Python

Poziom średniozaawansowany

Autor: Michał Nowotka

Plan na dzisiaj

1. Dekoratory
2. Obsługa błędów w Pythonie
3. Praca z plikami tekstowymi (.txt) – część 1

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.

WYJĄTKI – OBSŁUGA BŁĘDÓW W PROGRAMIE

Ćwiczenie na rozgrzewkę: napisz obsługę błędu błędnego użycia funkcji podaj_imie. Funkcja powinna przyjmowac jeden parametr imie typu str. Jeżeli użytkownik poda inny typ, funkcja zamiast wyprintować podane imię powinna wypisać komunikat o błędzie i zwrócić -1.

Obsługa wyjątków

- Wyjątek jest następstwem pewnej specjalnej sytuacji podczas wykonania programu.
- Sytuacja, która doprowadziła do podniesienia wyjątku nie należy do głównego scenariusza wykonania programu.
- Wyobraźmy sobie, że piszemy funkcję która jako argument przyjmuje ścieżkę do pliku a zwraca liczbę słów w danym pliku.
- Spodziewamy się, że w optymistycznym oraz najczęstszym scenariuszu plik, do którego podana jest ścieżka, będzie istniał.
- Może jednak zajść sytuacja w której ktoś popełni błąd i poda ścieżkę, która nie istnieje. W takim razie nasza funkcja może podnieść wyjątek, informując wywołującego o wystąpieniu specjalnej sytuacji.

Autor: Michał Nowotka

Obsługa wyjątków

- Każdą specjalną sytuację należy jakoś obsłużyć.
- Każdy wyjątek, który nie zostanie obsłużony spowoduje zakończenie działania programu.

```
In [2]: print(10/0)
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-2-fe01563e1bc6> in <module>
----> 1 print(10/0)

ZeroDivisionError: division by zero
```

Obsługa wyjątków

- Jeśli wywołujemy funkcję, o której wiemy, że może podnieść wyjątek i chcemy obsłużyć tę sytuację, to należy ją wywołać w bloku **try**.
- Po bloku try musi wystąpić blok **except** i/lub blok **finally**.

```
In [1]: try:
   ...:     print(10/0)
   ...: except:
   ...:     print("Nie dziel przez zero!!!")
   ...:
Nie dziel przez zero!!!
```

Obsługa wyjątków

- Jeśli wywołujemy funkcję, o której wiemy, że może podnieść wyjątek i chcemy obsłużyć tę sytuację, to należy ją wywołać w bloku **try**.
- Po bloku try musi wystąpić blok **except** i/lub blok **finally**.

```
In [3]: try:
   ...:     print(10/0)
   ...: except:
   ...:     print("Nie dziel przez zero!!!")
   ...: finally:
   ...:     print("Blok finally. Zawsze wykonywany.")
   ...:
Nie dziel przez zero!!!
Blok finally. Zawsze wykonywany.
```

Obsługa wyjątków

- Jeśli wywołujemy funkcję, o której wiemy, że może podnieść wyjątek i chcemy obsłużyć tę sytuację, to należy ją wywołać w bloku **try**.
- Po bloku try musi wystąpić blok **except** i/lub blok **finally**.

```
In [4]: try:
   ...:     print(10/10)
   ...: except:
   ...:     print("Nie dziel przez zero!!!")
   ...: finally:
   ...:     print("Blok finally. Zawsze wykonywany.")
   ...:
1.0
Blok finally. Zawsze wykonywany.
```

Obsługa wyjątków

- Instrukcje w bloku finally wykonają się zawsze, niezależnie od tego czy wyjątek zostanie wyrzucony czy nie.
- Blok finally jest bardzo ważną konstrukcją kiedy używamy cennych zasobów, które musimy zwolnić nawet jeśli wystąpi wyjątkowa sytuacja.

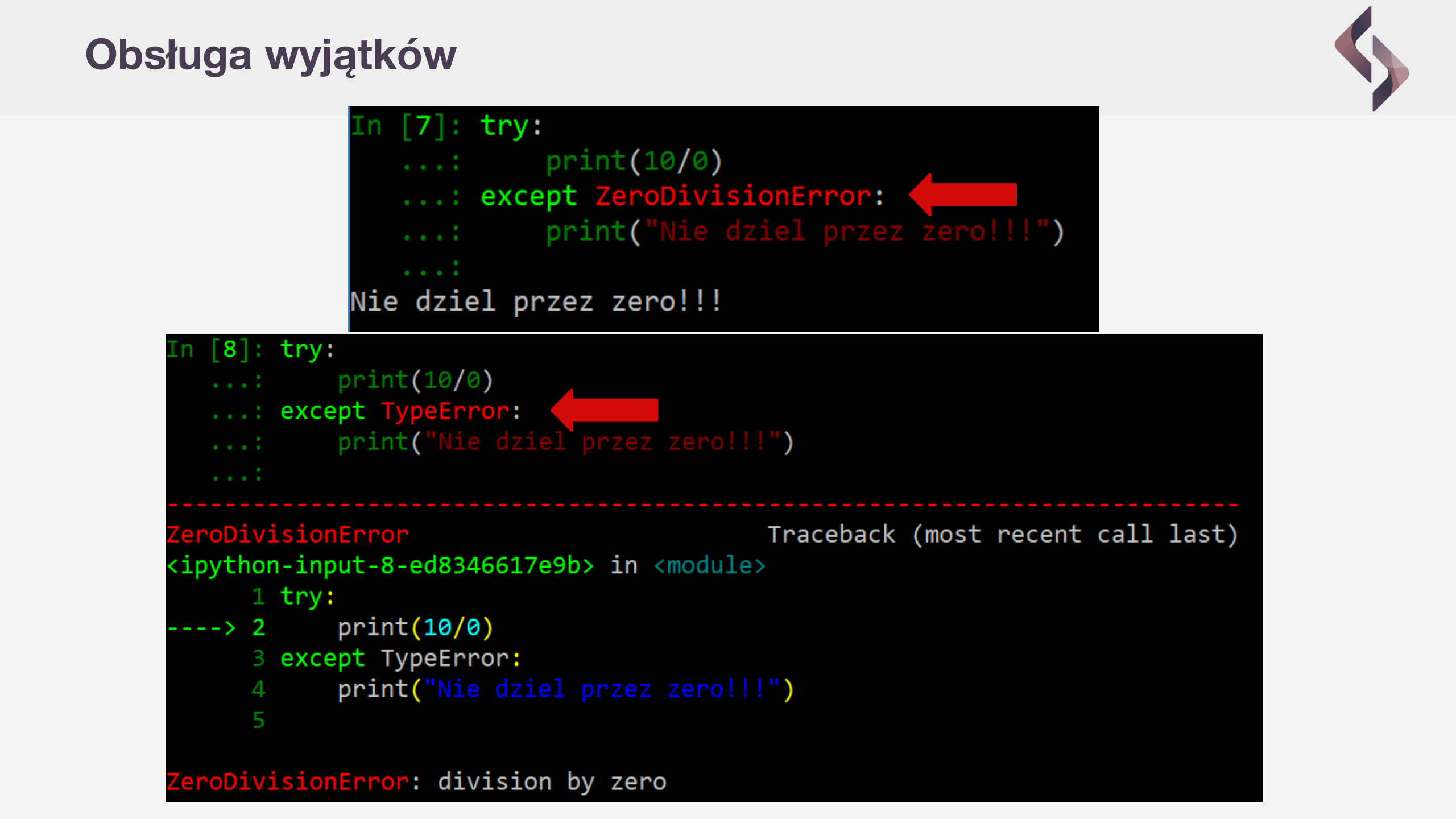
Autor: Michał Nowotka

Obsługa wyjątków

- W bloku **except** znajduje się kod obsługi przechwyconego wyjątku. Po słowie kluczowym **except** może zostać podana nazwa klasy wyjątku, którego obsługi dotyczy ten blok.
- Wszystkie podane niżej wyjątki (jak również pozostałe) dziedziczą po ogólnej klasie **`BaseException`**. Przy tworzeniu własnych wyjątków zaleca się dziedziczyć po typie **`Exception`**.

`TypeError, AssertionError, KeyError, ModuleNotFoundError, IndexError, NameError, ZeroDivisionError, SyntaxError`

Obsługa wyjątków



```
In [7]: try:
   ...:     print(10/0)
   ...: except ZeroDivisionError:
   ...:     print("Nie dziel przez zero!!!")
   ...:
Nie dziel przez zero!!!
```

```
In [8]: try:
   ...:     print(10/0)
   ...: except TypeError:
   ...:     print("Nie dziel przez zero!!!")
   ...:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-8-ed8346617e9b> in <module>
      1 try:
----> 2     print(10/0)
      3 except TypeError:
      4     print("Nie dziel przez zero!!!")
      5

ZeroDivisionError: division by zero
```

Obsługa wyjątków

- Można poinformować program, że spodziewamy się wystąpienia jednego z dwóch lubi więcej wyjątków (lepiej tego nie nadużywać, jeśli nie jesteśmy pewni)

```
In [9]: try:
   ...:     print(10/0)
   ...: except (ZeroDivisionError, TypeError):
   ...:     print("Nie dziel przez zero!!!")
   ...:
Nie dziel przez zero!!!
```

Obsługa wyjątków - przykłady

- TypeError – operacja/funkcja wykonywana ze złym typem obiektu (np. ‘2’ + 2)
- KeyError – klucz (np. w słowniku) nie może być znaleziony
- IndexError – próba dostania się do nieistniejącego elementu (np. w liście)
- ModuleNotFoundError – importowany (cały) moduł nie może być znaleziony
- AssertionError – asercja (wykorzystywana np. w testach) kończy się niepowodzeniem
- NameError – obiekt nie może być znaleziony (np. zmienna przed deklaracją)
- SyntaxError – niepoprawna składnia (np. wyrażenie: `a(=):1+2)`)
- ZeroDivisionError – dzielenie przez zero

Obsługa wyjątków

- Aby własnoręcznie podnieść wyjątek (wymusić go) w naszym kodzie, używamy słowa kluczowego **raise**. Podajemy po nim nazwę klasy wyjątku, który chcemy podnieść albo konkretną instancję tej klasy.

```
In [15]: try:
    ...:     raise SyntaxError
    ...: except SyntaxError:
    ...:     print("Nastąpił błąd syntaktyczny!")
    ...:
Nastąpił błąd syntaktyczny!
```

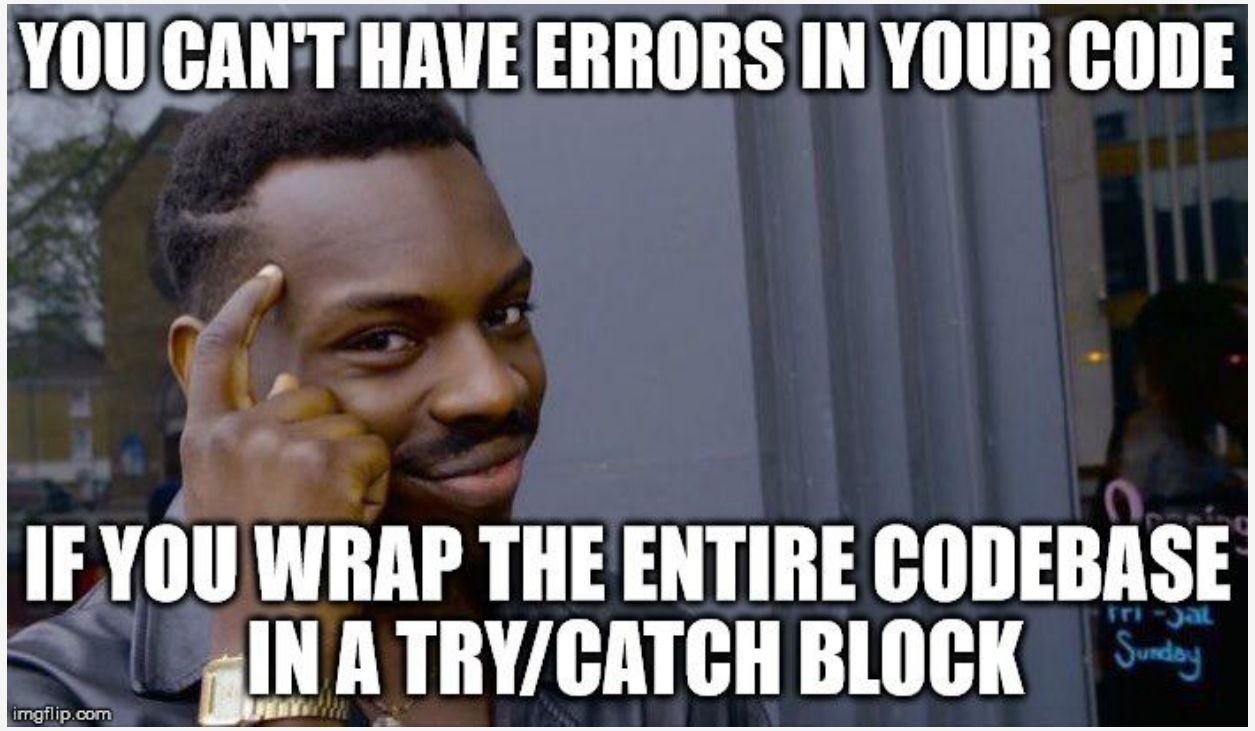
Obsługa wyjątków

- Bloków **except** może być wiele, np. po jednym bloku na każdy spodziewany typ wyjątku.
- Jeśli kilka różnych typów wyjątków będzie obsłużonych w tym samym bloku, ich typy można podać w postaci krotki.
- Po typie wyjątku można użyć słowa kluczowego **as** oraz nazwy zmiennej, do której zostanie przypisana konkretna instancja wyrzuconego wyjątku. Możemy jej użyć w kodzie obsługi np. aby zbadać treść komunikatu o błędzie.

```
In [21]: try:
    ...:     print(10/0)
    ...: except ZeroDivisionError as e:
    ...:     print(e, e.args, type(e))
    ...:
    ...:
division by zero ('division by zero',) <class 'ZeroDivisionError'>
```

Obsługa wyjątków

- Samo słowo **except** bez podania typu albo konstrukcja **except Exception** obsłużą każdy wyjątek, niezależnie od jego typu. Takie przechwytywanie wszystkich wyjątków, nie zważając na ich typ jest nazywane Pokemon exception handling i powszechnie uważane za antywzorzec programowania!



Obsługa wyjątków

- Tworzenie własnych wyjątków polega na dziedziczeniu po klasie **`Exception`** .
- Takie wyjątki możemy podnosić jak każde inne w naszym kodzie.

```
In [3]: class MyNewException(Exception):
   ...:     pass
   ...:

In [4]: raise MyNewException("Mayday!")
---------------------------------------------------------------------------
MyNewException                            Traceback (most recent call last)
<ipython-input-4-a19b402f2158> in <module>
----> 1 raise MyNewException("Mayday!")

MyNewException: Mayday!
```

Spróbujmy wymusić zwrócenie przez program kilku wyjątków…

Napisz funkcję przyjmującą jeden argument. Sprawdź jego typ we wnętrzu funkcji, w przypadku kiedy będzie to:

- typ str – podnieś własny wyjątek StrArgTypeError,
- typ int/float – podnieś własny wyjątek NumberArgTypeError,
- typ bool – podnieś własny wyjątek BoolArgTypeError.

Jeżeli argument będzie miał długość większą niż 1 (np. jako lista, tuple'a, set itd) podnieś wyjątek LenGreaterThanOneError.

Funkcja powinna wyprintować 'None jest okej', jeżeli argument będzie miał wartość None. Wszystkie wyjątki należy obsłużyć wewnątrz bloku if __name__ == '__main__' przy wywoływaniu 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 to 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 python_zen.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.