dodawać kolejne. Nazwy i hierarchię klas wyjątków oraz opisy powodu wystąpienia można znaleźć w dokumentacji: https://docs.python.org/3/library/exceptions.html#. Poradniki także publikują opisy wyjątków, chociaż mniej szczegółowo, np.: https://www.programiz.com/python-programming/exceptions

Protokół zarządzania kontekstem

Protokół zarządzania kontekstem jest nieformalnym interfejsem, implementowanym przez funkcje specjalne __enter__ i __exit__ (w konwencji "kaczego typowania"). Jest powiązany z instrukcją with. Może obejmować również obsługę wyjątków – ale niekoniecznie, a jeśli już to niekoniecznie wszystkich.
Zazwyczaj wiąże się z wykorzystaniem zasobów zewnętrznych, np. obiektów synchronizacji wątków lub plików. Można także definiować własne klasy tego typu

Instrukcja with

Może być stosowana wyłącznie dla obiektów implementujących protokół zarządzania kontekstem:

```
with <wyrażenie> [ as <zmienna> ]:
    <blok with>
```

Albo, prawie równoważnie:

```
<zmienna> = <wyrażenie>
with <zmienna>:
    <blok with>
```

Wykonując instrukcję with, program najpierw wykonuje <wyrażenie>, którego rezultatem powinien być obiekt implementujący zarządzanie kontekstem, inaczej rzuca wyjątek. Następnie wykonywany jest protokół wejścia, przez wywołanie funkcji specjalnej __enter__, enter(zmienna), potem blok instrukcji <blok with> oraz protokół wyjścia, jak można się domyślić przy użyciu funkcji specjalnej __exit__, exit(zmienna).

Protokół wyjścia zostanie wykonany nawet wówczas, gdy podczas wykonywania bloku zostanie rzucony wyjątek: wewnętrzna implementacja instrukcji with go przechwyci, a informacje o typie wyjątku przekaże funkcji wyjścia __exit__. Jeżeli funkcja wyjścia uzna, że sama "posprząta" , zwraca True, co kończy instrukcję with, natomiast jeżeli nie, zwraca False i ten sam wyjątek jest rzucany ponownie, już poza instrukcję with.

Instrukcja with NIE obsługuje wyjątków otwarcia/odczytu/zapisu plików; Zapewnia jedynie zamknięcie pliku, jeżeli uda się go otworzyć, a wyjątek nastąpi podczas odczytu lub zapisu pliku – czego nie da się łatwo osiągnąć przy użyciu instrukcji try.

Pliki

Podstawowe funkcje obsługi plików są zawarte w szkielecie Pythona (odczyt i zapis plików tekstowych i binarnych), natomiast biblioteka standardowa dostarcza wiele dodatkowych funkcji, np. operacje na plikach i katalogach, odczyt i zapis specyficznych formatów plików itp. Biblioteka standardowa zapewnia obsługę plików, m.in.: zip, gzip, tar, lzma, csv, wav oraz dane w formatach xml, json. W sieci jest dostępnych wiele modułów do obsługi plików, w różnych formatach, m.in.: pdf, xls/xlsx, doc/docx, gif, jpg, mp3, mp4, …

Do obsługi plików przydają się instrukcje try i with:

- try
  obsługa wyjątków zawsze jest dobrym zwyczajem, jednak w przypadku korzystania z zasobów zewnętrznych, np. plików, wręcz powinnością: operacje na plikach mogą zakończyć się niepowodzeniem nie z winy programu i zawsze powinny być chronione instrukcją try
- with
  instrukcja with służy do obsługi obiektów implementujących protokół zarządzania kontekstem, do których w Pythonie należą m.in. pliki; umożliwia obsłużenie błędu powstającego przy próbie otwarcia pliku, czego nie zapewnia instrukcja with

Funkcja open i metody wbudowane do obsługi plików

Proste operacje na plikach tekstowych i binarnych można zrealizować korzystając z funkcji i metod wbudowanych Pythona: Funkcja open otwiera plik, natomiast metody, m.in. read, seek, tell, write i close, wykonują działania na otwartym wcześniej pliku.

- open(file, mode='rt')
  otwiera wskazany plik i zwraca obiekt go reprezentujący file – ścieżka i nazwa pliku (ścieżka może być względna), mode – tryb otwarcia pliku ('r' – tylko do odczytu, plik musi istnieć, 'w' – tylko do zapisu, plik nie musi istnieć, 'r+', 'w+' – do odczytu i zapisu, ale 'w+' czyści plik, 't' – tryb tekstowy, 'b' – tryb binarny); domyślnie plik otwierany jest w trybie 'rt'. Jeżeli pliku nie uda się otworzyć rzuca wyjątek OSError

  ```
  plik = open('d:/archiwum/plik.txt', 'r+t')
  plik = open('./plik.txt')                    # mode='rt'
  ```

- read(size=-1)
  odczytuje i zwraca tekst o długości size (domyślnie cały tekst pliku) jako jeden łańcuch znaków

- readlines(hint=-1)
  odczytuje i zwraca cały tekst pliku jako listę łańcuchów znaków; jeżeli liczba znaków osiągnie hint, kończy na najbliższym końcu linii; jest też iteratorem pliku
- readline(size=-1)
  odczytuje i zwraca linię, do znaku końca linii włącznie, ale nie więcej niż size znaków; Python nie ma funkcji w rodzaju EndOfStream – po dojściu do końca pliku funkcje czytające plik zwracają pusty łańcuch znaków

```
plik = open('plik.txt')
for line in plik:
    print(line)
```

- tell()
  zwraca aktualną pozycję punktu odczytu/zapisu pliku
- seek(offset, whence=0)
  przesuwa pozycję punktu odczytu/zapisu pliku, na offset znaków licząc miejsca wskazanego przez whence (0 – od początku pliku, 1 – od bieżącej pozycji, 2 – od końca pliku);

```
plik = open('plik.txt')

plik.seek(7)
second7 = plik.read(7)

plik.seek(0)
first7 = plik.read(7)
```

- write(text)
  zapisuje do pliku łańcuch znaków na bieżącej pozycji, nie dodaje końca linii
- writelines(lines)
  zapisuje do pliku kolekcję łańcuchów znaków na bieżącej pozycji, nie dodaje końca linii;

```
dane = [ 'alfa', 283, 'bravo', 96528.75 ]

plik = open('plik.txt', 'r+')
plik.seek(0, 2)
plik.writelines( str(d)+'\n' for d in dane )
```

- closed
  właściwość informująca czy plik jest zamknięty
- writable()
  sprawdza czy plik jest otwarty do zapisu
- readable()
  sprawdza czy plik jest otwarty do odczytu
- seekable()
  sprawdza czy można przesuwać punkt odczytu/zapisu pliku
- close()
  zamyka plik;

```
plik = open(…)
# …
```

```
if not plik.closed and plik.writable():
    plik.seek(0, 2)
    plik.write('===\n')
```

Obsługa wyjątków przy obsłudze plików

Obsługa wyjątków w przypadku plików nie jest trywialna. Użycie samej instrukcji try może nie wystarczyć. Pokazana poniżej konstrukcja zadziała tylko wtedy, gdy uda się otworzyć plik i błąd nastąpi podczas odczytu lub zapisu, ale jeżeli nie uda się otworzyć pliku, to sekcja finally wygeneruje nowy wyjątek – ponieważ zmienna 'plik' nie istnieje

```
try:
    plik = open(…)
    plik.write(…)

except:
    print('nie udało się… ')

finally:
    plik.close()
```

Zalecane (choć nie jedyne możliwe) rozwiązanie tego problemu polega na połączeniu instrukcji try i instrukcji with zarządzania kontekstem.

```
try:
    with open(…) as plik:
        plik.write(…)

except:
    print('nie udało się… ')
```

Zadziała zawsze, ponieważ instrukcja with, w ramach protokołu wyjścia, zamknie plik, o ile jest on otwarty; Dodatkowe korzyści to fakt, że nie trzeba pamiętać o zamknięciu pliku – plik zawsze zostanie zamknięty, niezależnie od tego, czy błąd wystąpi, czy nie.

Metaprogramowanie

Metaprogramowanie to technika umożliwiająca programom tworzenie lub modyfikację kodu programów. Przedrostek meta- (z greckiego μετά – po, poza) – oznacza przekraczający, wskazujący na wyższy poziom lub szerszy zakres znaczenia. Metaprogramowanie może służyć dwóm różnym celom:

- tworzenie programu, który tworzy inne programy lub modyfikuje ich działanie
- tworzenie kodu, który modyfikuje działanie samego siebie

Z metaprogramowaniem wiąże się tzw. refleksja – jest to mechanizm, który daje wgląd do definicji klas, obiektów i funkcji w trakcie działania programu, przez co umożliwia zarządzanie kodem w podobny sposób, jak zwykłe programy zarządzają danymi;

Język Python wspiera metaprogramowanie w obu wymienionych aspektach. Dla przykładu, program GNU Radio Companion, popularny program języka Pthon służący do tworzenia modeli i prototypów radia programowalnego, umożliwia projektowanie diagramu przepływu sygnałów, po czym na podstawie tego diagramu tworzy samodzielny program, również języka Python, implementujący ów diagram. Z kolein przykładem kodu modyfikującego samego siebie są tzw. dekoratory, które zostaną przedstawione poniżej.

Domknięcie

Domknięcie (closure) to obiekt wiążący referencję funkcji oraz jej środowisko (lub inaczej otoczenie funkcji), tj. dane nielokalne, które mają wpływa na wykonanie funkcji; W praktyce: program zapamiętuje nieco więcej, niż wynika z przypisania.

W pierwszym przykładzie program zapamiętuje w zmiennej ref nie tylko adres funkcji, ale też adres obiektu. Interpreter tworzy domknięcie, ponieważ do wywołania funkcji przez zmienną ref są potrzebne oba adresy:

```
s1 = Student('Kowalski', 'Adam', …)
s2 = Student('Nowak', 'Michał', …)

ref = s1.getName   # referencja metody i obiektu
ref()              # jak s1.getName()
```

Drugi przykład pokazuje funkcję wrapper, zwracającą jako rezultat adres innej funkcji, funkcji inner. Również w tym przypadku interpreter utworzy domknięcie, ponieważ funkcja inner potrzebuje do działania wartości wr_arg, lokalnej zmiennej funkcji wrapper. Domknięcie zawiera zatem dwa adresy – funkcji inner i argumentu wr_arg.

- Przykład 2 – funkcja

```
def wrapper(arg)
    wr_arg = arg

    def inner():
        print(wr_arg)

    return inner

ref = wrapper('Hello!')  # referencja metody i obiektu
ref()                    # domknięcie, drukuje 'Hello'!
```

Dekoratory

Dekorator jest rodzajem metaprogramowania – dodaje nowe funkcjonalności do już istniejącego kodu; Podobne rozwiązania są dostępne w C#/Java (jako atrybuty metod). Dekorator jest funkcją, która otrzymuje jako argument funkcję dekorowaną i zwraca funkcję ją zastępującą. Python ma dedykowany dla dekoratorów, skrócony zapis ze znakiem "@":

```
@decorator
def function(…):
   pass
```

Zapis ten nie jest nowym elementem języka, powyższy zapis jest w pełni równoważny następującemu

```
def function(…):
    pass

function = decorator(function)
```

Przykład typowego prostego dekoratora i jego użycia przedstawiono poniżej. Dekorator zwraca funkcję inner, które zastępuje funkcję dekorowaną. Funkcja inner wywołuje funkcję dekorowaną (jej referencja 'org_fun' jest zapamiętana w domknięciu!), ale dodatkowo wykonuje swoje działania – może to zrobić tuż przed lub tuż po wywołaniu funkcji dekorowanej. Funkcja inner ma wgląd do wartości argumentów przekazywanych do funkcji dekorowanej – może zatem w określonych sytuacjach zablokować jej wywołanie. Może

również przechwycić rezultat zwracany przez funkcję dekorowaną i dowolnie go wykorzystać, a nawet podmienić.

```
def decorator(f):
    org_fun = f

    def inner(*args, **kwargs):
        print('-- inner begin --')
        res = org_fun(*args, **kwargs)
        print('-- inner end --')
        return res

    return inner

@decorator
def function():
  print('Hello')
```

Dekorator z argumentami (argumenty także są pamiętane w domknięciu) jest bardziej złożony: jest to funkcja, która zwraca właściwy dekorator, który z kolei zwraca funkcję zastępującą funkcję dekorowaną:

```
def decorator(x):
    deco_x = x
    def wrapper(f):
        org_fun = f
        def inner(*args, **kwargs):
            # można użyć deco_x
            return org_fun(*args, **kwargs)
        return inner
    return wrapper

@decorator(100)
def function():
    pass
```

Dekorator może zostać również zdefiniowany w postaci klasy, co ułatwia realizację niektórych funkcji dekoratorów, np. zbierania informacji o kolejnych wywołaniach funkcji dekorowanej. Dla każdej dekorowanej funkcji tworzony jest odrębny obiekt klasy dekoratora, a inicjalizator obiektu otrzymuje jako argument funkcję dekorowaną. Obiekt ten zastępuje funkcję dekorowaną, a w konsekwencji wywołanie oryginalnej funkcji jest zastępowane wywołaniem metody __call__ obiektu dekoratora. Obiekt klasy dekoratora może zbierać dowolne dane o kolejnych wywołaniach funkcji, w których pośredniczy, a później te dane udostępnić przy pomocy odrębnych, dedykowanych metod lub właściwości.

```
class decorator:
    def __init__(self, f):
        self.__f = f

    def __call__(self, *args, **kwargs):
        # zbieranie danych
        return self.__f(*args, **kwargs)

    def getInfo():
        # wyświetlenie zebranych danych
```

```
@decorator
def function()      # jak function = decorator(function)

function()          # jak function.__call__()

function.getInfo()  # wyświetlenie zebranych danych
```

Zastosowania dekoratorów

Dokoratory przydają się relatywnie rzadko, jednak w pewnych sytuacjach są niezastąpione:

- Modyfikacja dostępu do metod, @classmthod i @staticmethod
- Implementacja mechanizmów niedostępnych w składni Pythona, np. właściwości (dekorator @property) albo "przeciążanie" funkcji (dekorator @singledispatch z modułu functools)
- Cache – zapamiętywanie rezultatów zwracanych przez funkcję, dzięki czemu nie trzeba wielokrotnie obliczać tego samego, np. dekoratory @cache lub @lru_cache z modułu functools
- Profilowanie – pomiar czasu wykonywania funkcji oraz zliczanie wywołań funkcji, przydatne do optymalizacji programów

Kolekcja różnych dekoratorów jest dostępna wraz przykładami użycia na oficjalnej stronie Python Wiki: https://wiki.python.org/moin/PythonDecoratorLibrary.

Uzupełnienie

Poniżej kilka zagadnień, które będą przydatne przy rozwiązywaniu zadań.

Ścieżka do plików użytkownika

Moduł `os.path` zawiera kilka użytecznych funkcji związanych ze ścieżkami i plikami, m.in. umożliwia uzyskanie ścieżki do pulpitu oraz dokumentów użytkownika

```
import os
import glob

desktop = os.path.expanduser("~\\Desktop")
listaPlików = glob.glob(desktop + '\\*.*')

print(str.format('Zawartość folderu {0}:', desktop))
for plik in listaPlików:
    print(plik)


dokumenty = os.path.expanduser("~\\Documents")
```

Podział łańcucha znaków

Metoda `split(sep=None)` dzieli łańcuch na części i zwraca je w postaci listy; metoda `strip()` usuwa z łańcucha białe znaki wiodące i końcowe. Jeżeli separator nie zostanie podany w wywołaniu, to separatorami są „białe znaki” – spacja, tabulator i znak końca linii, przy czym kilka takich znaków pod rząd jest traktowanych jak jeden „odstęp”

```
line = input()
words = line.split()
```

Jeżeli separator zostanie podany i wystąpi kilka separatorów pod rząd, to w wyniku podziału powstają też łańcuchy puste lub zawierające tylko białe znaki; można je usunąć np. przy pomocy wyrażenia listowego

```
line = input()
words = [p.strip() for p in line.split(':') if p.strip()]
```

Pomiar czasu

Moduł `datetime` dostarcza definicje klas do przetwarzania daty i czasu. Do pobrania bieżącego czasu można użyć funkcji `datetime.now()`, a do obliczenia i wyświetlenia różnicy czasu wystarczy operator odejmowania, ale można też skorzystać z funkcji `total_seconds()` klasy `timedelta`

```
from datetime import datetime

begTime = datetime.now()
# …
endTime = datetime.now()
delta = endTime - begTime
print(delta)                    # 0:00:23.934369
print(delta.total_seconds()) # 23.934369
```

Przykładowe dane studentów

```
44405;44405@student.umg.edu.pl;Adamczyk;Norbert
44661;44661@student.umg.edu.pl;Kątniak;Krzysztof
43521;43521@student.umg.edu.pl;Wiśniewski;Krystian
44513;44513@student.umg.edu.pl;Wolak;Oskar
44833;44833@student.umg.edu.pl;Wojak;Igor
43728;43728@student.umg.edu.pl;Lewandowski;Kacper
44029;44029@student.umg.edu.pl;Dobrzyński;Krzysztof
45169;45169@student.umg.edu.pl;Czajkowski;Błażej
44364;44364@student.umg.edu.pl;Pruński;Bartosz
44323;44323@student.umg.edu.pl;Opiekun;Marija
44483;44483@student.umg.edu.pl;Popiel;Damian
44666;44666@student.umg.edu.pl;Groenwald;Michał
43319;43319@student.umg.edu.pl;Popiel;Damian
44306;44306@student.umg.edu.pl;Kątniak;Krzysztof
```

Ciąg Fibonacciego

```
def fibo(n):
    if n<=0:
        return 0
    elif n==1:
        return 1
    else:
        return fibo(n-1) + fibo(n-2)
```

Zliczanie liczb pierwszych

```
def CountPrimes(start, stop):
    count = 0
```

```
    for n in range(start, stop):
        nd = 0
        for d in range(1, n+1):
            if n%d == 0:
                nd += 1

        if nd==2:
            count += 1
    return count

setSize = 1000
for i in range(10):
    print(i, CountPrimes(i*setSize, (i+1)*setSize))
```

Zadania

Proszę napisać program, który…

1. Kopiuje zawartość pliku tekstowego na ekran
   Plik powinien być wcześniej utworzony w notatniku i zapisany na pulpicie
2. jw. ale nazwę pliku podaje użytkownik
3. jw. ale z obsługą wyjątków
   (należy połączyć instrukcję try-except i zarządzanie kontekstem – instrukcję with)
4. Zapisuje do pliku tekstowego na pulpicie liczby od 1 do 20 i ich kwadraty
5. jw. ale do pliku csv (w polskiej wersji językowej Windows dane należy rozdzielić średnikami, a nie przecinkami); plik powinien się otwierać w MS Excel
6. * Odczytuje z pliku tekstowego dane studentów i przekształca je w słownik,
   gdzie kluczem jest numer albumu, a wartością lista bądź krotka pozostałych danych studenta
7. * jw. ale z obsługą wyjątków
   (należy połączyć instrukcję try-except i zarządzanie kontekstem – instrukcję with)
8. Oblicza (funkcją rekurencyjną) i wyświetla pierwszych 100 liczb ciągu Fibonacciego
   Kiedy już się okaże, że to nie zadziała, obliczenia można przerwać (<Control> + C)
9. jw. ale z użyciem dekoratora lru_chache z modułu functools
10. Dodaje dekorator (w postaci funkcji) do funkcji zliczającej liczby pierwsze
    Należy wykorzystać prosty dekorator funkcyjny; dekorator powinien tylko wyświetlać komunikat przed oraz po wywołaniu oryginalnej funkcji
11. jw. ale dekorator oblicza i wyświetla czas wykonywania funkcji
12. *Dodaje dekorator (w postaci klasy) do funkcji zliczającej liczby pierwsze
    Należy wykorzystać prosty dekorator klasowy; dekorator powinien wyświetlać komunikat przed oraz po wywołaniu oryginalnej funkcji
13. jw. ale dekorator oblicza i wyświetla czas wykonywania funkcji
14. ** jw. ale dekorator zapamiętuje wszystkie czasy wykonania funkcji i udostępnia je przez metodę albo przez właściwość
15. *** jw. ale dodatkowo udostępnia metodę, która wyznacza i wyświetla minimalny, maksymalny i średni czas wykonywania funkcji
    Wskazówka: wartości minimum i maksimum można wyznaczyć bezpośrednio z listy różnic czasu, natomiast do obliczania wartości średniej należy użyć wyrażenia listowego, przekształcającego listę różnic czasu (które są typu timedelta) na listę wartości typu float, do czego przydaje się metoda total_seconds()
16. *** jw. ale z udoskonaloną funkcją CountPrimes(), która wykonuje obliczenia szybciej – co powinny potwierdzać wyniki pomiaru czasu wykonywania obliczeń
17. *** Wykonuje zadanie 9, ale z własnym dekoratorem klasowym, który udostępnia dodatkowo statystyki użycia funkcji, np. liczba wywołań dla określonego argumentu
    Wskazówka: klasa dekoratora powinna mieć prywatne pole w postaci słownika, gdzie kluczem jest krotka argumentów funkcji, a wartością – rezultat funkcji