Python

Poziom średniozaawansowany

Autor: Michał Nowotka

Plan na dzisiaj

1. Lambda – anonimowe funkcje
2. Dekoratory
3. Praca z plikami tekstowymi .txt
4. Praca z plikami testowymi .csv

LAMBDA – FUNKCJA ANONIMOWA

Wyrażenia lambda

```
In [60]: def identity(x):
    ...:     return x
    ...:
```

vs

```
identity = lambda x: x
```

Wyrażenia lambda

- Wyrażenie lambda to inaczej anonimowa funkcja.
- Jest to funkcja która, nie posiada nazwy i jest definiowana w miejscu jej użycia.
- Ponieważ nie jest zwykłą funkcją nie definiuje się jej za pomocą słowa kluczowego **def**. Służy do tego oddzielne słowo kluczowe **lambda**.
- Po słowie kluczowym następuje lista argumentów rozdzielonych przecinkami.
- Po liście argumentów pisze się dwukropek a po nim pojedyncze wyrażenie, które zwraca wartość funkcji.
- Lambda może mieć w swoim ciele tylko jedno wyrażenie.

```
In [1]: my_lambda = lambda x: x.lower()

In [2]: my_lambda('HA HA HA')
Out[2]: 'ha ha ha'

In [3]: square_lambda = lambda x: x**2

In [4]: square_lambda(4)
Out[4]: 16

In [5]: equals_lambda = lambda x, y: x == y

In [6]: equals_lambda(1, 1)
Out[6]: True

In [7]: equals_lambda(1, 2)
Out[7]: False
```

Autor: Michał Nowotka

map, reduce, filter

- Można zapytać do czego służą wyrażenia lambda?
- Najczęstszym miejscem ich użycia są funkcje **map**, **reduce** oraz **filter**.
- Wbudowana funkcja **map** przyjmuje dwa parametry. Pierwszym z nich jest jednoargumentowa funkcja, która zostanie użyta do przekształcenia (przemapowania) wszystkich elementów. Drugim parametrem jest kolekcja tych elementów. Funkcja ta przekształca elementy kolekcji używając podanej funkcji, którą najczęściej jest wyrażenie lambda.
- Funkcja **reduce** ze standardowego modułu **functools** redukuje kolekcję elementów podanych jako drugi parametr przy użyciu dwuargumentowej funkcji podanej jako pierwszy parametr. Funkcje **map** i **reduce** są często używane łącznie - najpierw przekształcamy elementy a później je redukujemy.

```
In [1]: items = [1, 2, 3, 4, 5]

In [2]: squared = list(map(lambda x: x**2, items))

In [3]: squared
Out[3]: [1, 4, 9, 16, 25]

In [4]: from functools import reduce

In [5]: square_sum = reduce((lambda x, y: x + y), squared)

In [6]: square_sum
Out[6]: 55
```

map

- dla każdego elementu z listy LISTA wykonaj funkcję FUNC,
- przetworzone elementy można przypisać do nowej listy,
- bez rzutowania list(map(…)) funkcja map zwraca generator.

```
map(FUNC, LISTA)
```

```
new_list = list(map(FUNC,LISTA)
```

map, reduce, filter

Z listy pierwszych dziesięciu liczb naturalnych, utwórz listę kwadratów pierwszych liczb naturalnych wykorzystując:

a) pętlę for,
b) listę składaną (list comprehension),
c) funkcję map oraz lambdę.

reduce

- redukuje listę LISTA do jednego elementu na podstawie funkcji FUNC,
- Funkcja FUNC przyjmuje dwa argumenty i zwraca wynik operacji między tymi argumentami.

```
reduce(FUNC, LISTA)
```

```
new_value = reduce(FUNC,LISTA)
```

map, reduce, filter

Na podstawie listy liczb od 10 do 20, wylicz iloczyn liczb się w niej znajdujących wykorzystując:

a) pętlę for,

b) funkcję reduce oraz lambdę.

Autor: Michał Nowotka

map, reduce, filter

- Wbudowana funkcja **filter**, podobnie jak dwie poprzednie również przyjmuje jako swoje parametry funkcję oraz kolekcję. Funkcja musi być jednoargumentowa.
- Jako wynik zwraca te elementy kolekcji, dla których funkcja zwróciła **True**. Te dla których funkcja zwróciła **False** zostały odfiltrowane.
- Na przykładzie obok funkcja odfiltrowuje parzyste elementy kolekcji liczb całkowitych z przedziału od jednego do stu.
- W efekcie zostały nam tylko liczby nieparzyste.

```
In [3]: odds = list(filter(lambda x: x%2, range(1, 101)))

In [4]: odds
Out[4]:
[1,
 3,
 5,
 7,
 9,
 11,
 13,
 15,
 17,
 19
```

Autor: Michał Nowotka

filter

- dla każdego elementu z listy LISTA, sprawdź czy spełnia on warunek z funkcji WARUNEK.
- WARUNEK to funkcja zwracająca True lub False w poszczególnych, zdefiniowanych przypadkach.
- Jeżeli dla danego elementu wartość zwrócona z funkcji WARUNEK będzie True, element nie zostanie odfiltrowany.
- przetworzone elementy można przypisać do nowej listy,
- bez rzutowania list(filter(…)) funkcja filter zwraca generator.

```
filter(WARUNEK, LISTA)
```

```
new_list = list(filter(WARUNEK,LISTA)
```

map, reduce, filter

Z listy pierwszych dziesięciu liczb naturalnych odfiltruj liczby niepodzielne przez 2 wykorzystując:

a) pętlę for,
b) listę składaną (list comprehension),
c) funkcję filter oraz lambdę.

map, reduce, filter - czy warto stosować?

- Obecnie używanie funkcji **map**, **reduce** oraz **filter** wychodzi z mody.
- Wynika to z potęgi wyrażeń typu comprehension (list, set, dict comprehensions).
- Większość wyrażeń napisanych przy użyciu **map**, **reduce** i **filter** da się zapisać za pomocą comprehensions, które są bardziej idiomatyczne w Pythonie.
- Dlatego też warto się zastanowić zanim użyjemy tych funkcji.

Funkcje sorted, min, max

- Wbudowane funkcje **sorted**, **min** oraz **max** poznaliśmy znacznie wcześniej.
- O czym nie wspomnieliśmy to fakt, że wszystkie trzy przyjmują opcjonalny argument o nazwie **key**.
- **key** jest funkcją, za pomocą której zostaną przekształcone elementy kolekcji, na których liczymy **min**, **max** czy **sorted**. Funkcja **min** zwróci ten element, którego wartość **key(element)** jest najmniejsza. Funkcja sorted również uporządkuje elementy według ich klucza.
- Domyślnie kluczem jest funkcja tożsamościowa, przekształcająca element na samego siebie.
- Czasami jednak mamy listę par i chcielibyśmy wybrać element, który ma najmniejszy drugi element pary albo posortować obiektu klasy **Employee** po imieniu.
- Do tego przyda nam się parameter **key**, do którego zazwyczaj przekazujemy wyrażenie lambda.

```
In [1]: pairs = [(1, 10), (2, 9), (3, 8)]

In [2]: min(pairs)
Out[2]: (1, 10)

In [3]: max(pairs)
Out[3]: (3, 8)

In [4]: min(pairs, key=lambda x:x[1])
Out[4]: (3, 8)

In [5]: max(pairs, key=lambda x:x[1])
Out[5]: (1, 10)

In [6]: min(pairs, key=lambda x: x[0] * x[1])
Out[6]: (1, 10)

In [7]: max(pairs, key=lambda x: x[0] * x[1])
Out[7]: (3, 8)
```

Autor: Michał Nowotka

Funkcje sorted, min, max

Zaimplementuj własną funkcję max (nazwij ją maximum), która przyjmuje dwa argumenty: listę oraz klucz, zgodnie z którym ma wyszukiwać największą wartość w liście. Obydwa parametry są obowiązkowe.

DEKORATORY

Dekoratory

- Do tej pory nauczyliśmy się używać gotowych dekoratorów, takich jak **@staticmethod**, **@classmethod** czy **@abstractmethod**.
- Teraz spróbujemy nauczyć się definiować własne dekoratory.
- To ćwiczenie może pozwolić nam zrozumieć jak potężnym narzędziem w Pythonie są funkcje.

Autor: Michał Nowotka

Funkcje - obiekty pierwszej kategorii

- W Pythonie funkcje są obiektami pierwszej kategorii. To znaczy, że funkcję (nie tylko jej wynik) można przypisać do zmiennej.
- Można ją przekazać jako argument do innej funkcji.
- Na przykładzie obok, funkcja **greet_kryśka**, przyjmuje jako argument inną funkcję, po czym wywołuje ją jako argument podając imię “Kryśka”.

```
In [1]: def say_hello(name):
   ...:     print(f'Siemka {name}!')
   ...:

In [2]: def say_goodbye(name):
   ...:     print(f'Trzymaj się, {name}!')
   ...:

In [3]: def greet_kryśka(greeting):
   ...:     return greeting('Kryśka')
   ...:

In [4]: greet_kryśka(say_hello)
Siemka Kryśka!

In [5]: greet_kryśka(say_goodbye)
Trzymaj się, Kryśka!
```

Autor: Michał Nowotka

Funkcje wewnętrzne

- W Pythonie można również tworzyć funkcje wewnątrz innych funkcji.
- Taka wewnętrzna funkcja działa na wyłączny użytek funkcji, we wnętrzu której jest zdefiniowana. Nie można się do niej dostać z zewnątrz w żaden sposób.
- Na przykładzie obok mamy funkcję, która w swoim ciele definiuje inną funkcję, po czym ją wywołuje.

```
In [1]: def outer_function(text):
   ...:     print('outer function')
   ...:     def inner_function(txt):
   ...:         print('inner function')
   ...:         return txt.upper()
   ...:     return inner_function(text)
   ...:

In [2]: outer_function('Ala ma kota')
outer function
inner function
Out[2]: 'ALA MA KOTA'
```

Zwracanie funkcji

- Jedna funkcja może zwracać inną funkcję jako wynik swojego działania.
- Taką funkcję nazywamy fabryką.

```
In [9]: def greeting(time):
   ...:
   ...:     def morning_greeting(name):
   ...:         return f'Good morning, {name}!'
   ...:
   ...:     def afternoon_greeting(name):
   ...:         return f'Good afternoon, {name}!'
   ...:
   ...:     def late_night_greeting(name):
   ...:         return f'Good night, {name}!'
   ...:
   ...:     if time is 'morning':
   ...:         return morning_greeting
   ...:     if time is 'afternoon':
   ...:         return afternoon_greeting
   ...:     return late_night_greeting
   ...:

In [10]: greeting_fun = greeting('morning')

In [11]: greeting_fun('John')
Out[11]: 'Good morning, John!'
```

Autor: Michał Nowotka

Prosty dekorator

- Nawet najprostszy dekorator używa wszystkich trzech poznanych technik:
  - przyjmuje funkcję jako swój argument
  - definiuje wewnętrzną funkcję
  - zwraca funkcję jako wynik swojego działania

```
In [1]: def my_decorator(func):
   ...:     def wrapper():
   ...:         return 'DECORATED --> ' + func() + ' <-- DECORATED'
   ...:     return wrapper
   ...:

In [2]: def hello_world():
   ...:     return 'Hello, world!'
   ...:

In [3]: print(hello_world())
Hello, world!

In [4]: print(my_decorator(hello_world)())
DECORATED --> Hello, world! <-- DECORATED
```

Prosty dekorator

- Dekorator jako jedyny argument, przyjmuje funkcję, którą ma udekorować.
- W środku definiuje funkcję tradycyjnie nazywaną **wrapper**, która wykonuje funkcję, którą dekorujemy ale oprócz tego wykonuje jakąś dodatkową czynność (w naszym przykładzie dopisuje dodatkowe napisy przed i po wyniku funkcji)
- Zwraca **wrapper** jako swój wynik.

```
In [1]: def my_decorator(func):
   ...:     def wrapper():
   ...:         return 'DECORATED --> ' + func() + ' <-- DECORATED'
   ...:     return wrapper
   ...:

In [2]: def hello_world():
   ...:     return 'Hello, world!'
   ...:

In [3]: print(hello_world())
Hello, world!

In [4]: print(my_decorator(hello_world)())
DECORATED --> Hello, world! <-- DECORATED
```

Autor: Michał Nowotka

Prosty dekorator

- Zauważmy, że po zastosowaniu dekoratora, zwraca on oryginalną funkcję owiniętą we wrapper. Zatem końcowy użytkownik nie dotyka teraz oryginalnej funkcji tylko wrappera.
- To znaczy, że wrapper musi przyjmować dokładnie takie argumenty co opakowana funkcja żeby mógł je do tej opakowanej funkcji przekazać.
- Problem polega na tym, że ciężko z góry przewidzieć jaką funkcję ktoś opakuje w dekorator, który piszemy i jakie ona będzie mieć argumenty.
- Dlatego też sygnatura wrappera powinna zawsze wyglądać tak:
  def wrapper(*args, **kwargs), dzięki czemu jesteśmy w stanie opakować dowolną funkcję.

```
In [1]: def my_decorator(func):
   ...:     def wrapper(*args, **kwargs):
   ...:         return 'DECORATED --> ' + func(*args, **kwargs) + ' <-- DECORATED'
   ...:     return wrapper
   ...:

In [2]: def hello_name(name):
   ...:     return f'Hello, {name}!'

In [3]: def hello_two_names(first_name, second_name):
   ...:     return f'Hello, {first_name} and {second_name}!'

In [4]: hello_name('Krzysiek')
Out[4]: 'Hello, Krzysiek!'

In [5]: my_decorator(hello_name)('Krzysiek')
Out[5]: 'DECORATED --> Hello, Krzysiek! <-- DECORATED'

In [6]: hello_two_names('Krzysiek', 'Olga')
Out[6]: 'Hello, Krzysiek and Olga!'

In [7]: my_decorator(hello_two_names)('Krzysiek', 'Olga')
Out[7]: 'DECORATED --> Hello, Krzysiek and Olga! <-- DECORATED'
```

Autor: Michał Nowotka

Prosty dekorator

- W dwóch poprzednich slajdach dekoratora używaliśmy w ten sposób: **dekorator(dekorowana_funkcja)(argumenty)**
- Takie użycie dekoratora nie jest zbyt czytelne, ale przecież Python daje nam zupełnie inny sposób.
- Kiedy mamy już zadeklarowany dekorator możemy go użyć dosłownie dekorując funkcję pisząc nad nią **@nazwa_dekoratora**.

```
In [1]: @my_decorator
   ...: def my_function(arg1, arg2):
   ...:     pass
   ...:
```

Prosty dekorator

Napisz dekorator o nazwie thrice, który powoduje wykonanie się opakowanej funkcji trzykrotnie.

Prosty dekorator

Napisz dekorator, który przed wykonaniem opakowywanej funkcji wypisze jej argumenty, a po wykonaniu wyprintuje komunikat „Wykonano z x argumentami”, gdzie x to liczba podanych argumentów do opakowywanej funkcji.

Prosty dekorator

Stwórz dekorator, który wykona funkcję opakowywaną lub nie, w zależności od wartości zmiennej globalnej SHOULD_BE_RUN (wartość True uruchamia funkcję, wartosc False oznacza, że powinien się pojawić tylko napis „Pomijam…”.

Prosty dekorator - debugowanie

- Każda funkcja ma kilka specjalnych atrybutów:
  - **__name__** - przechowuje nazwę funkcji
  - **__doc__** - przechowuje dokumentację (docstringi) funkcji drukowane wtedy, kiedy ktoś wywoła wbudowaną w Pythona funkcję **help**.
  - **__module__** - ścieżkę do modułu, w którym zdefiniowana jest funkcja.
- Niestety kiedy udekorujemy funkcję, wszystkie wymienione wyżej atrybuty będą pobrana nie z tej funkcji a z wrappera.
- To znacznie utrudnia debugowanie.
- Dlatego Python w swoim standardowym module o nazwie **functools** dostarcza dekoratora **@wraps**.
- Tym dekoratorem należy udekorować wrapper a wtedy zaciągnie on atrybuty z funkcji, którą opakowuje.

```
In [6]: hello_name.__name__
Out[6]: 'wrapper'

In [7]: hello_name.__module__
Out[7]: '__main__'

In [8]: print.__name__
Out[8]: 'print'

In [9]: print.__module__
Out[9]: 'builtins'
```

Autor: Michał Nowotka

Prosty dekorator - ostateczna wersja

```
In [1]: from functools import wraps

In [2]: def my_decorator(func):
   ...:     @wraps(func)
   ...:     def wrapper(*args, **kwargs):
   ...:         return 'DECORATED --> ' + func(*args, **kwargs) + ' <-- DECORATED'
   ...:     return wrapper
   ...:

In [3]: @my_decorator
   ...: def hello_name(name):
   ...:     return f'Hello, {name}!'
   ...:

In [4]: hello_name('Krzysiek')
Out[4]: 'DECORATED --> Hello, Krzysiek! <-- DECORATED'

In [5]: hello_name.__name__
Out[5]: 'hello_name'
```

Autor: Michał Nowotka

Prosty dekorator

Dodaj do poprzednich funkcji docstringi oraz sprawdź działanie atrybutów __name__, __doc__ oraz __module__. Opakuj funkcje z wykorzystaniem dekoratora wraps tak, by wartość __name__ była taka sama jak opakowywanej funkcji.

Prosty dekorator

Stwórz dekorator obliczający i wypisujący czas działania funkcji.

OPERACJE NA PLIKACH

Operacje na plikach

- Aby móc operować na pliku trzeba najpierw stworzyć do niego referencję. Służy do tego wbudowana funkcja **open**.
- **open** wymaga jednego parametru - jest nim ścieżka do pliku, który chcemy otworzyć. Ścieżka może być względna lub bezwzględna.
- Domyślnie plik otwarty jest w trybie tylko do odczytu i w formacie tekstowym. Jeśli chcemy operować na pliku binarnym albo zapisywać lub dopisywać do pliku, należy podać tryb otwarcia pliku jako drugi parametr.

```
In [1]: f = open('file.txt')

In [2]: f.write('ala ma kota')
---------------------------------------------------------------------------
UnsupportedOperation                                 Traceback (most recent call last)
<ipython-input-2-ff2e6ea74f55> in <module>
----> 1 f.write('ala ma kota')

UnsupportedOperation: not writable

In [3]: f.close()
```

Operacje na plikach - tryby odczytu, kodowanie

- Funkcja open posiada następujące tryby:
  - **r** - tylko do odczytu (domyślny)
  - **w** - do zapisu, jeśli plik nie istnieje to zostanie utworzony, jeśli istnieje jego stara zawartość zostanie usunięta.
  - **x** - do tworzenia, operacja nie powiedzie się, jeśli plik już istnieje.
  - **a** - dopisywanie, jeśli plik nie istnieje to zostanie utworzony, jeśli istnieje to nowa treść zostanie dopisana
  - **t** - tryb tekstowy (domyślny)
  - **b** - tryb binarny
  - **+** - oznacza, że po otwarciu kursor ustawi się na końcu pliku. Domyślnie ustawia się na początku.
- Tryby można stosować łącznie, np **a+b** oznacza otwarcie pliku binarnego do zapisu, przy czym plik, jeśli istnieje, nie zostanie skrócony a dopisywanie będzie się odbywać na końcu pliku.
- Ostatnim parameterem funkcji **open** jest **encoding**, czyli kodowanie plików. Domyślna wartość zależy od systemu operacyjnego, w systemie Windows jest to **cp1252** a pod Linuxem **utf-8**.

Autor: Michał Nowotka

Operacje na plikach - zarządzanie zasobami

- Po skończeniu pracy na pliku należy go zamknąć. Służy do tego metoda close na obiekcie typu File.
- Warto zauważyć, że poniższy kod nie jest bezpieczny:

```
In [1]: f = open('file.txt')

In [2]: # do stuff ...

In [3]: f.close()
```

- Nie ma żadnej gwarancji, że pomiędzy otwarciem a zamknięciem pliku nie wystąpi jakiś błąd, w konsekwencji którego nie wykona się ostatnia linijka i zostaniemy z otwartym plikiem.
- Lepszym podejściem byłoby otworzenie pliku w bloku try i zamknięcie w bloku finally. Mamy wtedy pewność, że niezależnie od tego co się stanie plik zostanie zamknięty (o tym wkrótce).

```
In [1]: try:
   ...:     f = open('file.txt')
   ...:     # do stuff...
   ...: finally:
   ...:     f.close()
   ...:
```

Autor: Michał Nowotka

Operacje na plikach - zarządzanie zasobami

- Oba sposoby podane na poprzednim slajdzie nie są specjalnie “pythonowe”.
- Najlepszym sposobem na otworzenie pliku jest użycie słowa kluczowego with.
- Słowo kluczowe with otwiera kontekst, który znajduje się we wciętym bloku poniżej niego.
- Kiedy opuścimy wcięty blok kontekstu, plik zostanie automatycznie zamknięty, także nie musimy wcale zamykać go sami.
- To eliminuje pomyłki spowodowane zapomnieniem o konieczności zamknięcia pliku kiedy nie używamy managera kontekstu.

```
In [1]: with open('file.txt', 'a+') as f:
   ...:     # do stuff...
   ...: # here the file is already closed
```

Operacje na plikach - zapis

- Aby zapisać jakąś treść do pliku należy go najpierw otworzyć w odpowiednim trybie (do zapisu albo do dopisania).
- Jeśli operacja otwarcia pliku powiedzie się (mamy odpowiednie uprawnienia, na dysku jest wystarczająco dużo miejsca) to możemy zapisać do niego treść używając metody write.
- Metoda write jako parametr przyjmuje ciąg znaków, który chcemy zapisać do pliku.
- Na przykładzie obok otworzyliśmy plik o nazwie file.txt w katalogu bieżącym (podaliśmy względną ścieżkę).
- Plik był otwarty w trybie do zapisu a więc jeśli nie istniał do został utworzony a jeśli istniał to jego stara zawartość została usunięta.
- We wciętym bloku zapisaliśmy dwie linijki treści, po opuszczeniu bloku plik został automatycznie zamknięty.

```
In [1]: with open('file.txt', 'w') as f:
   ...:     f.write('Ala ma kota\n')
   ...:     f.write('Kot ma Alę\n')
   ...:

In [2]: !cat file.txt
Ala ma kota
Kot ma Alę
```

Autor: Michał Nowotka

Operacje na plikach - odczyt

- Aby odczytać treść z pliku można użyć metody **read**.
- Jeśli metoda **read** zostanie wywołana bez parametrów, wczyta na raz całą zawartość pliku.
- Jeśli plik jest bardzo duży to może być potencjalnie groźna operacja dlatego opcjonalnie można podać liczbę, która oznacza maksymalną ilość bajtów, które zostaną wczytane.
- Mówmy maksymalną, ponieważ zawartość pliku może być krótsza niż ilość bajtów, które zażądaliśmy przeczytać.
- Kolejne wywołania funkcji **read** przesuwają kursor odczytu do przodu, także zaczynamy czytać od momentu, w którym skończyliśmy.
- Kiedy dotrzemy do końca pliku, metoda **read** będzie zwracać pusty string.
- Aby przesunąć kursor w zadane miejsce od początku pliku należy użyć metody **seek**.
- Aby poznać obecną pozycję kursora w pliku należy użyć metody **tell**.

```
In [1]: with open('file.txt', 'r') as f:
   ...:     content = f.read(4)
   ...:     print(f'Content I read is: {content}')
   ...:     print(f"This is where I'm in the file: {f.tell()}")
   ...:     content = f.read()
   ...:     print(f'Some more content I read: {content}')
   ...:     print(f"And now I'm here: {f.tell()}")
   ...:     content = f.read()
   ...:     print(f'Even more content I read: {content}')
   ...:     print(f"And where I am now: {f.tell()}")
   ...:     f.seek(0)
   ...:     print(f"I should be back now: {f.tell()}")
   ...:
Content I read is: Ala
This is where I'm in the file: 4
Some more content I read: ma kota
Kot ma Alę

And now I'm here: 24
Even more content I read:
And where I am now: 24
I should be back now: 0
```

Autor: Michał Nowotka

Operacje na plikach - odczyt

- W przypadku plików tekstowych bardzo często chcemy czytać plik linijka po linijce a nie bajt po bajcie.
- Istnieją trzy główne sposoby czytania pliku linijka po linijce.
  - Podejście prehistoryczne: użycie metody **readline**
  - Podejście średniowieczne: użycie metody **readlines**. **readlines** zwraca listę, której każdy element jest pojedynczą linijką przeczytaną z pliku. Wciąż może być przydatna kiedy nie chcemy czytać linijka po linijce ale poruszać się po pliku w wybrany przez nas sposób. Należy jednak pamiętać, że **readlines** wczyta cały plik, niezależnie od jego wielkości.
  - Podejście standardowe: użycie leniwego iteratora zwracanego przez obiekt typy File. Sam deskryptor pliku zwraca iterator przechodzący linijka po linijce przy czym jest on lepszy od readlines ponieważ nie wczytuje na raz całego pliku. Ten sposób jest preferowany.

```
In [1]: f = open('file.txt')

In [2]: while(True):
   ...:     line = f.readline()
   ...:     if not line:
   ...:         break
   ...:     print(line)
   ...:
Ala ma kota

Kot ma Alę
```

```
In [1]: with open('file.txt') as f:
   ...:     lines = f.readlines()
   ...:     for line in lines:
   ...:         print(line)
   ...:
Ala ma kota

Kot ma Alę
```

```
In [1]: with open('file.txt') as f:
   ...:     for line in f:
   ...:         print(line)
   ...:
Ala ma kota

Kot ma Alę
```

Autor: Michał Nowotka

Na podstawie pliku zen_python.txt z repozytorium, wykonaj szereg operacji:

a) Napisz funkcję wczytującą plik i wyliczającą ilość linii oraz ilość znaków niebędących spacjami
b) Napisz funkcję, która podmieni wszystkie słowa „is” na słowa „was” i zapisze wynik do nowego pliku
c) Napisz funkcję, która na podstawie parametru będącego liczbą, usunie z pliku linię podaną jako argument (jeśli drugi parametr should_remove będzie ustawiony na True), w przeciwnych wypadku zamieni wszystkie litery z tej linii na wielkie.

Pliki CSV

- CSV to skrót od *comma separated values* (wartości rozdzielone przecinkiem).
- Są to pliki tekstowe o specjalnej strukturze przeznaczone do przechowywania danych tabularycznych.
- Wartości oddzielone są przecinkiem (lub innym separatorem np. tabulatorem, średnikiem).
- Jest to format przechowywania danych w plikach typu text/csv (to znaczy, że jest to jakiś znany i uznawany format).

```
1 "Index", "Year", "Age", "Name", "Movie"
2  1, 1928, 44, "Emil Jannings", "The Last Command, The Way of All Flesh"
3  2, 1929, 41, "Warner Baxter", "In Old Arizona"
4  3, 1930, 62, "George Arliss", "Disraeli"
5  4, 1931, 53, "Lionel Barrymore", "A Free Soul"
6  5, 1932, 47, "Wallace Beery", "The Champ"
```

Pliki CSV

UWAGA:

- Spacje i inne białe znaki (w szczególności te przyległe do separatorów) należą do pól.
- Pierwsza linia może stanowić nagłówek zawierający nazwy pól rekordów, jednak pierwszy wiersz pliku CSV wg standardu ma takie samo znaczenie jak pozostałe.
- Zazwyczaj pierwszy wiersz określa nazwy kolumn.

```
1 "Index", "Year", "Age", "Name", "Movie"
2  1, 1928, 44, "Emil Jannings", "The Last Command, The Way of All Flesh"
3  2, 1929, 41, "Warner Baxter", "In Old Arizona"
4  3, 1930, 62, "George Arliss", "Disraeli"
5  4, 1931, 53, "Lionel Barrymore", "A Free Soul"
6  5, 1932, 47, "Wallace Beery", "The Champ"
```

Pliki CSV

- Do obsługi plików csv służy moduł csv.
- Plik otwieramy tak jak w przypadku normalnych plików tekstowych (zalecane: menadżer kontekstu with).
- Funkcja do odczytu pliku nazywa się reader i znajduje się w module csv.
- Jako argumenty podajemy jej uprzednio otwarty plik oraz znak delimitera (przecinek, tabulator, średnik).

```
In [1]: import csv

In [2]: with open("plik.csv", "r") as csvfile:
   ...:     csvreader = csv.reader(csvfile, delimiter=",")
   ...:
```

Pliki CSV

- Po wykonaniu funkcji reader zwracającej iterator, możemy po nim przeiterować i uzyskać dostęp do każdego wiersza z pliku po kolei.

```
In [11]: with open("F:\pdf\plik.csv", "r") as csvfile:
    ...:     csvreader = csv.reader(csvfile, delimiter=",")
    ...:     for row in csvreader:
    ...:         print(row)
```

```
['Index', ' "Year"', ' "Age"', ' "Name"', ' "Movie"']
[' 1', ' 1928', ' 44', ' "Emil Jannings"', ' "The Last Command', ' The Way of All Flesh"']
[' 2', ' 1929', ' 41', ' "Warner Baxter"', ' "In Old Arizona"']
[' 3', ' 1930', ' 62', ' "George Arliss"', ' "Disraeli"']
[' 4', ' 1931', ' 53', ' "Lionel Barrymore"', ' "A Free Soul"']
[' 5', ' 1932', ' 47', ' "Wallace Beery"', ' "The Champ"']
```

- Każdy wiersz zostaje zapisany jako lista elementów (na podstawie delimitera (najczęściej przecinka) z pliku; UWAGA – liczby są zapisywane jako stringi!

Pliki CSV

- Oprócz funkcji reader z modułu csv, do odczytu danych z pliku csv możemy wykorzystać klasę DictReader.

```
with open("F:\pdf\plik.csv", "r") as csvfile:
    csvreader = csv.DictReader(csvfile)
    for row in csvreader:
        print(row)
```

Pliki CSV

- Obiekt typu DictReader odczytuje zawartość pliku csv z wykorzystaniem słownika, w którym kluczami są nazwy kolumn z pliku (ustalane na podstawie wartości pierwszego wiersza w pliku).

```
"Index", "Year", "Age", "Name", "Movie"
 1, 1928, 44, "Emil Jannings", "The Last Command, The Way of All Flesh"
 2, 1929, 41, "Warner Baxter", "In Old Arizona"
 3, 1930, 62, "George Arliss", "Disraeli"
 4, 1931, 53, "Lionel Barrymore", "A Free Soul"
 5, 1932, 47, "Wallace Beery", "The Champ"
```

- Oto przykład odczytania pierwszego wiersza z pliku z wykorzystaniem klasy DictReader.

```
OrderedDict([('Index', ' 1'), (' "Year"', ' 1928'), (' "Age"', ' 44'), (' "Name"', ' "Emil Jannings"'),
(' "Movie"', ' "The Last Command'), (None, [' The Way of All Flesh"'])])
```

- OrderedDict to słownik, który pamięta kolejność par klucz: wartość, które zostały do niego dodane (poza tym zachowuje się jak zwykły dict).

Odczytaj zawartość pliku students.csv z repozytorium i znajdź ucznia oraz uczennicę o najwyższym wzroście.

Pliki CSV

- Celem zapisania danych do formatu csv, korzystamy z funkcji writer z modułu csv.

```
with open("F:\pdf\plik.csv", mode="w") as csvfile:
    writer = csv.writer(csvfile, delimiter=",", quotechar='"')
    writer.writerow([6, 1950, 44, "An actor", "Any film"])
```

- Argumenty: csvfile – nasz otwarty plik, delimiter – znak oddzielający od siebie kolumny danych, quotechar – jakimi znakami będziemy oznaczać napisy (albo ‘napis’ albo ”napis”)
- Funkcja writerow obiektu zwróconego przez funkcję csv.writer zapisuje do pliku pojedynczy wiersz: każdy element to nowa kolumna w wierszu oddzielona od innych znakiem delimitera

Pliki CSV

- Podobnie jak w przypadku odczytu i przy zapisywaniu możemy skorzystać z słownikowego zarządzania plikami csv – zapis do pliku jest możliwy z wykorzystaniem klasy DictWriter.

```
with open('F:\pdf\plik.csv', mode='w', newline="") as csv_file:
    column_names = ['id', 'name']
    writer = csv.DictWriter(csv_file, fieldnames=column_names)

    writer.writeheader()
    writer.writerow({"id": 1, "name": "Anna"})
```

- Na początku tworzymy listę z nazwami kolumn, jakie będziemy chcieli umieścić w pliku (u nas lista nazywa się column_names)
- Przekazujemy tę listę jako parametr fieldnames, po czym metodą writeheader obiektu writer zapisujemy pierwszy wiersz z nazwami kolumn do pliku csv.

Pliki CSV

- writer.writerow zapisuje każdy kolejny wiersz do pliku.
- Metoda ta wymaga podania słownika z nazwami kluczy odpowiadającymi nazwom kolumn.

```
with open('F:\pdf\plik.csv', mode='w', newline="") as csv_file:
    column_names = ['id', 'name']
    writer = csv.DictWriter(csv_file, fieldnames=column_names)

    writer.writeheader()
    writer.writerow({"id": 1, "name": "Anna"})
```

- Parametr newline w funkcji open zapobiega dodatkowemu dodaniu nowej linii po każdym nowym wierszu w pliku csv.

Pobierz zawartość pliku snake_game.csv z repozytorium za pomocą biblioteki csv. Zaproponuj i stwórz strukturę przechowującą dane na temat przebiegu gry. Dla takich danych napisz:

a) funkcję sortującą numer gry w zależności od zdobytych punktów (rosnąco),
b) funkcję obliczającą średnią ilość zdobytych punktów,
c) funkcję dopisującą nowy wynik do pliku (powinna samodzielnie zwiększać numer gry na podstawie ostatniego numeru).