ALTKOM AKADEMIA

ZAAWANSOWANE TECHNIKI PROGRAMOWANIA W JĘZYKU PYTHON

poziom II

Inżynieria Oprogramowania
Altkom Akademia

Plan szkolenia

1 PROGRAMOWANIE FUNKCYJNE

2 PROGRAMOWANIE OOP

3 POMOCNE NARZĘDZIA

4 KOLEKCJE

5 WYRAŻENIA REGULARNE

6 PRZETWARZANIE DANYCH

7 BAZY DANYCH

8 WĄTKI I PROCESY

9 ASYNCHRONICZNY PYTHON

10 WSTĘP DO TESTÓW

Plan modułu

1 PROGRAMOWANIE FUNKCYJNE

- funkcje ze zmienną liczbą parametrów
- rozpakowywanie argumentów i kolekcji
- funkcja jako parametr
- funkcje lambda
- wyrażenia listowe, słownikowe, etc.
- generatory i iteratory
- moduł *itertools*
- wzorzec dekoratora
- moduł *functools*

Definiowanie funkcji

- Własne funkcje definiuje się z użyciem słowa kluczowego **`def`**

Składnia definicji funkcji

```
def nazwa_funkcji(parametry):
    instrukcje
```

nazwa_funkcji	identyfikator funkcji jest zmienną, która zostaje powiązana z obiektem funkcji w momencie wykonania deklaracji **`def`**
parametry	opcjonalna lista parametrów funkcji
instrukcje	ciało funkcji

Parametry funkcji

- Funkcja może posiadać dowolną (w tym zerową) liczbę **parametrów**, które podczas wywołania funkcji są inicjowane podanymi wartościami, tzw. **argumentami**

typy parametrów	przeznaczenie
pozycyjne (*positional parameters*)	podczas wywołania funkcji muszą być obowiązkowo związane z argumentami są podawane w pierwszej kolejności
nazwane (*keyword parameters*)	parametry, dla których podano wartości domyślne mają postać: *nazwa=wartość* podczas wywołania nie muszą być związane z argumentami – zostaną użyte wartości domyślne

Funkcje ze zmienną liczbą parametrów

- Można tworzyć funkcje ze zmienną (nieokreśloną) liczbą parametrów
- Na liście parametrów mogą wystąpić deklaracje:

**args*	reprezentuje krotkę zawierającą nienazwane argumenty, które nie zostały związane z zadeklarowanymi jawnie parametrami
***kwds*	reprezentuje słownik zawierający nazwane argumenty, które nie zostały związane z zadeklarowanymi jawnie parametrami

Parametry *keywords-only*

- Python 3 umożliwia zdefiniowanie parametrów funkcji, które, jeśli funkcja zostanie wywołana, **muszą zostać powiązane z nazwanymi argumentami**
- Takie parametry powinny wystąpić pomiędzy opcjonalnymi deklaracjami parametrów **args* i ***kwds*
- Są one określane terminem *keywords-only parameters*

- Jeśli w sygnaturze funkcji nie występuje parametr **args*, to do odseparowania parametrów obowiązkowych od parametrów *keywords-only* należy użyć parametru *
- Parametr * nie jest wiązany z żadnym z argumentów

Parametry *positional-only*

- Podczas wywołania funkcji do parametrów pozycyjnych można także odwołać się poprzez nazwę → wtedy kolejność podania argumentów nie jest istotna

- Można także zadeklarować parametry pozycyjne, które **muszą być dostarczone obowiązkowo z pominięciem nazwy**
- Takie parametry noszą nazwę *positional-only parameters*
- Są dostępne od Pythona 3.8

- Do ich odseparowania od parametrów pozycyjnych stosuje się parametr /
- Nie jest on wiązany z żadnym z argumentów

Operatory rozpakowania

- Operator * można wykorzystać do rozpakowania obiektów iterowalnych

Użycie operatora *

```
def suma(skladnik1, skladnik2, skladnik3):
    return skladnik1 + skladnik2 + skladnik3

liczby = [1, 2, 3]

# zamiast:
wynik = suma(liczby[0], liczby[1], liczby[2])  # 6

# można prościej:
wynik = suma(*liczby)                          # 6
```

Operatory rozpakowania

- Sekwencję można przypisać do krotki zmiennych i w ten sposób dokonać jej rozpakowania
- Takie zmienne nazywa się **celami** (*targets*) operacji rozpakowania
- Jeden z celów wypakowania można poprzedzić *, czyli **operatorem wypakowania sekwencji** (*sequence unpacking operator*)
- Do wyrażenia postaci **target* (*starred expression*) zostanie przypisana lista elementów sekwencji, które nie stały się celami innych zmiennych

Operatory rozpakowania

- W miejsce argumentów funkcji można także przekazać rozpakowany słownik
- Operatorem rozpakowania słownika jest **
- Nazwy kluczy słownika muszą zgadzać się z parametrami funkcji

Użycie operatora **

```
def suma(skladnik1, skladnik2, skladnik3):
    return skladnik1 + skladnik2 + skladnik3

slownik = {'skladnik1': 1, 'skladnik2': 2, 'skladnik3': 3}

wynik = suma(**slownik)  # 6
```

Operatory rozpakowania

- Operatory rozpakowania można także wykorzystać do złączenia sekwencji i słowników:

Złączenie sekwencji i słowników

```
sekwencja1 = 1, 2
sekwencja2 = [3, 4]
sekwencja = [*sekwencja1, *sekwencja2]  # [1, 2, 3, 4]

slownik1 = {'a': 1, 'b': 2}
slownik2 = {'b': 3, 'c': 4}
slownik = {**slownik1, **slownik2}  # {'a': 1, 'b': 3, 'c': 4}
```

Funkcje jako parametry

- Funkcje są **obiektami pierwszej kategorii** (*first-class object, first-class citizen*)
- Oznacza to, że:
  - funkcję można przekazać jako argument do innej funkcji
  - funkcja może w wyniku wywołania zwrócić funkcję
  - funkcję można związać ze zmienną
  - funkcja może być elementem kontenera lub pełnić rolę klucza w słowniku
  - funkcja może być atrybutem obiektu (metodą)

Funkcja lambda

- Proste funkcje można definiować za pomocą **wyrażeń lambda**
- Taka definicja tworzy nowy obiekt funkcji, który można później wywołać – jest to **anonimowy odpowiednik funkcji**, której ciało składa się tylko z jednego wyrażenia `return`
- Funkcja lambda ma postać:

Składnia funkcji lambda

```
lambda parametry: wyrażenie
```

parametry	opcjonalne parametry separowane przecinkami
wyrażenie	wyrażenie niezawierające: – pętli (wyrażenie warunkowe jest dopuszczalne) – instrukcji `return` – instrukcji `yield`

Funkcje lambda

- Wyrażenie w funkcji lambda powinno zwracać wartość
- Jego wyliczenie jest odwlekane do momentu wywołania funkcji

Przykład

```
lista = lambda a, b=0, *c, **d: [a, b, c, d]

print(lista(1, 2, 3, 4, x=5, y=6))

-> [1, 2, (3, 4), {'x': 5, 'y': 6}]
```

Wyrażenia listowe

- **Wyrażenia listowe** (*list comprehensions*) pozwalają w zwięzły sposób odwzorować pewną sekwencję na listę, wykonując na każdym jej elemencie pewną funkcję
- W wyrażeniach listowych wykorzystuje się pętlę `for` do generowania elementów nowej listy
- Wyrażenia listowe mają następującą składnię:

Składnia wyrażenia listowego

```
lista = [wyrażenie for element in obiekt_iterowalny]

# lub:
lista = [wyrażenie for element in obiekt_iterowalny if warunek]
```

Zagnieżdżone wyrażenia listowe

Wyrażenia listowe można zagnieżdżać...

```
lista = [x + str(y) for x in 'abc' for y in range(3)]
print(lista)

-> ['a0', 'a1', 'a2', 'b0', 'b1', 'b2', 'c0', 'c1', 'c2']
```

Równoważnie...

```
lista = []
for x in 'abc':
    for y in range(3):
        lista.append(x + str(y))
print(lista)

-> ['a0', 'a1', 'a2', 'b0', 'b1', 'b2', 'c0', 'c1', 'c2']
```

Wyrażenia zbiorowe

- W podobny sposób do wyrażeń listowych można tworzyć zbiory
- Takie konstrukcje to **wyrażenia zbiorowe** (*set comprehensions*)

Składnia wyrażenia zbiorowego

```
zbior = {wyrażenie for element in obiekt_iterowalny}

# lub:
zbior = {wyrażenie for element in obiekt_iterowalny if warunek}
```

Przykład

```
zbior = {litera.upper() for litera in 'kakao'}
print(zbior)

-> {'K', 'A', 'O'}
```

Wyrażenia słownikowe

- Analogicznie do list i zbiorów, słowniki można tworzyć za pomocą **wyrażeń słownikowych** (*dictionary comprehensions*)

Składnia wyrażenia słownikowego

```
slownik = {wyrazenie1:wyrazenie2
                      for element in obiekt_iterowalny}

# lub:
slownik = {wyrazenie1:wyrazenie2
                    for element in obiekt_iterowalny if warunek}
```

  - *wyrazenie1* tworzy dla danego elementu klucz słownika
  - *wyrazenie2* tworzy dla danego elementu wartość słownika

Obiekty iterowalne

- Wiele operacji i funkcji w Pythonie oczekuje jako argumentów obiektów iterowalnych, np.:
  - pętla *for-each*
  - funkcje wbudowane *filter*, *map*
  - i wiele innych...
- **Obiekty iterowalne** (*iterables*) zachowują się jak kontenery obiektów, umożliwiając sekwencyjny dostęp do swoich elementów lub zwracając kolejne obiekty "na życzenie"

Protokół iteracji

- Obiekt iterowalny musi zaimplementować metodę *__iter__()*, która zwraca iterator
- **Iterator** umożliwia dostęp do kolejnych elementów obiektu iterowalnego poprzez sukcesywne wywoływanie metody *__next__()*
- Wyczerpanie się dostępnych elementów jest sygnalizowane przez iterator wyrzuceniem wyjątku *StopIteration*
- Często protokół iteracji implementuje się tak, że obiekt iterowalny pełni jednocześnie rolę iteratora
- **Protokół iteracji** wymaga implementacji obu metod: *__iter__()* oraz *__next__()*

Implementacja iteratora

Przykład

```
class LetterIterator:
    def __init__(self, n=26):
        if n > 26:
            n = 26
        self.__current = ord('A') - 1
        self.__last = ord('A') + n

    def __iter__(self):
        return self

    def __next__(self):
        self.__current += 1
        if self.__current < self.__last:
            return chr(self.__current)
        else:
            raise StopIteration
```

Użycie iteratora

Przykład

```
iterator = LetterIterator(3)
print(next(iterator))
-> A
print(next(iterator))
-> B
print(next(iterator))
-> C
print(next(iterator))
-> Traceback (most recent call last):
->   File "<stdin>", line 1, in <module>
->   File "<stdin>", line 14, in __next__
-> StopIteration

iterator = LetterIterator(10)
print(list(iterator))
-> ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']
```

Funkcja generatora

- **Funkcja generatora** (*generator function*) – funkcja zawierająca przynajmniej jedną instrukcję *yield*
- Ma ona postać:

Składnia instrukcji *yield*

```
yield wyrażenie
```

Funkcja generatora

- Wywołanie funkcji generatora nie powoduje wykonania jej ciała, lecz zwraca **obiekt generatora** (*generator object*) – specjalny obiekt iteratora

Przykład

```
# funkcja generatora
def countdown(n):
    while n > 0:
        yield n
        n -= 1

# utworzenie obiektu generatora
generator = countdown(5)
```

Funkcja generatora

- Generator jest praktycznym, wygodnym i bardzo prostym narzędziem do stworzenia własnego, specjalizowanego iteratora
- Interpreter przejmuje obowiązki obsługi iteratora
- Generator produkujący obiekt iteratora dostarcza implementacji metod *__iter__()*, *__next__()* (funkcji *next()*), obsługi wyjątku *StopIteration*, itd. . .

Funkcja generatora

- Wywołanie metody *__next__* na generatorze powoduje wykonanie ciała funkcji od bieżącego miejsca do pierwszej napotkanej instrukcji **`yield`**
- Wtedy następuje:
  - przerwanie dalszego przetwarzania
  - zapamiętanie bieżącego stanu
  - zwrócenie wartości wyrażenia **`yield`** (lub wartości **`None`** – w razie braku wyrażenia)

Funkcja generatora

- Kolejne wywołanie funkcji *__next__*:
  - powoduje wznowienie przetwarzania od miejsca zatrzymania
  - przetwarzanie trwa do momentu napotkania kolejnej instrukcji `yield`
- Zakończenie wykonywania funkcji generatora lub wykonanie instrukcji `return` powoduje wyrzucenie wyjątku *StopIteration*
- W Pythonie 3 instrukcja `return` może mieć opcjonalne wyrażenie, które jest przekazywane jako argument do wyjątku

Użycie generatora

Przykład

```
>>> generator = countdown(3)
>>> print(next(generator))
3
>>> print(next(generator))
2
>>> print(next(generator))
1
>>> print(next(generator))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Generatory

- Generatory są bardziej uniwersalne od funkcji zwracających listy
- Generator może zwrócić **nieograniczony iterator** (*unbounded iterator*), za pomocą którego można uzyskać nieskończony strumień wartości
- Generator działa **w sposób leniwy** (*lazy evaluation*) – wylicza kolejne elementy dopiero wtedy, gdy jest to konieczne, podczas, gdy analogiczna funkcja wykonuje wszystkie wyliczenia zawczasu, co może wymagać dużej ilości pamięci do przechowania listy

Tworzenie generatorów

- Python 3 upraszcza tworzenie generatorów dzięki instrukcji:

Instrukcja *yield from*

```
yield from obiekt_iterowalny
```

- Jest ona równoważna konstrukcji:

Odpowiednik instrukcji *yield from*

```
for element in obiekt_iterowalny:
    yield element
```

Funkcja generatora – przykład

Definicja generatora

```
def updown(N):
    yield from range(1, N)
    yield from range(N, 0, -1)
```

Użycie generatora

```
g = updown(3)   # utworzenie obiektu generatora

print(list(g))
-> [1, 2, 3, 2, 1]
```

Wyrażenia generatorowe

- Proste generatory można utworzyć za pomocą **wyrażeń generatorowych** (*generator expressions, genexps*)
- Wyrażenia generatorowe mają składnię zbliżoną do wyrażeń listowych:

Składnia wyrażeń generatorowych

```
(wyrażenie for element in obiekt_iterowalny)
(wyrażenie for element in obiekt_iterowalny if warunek)
```

- Wyrażenie generatorowe tworzy iterator, zwracający w danej chwili **tylko jeden element**
- Wyrażenie listowe tworzy listę zawierającą **wszystkie elementy**
  $\rightarrow$ większe obciążenie pamięci

Moduł *itertools*

- Moduł *itertools* umożliwia **wydajne tworzenie iteratorów** (wzorowany na językach takich jak *APL*, *Haskell*, czy *SML*)
- Tworzy tzw. algebrę iteratorów
- Pełna dokumentacja modułu jest dostępna pod tym linkiem

Moduł *itertools*

Tworzenie iteratorów **nieskończonych**:

- *count(start=0, step=1)*
  - iterator zwraca równomiernie rozmieszczone wartości, zaczynając od wartości *start*
- *cycle(iterable)*
  - iterator zwraca elementy obiektu iterowalnego kopiując je, a następnie zwraca elementy z kopii (powtarza to nieskończoną ilość razy)
- *repeat(object, times=None)*
  - iterator zwracający nieskończoną ilość razy podany obiekt, chyba, że zostanie podany parametr *times*

Moduł *itertools*

Tworzenie iteratorów **kończących działanie wraz z krótszą sekwencją wejściową**:

- *accumulate(iterable[, func, *, initial=None])*
  - iterator zwraca zakumulowane sumy elementów lub zakumulowane sumy wyników podanej funkcji dwuargumentowej
- *chain(*iterables)*
  - iterator zwraca elementy z pierwszej sekwencji, a następnie z kolejnych, aż do wyczerpania
- *from_iterable(iterable)*
  - metoda klasy, alternatywny konstruktor w stosunku do *chain*
- *compress(data, selectors)*
  - iterator zwraca te elementy sekwencji *data*, które posiadają odpowiedniki w sekwencji *selectors*, o kontekście logicznym *True*

Moduł *itertools*

- *dropwhile(predicate, iterable)*
  - iterator filtrujący sekwencję *iterable*, tak że pomija te elementy, dla których predykat jest prawdziwy i zwraca pozostałe
- *filterfalse(predicate, iterable)*
  - iterator filtrujący sekwencję *iterable*, tak że zwraca te elementy, dla których predykat jest fałszywy
  - gdy predykatem jest `None`, zwracana są te elementy których kontekst logiczny jest fałszywy
- *groupby(iterable, key=None)*
  - iterator zwraca kolejne klucze i powiązane z nimi grupy elementów
- *islice(iterable, start, stop[, step])*
  - iterator zwraca kolejne elementy wycinka sekwencji (parametry nie mogą być ujemne)

Moduł *itertools*

- *starmap(function, iterable)*
  - iterator zwraca wyniki zastosowania funkcji do argumentów pobranych z sekwencji
  - funkcja używana zamiast *map*, gdy argumenty są zgrupowane w krotki
- *takewhile(predicate, iterable)*
  - iterator zwraca elementy sekwencji tak długo, gdy predykat jest prawdziwy
- *tee(iterable, n=2)*
  - zwraca *n* niezależnych iteratorów dla podanej sekwencji

    iteratory nie są bezpieczne wielowątkowo
- *zip_longest(*iterables, fillvalue=None)*
  - iterator agregujący wartości z podanych sekwencji
  - jeśli długości sekwencji są różne, to krótsza sekwencja jest uzupełniana podaną wartością *fillvalue*

Moduł *itertools*

- *product(*iterables, repeat=1)*
  - iloczyn kartezjański podanych sekwencji
- *permutations(iterable, r=None)*
  - zwraca kolejne permutacje (o długości *r*) elementów sekwencji
- *combinations(iterable, r)*
  - zwraca podsekwencje o długości *r*
- *combinations_with_replacement(iterable, r)*
  - zwraca podsekwencje o długości *r*, przy czym te same elementy mogą występować więcej niż raz

Wzorzec dekoratora

- Ponieważ funkcje są obiektami pierwszej klasy, więc można utworzyć funkcję, która:
  - jako argument przyjmie inną funkcję
  - opakuje ją w inną funkcję (*wrapper function*)
  - zwróci nową funkcję
- Ta nowa funkcja może zastąpić oryginalną

Wzorzec dekoratora

- Funkcję, która zwraca funkcję opakowaną nazywamy **dekoratorem** (*decorator*)

Funkcja bez dekoratora

```
def powitanie(imie):
    return f'Cześć {imie}'

komunikat = powitanie('Tomasz')
print(komunikat)

-> Cześć Tomasz
```

Wzorzec dekoratora

Funkcja “udekorowana”

```
# dekorator
def wersaliki(funkcja):
    print(f'dekorowanie funkcji {funkcja.__name__}')
    def wrapper(*args, **kwargs):
        print(f'wywołanie funkcji {funkcja.__name__}')
        wynik = funkcja(*args, **kwargs)
        return wynik.upper()
    return wrapper

powitanie = wersaliki(powitanie)  # dekorowanie funkcji powitanie
komunikat = powitanie('Tomasz')   # wywołanie funkcji powitanie
print(komunikat)

-> CZEŚĆ TOMASZ
```

Wzorzec dekoratora

- Od Pythona 2.4 do deklarowania dekoratorów można użyć alternatywnej składni

Zamiast deklaracji...

```
def funkcja():
    ...

funkcja = dekorator(funkcja)
```

można użyć składni...

```
@dekorator
def funkcja():
    ...
```

Dekoratory zagnieżdżone

- Dekoratory można zastosować do dowolnej funkcji, w tym także do metod
- Dekoratory można zagnieżdżać:

Zagnieżdżone dekoratory

```
@a
@b
@c
def funkcja(): ...

# jest równoważne z:
def funkcja(): ...
funkcja = a(b(c(funkcja)))
```

- Dekorator trzeba umieścić w linii poprzedzającej dekorowaną funkcję

Metody fabryki

- Dekoratory mogą przyjmować listę argumentów
- Muszą zwracać funkcję
- W takim przypadku tworzą one **metodę fabryki** (*factory method*)

Moduł *functools*

- Moduł *functools* jest przeznaczony dla funkcji wyższego rzędu, tzn. funkcji, które działają na innych funkcjach lub je zwracają
- Moduł definiuje dwie klasy:
  - *partial*
  - *partialmethod*
- Pełna dokumentacja modułu jest dostępna pod tym linkiem

Moduł *functools*

- Klasa *partial* umożliwia tworzenie obiektów wykonywalnych (*callable objects*)
- Posiada atrybuty tylko-do-odczytu:

partial.func	obiekt wykonywalny lub funkcja wywołania *partial* będą przekierowywane do tej funkcji z nową listą parametrów
partial.args	argumenty pozycyjne, które zostaną dodane do listy parametrów pozycyjnych podanych do *partial*
partial.keywords	argumenty nazwane, które są podawane podczas wywołania *partial*

- Atrybuty *__name__* oraz *__doc__* nie są tworzone automatycznie
- Obiekty *partial* zachowują się jak metody statyczne

Moduł *functools*

Składnia *partial*

```
partial(func, /, *args, **keywords)
```

Przykład

```
from functools import partial

def potega(a, b):
    return a ** b

# funkcje partial
szescian =  partial(potega, b = 3)
potega3 = partial(potega, 3)

szescian(2)   # 2 ** 3 = 8
potega3(2)    # 3 ** 2 = 9
```

Moduł *functools*

- Klasa *partialmethod* działa podobnie do *partial*, ale jest przeznaczona do pracy z definicjami metod, a nie obiektów wykonywalnych

Składnia

```
partialmethod(func, /, *args, **keywords)
```

Moduł *functools* – wybrane funkcje

- *@lru_cache(user_function)*
- *@lru_cache(maxsize=128, typed=False)*
  - dekorator opakowujący funkcję w obiekt *callable* zapamiętujący *maxsize* ostatnich wywołań funkcji
  - ponieważ do cache'owania wyników wykorzystywany jest słownik, to argumenty pozycyjne i nazwane muszą być hash'owalne
  - przy *maxsize=None*, rozmiar cache'a jest nieograniczony
  - przy *typed=True*, argumenty funkcji różnych typów są cache'owane niezależnie
  - do określenia efektywności cache'a można użyć funkcję *cache_info()*

Moduł *functools* – wybrane funkcje

- *@cache(user_function)*
  - nieograniczony cache funkcji
  - daje te same wyniki co *lru_cache(maxsize=None)*
  - od Python'a 3.9
- *@cached_property(func)*
  - przekształca metodę we właściwość (tak jak *property*), której wartość jest wyliczana jednorazowo, a następnie cache'owana
  - funkcja użyteczna, gdy wyliczenie wartości jest czasochłonne

Moduł *functools* – wybrane funkcje

- *cmp_to_key(func)*
  - konwertuje funkcję komparatora na funkcję klucza (funkcję dyskryminatora używaną do sortowania lub porządkowania)
  - funkcję klucza wykorzystują np.: *sorted()*, *min()*, *max()*, *heapq.nlargest()*, *heapq.nsmallest()*, *itertools.groupby()*
- *@total_ordering*
  - dekorator dostarczający implementacji brakujących relacyjnych metod specjalnych
  - należy zaimplementować metodę *__eq__* oraz jedną metodę spośród: *__lt__*, *__le__*, *__gt__*, *__ge__*
- *reduce(function, iterable[, initializer])*
  - umożliwia redukcję sekwencji *iterable* do pojedynczej wartości z wykorzystaniem dwuargumentowej funkcji *function*

Moduł *functools* – wybrane funkcje

- *@singledispatch*
  - dekorator przekształcający funkcję w funkcję generyczną, która może mieć różne zachowanie w zależności od typu pierwszego argumentu
  - funkcja dekorowana dostarcza implementacji domyślnej
  - do dodania przeciążonej wersji implementaci funkcji służy atrybut *register()* funkcji generycznej
- *@singledispatchmethod(func)*
  - dekorator przekształcający metodę w metodę generyczną
  - działa podobnie do dekoratora *@singledispatch* (ale w stosunku do metod)
  - metoda może być opisana wieloma dekoratorami, ale ten musi być najbardziej zewnętrzny

Moduł *functools* – wybrane funkcje

Przykład przeciążania funkcji

```
from functools import singledispatch

# funkcja generyczna: funkcja(arg: T)
@singledispatch
def funkcja(arg): print('argument typu ??? =', arg)

# funkcja przeciążona: funkcja(arg: int)
@funkcja.register(int)
def _(arg): print('argument typu int =', arg)

# funkcja przeciążona: funkcja(arg: str)
@funkcja.register(str)
def _(arg): print('argument typu str =', arg)

funkcja(123)        # argument typu int = 123
funkcja('abc')      # argument typu str = abc
funkcja(['a', 1])   # argument typu ??? = ['a', 1]
```

Moduł *functools* – wybrane funkcje

- *update_wrapper(wrapper, wrapped, assigned=WRAPPER_ASSIGNMENTS, updated=WRAPPER_UPDATES)*
  - funkcja, która aktualizuje funkcję opakowującą (*wrapper*) tak, by wyglądała jak funkcja opakowywana (*wrapped*)
  - opcjonalne argumenty to krotki określające, które atrybuty oryginalnej funkcji są przypisywane bezpośrednio do pasujących atrybutów funkcji opakowującej, a które atrybuty funkcji opakowującej są aktualizowane przy użyciu odpowiednich atrybutów z oryginalnej funkcji
- *@wraps(wrapped, assigned=WRAPPER_ASSIGNMENTS, updated=WRAPPER_UPDATES)*
  - dekorator, który stosuje funkcję *update_wrapper* w stosunku do dekorowanej funkcji
  - równoważne wywołaniu: *partial(update_wrapper, wrapped=wrapped, assigned=assigned, updated=updated)*

Plan szkolenia

1 PROGRAMOWANIE FUNKCYJNE

2 PROGRAMOWANIE OOP

3 POMOCNE NARZĘDZIA

4 KOLEKCJE

5 WYRAŻENIA REGULARNE

6 PRZETWARZANIE DANYCH

7 BAZY DANYCH

8 WĄTKI I PROCESY

9 ASYNCHRONICZNY PYTHON

10 WSTĘP DO TESTÓW

Plan modułu

2 PROGRAMOWANIE OOP

- dokumentowanie kodu
- atrybuty klas
- dziedziczenie wielobazowe oraz MRO
- metoda *super*
- dostęp do atrybutów
- metody specjalne i przeciążanie operatorów
- dekoratory klasowe
- deskryptory i właściwości
- metaklasy
- moduł *abc* – klasy abstrakcyjne

Komentarze dokumentujące

- Do tworzenia treści dokumentacji wykorzystuje się komentarze dokumentujące (*documentation strings, docstrings*)
- Tekst ujmuje się w potrójne cudzysłowy """ lub apostrofy '''
- Komentarze mogą obejmować wiele linii
- Znaki białe są w nich traktowane dosłownie
- Jeśli instrukcją w pierwszej linii ciała funkcji/klasy/metody/modułu jest literał tekstowy, to jest on traktowany jak komentarz dokumentujący

Komentarze dokumentujące

- Zgodnie z konwencją:
  - **pierwsza linia** powinna być zwartym opisem przeznaczenia funkcji, zaczynającym się dużą literą i kończącym kropką
  - nie powinna zawierać nazwy funkcji
  - jeśli opis obejmuje wiele linii, to **druga linia** powinna być pusta
  - **kolejne linie** powinny być uformowane w akapity odseparowane pustymi liniami
  - powinny zawierać takie informacje jak: parametry, warunki wstępne, zwracana wartość, efekty uboczne
  - **na końcu** mogą się znaleźć: dalsze wyjaśnienia, odnośniki do bibliografii, przykłady użycia
- Bardziej szczegółowy format komentarzy dokumentujących definiuje narzędzie *sphinx*

Komentarze dokumentujące

- Komentarze dokumentujące stanowią informację na użytkownika końcowego
- Komentarze są dostępne w czasie wykonania aplikacji (chyba, że została ona uruchomiona z opcją *–OO*)
- Środowiska zintegrowane i programy narzędziowe mogą wykorzystywać komentarze, aby wspierać programistów w pracy

Moduł *pydoc*

- Do utworzenia dokumentacji na podstawie komentarzy dokumentujących można wykorzystać moduł *pydoc*
- Dokumentację można:
  - przedstawić jako strony pomocy na konsoli
  - zapisać do plików HTML
  - przedstawić na serwerze WWW
- Pełna dokumentacja modułu jest dostępna pod tym linkiem

Moduł *pydoc*

- Opcje programu *pydoc*:

–w	dokumentacja w formacie HTML zostanie zapisana w pliku
–k <key>	umożliwia podanie poszukiwanego klucza
–p <port>	startuje serwer HTTP na podanym porcie
–n <hostname>	startuje serwer HTTP na podanym adresie
–b	startuje serwer HTTP i uruchamia przeglądarkę ze spisem modułów

Klasy i instancje

- **Klasy** w programowaniu zorientowanym obiektowo są definicjami nowych typów danych – typów użytkownika (np. klasa *Samochod*)
- Na podstawie klasy można tworzyć **instancje**, które są konkretnymi "egzemplarzami" klasy (np. konkretny samochód z jakimś numerem rejestracyjnym)

Atrybuty klasy

- Zarówno klasy, jak i ich instancje mogą przechowywać dane
- Zmienne tworzone wewnątrz klasy to **atrybuty klasy**
- Ich wartości są współdzielone przez instancje klasy

Atrybuty klasy

```
class Samochod:
    liczba_kol = 4          # atrybut klasy

print(Samochod.liczba_kol)
-> 4

# Atrybyt można dodeklarować do wcześniej utworzonej klasy...
class Samochod: pass
Samochod.liczba_kol = 4     # atrybut klasy

print(Samochod.liczba_kol)
-> 4
```

Atrybuty instancyjne

- **Atrybuty instancyjne** to dane specyficzne dla danej instancji
- Nie są współdzielone pomiędzy instancjami
- Typowo są tworzone w metodzie *__init__*

Atrybuty instancji

```
class Samochod:
    def __init__(self, nr_rej):
        self.nr_rej = nr_rej  # atrybut instancyjny

s = Samochod('WF12345')       # utworzenie instancji
print(s.nr_rej)
-> WF12345

# Atrybyt można dodeklarować do wcześniej utworzonej instancji...
class Samochod: pass
s = Samochod()                # utworzenie instancji
s.nr_rej = 'WF12345'          # atrybut instancyjny
print(s.nr_rej)
-> WF12345
```

Metody instancyjne

- Zbiór wartości atrybutów instancji definiuje jej **stan**
- Wartościami atrybutów mogą być także obiekty funkcji
- Funkcje definiowane wewnątrz klasy są nazywane **metodami**
- Metody definiują **zachowanie** klasy

- Najczęściej wewnątrz klasy definiuje się **metody instancyjne**
- Wtedy pierwszy argument metody jest traktowany jak **referencja do bieżącej instancji**
- Zwyczajowo nosi on nazwę *self*

Metody instancyjne

- Metody instancyjne wołane są na rzecz instancji

Przykład - metody instancyjne

```
class Samochod:
    # metody instancyjne
    def __init__(self, nr_rej):
        self.nr_rej = nr_rej

    def jaki_nr_rej(self):
        return self.nr_rej

s = Samochod('WF12345')
print(s.jaki_nr_rej())
->  WF12345
```

- Metody instancyjne mają dostęp zarówno do atrybutów instancyjnych, jak i atrybutów klasy

Metody klasy

- **Metoda klasy** (*class method*) – w odróżnieniu od metody instancyjnej – nie przyjmuje jako pierwszego parametru obiektu bieżącej instancji, tylko **obiekt bieżącej klasy**
- Zgodnie z konwencją, parametrowi nadaje się nazwę *cls*
- Metodę klasy można wywołać na rzecz klasy lub dowolnej jej instancji (ale nie uzyska ona dostępu do atrybutów instancji)

- Nie ma konieczności definiowania metod klasy – można zawsze na zewnątrz klasy utworzyć funkcję, która jako pierwszy parametr przyjmie obiekt klasy
- Metody klasy są elegancką alternatywą dla takich funkcji, zwł. że można je przedefiniować w podklasach

Metody klasy

- Do utworzenia metody klasy wykorzystuje się wbudowany typ *classmethod*, którego wynik wywołania należy powiązać z atrybutem klasy
- Jedynym argumentem jest funkcja, którą Python wykona podczas wywołania metody klasy

Metoda klasy

```
class A:
    def f(cls):
        return 'metoda klasy z ' + cls.__name__

    f = classmethod(f)

a = A()
print(A.f(), a.f(), sep=', ')
-> metoda klasy z A, metoda klasy z A
```

Metody klasy

- Równoważnie, począwszy od Pythona 2.4, do zadeklarowania metody klasy można użyć dekoratora `@classmethod`:

Metoda klasy

```
class A:
    @classmethod
    def f(cls):
        return 'metoda klasy z ' + cls.__name__

a = A()
print(A.f(), a.f(), sep=', ')
-> metoda klasy z A, metoda klasy z A
```

Metody statyczne

- **Metoda statyczna** (*static method*) – może mieć dowolną sygnaturę
- Może w ogóle nie posiadać parametrów, a jeśli je posiada, to pierwszy parametr nie odgrywa żadnej specjalnej roli – nie ma więc bezpośredniego dostępu do atrybutów klasy i jej instancji
- Metoda statyczna zachowuje się jak **zwykła funkcja**, z tą różnicą, że jest zdefiniowana wewnątrz klasy

- Nie ma konieczności stosowania metod statycznych – można się bez nich obyć, wykorzystując funkcje zdefiniowane na zewnątrz klasy
- Metody statyczne stanowią elegancką alternatywę składniową, w sytuacji, gdy **cel metody jest ściśle powiązany z klasą**

Metody statyczne

- Do utworzenia metody statycznej wykorzystuje się wbudowany typ *staticmethod*, a wynik wywołania wiąże z atrybutem klasy
- Jedynym argumentem jest funkcja, którą Python wykona podczas wywołania metody statycznej

Metoda statyczna

```
class A:
    def f():
        return 'metoda statyczna'

    f = staticmethod(f)

a = A()
print(A.f(), a.f(), sep=', ')
-> metoda statyczna, metoda statyczna
```

Metody statyczne

- Alternatywnie, do utworzenia metody statycznej można użyć dekoratora `@staticmethod`:

Metoda statyczna

```
class A:
    @staticmethod
    def f():
        return 'metoda statyczna'

a = A()
print(A.f(), a.f(), sep=', ')
-> metoda statyczna, metoda statyczna
```

Relacja dziedziczenia

- Python wspiera **relację dziedziczenia** (*inheritance*)
- Relacja umożliwia tworzenie nowych klas na podstawie klas istniejących
- Klasa potomna, tzw. **podklasa** (*subclass*), specjalizuje klasę wyjściową, tzw. **nadklasę** (*superclass*)
- W Pythonie mamy **dziedziczenie wielobazowe** (*multiple inheritance*), tzn. że klasa potomna może posiadać wielu "rodziców"
- Nadklasą wszystkich klas w Pythonie jest klasa *object*

Relacja dziedziczenia

- Podklasa przejmuje zachowanie klas swoich przodków
- Oznacza to, że instancja podklasy ma dostęp do atrybutów (stanu oraz metod) wszystkich swoich nadklas
- Podczas odwołania się do atrybutu czy metody konieczny jest mechanizm jednoznacznego wyszukiwania, w ustalonym porządku, atrybutów o żądanej nazwie

Identyfikacja atrybutów

- Gdy w programie wystąpi odwołanie do atrybutu, w celu jego identyfikacji podejmowanych jest w kolejności kilka kroków:
  - najpierw atrybut jest poszukiwany w słowniku *__dict__*
  - jeśli nie zostanie tam odnaleziony, to poszukiwanie rozciąga się na wszystkie klasy wymienione w atrybucie *__bases__* w **określonej kolejności**
  - ponieważ klasy bazowe również mogą posiadać swoich przodków, to proces poszukiwania niejawnie dotyczy wszystkich przodków (niezależnie od pokolenia)
  - poszukiwanie kończy się w momencie znalezienia atrybutu o podanej nazwie

Kolejność poszukiwania – MRO

- Porządek przeszukiwania dotyczy **wszystkich typów atrybutów**, choć historycznie stosuje dla niego termin **MRO** (*method resolution order*)
- Poszukiwanie atrybutu o podanej nazwie odbywa się poprzez przeglądanie klas bezpośrednich przodków w kolejności **od lewej do prawej**
- Przed przejściem do kolejnej klasy na tym samym poziomie następuje analiza **w głąb** (zgodnie z relacją dziedziczenia – w kierunku klasy *object*)

Kolejność poszukiwania – MRO

- Każda klasa i wbudowany typ posiada atrybut tylko-do-odczytu *__mro__*, którego wartością jest krotka przeszukiwanych, w odpowiedniej kolejności klas
- Można także użyć wbudowanej, bezargumentowej metody *mro*
- Można ją wywołać na rzecz obiektu klasy
- Metoda jest wywoływana przy instancjonowaniu klasy, a jej wynik zapamiętywany w atrybucie *__mro__*

Nadpisywanie atrybutów

- Poszukiwanie atrybutu odbywa się zgodnie z porządkiem MRO (typowo w górę drzewa hierarchii dziedziczenia) i kończy w momencie jego odnalezienia
- Klasy potomne przeszukiwane są zawsze wcześniej – przed klasami przodków
- W konsekwencji, jeśli podklasa zdefiniuje atrybut o identycznej nazwie co nadklasa, to zostanie znaleziona definicja w podklasie i poszukiwanie tu się zakończy
- Taki mechanizm jest nazywany **przedefiniowywaniem** lub **nadpisywaniem** (*override*) atrybutów

Metoda *super*

- Metoda nadpisująca w podklasie może wywołać oryginalną metodę z nadklasy
- Można do tego wykorzystać delegację
- Wszystko działa dobrze dopóki w hierarchii dziedziczenia nie wystąpią grafy w kształcie diamentów (*diamond-shaped graphs*)

Metoda *super*

Przykład delegacji

```
class A:
    def jakas_metoda(self):
        print('jakas_metoda() z klasy A')

class B(A):
    def jakas_metoda(self):
        print('jakas_metoda() z klasy B')
        A.jakas_metoda(self)  # delegacja

class C(A):
    def jakas_metoda(self):
        print('jakas_metoda() z klasy C')
        A.jakas_metoda(self)  # delegacja

class D(B, C):
    def jakas_metoda(self):
        print('jakas_metoda() z klasy D')
        B.jakas_metoda(self)  # delegacja
        C.jakas_metoda(self)  # delegacja

d = D()
d.jakas_metoda()
```

Metoda *super*

- Taka konstrukcja klas powoduje, że podczas wywołania metody z klasy *D*, metoda z klasy *A* jest wywoływana dwukrotnie
- Wywołanie metody na rzecz obiektu proxy zwracanego przez metodę *super*, powoduje że takie wywołanie nastąpi tylko raz

Metoda *super*

Przykład delegacji

```
class A:
    def jakas_metoda(self):
        print('jakas_metoda() z klasy A')

class B(A):
    def jakas_metoda(self):
        print('jakas_metoda() z klasy B')
        super().jakas_metoda()  # delegacja

class C(A):
    def jakas_metoda(self):
        print('jakas_metoda() z klasy C')
        super().jakas_metoda()  # delegacja

class D(B, C):
    def jakas_metoda(self):
        print('jakas_metoda() z klasy D')
        super().jakas_metoda()  # delegacja

d = D()
d.jakas_metoda()
```

Dostęp do atrybutów

- Do **dostępu do atrybutów** można wykorzystać operator kropki
- Równoważnie można stosować funkcje:

Funkcje dostępu do atrybutów

```
getattr(obj, name[, default])
setattr(obj, name, value)
delattr(obj, name)

hasattr(obj, name)
```

- Parametr *obj* może wskazywać **instancję** lub **obiekt klasy**

Prywatne identyfikatory

- Identyfikatory rozpoczynające się pojedynczym znakiem podkreślenia _ służą do definiowania **prywatnych** zmiennych, funkcji, metod i klas
- Niektóre instrukcje importu pomijają takie identyfikatory $\rightarrow$ prywatność na poziomie modułu
- **Zgodnie z konwencją**, są one prywatne względem zasięgu w którym zostały zdefiniowane
- Kompilator **nie wymusza i nie pilnuje w żaden sposób realizacji prywatności** – jest to **tylko umowa** respektowana przez programistów

Silnie prywatne identyfikatory

- Identyfikatory rozpoczynające się, ale nie kończące, dwoma znakami podkreślenia __ są **silnie prywatne**
- Kompilator Pythona niejawnie przekształca takie nazwy zgodnie z regułą:

  $$\_\_identyfikator \rightarrow \_NazwaKlasy\_\_identyfikator$$

- Zmniejsza to ryzyko przypadkowego zdublowania nazw atrybutów, metod, czy zmiennych globalnych
- Ma to szczególne zastosowanie w podklasach

Metody specjalne

- **Metoda specjalna** (metoda magiczna) – metoda o specyficznej nazwie z dwoma wiodącymi i dwoma końcowymi podkreśleniami (*dunder method = double-underscored method*)
- Python niejawnie wywołuje takie specjalne metody, (klasa może je nadpisać), gdy na instancjach tej klasy wykonywane są różnego rodzaju operacje
- Znajomość tych metod jest niezbędna, jeśli chcemy “wzbogacić” zestaw dozwolonych operacji

Metody specjalne

- Python definiuje sporo metod specjalnych
- Przykładowo: klasy zachowujące się jak kontenery (sekwencje, słowniki, zbiory) standardowo powinny dostarczyć implementacji następujących metod:

Metody specjalne kontenerów

```
__getitem__(self, key)
__setitem__(self, key, value)
__delitem__(self, key)
__len__(self)
__contains__(self, item)
__iter__(self)
```

Metody specjalne

Przykład

```
class Kontener:
    def __init__(self, liczba_elementow=1):
        self.__lista = [None] * liczba_elementow

    def __str__(self):
        return ', '.join([str(element) for element in self.__lista])

    def __setitem__(self, indeks, element):
        self.__lista[indeks] = element

    def __getitem__(self, indeks):
        return self.__lista[indeks]


kontener = Kontener(3)

kontener[0] = 'abc'   # (wywołanie __setitem__)
kontener[1] = 'xyz'   # (wywołanie __setitem__)

print(kontener[0])    # (wywołanie __getitem__)
-> abc

print(kontener)       # (wywołanie __str__)
-> abc, xyz, None
```

Przeciążanie operatorów

- Metody specjalne dostarczają także implementacji operatorów
- Ze wszystkimi operatorami (arytmetycznymi, logicznymi, relacyjnymi, bitowymi, operacji na kontenerach, itp.) związane są odpowiednie metody specjalne
- We własnych klasach można te metody nadpisać (przedefiniować) i tym samym dodać nowe zachowanie (lub zmienić istniejące zachowanie) dla instancji klasy
- Tego typu możliwości noszą nazwę **przeciążania operatorów** (*operator overloading*)

Dekoratory funkcyjne

- Przykład dekoratora realizowanego poprzez funkcję

Przykład dekoratora funkcyjnego

```
# dekorator funkcyjny
def duze_litery(funkcja):
    def wrapped(*args, **kwargs):
        wynik = funkcja(*args, **kwargs)
        return wynik.upper()
    return wrapped

@duze_litery
def powitanie(imie):
    return f'Cześć {imie}'

komunikat = powitanie('Tomasz')
print(komunikat)
-> CZEŚĆ TOMASZ
```

Dekoratory klasowe

- Dekoratory można także zrealizować za pomocą klasy
- W tym przypadku dekorowaną funkcję przekazuje się do metody *__init__*
- Klasa musi implementować metodę specjalną *__call__*, gdyż instancja dekoratora musi zachowywać się jak funkcja (musi być *callable*)
- Metoda *__call__* jest wywoływana za każdym razem, gdy wywoływana jest udekorowana funkcja
- Argumenty podawane udekorowanej funkcji są przekazywane do metody *__call__*

Dekoratory klasowe bez arguentów

Przykład dekoratora klasowego bez argumentów

```
# dekorator klasowy (bez argumentów)
class DuzeLitery:
    def __init__(self, funkcja):
        self.__funkcja = funkcja

    def __call__(self, *args, **kwargs):
        wynik = self.__funkcja(*args, **kwargs)
        return wynik.upper()

@DuzeLitery
def powitanie(imie):
    return f'Cześć {imie}'

komunikat = powitanie('Tomasz')
print(komunikat)
-> CZEŚĆ TOMASZ
```

Dekoratory klasowe z argumentami

- Można także utworzyć dekorator klasowy do którego zostaną przekazane parametry
- W takiej sytuacji parametry przekazuje się do metody *__init__* – nie przekazuje się obiektu dekorowanej funkcji
- Obiekt funkcji przekazuje się do metody *__call__*
- Metoda *__call__* jest wywoływana tylko raz, jako część procesu dekorowania

Dekoratory klasowe z arguentami

Przykład dekoratora klasowego z argumentami

```
def powitanie(imie):
    return f'Cześć {imie}'

# dekorator
class DuzeLitery:
    def __init__(self, wersaliki):
        self.wersaliki = wersaliki

    def __call__(self, funkcja):
        def funkcja_udekorowana(*args, **kwargs):
            wynik = funkcja(*args, **kwargs)
            return wynik.upper() if self.wersaliki else wynik.title()
        return funkcja_udekorowana

powitanie = DuzeLitery(True)(powitanie)
komunikat = powitanie('Tomasz')
print(komunikat)
-> CZEŚĆ TOMASZ

powitanie = DuzeLitery(False)(powitanie)
komunikat = powitanie('Tomasz')
print(komunikat)
-> Cześć Tomasz
```

Deskryptory

- W ogólności **deskryptor** (*descriptor*) można traktować jak atrybut z dołączonym zachowaniem – umożliwia tworzenie “zarządzanych atrybutów”
- Atrybuty zarządzane są wykorzystywane do:
  - ochrony atrybutu przed zmianami
  - automatycznej aktualizacji wartości zależnego atrybutu

Deskryptory

- Aby klasa mogła być deskryptorem musi wypełnić następujący protokół:

Metody deskryptora

```
__get__(self, instance, owner)
__set__(self, instance, value)
__delete__(self, instance)
```

- Protokół określa, co się stanie, gdy odwołamy się do atrybutu
- Wystarczy, aby klasa zaimplementowała **przynajmniej jedną z powyższych metod**

Deskryptory

metoda	opis
__get__	zostanie wywołana w celu zwrócenia wartości atrybutu klasy lub obiektu – w przeciwnym razie powinna zgłosić wyjątek *AttributeError* parametr *instance* odnosi się do instancji właściciela parametr *owner* odnosi się do klasy właściciela
__set__	zostanie wywołana w celu nadania wartości atrybutowi
__delete__	zostanie wywołana w celu usunięcia atrybutu

- Powyższe metody mają zastosowanie, gdy instancję klasy deskryptora przypiszemy do **atrybutu klasowego** innej klasy (tzw. klasy właściciela)

Definicja deskryptora – przykład

- Jeśli deskryptor zostanie zdefiniowany następująco:

Definicja deskryptora

```
class Deskryptor:
    def __init__(self):
        self.__imie = ''

    def __get__(self, instance, owner):
        print('wywołanie __get__')
        return self.__imie

    def __set__(self, instance, imie):
        print('wywołanie __set__')
        self.__imie = imie.title()

class Osoba:
    imie = Deskryptor()
```

Użycie deskryptora – przykład

Użycie deskryptora

```
osoba = Osoba()
osoba.imie = 'jan'
-> wywołanie __set__

print(osoba.imie)
-> wywołanie __get__
-> Jan
```

Użycie deskryptora – przykład

- Jednakże...

Użycie deskryptora

```
osoba1 = Osoba()
osoba2 = Osoba()

osoba1.imie = 'jan'
-> wywołanie __set__
print(osoba1.imie, osoba2.imie)
-> wywołanie __get__
-> wywołanie __get__
-> Jan Jan

osoba2.imie = 'anna'
-> wywołanie __set__
print(osoba1.imie, osoba2.imie)
-> wywołanie __get__
-> wywołanie __get__
-> Anna Anna
```

Zachowanie deskryptora

- Dzieje się tak, gdyż mamy tylko **jedną instancję deskryptora** dla wielu instancji typu *Osoba*
- Każda próba odczytu odczytu atrybutu *imie* z poziomu którejkolwiek instancji *Osoba* powoduje wywołanie metody *__get__* deskryptora i zwrócenie **tej samej wartości** *_name*

- Aby móc przechować osobne wartości, specyficzne dla każdej instancji, deskryptor musi **zaimplementować słownik**, którego kluczami są instancje (w tym przypadku typu *Osoba*), a wartościami – wartości atrybutu
- W takim rozwiązaniu należy zadbać o usuwanie wpisów ze słownika, gdy usuwane są instancje (wykorzystuje się tu klasę *WeakRefDictionary* z modułu *weakref*)

Właściwości

- Python dostarcza **wbudowany typ deskryptora nadpisującego**, za pomocą którego można zdefiniować właściwości dla instancji klasy
- **Właściwość** (*property*) – atrybut instancyjny ze specyficzną funkcjonalnością:
  - dostęp do właściwości odbywa się z wykorzystaniem operatora kropki
  - faktycznie ta notacja powoduje wywołanie metod na danej instancji (służących do odczytu, nadania wartości lub usunięcia wartości)

Właściwości

- Do utworzenia właściwości wykorzystuje się wbudowaną funkcję *property*:

Funkcja *property*

```
property(fget = None, fset = None, fdel = None, doc = None)
```

- Parametry:

fget	obiekt funkcji zwracającej wartość atrybutu
fset	obiekt funkcji ustawiającej wartość atrybutu
fdel	obiekt funkcji usuwającej wartość atrybutu
fdoc	obiekt funkcji tworzącej *docstring*

Właściwości – przykład

Implementacja właściwości

```
class Osoba:
    def __init__(self):
        self.__imie = ''

    def __jakie_imie(self):
        print('odczyt imienia')
        return self.__imie

    def __ustaw_imie(self, imie):
        print('ustawianie imienia')
        self.__imie = imie.title()

    def __usun_imie(self):
        print('usunięcie imienia')

    imie = property(__jakie_imie, __ustaw_imie, __usun_imie,
                    'imię osoby')
```

Właściwości – przykład

Dostęp do atrybutu

```
osoba = Osoba()

osoba.imie = 'jan'
-> ustawianie imienia

print(osoba.imie)
-> odczyt imienia
-> Jan
```

Właściwości

- Funkcja *property* tworzy specjalny deskryptor nadpisujący, zawierający standardowe metody deskryptora *__get__*, *__set__* oraz *__delete__*
- Oprócz nich dostępne są jeszcze metody: *getter*, *setter* oraz *deleter*
  - wywołanie którejś z tych metod zwraca kopię deskryptora z nadpisaną jedną z trzech metod standardowych
  - dzięki tym metodom istnieje możliwość utworzenia właściwości za pomocą dekoratorów

Właściwości – przykład

Implementacja właściwości za pomocą dekoratorów

```
class Osoba:
    def __init__(self):
        self.__imie = ''

    @property
    def imie(self):
        print('odczyt imienia')
        return self.__imie

    @imie.setter
    def imie(self, imie):
        print('ustawianie imienia')
        self.__imie = imie.title()

    @imie.deleter
    def imie(self):
        print('usunięcie imienia')
```

Metaklasy

- Każdy obiekt, także obiekt klasy, ma swój typ
- Typ obiektu klasy jest nazywany **metaklasą** (*metaclass*)

Typy

```
class A: pass

print(type(A()))   # typ instancji
-> <class '__main__.A'>

print(type(A))     # typ obiektu klasy
-> <class 'type'>
```

- W powyższym przykładzie *type* jest metaklasą
- Jej instancjami są obiekty innych klas

Metaklasy

- W Pythonie typy i klasy są obiektami pierwszej kategorii
- Zachowanie obiektu zależy głównie od typu obiektu
- Dotyczy to również klas – zachowanie klasy jest głównie określane przez jej metaklasę
- Metaklasy są zasadniczo podklasami typu *type*

Metaklasa

```
class Meta(type): pass
```

- Kod metaklasy jest wykonywany pod koniec instrukcji **`class`**

Metaklasy

- Klasa może jawnie wskazać swoją metaklasę

W Pythonie 3.x . . .

```
class A(metaclass=Meta): pass

print(type(Meta))
-> <class 'type'>

print(type(A))
-> <class '__main__.Meta'>
```

W Pythonie 2.x . . .

```
class A(object):
    __metaclass__ = Meta
```

Metaklasy

- Można znaleźć wiele ciekawych zastosowań dla własnych metaklas
- Jednym z nich jest implementacja wzorca *Singleton*...

Metaklasy

Implementacja singletona

```
class Singleton(type):
    __inst = {}

    def __call__(cls, *args, **kwds):
        if cls not in cls.__inst:
            cls.__inst[cls] = super().__call__(*args, **kwds)
        return cls.__inst[cls]


class A(metaclass=Singleton):
    pass


a1 = A()
a2 = A()

print(a1 is a2)
-> True
```

Klasy abstrakcyjne

- **Klasy abstrakcyjne** to klasy zawierające jedną lub więcej metod abstrakcyjnych
- **Metoda abstrakcyjna** to metoda, która została zadeklarowana, ale nie zawiera implementacji (dokładniej: metoda, której implementacja **musi** być dostarczona w podklasie)
- Po klasach abstrakcyjnych można dziedziczyć i uzupełnić “brakujące” implementacje metod
- W ten sposób powstaje klasa kompletna (rzeczywista)

Klasy abstrakcyjne

- Ponieważ klasy abstrakcyjne są niekompletne (nie zawierają sensownej, oczekiwanej implementacji metod abstrakcyjnych), więc **nie można ich instancjonować**
- To ograniczenie nie dotyczy klas rzeczywistych
- Klasy abstrakcyjne **definiują kontrakt**, jaki muszą wypełnić klasy potomne – można narzucić jakie metody muszą być zaimplementowane w podklasach

Moduł *abc*

- Python bezpośrednio nie umożliwia definiowania klas abstrakcyjnych
- Taką infrastrukturę udostępnia moduł *abc*
- Nazwa jest akronimem pochodzącym od terminu *Abstract Base Classes*
- Pełną dokumentację do tego modułu można znaleźć pod tym linkiem

Definiowanie klas abstrakcyjnych

- Klasę abstrakcyjną tworzy się wskazując typ *ABCMeta* jako metaklasę

Klasa abstrakcyjna

```
from abc import ABCMeta

class KlasaAbstrakcyjna(metaclass=ABCMeta): pass
```

- Można też alternatywnie dziedziczyć po klasie *ABC*

Klasa abstrakcyjna

```
from abc import ABC

class KlasaAbstrakcyjna(ABC): pass
```

Klasy wirtualne

- Klasy abstrakcyjne umożliwiają rejestrację klas, jako tzw. **podklas wirtualnych**, dzięki metodzie *register*

Rejestracja wirtualnej podklasy

```
from abc import ABCMeta

class A(metaclass=ABCMeta): pass

@A.register
class B: pass

print(issubclass(B, A))
-> True
print(isinstance(B(), A))
-> True
```

- Podklasy wirtualne nie dziedziczą żadnych metod ani atrybutów (nie występują też w hierarchii *__mro__*)

Metody abstrakcyjne

- Moduł *abc* dostarcza kilku dekoratorów za pomocą których można zdefiniować metody abstrakcyjne

@abstractmethod	dekoruje instancyjną metodę abstrakcyjną
@abstractclassmethod	równoważny dekoratorom: *@classmethod* *@abstractmethod*
@abstractstaticmethod	równoważny dekoratorom: *@staticmethod* *@abstractmethod*
@abstractproperty	równoważny dekoratorom: *@property* *@abstractmethod*

Od Pythona 3.3 użycie dekoratorów *@abstractclassmethod*, *@abstractstaticmethod* oraz *@abstractproperty* nie jest zalecane

Klasy abstrakcyjne

- Wiele klas abstrakcyjnych można znaleźć w module *collections* (a dokładniej w *collections.abc*)

Callable	dowolna klasa z metodą *__call__*
Container	dowolna klasa z metodą *__contains__*
Hashable	dowolna klasa z metodą *__hash__*
Iterable	dowolna klasa z metodą *__iter__*
Sized	dowolna klasa z metodą *__len__*

Plan szkolenia

1 PROGRAMOWANIE FUNKCYJNE

2 PROGRAMOWANIE OOP

3 POMOCNE NARZĘDZIA

4 KOLEKCJE

5 WYRAŻENIA REGULARNE

6 PRZETWARZANIE DANYCH

7 BAZY DANYCH

8 WĄTKI I PROCESY

9 ASYNCHRONICZNY PYTHON

10 WSTĘP DO TESTÓW

Plan modułu

3 POMOCNE NARZĘDZIA

- moduł *typing* – adnotacje typów
- moduł *timeit* – pomiar czasu wykonania kodu
- moduł *logging* – praca z dziennikiem zdarzeń
- moduł *os* – powtórzenie oraz dodatkowe informacje
- moduł *sys*

Podpowiedzi typów

- Python jest językiem dynamicznie typowanym – nie ma konieczności deklaracji zmiennych, ani parametrów i określania ich typów
- W Pythonie 3.5 (PEP 483, PEP 484) wprowadzono nową funkcjonalność – możliwość **udzielania wskazówek dotyczących typów** (*Type Hints*) $\rightarrow$ można określać typy parametrów funkcji oraz typ zwracanej wartości
- W Pythonie 3.6 (PEP 526) rozszerzono te możliwości na precyzowanie typów zmiennych
- Kolejne rozszerzenia to dokumenty PEP 544, PEP 586, PEP 589 oraz PEP 591
- Pełny opis modułu *typing* dostępny jest pod tym linkiem

Podpowiedzi typów

- Po nazwie zmiennej lub parametru można dodać dwukropek i nazwę typu
- Typ zwracanej przez funkcję/metodę wartości określa się po znaku –>

Przykład

```
def czy_dorosly(wiek: int) -> bool:
    return wiek >= 18;
```

- Python nie wymusza używania podpowiedzi typów – mogą być one wykorzystywane np. przez IDE, narzędzia statycznej analizy kodu

Moduł *typing*

- Moduł *typing* został stworzony jako ujednolicona przestrzeń nazw dla podpowiedzi typów
- Podstawowymi elementami, które zawiera moduł są: *Any*, *Union*, *Tuple*, *Callable*, *TypeVar* i *Generic*
- Moduł zawiera również generyczne warianty wbudowanych kolekcji, m.in. takie jak: *Dict*, *DefaultDict*, *List*, *Set*, *FrozenSet*, ...

Moduł *typing*

Any	typ specjalny, zgodny z każdym innym typem z założenia każdy parametr funkcji bez adnotacji jest traktowany jako parametr tego typu
Union	typ oznaczający zbiór możliwych typów przyjmowanych przez argument typy są wymienione w nawiasach kwadratowych
Optional	deklaracja *Optional[X]* jest równoważna z *Union[X, None]*
Tuple	generyczny typ reprezentujący krotkę umożliwia zdefiniowanie struktury krotki (typów elementów krotki) od Pythona 3.9 niezalecany do użycia (*deprecated*), gdyż aktualnie typ wbudowany *tuple* wspiera []

Moduł *typing*

Callable	typ reprezentujący wykonywalny obiekt np. funkcję pierwszy argument – krotka typów argumentów jakie przyjmuje funkcja drugi argument – zwracany typ od Pythona 3.9 niezalecany do użycia (*deprecated*), gdyż aktualnie *collections.abc.Callable* wspiera []
TypeVar	fabryka typów używana do definiowania aliasów typów lub typów generycznych
Generic	pozwala na zdefiniowanie klasy jako typu generycznego

Moduł *typing*

- Do deklaracji zwracanych wartości zaleca się stosowanie rzeczywistych typów generycznych, np.:

TYP GENERYCZNY	ODPOWIADA TYPOWI...	Z MODUŁU...
Dict	*dict*	typ wbudowany
List	*list*	typ wbudowany
Set	*set*	typ wbudowany
FrozenSet	*frozenset*	typ wbudowany
DefaultDict	*defaultdict*	*collections*
OrderedDict	*OrderedDict*	*collections*
ChainMap	*ChainMap*	*collections*
Counter	*Counter*	*collections*
Deque	*Deque*	*collections*

Od Pythona 3.9 powyższe typy generyczne nie są zalecane do użycia (*deprecated*) – typy oryginalne aktualnie wspierają []

Moduł *typing*

- Do adnotowania argumentów można użyć generycznych typów abstrakcyjnych, np.:

TYP GENERYCZNY	ODPOWIADA TYPOWI...	Z MODUŁU...
AbstractSet	*Set*	*collections.abc*
Collection	*Collection*	*collections.abc*
ItemsView	*ItemsView*	*collections.abc*
KeysView	*KeysView*	*collections.abc*
ValuesView	*ValuesView*	*collections.abc*
Mapping	*Mapping*	*collections.abc*
Sequence	*Sequence*	*collections.abc*
Iterable	*Iterable*	*collections.abc*
Iterator	*Iterator*	*collections.abc*

Od Pythona 3.9 powyższe typy generyczne nie są zalecane do użycia (*deprecated*) – typy oryginalne aktualnie wspierają []

Moduł *typing*

Przykład

```
from typing import Iterable, List

class Osoba:
    def __init__(self, imie: str, nazwisko: str, wiek: int):
        self.imie = imie
        self.nazwisko = nazwisko
        self.wiek = wiek

    def __str__(self):
        return f'{self.imie} {self.nazwisko}, wiek: {self.wiek}'


def dorosli(osoby: Iterable[Osoba]) -> List[Osoba]:
    return [osoba for osoba in osoby if osoba.wiek >= 18]

grupa = Osoba('Jan', 'Kowalski', 25), \
        Osoba('Anna', 'Nowak', 17)
pelnoletni = dorosli(grupa)
```

Moduł *typing*

- Tworzenie aliasów

Przykład

```
from typing import Iterable, Union

U = Union[int, float, complex]  # alias
Vector = Iterable[U]            # alias


def iloczyn_skalarny(v1: Vector, v2: Vector) -> U:
    return sum(x * y for x in v1 for y in v2)


x = iloczyn_skalarny((1, 2, 3), [4.0, 5.5, 6.0])  # 93.0
```

Moduł *typing*

Przykład

```
from typing import Iterable, TypeVar

T = TypeVar('T', int, float, complex)  # typ generyczny
Vector = Iterable[T]                   # alias


def iloczyn_skalarny(v1: Vector[T], v2: Vector[T]) -> T:
    return sum(x * y for x in v1 for y in v2)


x = iloczyn_skalarny((1, 2, 3), [4, 5, 6])  # 90
```

Moduł *timeit*

- Moduł *timeit* umożliwia pomiar czasu wykonania niewielkich fragmentów kodu
- Definiuje klasę *Timer* oraz kilka funkcji pomocniczych
- Instancje klasy *Timer* reprezentują kod podlegający pomiarowi oraz określają metodę pomiaru
- Pełny opis modułu dostępny jest pod tym linkiem

Moduł *timeit*

Klasa *Timer*

```
Timer(stmt='pass', setup='pass', timer=<timer function>,
      globals=None)
```

PARAMETR	ZNACZENIE
stmt	kod, którego czas wykonania ma być mierzony
setup	kod wykonywany jednorazowo przed rozpoczęciem pomiarów czas jego wykonania jest pomijany
timer	funkcja timer'a
globals	przestrzeń nazw w której wykonywany jest kod

Moduł *timeit*

Metody klasy *Timer*

```
timeit(number=1000000)
autorange(callback=None)
repeat(repeat=5, number=1000000)
print_exc(file=None)
```

timeit	umożliwia dokonanie pomiaru skumulowanego czasu zadanej ilości (parametr *number*) wykonań kodu
autorange	metoda pomocnicza automatycznie dobierająca ilość wykonań kodu, tak by czas skumulowany był nie krótszy niż 0.2s
repeat	metoda pomocnicza umożliwiająca wielokrotne powtórzenie pomiarów czasu wywołuje metodę *timeit* zadaną ilość razy (parametr *repeat*)
print_exc	metoda pomocnicza wypisująca stosu

Moduł *timeit*

- Moduł dostarcza także funkcji pomocniczych które tworzą instancję *Timer*, a następnie wywołują na niej odpowiednie metody

Funkcja *timeit*

```
timeit(stmt='pass', setup='pass', timer=<default timer>,
       number=1000000, globals=None)

repeat(stmt='pass', setup='pass', timer=<default timer>,
       repeat=5, number=1000000, globals=None)
```

Moduł *timeit*

Przykład

```
import timeit

init = 'from math import sqrt'

kod = '''
def utworz_liste():
    lista = []
    for x in range(1000):
        lista.append(sqrt(x))
    return lista

utworz_liste()
'''

timer = timeit.Timer(setup=init, stmt=kod)
czas = timer.timeit(number=10_000))
```

Moduł *timeit*

Przykład

```
import timeit

init = 'from math import sqrt'

kod = '''
def utworz_liste():
    lista = []
    for x in range(1000):
        lista.append(sqrt(x))
    return lista

utworz_liste()
'''

czas = timeit.timeit(setup=init, stmt=kod, number=10_000))
```

Moduł *timeit*

Przykład

```
import timeit
from math import sqrt

def utworz_liste():
    lista = []
    for x in range(1000):
        lista.append(sqrt(x))
    return lista

czas = timeit.timeit(stmt=utworz_liste, number=10_000,
                     globals=globals())
```

Moduł *logging*

- Moduł *logging* dostarcza klas i funkcji umożliwiających stworzenie systemu logującego – dzienika zdarzeń
- Pełny opis modułu dostępny jest pod tym linkiem

Moduł *logging*

- System logujący wykorzystuje następujące typy obiektów:

Logger	reprezentuje obiekty poprzez które kierujemy komunikaty do systemu udostępnia API wykorzystywane przez aplikację komunikaty są formowane do rekordów (obiektów *LogRecord*)
Handler	jest odpowiedzialny za dostarczenie komunikatów do celu, z którym jest związany
Filter	umożliwia szczegółowe zdefiniowanie kryteriów jakie muszą spełniać rekordy dostarczane na wyjście
Formatter	jest odpowiedzialny za konwersję rekordów do postaci tekstowej odpowiedniej dla człowieka lub aplikacji, która je przetwarza

Moduł *logging*

- Przesyłanym komunikatom przypisuje się poziomy ważności

POZIOM	WARTOŚĆ NUMERYCZNA	PRZEZNACZENIE
CRITICAL	50	błąd krytyczny, unimożliwiający dalsze działanie aplikacji
ERROR	40	poważniejszy problem, aplikacja nie jest w stanie wykonać jakiejś funkcjonalności
WARNING	30	wydarzyło się coś nieoczekiwanego lub zasygnalizowanie jakiegoś problemu, ale aplikacja wciąż działa
INFO	20	potwierdzenie, że wszystko działa poprawnie
DEBUG	10	szczegółowe informacje dla celów diagnostycznych
NOTSET	0	

- Można też definiować własne poziomy ważności

Moduł *logging*

- Do wysłania komunikatu można użyć metody ogólnego zastosowania lub metod dedykowanych powiązanych z określonym poziomem ważności

Metody logujące

```
log(level, msg, *args, **kwargs)
exception(msg, *args, **kwargs)

# metody pomocnicze
debug(msg, *args, **kwargs)
info(msg, *args, **kwargs)
warning(msg, *args, **kwargs)
error(msg, *args, **kwargs)
critical(msg, *args, **kwargs)
```

Moduł *logging*

Przykład

```
from logging import *

# konfiguracja loggera
logger = getLogger(__name__)
logger.setLevel(WARNING)

file_handler = FileHandler('logfile.log')
format = '%(asctime)s : %(levelname)s : %(name)s : %(message)s'
formatter = Formatter(format)
file_handler.setFormatter(formatter)
logger.addHandler(file_handler)

# użycie loggera
logger.debug('Rozpoczelismy testowanie dzialania loggera')
logger.info('Do tego miejsca wszystko jest OK')
logger.warning('Cos jest nie tak')
logger.error('Funkcja nie dziala, ale pracujemy dalej')
logger.critical('Blad krytyczny - konczymy dzialanie')
```

Moduł *logging*

Zawartość pliku *logfile.log*

```
2020-11-04 08:48:23,339 : WARNING : __main__ : Cos jest nie tak
2020-11-04 08:48:23,339 : ERROR : __main__ : Funkcja nie dziala, ale pracujemy dalej
2020-11-04 08:48:23,339 : CRITICAL : __main__ : Blad krytyczny - konczymy dzialanie
```

Moduł *os*

- Moduł *os* jest podstawowym interfejsem usług systemu operacyjnego zarówno w Pythonie 3.X, jak i 2.X
- Zapewnia ogólną obsługę systemu operacyjnego i standardowy, niezależny od platformy zestaw narzędzi systemu operacyjnego
- Moduł zawiera narzędzia dla środowiska, procesów, plików, poleceń powłoki, itd.
- Zawiera również zagnieżdżony moduł podrzędny *os.path*, który zapewnia przenośny interfejs do narzędzi przetwarzania katalogów
- Pełny opis modułu dostępny jest pod tym linkiem

Moduł *os*

- Obiekt *environ* – słownik dający dostęp do zmiennych środowiska powłoki
- Jest inicjalizowany przy uruchomieniu programu
- Zmiany wartości są eksportowane na zewnątrz procesu Pythona i są dziedziczone przez wszystkie uruchamiane później procesy

Dostęp do zmiennych środowiska

```
import os

print(os.environ['OS'])
-> Windows NT

os.environ['HOME'] = r'C:\Users\jkowalski'
```

Moduł *os*

- Do modyfikacji zmiennych środowiska można także użyć metod:

Dostęp do zmiennych środowiska

```
putenv(key, value)
unsetenv(key)
getenv(key, default=None)
```

- Zmiany dokonane za pomocą metody *putenv* mają wpływ na procesy wystartowane poprzez wywołania: *system*, *popen*, *fork* oraz *execv*, ale nie aktualizują słownika *environ*

Moduł *os*

Inne metody związane ze środowiskiem

```
getcwd()
chdir(path)
strerror(code)
times()
umask(mask)
```

getcwd	zwraca nazwę bieżącego katalogu roboczego
chdir	umożliwia zmianę bieżącego katalogu roboczego dla danego procesu
strerror	zwraca komunikat dla danego kodu błędu
times	zwraca 5-elementową krotkę zawierającą informacje nt. wykorzystania czasu procesora (w sekundach) przez wywołujący proces krotka: *user, system, children_user, children_system, elapsed*
umask	ustawia nową maskę uprawnień i zwraca poprzednią

Moduł *os*

- Moduł definiuje wiele stałych, które ułatwiają przenoszenie aplikacji na inne systemy

os.name	nazwa systemu operacyjnego
os.curdir	symbol reprezentujący bieżący katalog
os.pardir	symbol reprezentujący katalog nadrzędny
os.sep	separator katalogów
os.extsep	separator nazwy pliku i rozszerzenia
os.pathsep	separator ścieżek
os.linesep	sekwencja znaków kończąca linię

Moduł *os*

- Do wywołania zewnętrznego polecenia można wykorzystać funkcję:

Składnia polecenia

```
system(command)
```

- Wykonuje ona polecenie w podpowłoce
- Jest to realizowane przez wywołanie standardowej funkcji C (*system*) i ma te same ograniczenia
- Główny proces Pythona czeka na zakończenie procesu potomnego
- Proces potomny dziedziczy strumienie *sys.stdin, sys.stdout, sys.stderr*

Moduł *os*

Przykład

```
import os

os.system('ping python.org')
```

```
-> Pinging python.org [45.55.99.72] with 32 bytes of data:
-> Reply from 45.55.99.72: bytes=32 time=120ms TTL=52
-> Reply from 45.55.99.72: bytes=32 time=120ms TTL=52
-> Reply from 45.55.99.72: bytes=32 time=123ms TTL=52
-> Reply from 45.55.99.72: bytes=32 time=124ms TTL=52
->
-> Ping statistics for 45.55.99.72:
->     Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
-> Approximate round trip times in milli-seconds:
->     Minimum = 120ms, Maximum = 124ms, Average = 121ms
-> 0
```

Moduł *os*

- Moduł *subprocess* zapewnia bardziej zaawansowane narzędzia do tworzenia nowych procesów i odbierania ich wyników

Przykład

```
import subprocess

subprocess.run('ping python.org')  # rozwiązanie zalecane
                                   # w porównaniu z poprzednim
-> Pinging python.org [45.55.99.72] with 32 bytes of data:
-> Reply from 45.55.99.72: bytes=32 time=120ms TTL=52
-> Reply from 45.55.99.72: bytes=32 time=118ms TTL=52
-> Reply from 45.55.99.72: bytes=32 time=121ms TTL=52
-> Reply from 45.55.99.72: bytes=32 time=121ms TTL=52
->
-> Ping statistics for 45.55.99.72:
->     Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
-> Approximate round trip times in milli-seconds:
->     Minimum = 118ms, Maximum = 121ms, Average = 120ms
-> CompletedProcess(args='ping python.org', returncode=0)
```

Moduł *os*

- Moduł *os* oferuje także całą rodzinę funkcji *exec** służących do uruchomienia programów

Funkcje *exec**

```
execl(path, arg0, arg1, ...)
execle(path, arg0, arg1, ..., env)
execlp(file, arg0, arg1, ...)
execlpe(file, arg0, arg1, ..., env)
execv(path, args)
execve(path, args, env)
execvp(file, args)
execvpe(file, args, env)
```

- Funkcje nadpisują bieżący proces Pythona
- Po zakończeniu operacji nie wracają – kod za wywołaniem nie może być wykonany

Moduł *os*

- Funkcje *exec** różnią się listą argumentów
- Te które w nazwie zawierają literę...

LITERA	DZIAŁANIE
l	kolejne parametry można podać w wywołaniu jeden za drugim
v	parametry są przekazywane w formie listy lub krotki
p	program jest poszukiwany na liście katalogów zmiennej systemowej *PATH* w pozostałych przypadkach konieczna jest pełna ścieżka do programu

Moduł *os*

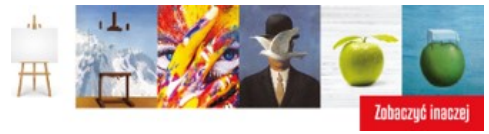
Przykład

```
import os

os.execlp('ping', 'ping', 'python.org')

-> C:\Users\student>
-> Pinging python.org [45.55.99.72] with 32 bytes of data:
-> Reply from 45.55.99.72: bytes=32 time=119ms TTL=52
-> Reply from 45.55.99.72: bytes=32 time=118ms TTL=52
-> Reply from 45.55.99.72: bytes=32 time=119ms TTL=52
-> Reply from 45.55.99.72: bytes=32 time=124ms TTL=52
->
-> Ping statistics for 45.55.99.72:
->     Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
-> Approximate round trip times in milli-seconds:
->     Minimum = 118ms, Maximum = 124ms, Average = 120ms
```

Moduł *sys*

- Moduł *sys* zapewnia dostęp do niektórych, specyficznych dla systemu parametrów używanych lub obsługiwanych przez interpreter oraz do funkcji, które wpływają bezpośrednio na działanie interpretera
- Pełna dokumentacja modułu jest dostępna pod tym linkiem

Moduł *sys* – wybrane parametry i funkcje



argv	lista argumentów wiersza poleceń przekazanych do skryptu
builtin_module_names	krotka nazw modułów wkompilowanych w interpreter Pythona
excepthook(type, value, traceback)	gdy wystąpi nieobsłużony wyjątek, następuje jego propagacja w górę stosu a bezpośrednio przed przekazaniem sterowania do systemu operacyjnego wywoływana jest funkcja *excepthook* można ją przedefiniować i zmienić sposób logowania informacji o wyjątku oryginalna wartość jest dostępna pod zmienną *__excepthook__*
exc_info()	funkcja dla wątka obsługującego wyjątek zwraca krotkę zawierającą: obiekt klasy, instancję wyjątku oraz ślad stosu

Moduł *sys* – wybrane parametry i funkcje

exit(arg=0)	zgłasza wyjątek *SystemExit*, który normalnie kończy działanie programu (po wykonaniu handlerów) argument o wartości 0 oznacza prawidłowe zakończenie
getrefcount(object)	zwraca liczbę referencji do obiektu
getrecursionlimit()	zwraca aktualny limit głębokości stosu Pythona
getsizeof(obj,[default])	zwraca rozmiar obiketu w bajtach (bez uwzględniania rozmiaru atrybutów)
modules	słownik zawierający nazwy i obiekty załadowanych modułów
path	lista katalogów i plików ZIP przeglądanych w poszukiwaniu modułu do załadowania
setrecursionlimit(limit)	ustawia głębokość stosu Pythona
stdin, stdout, stderr	predefiniowane strumienie: wejściowy, wyjściowy i konsola błędów

Moduł *sys*

Przykłady

```
import sys

print(sys.byteorder)
-> little
print(sys.float_info)
-> sys.float_info(max=1.7976931348623157e+308, max_exp=1024,
->                 max_10_exp=308, min=2.2250738585072014e-308,
->                 min_exp=-1021, min_10_exp=-307, dig=15,
->                 mant_dig=53, epsilon=2.220446049250313e-16,
->                 radix=2, rounds=1)
print(sys.maxsize)
-> 9223372036854775807
print(sys.maxunicode)
-> 1114111
print(sys.platform)
-> win32
print(sys.version)
-> 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40)
->       [MSC v.1927 64 bit (AMD64)]
```

Plan szkolenia

1 PROGRAMOWANIE FUNKCYJNE

2 PROGRAMOWANIE OOP

3 POMOCNE NARZĘDZIA

4 KOLEKCJE

5 WYRAŻENIA REGULARNE

6 PRZETWARZANIE DANYCH

7 BAZY DANYCH

8 WĄTKI I PROCESY

9 ASYNCHRONICZNY PYTHON

10 WSTĘP DO TESTÓW

Plan modułu

4 KOLEKCJE

- moduł *collections*
- *namedtuple* (moduł *collections*)
- *defaultdict* (moduł *collections*)
- *deque* (moduł *collections*)
- *Counter* (moduł *collections*)

Moduł *collections*

- Moduł *collections* definiuje specjalizowane typy danych kontenerów, które stanowią alternatywę dla typów wbudowanych, takich jak: *dict*, *list*, *set* oraz *tuple*
- Pełny opis modułu *collections* dostępny jest pod tym linkiem

Funkcja *namedtuple*

- **Nazwane krotki** przypisują znaczenie każdej pozycji w krotce i pozwalają na tworzenie bardziej czytelnego, samodokumentującego się kod
- Mogą być używane wszędzie tam, gdzie używane są zwykłe krotki i dodają możliwość dostępu do pól po nazwie (zamiast po indeksie)
- Funkcja *namedtuple* – funkcja fabryki służąca do tworzenia podklas krotek (klasy *tuple*) z nazwanymi polami

Funkcja *namedtuple*

Funkcja *namedtuple*

```
namedtuple(typename, field_names, *, rename=False, defaults=None,
           module=None)
```

typename	nazwa typu (podklasy *tuple*)
field_names	sekwencja lub łańcuch znakowy z nazwami pól pola krotki mogą być dostępne poprzez nazwy, być indeksowane lub iterowalne
rename	gdy `True`, błędne nazwy są automatycznie zamieniane na nazwy pozycyjne
defaults	wartość `None` lub obiekt iterowalny z wartościami domyślnymi wartości domyślne są wiązane z identyfikatorami od końca
module	wartość przypisywana do atrybutu *__module__* krotki

Funkcja *namedtuple*

Przykład

```
from collections import namedtuple

Prostokat = namedtuple('Prostokat', ['a', 'b'], defaults=[1])
p1 = Prostokat(4, b=7)      # Prostokat(a=4, b=7)

# użycie indeksów
pole  = p1[0] * p1[1]       # 28

# użycie nazw pól
obwod = 2 * (p1.a + p1.b)   # 22

x, y = p1                   # x = 4, y = 7

print(p1._fields)
-> ('a', 'b')

print(p1._field_defaults)
-> {'b': 1}
```

Funkcja *namedtuple*

Metody dodatkowe

```
_make(iterable)
_asdict()
_replace(**kwargs)
```

_make	tworzy nazwaną krotkę na podstawie sekwencji lub obiektu iterowalnego jest to metoda klasy
_asdict	konwertuje nazwaną krotkę na słownik
_replace	tworzy nową instancję krotki zamieniając wartości podanych pól

Funkcja *namedtuple*

Przykłady

```
from collections import namedtuple

Prostokat = namedtuple('Prostokat', 'a b')
wymiary = [3, 5]

p2 = Prostokat._make(wymiary)
print(p2)
-> Prostokat(a=3, b=5)

d = p2._asdict()
print(d)
-> {'a': 3, 'b': 5}

p3 = p2._replace(b=2)
print(p3)
-> Prostokat(a=3, b=2)
```

Klasa *defaultdict*

- Słowniki w Pythonie (klasa *dict*) to nieuporządkowane kolekcje par *klucz-wartość*
- Klucze muszą być unikalne i niemutowalne
- Próba odwołania się do słownika poprzez błędny klucz skutkuje wyrzuceniem wyjątku *KeyError*
- Sposobem na uniknięcie tego typu problemów jest użycie słownika *defaultdict*

Klasa *defaultdict*

- Klasa *defaultdict* jest podklasą klasy *dict*
- Funkcjonalności obu klas są prawie identyczne, ale podczas standardowego użycia słownika *defaultdict* nigdy nie jest zgłaszany wyjątek *KeyError*
- Klasa dostarcza automatycznie wartości domyślnej dla klucza, który nie istnieje

Klasa *defaultdict*

Klasa *defaultdict*

```
defaultdict([default_factory[, ...]])
```

- Argument *default_factory* inicjalizuje atrybut *default_factory* (w razie jego braku, domyślną wartością atrybutu jest **None**)
- Jego zadaniem jest przekazanie funkcji fabryki, dostarczającej domyślnych wartości dla kluczy, których nie ma w słowniku
- Pozostałe argumenty mają takie samo znaczenie, jak argumenty słownika *dict*

Klasa *defaultdict*

- Słownik *defaultdict*, oprócz metod słownika *dict*, definiuje metodę *__missing__(key)*
- Metoda wykorzystuje atrybut *default_factory* do utworzenia domyślnej wartości klucza (w razie braku fabryki zgłasza wyjątek *KeyError*)
- Jest wywoływana przez metodę *__getitem__*, która zwraca jej wynik (lub wyjątek)

Klasa *defaultdict*

Przykład

```
from collections import defaultdict

slownik = defaultdict(lambda: 'wartość domyślna')
slownik['a'] = 1
slownik['b'] = 2

print(slownik['a'])
-> 1

print(slownik['b'])
-> 2

print(slownik['c'])
-> wartość domyślna
```

Klasa *defaultdict*

Przykład

```
from collections import defaultdict

slownik = defaultdict(list)
miesiace = ('styczeń', 'luty', 'marzec', 'kwiecień', 'maj', 'czerwiec',
            'lipiec', 'sierpień', 'wrzesień', 'październik', 'listopad', 'grudzień')

for miesiac in miesiace:
    slownik[len(miesiac)].append(miesiac)

for dlugosc in range(1, 12):
    print(f'{dlugosc:2}: {slownik[dlugosc]}')
```

```
->  1: []
->  2: []
->  3: ['maj']
->  4: ['luty']
->  5: []
->  6: ['marzec', 'lipiec']
->  7: ['styczeń']
->  8: ['kwiecień', 'czerwiec', 'sierpień', 'wrzesień', 'listopad', 'grudzień']
->  9: []
-> 10: []
-> 11: ['październik']
```

Klasa *deque*

- Klasa *deque* dostarcza implementacji listy podwójnie wiązanej (*double-ended queue*)
- Umożliwia efektywną, w porównaniu z listami typu *list*, realizację kolejek (typu FIFO), jak i stosu (czyli kolejek LIFO)
- Klasa dostarcza metod wstawiania i usuwania elementów z początku i z końca kolekcji (złożoność czasowa *O(1)*, podczas gdy w przypadku *list* ta złożoność to *O(N)*)
- Operacje na kolejce są bezpieczne wielowątkowo

Klasa *deque*

Klasa *deque*

```
deque([iterable[, maxlen]])
```

- Kolejkę można utworzyć pustą lub wstępnie wypełnić ją elementami z podanego obiektu iterowalnego *iterable* (kolejne elementy są dodawane na końcu)
- Kolejka może być nieograniczona (gdy argument *maxlen* został pominięty lub ma wartość `None`)
- W przypadku kolejki ograniczonej dodanie kolejnego elementu do pełnej kolejki powoduje porzucenie elementu z drugiego końca

Klasa *deque*

- Oprócz metod klasy *list*, klasa *deque* oferuje metody i atrybuty specyficzne dla siebie

Elementy specyficzne klasy *deque*

```
# metody:
appendleft(x)
extendleft(iterable)
popleft()
rotate(n=1)

# atrybuty:
maxlen
```

Klasa *Counter*

- Klasa *Counter* to słownik (podklasa *dict*) umożliwiająca zliczanie obiektów hash'owalnych
- Podawane elementy stają się kluczami słownika, a wartościami powiązanymi z kluczami są ilości wystąpień
- Wartości mogą być dowolnymi liczbami całkowitymi (także ujemnymi i zerami)

Klasa *Counter*

Klasa *Counter*

```
Counter([iterable-or-mapping])
```

- Instancje klasy *Counter* można utworzyć wypełniając je wstępnie wartościami na podstawie podanej sekwencji lub słownika:

Przykłady

```
from collections import Counter

pusty = Counter()                           # Counter()
licznik_liter = Counter('kakao')            # Counter({'k': 2, 'a': 2, 'o': 1})
magazyn = Counter({'tv': 4, 'laptop': 7})   # Counter({'laptop': 7, 'tv': 4})
zakupy = Counter(mleko=2, bułki=6)          # Counter({'bułki': 6, 'mleko': 2})
```

Klasa *Counter*

- Klasa *Counter* rozszerza funkcjonalność klasy *dict* o kilka dodatkowych metod

Metody specyficzne dla klasy *Counter*

```
elements()
most_common([n])
subtract([iterable-or-mapping])
update([iterable-or-mapping])

fromkeys(iterable)   # metoda niezaimplementowana
```

Klasa *Counter*

elements	zwraca iterator po elementach słownika, powtarzający elementy tyle razy, ile wynosi ich liczebność
most_common	zwraca listę najliczniejszych elementów wraz z ich krotnościami lista może zawierać wszystkie elementy lub ich zadaną liczbę
subtract	odejmuje stan dwóch liczników, lub pomniejsza stan licznika o elementy podane w obiekcie iterowalnym
update	sumuje stan dwóch liczników, lub powiększa stan licznika o elementy podane w obiekcie iterowalnym

Klasa *Counter*

Przykłady

```
from collections import Counter

c = Counter('ananas')
print(c)
-> Counter({'a': 3, 'n': 2, 's': 1})

print(*c.elements())
-> a a a n n s

print(c.most_common(2))
-> [('a', 3), ('n', 2)]

c.subtract('nss')
print(c)
-> Counter({'a': 3, 'n': 1, 's': -1})

c.update(n=2, s=2)
print(c)
-> Counter({'a': 3, 'n': 3, 's': 1})
```

Plan szkolenia

1 PROGRAMOWANIE FUNKCYJNE

2 PROGRAMOWANIE OOP

3 POMOCNE NARZĘDZIA

4 KOLEKCJE

5 WYRAŻENIA REGULARNE

6 PRZETWARZANIE DANYCH

7 BAZY DANYCH

8 WĄTKI I PROCESY

9 ASYNCHRONICZNY PYTHON

10 WSTĘP DO TESTÓW

Plan modułu

5 WYRAŻENIA REGULARNE

- składnia – symbole, budowa wyrażeń regularnych
- moduł *re*
- narzędzia on-line

Wyrażenia regularne

- **Wyrażenia regularne** (*regular expressions*) – sekwencje znaków (wzorce) stosujące zwartą notację do definiowania łańcuchów symboli
- Jedno wyrażenie może opisywać nieograniczoną liczbę łańcuchów znakowych
- Wyrażenia regularne to potężne narzędzie w przetwarzaniu tekstów
  - przyspieszają proces programowania
  - są szybkie w działaniu
  - mogą być bardzo rozbudowane

Składnia wzorców

- W wyrażeniach regularnych mogą występować:
  - zwykłe znaki (*regular characters*) – rozumiane dosłownie
  - **metaznaki** (*metacharacters*) – znaki o specjalnym znaczeniu
- Litery alfabetu i cyfry są traktowane literalnie
- Metaznaki mają specjalne znaczenie w wyrażeniach regularnych i jeśli mają być traktowane literalnie, to trzeba je zamaskować, poprzedzając je odwrotnym ukośnikiem \
- Są nimi: \, ^, $, ., |, ?, *, +, (,), [, {

Składnia wzorców

- Ponieważ w wyrażeniach regularnych często występują odwrotne ukośniki, to wygodniej jest je zapisywać w składni *raw-strings*, np. zamiast '\\t' można napisać r'\t'
- Dokładne znaczenie elementów wzorców może ulec zmianie, gdy wraz ze wzorcem zostaną podane opcjonalne flagi

Metaznaki

metaznak	oznacza...
.	dowolny znak (oprócz znaku nowej linii) dowolny znak (jeśli podana flaga *DOTALL*) wewnątrz nawiasów kwadratowych oznacza kropkę
\t	tabulator <TAB>
\n	znak nowej linii <LF>
\r	znak powrotu karetki <CR>
\f	znak *form feed* <FF>
\v	znak pionowego tabulatora <VTAB>

Kotwice

metaznak	oznacza...
^	początek tekstu jeśli podana flaga *MULTILINE*, to obejmuje także tekst za \n
$	koniec tekstu jeśli podana flaga *MULTILINE*, to obejmuje także tekst przed \n
\A	pasuje do pustego łańcucha na początku całego tekstu
\Z	pasuje do pustego łańcucha na końcu całego tekstu
\b	pasuje do pustego łańcucha na początku lub końcu słowa
\B	pasuje do pustego łańcucha na pozycji innej niż początek lub koniec słowa

Grupy znaków

metaznak	oznacza...
[...]	dowolny, pojedynczy znak umieszczony pomiędzy nawiasami znaki można wyliczyć, np. [abcd] lub określić zakres, np. [a-d]
[^...]	negacja skrótów, dopełnienie zbioru dowolny, pojedynczy znak, inny niż te umieszczone pomiędzy nawiasami
(...)	wyrażenie regularne pomiędzy nawiasami wskazuje grupę
number	pasuje do wcześniej dopasowanej grupy o numerze *number* grupy są numerowane od 1 do 99
|	alternatywa – wyrażenie regularne z lewej lub z prawej strony

Klasy znaków

metaznak	oznacza...
\d	pojedyncza cyfra
\D	pojedynczy znak niebędący cyfrą
\s	znak biały równoważne wyrażeniu [\t\n\r\f\v]
\S	pojedynczy znak inny niż znak biały
\w	pojedynczy znak alfanumeryczny równoważne wyrażeniu [a-zA-Z0-9_]
\W	znak inny niż alfanumeryczny (inny niż \w)

Kwantyfikatory

- Kwantyfikatory w wyrażeniach regularnych umieszczane są za wyrażeniami i służą określeniu **krotności** dopasowań
- Brak kwantyfikatora oznacza jednokrotne dopasowanie

*	zero lub więcej wystąpień poprzedniego wyrażenia	$< 0; \infty)$
+	jedno lub więcej wystąpień poprzedniego wyrażenia	$< 1; \infty)$
?	zero lub jedno wystąpienie poprzedniego wyrażenia	$< 0; 1 >$
{n,}	conajmniej n wystąpień	$< n; \infty)$
{,n}	nie więcej niż n wystąpień	$< 0; n >$
{n}	dokładnie n wystąpień	n
{m, n}	pomiędzy m, a n wystąpień poprzedniego wyrażenia	$< m; n >$

Kwantyfikatory

- Wymienione kwantyfikatory są **zachłanne** (*greedy*)
- Warianty **leniwe** (*reluctant, non-greedy*) to odpowiednio: *?, +?, ?? oraz {m, n}?

Opcjonalne flagi

Python 3.x	Python 2.x	Znaczenie
I	IGNORECASE	w dopasowaniu nie jest uwzględniana wielkość liter
M	MULTILINE	opcja decyduje, czy symbole specjalne ^ i $ oznaczają początek i koniec linii, czy całego tekstu
S	DOTALL	powoduje, że symbol specjalny . reprezentuje dowolny znak, łącznie ze znakiem nowej linii
X	VERBOSE	powoduje pominięcie znaków białych we wzorcu, chyba, że są zamaskowane lub są elementami zbioru

Moduł *re*

- Moduł *re* dostarcza wsparcia dla wyrażeń regularnych w Pythonie
- Z wyrażeniami regularnymi współpracują dwa obiekty:

obiekt typu *Pattern*	obiekt wzorca reprezentuje skompilowane wyrażenie regularne
obiekt typu *Match*	reprezentuje dopasowany wzorzec

Obiekt wzorca

- Aby rozpocząć wykorzystywanie wzorców trzeba najpierw **skompilować** wyrażenie regularne

Składnia polecenia

```
compile(pattern, flags=0)
```

- Zachowanie wyrażenia można zmodyfikować za pomocą flag (można je łączyć za pomocą bitowego operatora “lub”)
- W wyniku otrzymamy reużywalny **obiekt wzorca**, który można wykorzystać do typowych operacji
- Moduł *re* dostarcza także dla każdej z tych operacji funkcji pomocniczych, dzięki którym można uniknąć kompilacji

Dopasowywanie wzorca

- Funkcje służące do wyszukiwania wzorców w tekście

Funkcje wyszukiwania

```
match(string[, pos[, endpos]])
search(string[, pos[, endpos]])
findall(string[, pos[, endpos]])
finditer(string[, pos[, endpos]])
```

Wyszukiwanie

FUNKCJA	DZIAŁANIE
match	próbuje dopasować skompilowany wzorzec **do początku tekstu**
search	stara się dopasować wzorzec **na dowolnej pozycji w tekście**

- Jeśli się powiedzie, to funkcje zwrócą obiekt typu *Match*, w przeciwnym razie – *None*
- Dodatkowe, opcjonalne parametry pozwalają określić zakres poszukiwań
- Polecenia *match* i *search* umożliwiają pojedyncze wyszukanie

Wyszukiwanie – przykład

Przykład

```
import re

wzorzec = re.compile(r'\w+[-\s]+\d{4}')
tekst = 'Szkolenie z Pythona - 2020'

if wzorzec.search(tekst):
    print('OK')

-> OK
```

Wyszukiwanie

Funkcja	Działanie
findall	zwraca **listę** wszystkich, nienakładających się **dopasowań**
finditer	działa podobnie do *findall*, ale w wyniku zwraca iterator po obiektach typu *Match*

Przykład

```
import re

wzorzec = re.compile(r'a*')
lista_znalezionych = wzorzec.findall('agawa')

print(lista_znalezionych)
-> ['a', '', 'a', '', 'a', '']
```

Modyfikacja tekstu

- Funkcje umożliwiające modyfikację tekstu:

Funkcje modyfikujące

```
split(string, maxsplit=0)
sub(repl, string, count=0)
subn(repl, string, count=0)
```

Modyfikacja tekstu

Funkcja	Działanie
split	umożliwia podział tekstu na części zgodnie z dopasowaniami wzorca jeśli funkcja ma zwrócić nie tylko podzielony tekst, ale również dopasowane wzorce (separatory), to należy użyć grup
sub	zwraca nowy tekst powstały przez zastąpienie dopasowanych wzorców w tekście źródłowym (parametr *string*) tekstem zastępczym (parametr *repl*)
subn	działa podobnie do *sub*, ale zwraca krotkę z nowym tekstem oraz liczbę dokonanych zamian

Modyfikacja tekstu – przykład

Przykład

```
import re

tekst1 = 'Programmer of Python'
tekst2 = re.sub(r'(\w+) of (\w+)', r'\2 \1', tekst1)

print(tekst2)
-> Python Programmer

tekst3 = re.sub(r'(\w+) of (\w+)',
                lambda m: f'{m.group(2)} {m.group(1)}', tekst1)

print(tekst3)
-> Python Programmer
```

- Dopasowania \1, \2, . . . to tzw. **referencje wsteczne** (*backreferences*)

Dopasowania

- Obiekt typu *Match* reprezentuje dopasowany wzorzec
- Jest wynikiem operacji: *match*, *search*, *finditer*
- Dostarcza szeregu metod dot. przechwyconych grup, zakresu gdzie nastąpiło dopasowanie, itp.

Funkcje dopasowań

```
group([group1, ...])
groups([default])
groupdict([default])
start([group])
end([group])
span([group])
expand(template)
```

Dopasowania

Funkcja	Działanie
match	zwraca podgrupy dopasowania jeśli nie zostaną podane żadne parametry (lub 0), wtedy zwracane jest pełne dopasowanie
groups	działa podobnie do *group*, ale zwraca krotkę wszystkich podgrup w dopasowaniu
groupdict	funkcja stosowana w przypadku użycia grup nazwanych zwraca słownik ze wszystkimi znalezionymi grupami
start	zwraca indeks początku dopasowania
end	zwraca indeks końca tekstu dopasowanego przez grupę
span	zwraca krotkę z indeksami początkowym i końcowym
expand	funkcja podobna w działaniu do *sub* zwraca tekst po zastąpieniu go referencjami wstecznymi w szablonie

Dopasowania

Przykład

```
import re

m = re.match(r'(\d+)\.(\d+)', '10.20')

print(m.groups())
-> ('10', '20')

print(m.group(0))
-> 10.20

print(m.group(1))
-> 10
```

Funkcje modułu

- Użyteczne funkcje modułu

Funkcje modułu

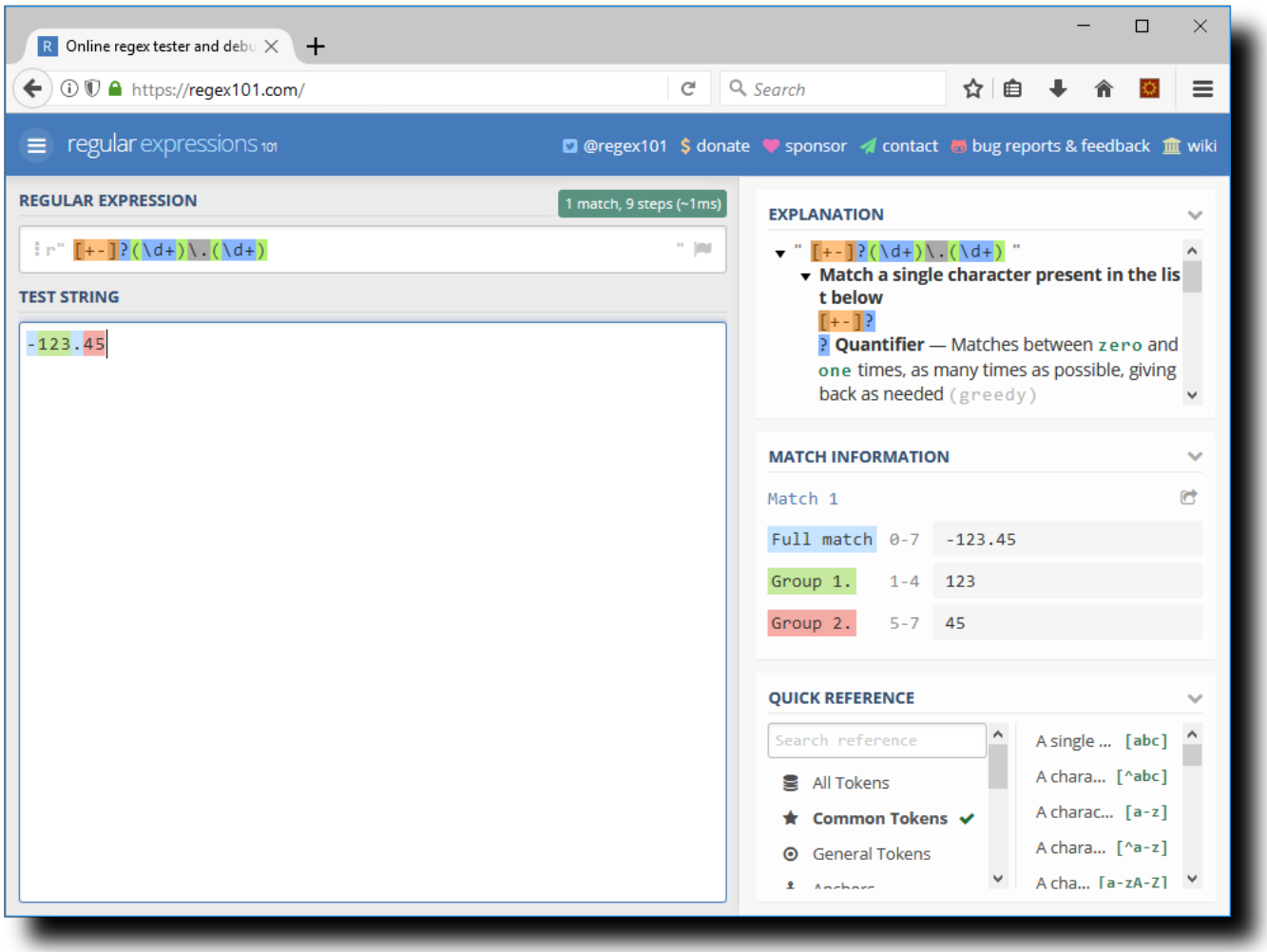
```
escape()
purge()
```

escape	“maskuje” literały, jakie mogą wystąpić w wyrażeniach
purge	opróżnia cache wyrażeń regularnych

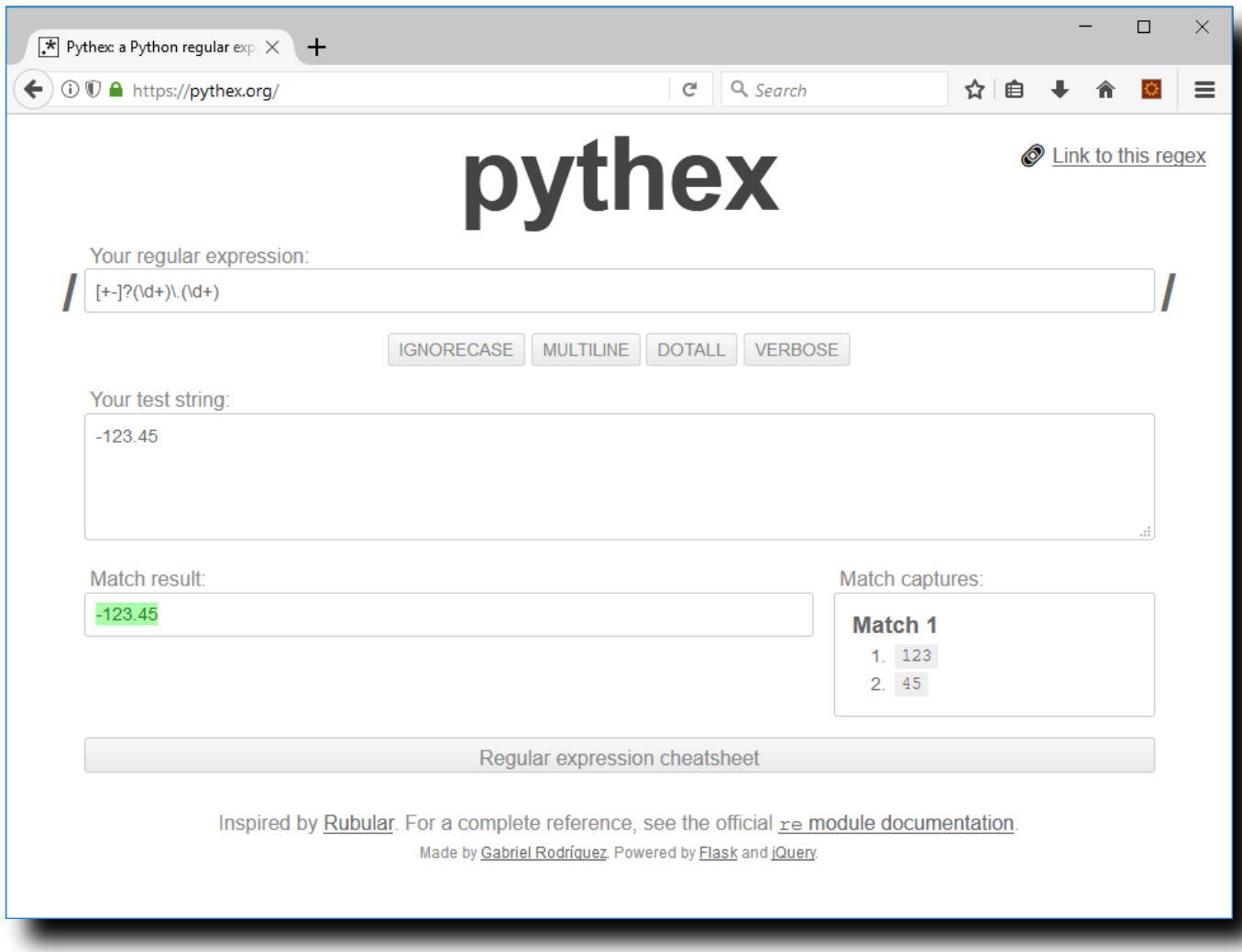
Narzędzia on-line do tworzenia wyrażeń regularnych

- Istnieje wiele dostępnych narzędzi on-line wspierających tworzenie wyrażeń regularnych w Pythonie
- *regex101*
  - autor: Firas Dib
  - narzędzie jest dostępne pod tym linkiem
- *pythex*
  - autor: Gabriel Rodríguez Alberich
  - narzędzie jest dostępne pod tym linkiem

Narzędzie *regex101*



Narzędzie *pythex*



Plan szkolenia

1 PROGRAMOWANIE FUNKCYJNE

2 PROGRAMOWANIE OOP

3 POMOCNE NARZĘDZIA

4 KOLEKCJE

5 WYRAŻENIA REGULARNE

6 **PRZETWARZANIE DANYCH**

7 BAZY DANYCH

8 WĄTKI I PROCESY

9 ASYNCHRONICZNY PYTHON

10 WSTĘP DO TESTÓW

Plan modułu

6 PRZETWARZANIE DANYCH

- moduł *requests* – wsparcie protokołu HTTP
- biblioteka *Beautiful Soup* – web scraping
- moduł *paramiko* – połączenia SSH
- serializacja i deserializacja
- wstęp do biblioteki *pandas*
- przegląd innych bibliotek

Moduł *requests*

- Moduł *requests* upraszcza komunikację za pomocą protokołu HTTP
  - wspiera metody protokołu HTTP/1.1
  - nie ma potrzeby ręcznego budowania URL (w tym dołączania parametrów)
  - nie ma potrzeby kodowania danych w żądaniach POST i PUT
- Dokumentacja modułu jest dostępna pod tym linkiem

Wysyłanie żądań HTTP

- Do wysłania żądania HTTP można użyć metody ogólnego zastosowania lub którejś z pomocniczych metod dedykowanych

Wysyłanie żądań

```
# funkcja ogólnego zastosowania
request(method, url, **kwargs)

# funkcje pomocnicze
head(url, **kwargs)
get(url, params=None, **kwargs)
post(url, data=None, json=None, **kwargs)
put(url, data=None, **kwargs)
patch(url, data=None, **kwargs)
delete(url, **kwargs)
options(url, **kwargs)
```

- W odpowiedzi metody zwracają obiekt *Response*

Wysyłanie żądań HTTP

Żądanie GET

```
from requests import *

r = request('GET', 'https://www.python.org/')
# równoważnie: r = get('https://www.python.org/')

print(f'Kod statusu: {r.status_code}')
-> Kod statusu: 200
print(f'Nagłówki HTTP: {r.headers}')
-> Nagłówki HTTP: {
->     'Connection': 'keep-alive',
->     'Content-Length': '49325',
->     'Server': 'nginx',
->     'Content-Type': 'text/html; charset=utf-8',
->     'X-Frame-Options': 'DENY',
->     'Via': '1.1 vegur, 1.1 varnish, 1.1 varnish',
->     'Accept-Ranges': 'bytes',
->     'Date': 'Sat, 14 Nov 2020 15:05:50 GMT',
->     'Age': '3201',
->     'X-Served-By': 'cache-bwi5123-BWI, cache-fra19162-FRA',
->     'X-Cache': 'HIT, HIT',
->     'X-Cache-Hits': '2, 12',
->     'X-Timer': 'S1605366350.143188,VS0,VE0',
->     'Vary': 'Cookie',
->     'Strict-Transport-Security': 'max-age=63072000; includeSubDomains'
->  }
```

Wysyłanie żądań HTTP

- Wysyłanie żądania GET z parametrami

Żądanie GET z parametrami

```
from requests import *

parametry_zapytania = {
    'param1': 'wart1',
    'param2': 'wart2'
}
naglowki_zapytania = {'accept': 'text/html'}

r = get('https://www.python.org/',
        params=parametry_zapytania,
        headers=naglowki_zapytania)
print(r.url)
-> https://www.python.org/?param1=wart1&param2=wart2
```

Treść odpowiedzi

- Aby przeczytać treść odpowiedzi z serwera wystarczy odwołać się do atrybutu *text* obiektu odpowiedzi
- Do jego dekodowania zostanie użyte kodowanie określone w nagłówku HTTP
- Można je sprawdzić odwołując się do atrybutu *encoding*
- Można zmienić wartość tego atrybutu, a tym samym zmienić używane kodowanie przy każdym odczycie atrybutu *text*
- Odpowiedź w formie binarnej jest dostępna poprzez atrybut *content*
- Python dostarcza także dekoder JSON – metodę *json*

Biblioteka *Beautiful Soup*

- Biblioteka *Beautiful Soup 4* to biblioteka Pythona służąca do wydobywania danych z plików HTML i XML (*web scraping, web harvesting, web data extraction*)
- Przy użyciu wskazanego parsera umożliwia nawigację, wyszukiwanie i modyfikację drzewa dokumentu
- Dokumentacja biblioteki dostępna jest pod tym linkiem

Podstawowe klasy

- Podstawowe klasy biblioteki *Beautiful Soup*:

KLASA	REPREZENTUJE...
BeautifulSoup	przeparsowany dokument
Tag	element HTML/XML
NavigableString	zawartość tekstową elementu

Podstawowe klasy

- Klasa *NavigableString* posiada kilka podtypów szczegółowych:

KLASA	REPREZENTUJE...
Comment	komentarz
CData	blok CDATA
ProcessingInstruction	instrukcja przetwarzania
Declaration	deklaracja
DocType	typ dokumentu DOCTYPE

- Biblioteka dostarcza także wsparcia dla typów kaskadowych CSS (klasy: *Stylesheet*, *Script* oraz *TemplateString*)

Wybór parsera

- Do parsowania można wykorzystać wbudowany parser lub inny zainstalowany:

PARSER	SPOSÓB UŻYCIA
wbudowany parser HTML	*BeautifulSoup(markup, 'html.parser')*
lxml parser HTML	*BeautifulSoup(markup, 'lxml')*
lxml parser XML	*BeautifulSoup(markup, 'lxml-xml')* *BeautifulSoup(markup, 'xml')*
html5lib parser HTML	*BeautifulSoup(markup, 'html5lib')*

Parsowanie dokumentu

- Dane do parsowania można pobrać z pliku lub podać bezpośrednio

Parsowanie dokumentu z pliku

```
from bs4 import BeautifulSoup

with open('index.html') as plik:
    bs = BeautifulSoup(plik, 'html.parser')
```

Parsowanie dokumentu

Parsowanie podanej treści

```
from bs4 import BeautifulSoup

markup = (
    '<html>'
    '  <head>'
    '    <title>Strona HTML</title>'
    '  </head>'
    '  <body color="white">To jest dokument HTML</body>'
    '</html>')
bs = BeautifulSoup(markup, 'html.parser')

print(type(bs))
-> <class 'bs4.BeautifulSoup'>
```

Dostęp do składników dokumentu

- Element posiada kilka przydatnych atrybutów:

ATRYBUT	PRZEZNACZENIE
name	nazwa elementu
string	zawartość tekstowa elementu
attrs	słownik atrybutów elementu

Dostęp do składników dokumentu

- Nawiązując do ostatniego przykładu...

Dostęp do składników dokumentu

```
element = bs.body
print(element)
->  <body color="white">To jest dokument HTML</body>

print(type(element))
-> <class 'bs4.element.Tag'>

print(element.name)
-> body

print(element.string)
-> To jest dokument HTML

print(element.attrs)
-> {'color': 'white'}
```

Dostęp do składników dokumentu

- Do wartości wybranego atrybutu elementu można dotrzeć traktując element jak słownik (nazwa atrybutu pełni rolę klucza)

Dostęp do składników dokumentu

```
print(element['color'])
-> white
```

- Atrybuty można także dodawać, usuwać i modyfikować

Wyszukiwanie elementów

Wyszukiwanie elementów

```
find_all(name, attrs, recursive, string, limit, **kwargs)
find_all_next(name, attrs, string, limit, **kwargs)
find_all_previous(name, attrs, string, limit, **kwargs)

find(name, attrs, recursive, string, **kwargs)
find_next(name, attrs, string, **kwargs)
find_previous(name, attrs, string, **kwargs)

find_parents(name, attrs, string, limit, **kwargs)
find_parent(name, attrs, string, **kwargs)

find_next_siblings(name, attrs, string, limit, **kwargs)
find_previous_siblings(name, attrs, string, limit, **kwargs)

find_next_sibling(name, attrs, string, **kwargs)
find_previous_sibling(name, attrs, string, **kwargs)
```

Wyszukiwanie elementów

- Parametr *name* reprezentuje nazwę poszukiwanego elementu
- Jeśli zostanie podany parametr nazwany (*klucz=wartość*), to wyszukiwanie będzie dotyczyło elementów, które posiadają podany atrybut o danej wartości
- Wartość atrybutu można podać w formie tekstu, wyrażenia regularnego, listy, funkcji lub wartości logicznej (wtedy wartość atrybutu jest nieistotna – określamy tylko, czy atrybut o podanej nazwie musi, czy nie może występować)

Wyszukiwanie elementów

- Wywołanie metody *find_all* można pominąć i potraktować dokument lub element jak funkcję

Wyszukiwanie elementów

```
print(bs.find_all('title'))
-> [<title>Strona HTML</title>]
print(bs('title'))
-> [<title>Strona HTML</title>]

print(bs.find_all(color=True))
-> [<body color="white">To jest dokument HTML</body>]
print(bs(color=True))
-> [<body color="white">To jest dokument HTML</body>]
```

Moduł *paramiko*

- *Paramiko* – implementacja protokołu SSH v.2 w Pythonie
- Umożliwia stworzenie kodu po stronie serwera, jak i klienta
- Pełna dokumentacja tej biblioteki jest dostępna pod tym linkiem

Budowa klienta SSH

- Standardowy sposób budowy klienta SSH:
  - utworzenie instancji *SSHClient* – wysokopoziomowej reprezentacji sesji z serwerem SSH
  - ustawienie polityki akceptacji nieznanych kluczy SSH
  - nawiązanie połączenia ze wskazanym serwerem na danym porcie – opcjonalnie można podać login i hasło
  - wysłanie polecenia do wykonania – w odpowiedzi zwracana jest krotka strumieni do komunikacji z serwerem
  - odebranie odpowiedzi
  - na zakończenie komunikacji – zamknięcie połączenia
- Można wykorzystać fakt, że klient jest menedżerem kontekstu

Budowa klienta SSH

Budowa klienta SSH

```
from paramiko import *

host = 'localhost'
port = 22
login = 'username'
haslo = 'password'
komenda = 'dir'
with SSHClient() as klient:
    klient.set_missing_host_key_policy(AutoAddPolicy())
    klient.connect(host, port, login, haslo)
    stdin, stdout, stderr = klient.exec_command(komenda)
    odpowiedz = stdout.readlines()
    print(odpowiedz)
```

- Standardowo biblioteka wykorzysta kodowanie ’utf-8’
- Jeśli serwer wspiera inne kodowanie, można je wskazać, np.: *odpowiedz = stdout.read().decode(’cp1250’)*

Serializacja

- **Serializacja** to proces konwersji hierarchii obiektów do strumienia danych
- **Deserializacja** to proces odwrotny, umożliwiający odtworzenie struktury obiektów ze strumienia danych
- Serializacja (i deserializacja) może być **binarna** lub **tekstowa** – w zależności od typu strumienia danych

Serializacja binarna

- Protokoły binarne umożliwiające serializację i deserializację obiektów Pythona są zaimplementowane w modułach: *pickle* oraz *marshal*
- Pełna dokumentacja możliwości modułu *pickle* jest dostępna pod tym linkiem
- Opis możliwości modułu *marshal* można znaleźć pod tym linkiem
- Możliwości modułu *marshal* są ograniczone w porównaniu z modułem *pickle* – zalecane jest użycie modułu *pickle*

Serializacja binarna

- Porównanie modułów *pickle* i *marshal*
  - moduł *pickle* śledzi obiekty, które zostały zserializowane, więc ten sam obiekt zostanie zserializowany tylko raz (nie ma problemu z rekurencją obiektów, ani obiektami współdzielonymi)
  - moduł *marshal* nie umożliwia serializacji klas i instancji użytkownika
  - format serializacji modułu *marshal* nie gwarantuje przenośności pomiędzy różnymi wersjami Pythona

Serializacja binarna

- Funkcje modułu *pickle* umożliwiające serializację

Serializacja

```
# serializacja obiektu do pliku
dump(obj, file, protocol=None, *, fix_imports=True,
     buffer_callback=None)

# serializacja obiektu do obiektu typu bytes
dumps(obj, protocol=None, *, fix_imports=True,
      buffer_callback=None)
```

Serializacja binarna

Przykład serializacji binarnej obiektu

```
from pickle import dump

dane = {
    'a': [1, 2.0, True],
    'b': ('tekst', b'\xc5\xbc\xc3\xb3\xc5\x82w'),
    'c': None
}

with open('dane.pkl', 'wb') as plik:
    dump(dane, plik)
```

Deserializacja binarna

- Funkcje modułu *pickle* umożliwiające deserializację

Deserializacja

```
# deserializacja obiektu z pliku
load(file, *, fix_imports=True, encoding="ASCII",
     errors="strict", buffers=None)

# deserializacja obiektu z obiektu typu bytes
loads(data, /, *, fix_imports=True, encoding="ASCII",
      errors="strict", buffers=None)
```

Deserializacja binarna

Przykład deserializacji binarnej obiektu

```
from pickle import load

with open('dane.pkl', 'rb') as plik:
    dane = load(plik)

print(type(dane))
-> <class 'dict'>

for k, v in dane.items():
    print(f'{k}: {v}')

-> a: [1, 2.0, True]
-> b: ('tekst', b'\xc5\xbc\xc3\xb3\xc5\x82w')
-> c: None
```

Serializacja tekstowa

- Użycie modułu *pickle* **nie jest bezpieczne** – możliwe jest takie spreparowanie danych binarnych, że podczas deserializacji zostanie wykonany dodatkowy kod
- Dlatego nie należy dokonywać deserializacji danych pochodzących z niezaufanych źródeł
- W takim przypadku lepiej użyć serializacji i deserializacji tekstowej

Serializacja do JSON

- **JSON** (*JavaScript Object Notation*) to popularny format danych używany do reprezentowania danych strukturalnych
- Przesyłanie i odbieranie danych między serwerem a aplikacją internetową w formacie JSON jest powszechne
- Konwersję obiektu do formatu JSON (i deserializację) umożliwia moduł *json*
- Pełna dokumentacja tego modułu jest dostępna pod tym linkiem

Serializacja do JSON

- Funkcje do serializacji i deserializacji przypominają w działaniu te z modułu *pickle*

Funkcje serializacji obiektu do formatu JSON

```
# serializacja obiektu do pliku (w formacie JSON)
dump(obj, fp, *, skipkeys=False, ensure_ascii=True,
     check_circular=True, allow_nan=True, cls=None, indent=None,
     separators=None, default=None, sort_keys=False, **kw)

# serializacja obiektu do tekstu (w formacie JSON)
dumps(obj, *, skipkeys=False, ensure_ascii=True,
      check_circular=True, allow_nan=True, cls=None, indent=None,
      separators=None, default=None, sort_keys=False, **kw)
```

Serializacja do JSON

- Podczas konwersji do formatu JSON stosowana jest następująca tablica konwersji:

PYTHON	JSON
dict	*object*
list, *tuple*	*array*
str	*string*
int, *float*, numeryczne *Enum*	*number*
True	*true*
False	*false*
None	*null*

Serializacja do JSON

Przykład serializacji do formatu JSON

```
from json import dumps

dane = ['Jan Kowalski',
        {
          'adres': ('ul. Morska 123', '82-103', 'Stegna'),
          'inne_dane': (None, 1, 2.0, False)
        }
]

json = dumps(dane)
print(json)
-> ["Jan Kowalski", {"adres": ["ul. Morska 123", "82-103",
->  "Stegna"], "inne_dane": [null, 1, 2.0, false]}]
```

- W przypadku serializacji klas lub instancji własnych typów trzeba wskazać funkcję konwersji (parametr *default*) lub klasę kodera rozszerzającą typ *JSONEncoder* (parametr *cls*)

Deserializacja z JSON

Funkcje deserializacji obiektu z formatu JSON

```
# deserializacja obiektu z pliku z zawartością JSON
load(fp, *, cls=None, object_hook=None, parse_float=None,
     parse_int=None, parse_constant=None,
     object_pairs_hook=None, **kw)

# deserializacja obiektu ze stringa z JSONem
loads(s, *, cls=None, object_hook=None, parse_float=None,
      parse_int=None, parse_constant=None,
      object_pairs_hook=None, **kw)
```

Deserializacja z JSON

- Podczas deserializacji z formatu JSON do obiektu Pythona stosowana jest następująca tablica konwersji:

JSON	PYTHON
object	*dict*
array	*list*
string	*str*
number (int)	*int*
number (real)	*float*
true	*True*
false	*False*
null	*None*

Deserializacja z JSON

Przykład deserializacji z formatu JSON

```
from json import loads

json = '["Jan Kowalski", {"adres": ["ul. Morska 123", ' \
     '"82-103", "Stegna"], "inne_dane": [null, 1, 2.0, false]}]'

obj = loads(json)

print(obj)
-> ['Jan Kowalski', {'adres': ['ul. Morska 123', '82-103',
->  'Stegna'], 'inne_dane': [None, 1, 2.0, False]}]
```

- W przypadku deserializacji klas lub instancji własnych typów trzeba wskazać funkcję konwersji (parametr *object_hook* lub *object_pairs_hook*) lub klasę dekodera rozszerzającą typ *JSONDecoder* (parametr *cls*)

Serializacja do YAML

- **YAML** (*YAML Ain't Markup Language*) to tekstowy format serializacji danych
- Jest powszechnie używany w plikach konfiguracyjnych, ale służy również do przechowywania i transmisji danych
- Popularnym modułem Pythona wspierającym YAML jest *PyYAML*
- Pełna dokumentacja dla tego modułu jest dostępna pod tym linkiem

Składnia YAML

- YAML natywnie obsługuje trzy podstawowe typy danych: wartości skalarne (np. łańcuchy, liczby całkowite i zmiennoprzecinkowe), listy oraz tablice asocjacyjne

Listy

```
--- # lista
- element1
- element2
- element3
--- # równoważnie:
[element1, element2, element3]
```

Tablice asocjacyjne

```
--- # tablica asocjacyjna
  klucz1: wartość1
  klucz2: wartość2
--- # równoważnie:
{klucz1: wartość1, klucz2: wartość2}
```

Serializacja do YAML

- Do serializacji obiektów Pythona do formatu YAML można użyć funkcji:

Serializacja do YAML

```
dump(data, stream=None, Dumper=Dumper,
     default_style=None,
     default_flow_style=None,
     encoding=None
     explicit_start=None,
     explicit_end=None,
     version=None,
     tags=None,
     canonical=None,
     indent=None,
     width=None,
     allow_unicode=None,
     line_break=None)

dump_all(data, stream=None, Dumper=Dumper, ...)
```

Serializacja do YAML

Przykład

```
from yaml import dump

szkolenia = [
    {
        'kod': 'PYTH01',
        'nazwa': 'Podstawy programowania w jezyku Python',
        'poziom': 1
    },
    {
        'kod': 'PYTH02',
        'nazwa': 'Zaawansowane techniki programowania w jezyku Python',
        'poziom': 2
    }
]

print(dump(szkolenia))
-> - kod: PYTH01
->   nazwa: Podstawy programowania w jezyku Python
->   poziom: 1
-> - kod: PYTH02
->   nazwa: Zaawansowane techniki programowania w jezyku Python
->   poziom: 2
```

Deserializacja z YAML

- Do deserializacji danych YAML do obiektów Pythona można użyć funkcji:

Deserializacja z YAML

```
load(stream, Loader=Loader)

load_all(stream, Loader=Loader)
```

Deserializacja z YAML

Przykład

```
from yaml import load, SafeLoader

dane = '''
- kod: PYTH01
  nazwa: Podstawy programowania w jezyku Python
  poziom: 1
- kod: PYTH02
  nazwa: Zaawansowane techniki programowania w jezyku Python
  poziom: 2
'''

lista = load(dane, Loader=SafeLoader)
print(lista)
-> [
-> {'kod': 'PYTH01',
->  'nazwa': 'Podstawy programowania w jezyku Python',
->  'poziom': 1},
-> {'kod': 'PYTH02',
->  'nazwa': 'Zaawansowane techniki programowania w jezyku Python',
->  'poziom': 2}
-> ]
```

Biblioteka *pandas*

- Biblioteka *pandas* to biblioteka Pythona służąca do manipulacji i analizy danych
- W szczególności oferuje struktury danych i operacje do manipulacji tabelami numerycznymi i szeregami czasowymi
- Nazwa pochodzi od terminu *panel data* – terminu ekonometrycznego określającego zbiory danych, które obejmują obserwacje w wielu okresach dla tych samych osób
- Pełna dokumentacja biblioteki *pandas* jest dostępna pod tym linkiem

Typy danych

- Dwa podstawowe typy danych to:

Series	struktura jednowymiarowa reprezentuje ciąg danych
DataFrame	struktura dwuwymiarowa organizująca dane w wiersze i kolumny reprezentuje dane tabelaryczne

- Biblioteka potrafi współpracować z wieloma źródłami danych (m.in. CSV, JSON, dane Excela, . . .)

Serializacja *DataFrame*

- Klasa *DataFrame* udostępnia wiele metod umożliwiających konwersję danych do innych formatów (i zapisu do plików)

METODA	KONWERTUJE/KOPIUJE DANE DO...
to_csv([path_or_buf, sep, na_rep, ...])	formatu CSV
to_excel(excel_writer[, sheet_name, na_rep, ...])	arkusza Excel'a
to_html([buf, columns, col_space, header, ...])	tabeli HTML
to_json([path_or_buf, orient, date_format, ...])	formatu JSON
to_latex([buf, columns, col_space, header, ...])	tabeli LaTEX
to_pickle(path[, compression, protocol])	formatu binarnego Pickle
to_dict([orient, into])	słownika Pythona
to_clipboard([excel, sep])	systemowego schowka
...	...

Utworzenie instancji *DataFrame*

Utworzenie instancji *DataFrame*

```
import pandas as pd

kwartal1 = {
    'miesiac': ('styczeń', 'luty', 'marzec'),
    'liczba dni': (31, 28, 31)
}

df = pd.DataFrame(kwartal1, columns=['miesiac', 'liczba dni'])
df.index += 1

print(df)
->     miesiac  liczba dni
-> 1   styczeń          31
-> 2      luty          28
-> 3    marzec          31
```

- Powyższa instancja będzie wykorzystana w kolejnych przykładach

Format CSV

Serializacja do CSV

```
import pandas as pd

kwartal1 = {
    'miesiac': ('styczeń', 'luty', 'marzec'),
    'liczba dni': (31, 28, 31)
}

df = pd.DataFrame(kwartal1,
                  columns=['miesiac', 'liczba dni'])

df.to_csv('kwartal1.csv', index=False)
```

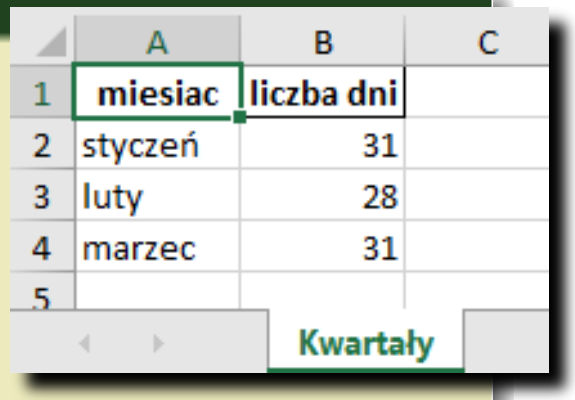
kwartal1.csv

```
1  miesiac,liczba dni
2  styczeń,31
3  luty,28
4  marzec,31
5
```

Format XLS

Serializacja do XLS

```
import pandas as pd

kwartal1 = {
    'miesiac': ('styczeń', 'luty', 'marzec'),
    'liczba dni': (31, 28, 31)
}

df = pd.DataFrame(kwartal1,
                  columns=['miesiac', 'liczba dni'])

writer = pd.ExcelWriter('kwartal1.xlsx', engine='xlsxwriter')
df.to_excel(writer, sheet_name='Kwartały', index=False)
writer.save()
```



	A	B	C
1	miesiac	liczba dni	
2	styczeń	31	
3	luty	28	
4	marzec	31	
5			

Kwartały

Format HTML

Serializacja do HTML

```
import pandas as pd

kwartal1 = {
    'miesiac': ('styczeń', 'luty', 'marzec'),
    'liczba dni': (31, 28, 31)
}


df = pd.DataFrame(kwartal1,
                  columns=['miesiac', 'liczba dni'])

df.to_html('kwartal1.html', index=False)
```

kwartal1.html

```
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th>miesiac</th>
      <th>liczba dni</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>styczeń</td>
      <td>31</td>
    </tr>
    <tr>
      <td>luty</td>
```

Format JSON

Serializacja do JSON

```
import pandas as pd

kwartal1 = {
    'miesiac': ('styczeń', 'luty', 'marzec'),
    'liczba dni': (31, 28, 31)
}

df = pd.DataFrame(kwartal1,
                  columns=['miesiac',
                           'liczba dni'])

with open('kwartal1.json', 'wt',
          encoding='utf-8') as plik:
    df.to_json(plik, index=False, orient='split',
               force_ascii=False, indent=2)
```

kwartal1.json

```
{
  "columns":[
    "miesiac",
    "liczba dni"
  ],
  "data":[
    [
      "styczeń",
      31
    ],
    [
      "luty",
```

Format JSON

Serializacja do JSON

```
import pandas as pd

kwartal1 = {
    'miesiac': ('styczeń', 'luty', 'marzec'),
    'liczba dni': (31, 28, 31)
}

df = pd.DataFrame(kwartal1,
                  columns=['miesiac',
                           'liczba dni'])

with open('kwartal1.json', 'wt',
          encoding='utf-8') as plik:
    df.to_json(plik, index=False, orient='table',
               force_ascii=False, indent=2)
```

kwartal1.json

```
{
  "schema":{
    "fields":[
      {
        "name":"miesiac",
        "type":"string"
      },
      {
        "name":"liczba dni",
        "type":"integer"
      }
    ],
    "pandas_version":"0.20.0"
  },
  "data":[
    {
      "miesiac":"styczeń",
      "liczba dni":31
    },
    {
      "miesiac":"luty",
```

Deserializacja

- W podobny sposób do przedstawionego do deserializacji danych do instancji *DataFrame* można wykorzystać funkcje:

FUNKCJA	DESERIALIZACJA DANYCH...
read_csv(filepath_or_buffer[, sep, ...])	w formacie CSV
*read_excel(*args, **kwargs)*	w pliku Excel'a
*read_html(*args, **kwargs)*	w formacie HTML
*read_json(*args, **kwargs)*	w formacie JSON
read_pickle(filepath_or_buffer[, compression])	binarnych w formacie Pickle
read_clipboard([sep])	w schowku i przekazanie ich do *read_csv*
...	...

Plan szkolenia

1 PROGRAMOWANIE FUNKCYJNE

2 PROGRAMOWANIE OOP

3 POMOCNE NARZĘDZIA

4 KOLEKCJE

5 WYRAŻENIA REGULARNE

6 PRZETWARZANIE DANYCH

7 **BAZY DANYCH**

8 WĄTKI I PROCESY

9 ASYNCHRONICZNY PYTHON

10 WSTĘP DO TESTÓW

Plan modułu

7 BAZY DANYCH

- DB API 2.0
- przegląd popularnych “connectorów” dla RDBMS
- obsługa zapytań
- połączenie z bazami nierelacyjnymi
- ORM w Pythonie

Systemy zarządzania relacyjnymi bazami danych

- **Systemy zarządzania relacyjnymi bazami danych** (*RDBMS – Relational Database Management System*), jak np. PostgreSQL lub Oracle, oferują zaawansowane podejście do przechowywania, wyszukiwania i odzyskiwania trwałych danych
- Relacyjne bazy danych wykorzystują różne dialekty SQL
- Pomimo istnienia standardów SQL, żadne dwa RDBMS nie implementują dokładnie tego samego dialektu SQL

Standard DB API 2.0

- Standardowa biblioteka Python'a nie jest wyposażona w interfejs RDBMS (wyjątkiem jest moduł *sqlite3*, który jest pełną implementcją, a nie tylko interfejsem)
- Jednak wiele modułów firm trzecich pozwala programom Python'a uzyskać dostęp do określonego RDBMS
- Takie moduły są w większości zgodne ze standardem **Python Database API v2.0**, znanym również jako *DB API v2.0* (PEP 249)

Standard DB API 2.0

BAZA DANYCH	MODUŁ PYTHONA	PROJEKT/DOKUMENTACJA
MySQL	*MySQL Connector/Python*	→ link
	mysqlclient	→ link
	pymysql	→ link
PostgreSQL	*psycopg2*	→ link
	PyGreSQL	→ link
	py-postgresql	→ link
	pg8000	→ link
Oracle	*cx_Oracle*	→ link

Nawiązanie połączenia

- Po zaimportowaniu dowolnego modułu zgodnego z DB API należy wywołać funkcję *connect* z parametrami specyficznymi dla bazy danych

Nawiązanie połączenia

```
connect(parametry...)
```

- Opcjonalne parametry:

database	nazwa bazy danych do której się podłączamy
dsn	nazwa używanego źródła danych
host	nazwa maszyny na której działa baza danych
user	nazwa użytkownika używanego do połączenia
password	hasło używane do połączenia

Nawiązanie połączenia – przykłady

Połączenie do MySQL

```
# zainstalowany moduł mysql-connector-python
import mysql.connector
conn = mysql.connector.connect(host='localhost',
                               user='root',
                               password='admin',
                               database='magazyn'
                               charset='utf8mb4')
```

Połączenie do PostgreSQL

```
import psycopg2

conn = psycopg2.connect(host="localhost",
                        user="postgres",
                        password="admin",
                        port="5432",
                        database="magazyn")
```

Operacje na obiekcie połączenia

- Funkcja *connect* zwraca instancję *Connection*, która reprezentuje połączenie z bazą danych
- Instancja połączenia dostarcza metod:

METODA	DZIAŁANIE
close()	zamyka połączenie
commit()	zatwierdza oczekujące transakcje
rollback()	wycofuje zmiany dokonane w ramach transakcji do punktu początkowego zamknięcie połączenia bez zatwierdzenia zmian spowoduje niejawnie wycofanie zmian
cursor()	zwraca obiekt kursora (instancję klasy *Cursor*) obiekt reprezentuje kursor bazy danych, który jest używany do zarządzania kontekstem operacji pobierania

Operacje na kursorze

- Kursor dostarcza metody i atrybuty używane do operacji na bazie danych

ATRYBUT	ZNACZENIE
description	sekwencja zawierająca 7-elementowe krotki każda krotka zawiera informacje opisujące jedną kolumnę wynikową (*name*, *type_code*, *display_size*, *internal_size*, *precision*, *scale*, *null_ok*)
rowcount	liczba wierszy, które zostały zwrócone (operacje DQL, jak np. *select*) lub zmodyfikowane/utworzone (operacje DML, jak np. *update*, czy *insert*)

Operacje na kursorze

Metody kursora

```
callproc(procname [, parameters])
close()
```

METODA	DZIAŁANIE
callproc	wywołuje procedurę składowaną wyniki są zwracane poprzez zmodyfikowane kopie parametrów wejściowych
close	powoduje zamknięcie kursora

Operacje na kursorze

Metody kursora

```
execute(operation [, parameters])
executemany(operation, seq_of_parameters)
```

METODA	DZIAŁANIE
execute	przygotowuje i wykonuje operacje na bazie (kwerendę lub polecenie) aby wstawić wiele wierszy można jako parametry przekazać listy krotek lub użyć polecenia *executemany()*
executemany	przygotowuje operację (kwerendę lub polecenia) i ją wykonuje z podanymi sekwencjami parametrów lub mapowaniami podobne działanie do wielokrotnych wywołań *execute()*

Operacje na kursorze

Metody kursora

```
fetchone()
fetchmany([size=curs.arraysize])
fetchall()
```

METODA	DZIAŁANIE
fetchone	pobiera następny wiersz zestawu wyników zapytania, zwracając pojedynczą sekwencję lub *None*, gdy nie ma już dostępnych danych
fetchmany	pobiera następny zestaw wierszy wyniku zapytania, zwracając sekwencję sekwencji (np. listę krotek) gdy nie ma już dostępnych wierszy zwracana jest pusta sekwencja
fetchall	pobiera wszystkie (lub pozostałe) wiersze wyniku zapytania, zwracając je jako sekwencję sekwencji (np. listę krotek)

Styl parametrów

- Moduł zgodny z DB API posiada atrybut *paramstyle*, który określa styl znaczników używanych jako symbole zastępcze parametrów

Atrybut *paramstyle*

```
select = 'SELECT * FROM TABELA WHERE '

c.execute(select + 'KOL=%s', (wart,))                    # format
c.execute(select + 'KOL=:param', {'param': wart})        # named
c.execute(select + 'KOL=:1', (wart,))                    # numeric
c.execute(select + 'KOL=%(param)s', {'param': wart})     # pyformat
c.execute(select + 'KOL=?', (wart,))                     # qmark
```

- W ten sposób można tworzyć szablony zapytań

Przykład

Przykład

```
from sqlite3 import connect

# podłączenie się do istniejącej bazy danych
conn = connect(r'C:\db\osoby.db')

# utworzenie tabeli
cursor = conn.cursor()
cursor.execute('create table if not exists osoba '
               '(imie, czy_mezczyzna, wiek)')
cursor.close()

# wstawianie nowych rekordów danych
prefix = 'insert into osoba values '
cursor = conn.cursor()
cursor.execute(prefix + "('Adam', 1, 30)")
cursor.execute(prefix + "('Anna', 0, 25)")
cursor.execute(prefix + "('Robert', 1, 19)")
conn.commit()
cursor.close()
```

Przykład cd.

Przykład

```
# wykonanie kwerendy
cursor = conn.cursor()
cursor.execute("select * from osoba where czy_mezczyzna = 1")

# odebranie i wyświetlenie danych
for (imie, plec, wiek) in cursor.fetchall():
    print(f'{imie}, '
          f'{"mężczyzna" if plec == 1 else "kobieta"}, '
          f'wiek: {wiek}')
cursor.close()

# usunięcie tabeli
cursor = conn.cursor()
cursor.execute('drop table osoba')
cursor.close()

conn.close()
```

Zagrożenia

- Uwaga: Należy **bezwzględnie walidować** zmienne generujące zapytania SQL, aby się ustrzec przed atakami typu *SQL Injection*

Przykładowe zapytanie

```
query = "SELECT * FROM salary WHERE name='{}'"
curs.execute(query.format(name))
```

- Przy poprawnej wartości parametru otrzymamy prawidłowy SQL:

Prawidłowe działanie

```
# dla parametru:
name = 'Jan Nowak'

# treść zapytania:
SELECT * FROM salary WHERE name='Jan Nowak';
```

Zagrożenia – ataki *SQL Injection*

Mamy problem...

```
# dla parametru:
name = "' or 1=1; SELECT * FROM passwords; --"

# treść zapytania:
SELECT * FROM salary WHERE name='' or 1=1;
SELECT * FROM passwords; --'
```

Mamy DUŻY problem...

```
# dla parametru:
name = "' or 1=1; DROP TABLE passwords; --"

# treść zapytania:
SELECT * FROM salary WHERE name='' or 1=1;
DROP TABLE passwords; --'
```

Zagrożenia

- Aby ustrzec się ataków typu *SQL Injection* nie należy parametrów ujmować w cudzysłowy lub apostrofy – lepiej ująć je w nawiasy:

Poprawne maskowanie

```
query = "SELECT * FROM salary WHERE name=?"
cur.execute(query, (name,))
```

Typy parametrów

- Parametry przekazywane do bazy danych za pomocą symboli zastępczych muszą zazwyczaj być właściwego typu: liczbowego, tekstowego lub *None* (aby reprezentować SQL NULL)
- Nie istnieje typ powszechnie używany do reprezentowania dat, czasu i dużych obiektów binarnych (BLOB)

Typy parametrów

- Moduł zgodny z DB API dostarcza funkcji fabryki do budowy takich obiektów

Metody fabryki

```
Binary(string)
Date(year, month, day)
DateFromTicks(s)
```

Binary	zwraca obiekt reprezentujący podany łańcuch bajtów jako BLOB
Date	zwraca obiekt reprezentujący podaną datę
DateFromTicks	zwraca obiekt reprezentujący datę po upływie s sekund od początku epoki (wg modułu *time*)

Typy parametrów

Metody fabryki

```
Time(hour, minute, second)
TimeFromTicks(s)
Timestamp(year, month, day, hour, minute, second)
TimestampFromTicks(s)
```

Time	zwraca obiekt reprezentujący podany czas
TimeFromTicks	zwraca obiekt reprezentujący czas po upływie *s* sekund od początku epoki (wg modułu *time*)
Timestamp	zwraca obiekt reprezentujący podaną datę i czas
TimestampFromTicks	zwraca obiekt reprezentujący datę i czas po upływie *s* sekund od początku epoki (wg modułu *time*)

Standard DB API 2.0

- Poza relacyjnymi bazami danych istnieje wiele baz NoSQL
- Wśród nich można wskazać m.in.:
  - obiektowe bazy danych, takie jak: ZODB, Dobbin
  - bazy dokumentów, np.: MongoDB

Moduł *pymongo*

- Wsparcia dla komunikacji z bazą MongoDB z poziomu Pythona dostarcza moduł *pymongo*
- Baza MongoDB to baza dokumentów
- Pełna dokumentacja modułu jest dostępna pod tym linkiem

Moduł *pymongo*

- Kolekcję dokumentów reprezentuje klasa *Collection*
- Klasa udostępnia wiele metod, w tym metody zliczające dokumenty w kolekcji, umożliwiające zmianę nazwy kolekcji oraz jej usunięcie

Kolekcja dokumentów

```
count_documents(filter, session=None, **kwargs)

rename(new_name, session=None, **kwargs)

drop(session=None)
```

Moduł *pymongo*

Kolekcja dokumentów

```
from pymongo import MongoClient

# utworzenie klienta MongoDB
mongo = MongoClient('mongodb://localhost:27017')

# wybór/utworzenie bazy danych
baza = mongo['magazyn']     # baza = mongo.magazyn

# wybór/utworzenie kolekcji dokumentów
klienci = baza['klienci']  # klienci = baza.klienci

# lista kolekcji dokumentów
kolekcje = baza.list_collection_names()

print('kolekcje =', kolekcje)
-> kolekcje = ['klienci']
```

Moduł *pymongo*

Metody zmieniające zawartość kolekcji dokumentów

```
insert_one(document, bypass_document_validation=False,
           session=None)
insert_many(documents, ordered=True,
            bypass_document_validation=False, session=None)

replace_one(filter, replacement, upsert=False,
            bypass_document_validation=False, collation=None,
            hint=None, session=None)

update_one(filter, update, upsert=False,
           bypass_document_validation=False, collation=None,
           array_filters=None, hint=None, session=None)
update_many(filter, update, upsert=False, array_filters=None,
            bypass_document_validation=False, collation=None,
            hint=None, session=None)

delete_one(filter, collation=None, hint=None, session=None)
delete_many(filter, collation=None, hint=None, session=None)
```

Moduł *pymongo*

Metody wyszukujące dokumenty

```
find(filter=None, projection=None, skip=0, limit=0, no_cursor_timeout=False,
     cursor_type=CursorType.NON_TAILABLE, sort=None, allow_partial_results=False,
     oplog_replay=False, modifiers=None, batch_size=0, manipulate=True,
     collation=None, hint=None, max_scan=None, max_time_ms=None, max=None, min=None,
     return_key=False, show_record_id=False, snapshot=False, comment=None,
     session=None)

find_one(filter=None, *args, **kwargs)

find_one_and_delete(filter, projection=None, sort=None, hint=None, session=None,
                    **kwargs)

find_one_and_replace(filter, replacement, projection=None, sort=None,
                     return_document=ReturnDocument.BEFORE, hint=None, session=None,
                     **kwargs)

find_one_and_update(filter, update, projection=None, sort=None,
                        return_document=ReturnDocument.BEFORE, array_filters=None,
                        hint=None, session=None, **kwargs)
```

Moduł *pymongo*

Przykład

```
klient = {'imie': 'Jan', 'nazwisko': 'Kowalski'}

# wstawienie dokumentu i odebranie jego id
id = klienci.insert_one(klient).inserted_id
print('id =', id)
-> id = 5fb812ef61e7c1ea241ad261

# wyszukanie dokumentu po id
klient = klienci.find_one({'_id': id})
print(klient)
-> klient = {'_id': ObjectId('5fb812ef61e7c1ea241ad261'),
->            'imie': 'Jan', 'nazwisko': 'Kowalski'}

# wyszukanie dokumentu wg kryteriów
klient = klienci.find_one({'nazwisko': 'Kowalski'})
print(klient)
-> klient = {'_id': ObjectId('5fb7f520bb58bdc4149bf92e'),
->            'imie': 'Jan', 'nazwisko': 'Kowalski'}
```

ORM

- **ORM** (*Object-Relational Mapping*) – to narzędzie mapowania obiektowo-relacyjnego
- Umożliwia komunikację z relacyjną bazą danych poprzez odwzorowanie modelu obiektowego, czyli klas encyjnych (model abstrakcyjny) na tabele w bazie danych (model fizyczny)
- W takim przypadku tabele są tworzone automatycznie przez usługę, podobnie jak zapytania SQL
- Istnieje kilka implementacji ORM dla Pythona, wśród nich:

MODUŁ	DOKUMENTACJA
peewee	→ link
SQLAlchemy	→ link

Biblioteka *SQLAlchemy*

- SQLAlchemy umożliwia stworzenie abstrakcji i uniezależnienie się od bazy danych i dialektu SQL który ona wykorzystuje
- Najpierw należy utworzyć silnik:

Utworzenie silnika

```
create_engine(*args, **kwargs)
```

- Funkcja wykorzystuje URL-a do bazy danych o postaci:
  dialect+driver://username:password@host:port/database[?key=value,...]
  gdzie:

dialect	np. *sqlite, mysql, postgresql, oracle, mssql*
driver	nazwa DB API, np.: *psycopg2, pyodbc, cx_oracle*

Biblioteka *SQLAlchemy*

- Opcjonalne parametry nazwane, to m.in.:

echo=False	logowanie poleceń na standardowe wyjście
encoding='utf8'	domyślne kodowanie znaków
convert_unicode=False	domyślne konwertowanie łańcuchów w bazie na Unicode
execution_options	np. *autocommit*
listeners	listenery na zdarzenia związane z pulą połączeń

Przykład

```
from sqlalchemy import create_engine

silnik = create_engine('mysql+pymysql://root:admin@localhost/'
                       'magazyn?charset=utf8mb4', echo=True)
```

Biblioteka *SQLAlchemy*

- Silnik posiada wiele metod:

Wybrane metody silnika

```
begin(close_with_result=False)
connect(**kwargs)
execute(statement, *multiparams, **params)
table_names(schema=None, connection=None)
transaction(callable_, *args, **kwargs)
```

begin	zwraca kontekst
connect	zwraca połączenie
execute	wykonuje polecenie
table_names	zwraca dostępne tabele w bazie
transaction	opakowuje funkcję w transakcję

Biblioteka *SQLAlchemy*

- Aby dokonać mapowania klas encyjnych na bazę danych potrzebujemy bazy deklaratywnej
- Będzie ona stanowiła nadklasę klas encyjnych
- Dalej pozostaje wygenerowanie schematu bazy danych (utworzenie tabel) i utworzenie instancji encji

Biblioteka *SQLAlchemy*

Klasa encyjna (moduł *encje*)

```
from sqlalchemy.ext.declarative import *

Baza = declarative_base()

# klasa encyjna
class Klient(Baza):
    __tablename__ = 'klienci'
    id = Column(Integer, primary_key=True)
    imie = Column(String(20))
    nazwisko = Column(String(30))
    wiek = Column(Integer)
    mezczyzna = Column(Boolean, default=True)

    def __init__(self, imie,nazwisko, wiek, mezczyzna=True):
        self.imie = imie
        self.nazwisko = nazwisko
        self.wiek= wiek
        self.mezczyzna = mezczyzna

    def __str__(self):
        return (f'{self.imie} {self.nazwisko}, '
                f'{"mężczyzna" if self.mezczyzna else "kobieta"}, '
                f'wiek: {self.wiek}')
```

Biblioteka *SQLAlchemy*

Instancje encji

```
from sqlalchemy import create_engine
from encje import Baza, Klient

silnik = create_engine('mysql+pymysql://root:admin@localhost/'
                       'magazyn?charset=utf8mb4', echo=False)

# utworzenie schematu bazy danych
Baza.metadata.create_all(silnik)

klient1 = Klient('Jan', 'Kowalski', 30)
klient2 = Klient('Anna', 'Nowakowska', 25, False)

print(klient1)
-> Jan Kowalski, mężczyzna, wiek: 30

print(klient2)
-> Anna Nowakowska, kobieta, wiek: 25
```

Biblioteka *SQLAlchemy*

- Wszystkie operacje na bazie danych są wykonywane poprzez obiekt sesji – reprezentuje go klasa *Session*

Tworzenie sesji

```
from sqlalchemy.orm import sessionmaker

Sesja = sessionmaker(bind = silnik)
sesja = Sesja()
```

Biblioteka *SQLAlchemy*

- Wybrane atrybuty funkcji *sessionmaker*:

bind	powiązanie sesji z konkretnym połączeniem
autoflush	natychmiastowe zmiany (ale nie zatwierdzone) zatwierdzenie na końcu po wywołaniu *commit*
autocommit	natychmiastowe zatwierdzanie zmian w odrębnych chwilowych transakcjach
expire_on_commit	wygasza sesję po zatwierdzeniu zmian

- Zanim stan obiektu zostanie utrwalony w bazie, musi być dodany do sesji

Biblioteka *SQLAlchemy*

- Wybrane metody sesji:

add	dodaje obiekt do sesji
add_all	dodaje kolekcję obiektów do sesji
begin	początek transakcji
begin_nested	początek transakcji zagnieżdżonej
commit	zatwierdzenie zmian
rollback	wycofanie się z zapisanych zmian
dirty	obiekty zmienione
new	obiekty nowe
execute	wykonanie polecenia
scalar	jak *execute*, ale z liczbą w wyniku
flush	zapis zmian, ale jeszcze nie zatwierdzony
query	zapytanie

Biblioteka *SQLAlchemy*

- Obiekt encyjny może znajdować się w jednym z poniższych stanów:

transient	instancja nie znajduje się w sesji i nie jest zapisana w bazie
pending	po wywołaniu metody *add*, jest w sesji, ale nie jest zapisana w bazie aż do wywołania metody *flush*
persistent	instancja jest w sesji i w bazie
detached	jest w bazie, ale nie ma jej w żadnej sesji

Biblioteka *SQLAlchemy*

Przykład

```
sesja.add_all([klient1, klient2])
sesja.commit()

klient = sesja.query(Klient) \
              .filter_by(nazwisko='Nowakowska') \
              .first()

print(f'{klient.id}: {klient}')
-> 2: Anna Nowakowska, kobieta, wiek: 25
```

Plan szkolenia

1 PROGRAMOWANIE FUNKCYJNE

2 PROGRAMOWANIE OOP

3 POMOCNE NARZĘDZIA

4 KOLEKCJE

5 WYRAŻENIA REGULARNE

6 PRZETWARZANIE DANYCH

7 BAZY DANYCH

8 **WĄTKI I PROCESY**

9 ASYNCHRONICZNY PYTHON

10 WSTĘP DO TESTÓW

Plan modułu

8 WĄTKI I PROCESY

- moduł *threading*
- moduł *multiprocessing*

Wątki

- **Wątek** (*thread*) – przepływ sterowania w programie, który współdzieli stan globalny (pamięć) z innymi wątkami
- Wątki działające na pojedynczym procesorze/rdzeniu tworzą iluzję, że są wykonywane jednocześnie, chociaż zwykle działają z pewnym przeplotem
- Akcja jest nazywana **atomową**, jeśli gwarantuje się, że między początkiem, a końcem akcji nie nastąpi przełączanie wątków

Wątki w Pythonie

- Python oferuje wielowątkowość w dwóch wersjach:
  - starszy moduł *_thread* oferuje niskopoziomową funkcjonalność i nie jest zalecany do bezpośredniego użycia
  - nowszy moduł *threading* – jest modułem wyższego poziomu, zbudowanym na bazie modułu *_thread*
- Zaleca się użycie modułu *threading*
- Jego pełna dokumentacja jest dostępna pod tym linkiem

Moduł *threading*

- Moduł *threading* definiuje kilka użytecznych funkcji:

Funkcje modułu *threading*

```
active_count()
current_thread()
enumerate()
stack_size([size])
```

FUNKCJA	DZIAŁANIE
active_count	liczba wątków aktywnych
current_thread	zwraca instancję bieżącego wątka
enumerate	lista wszystkich “żywych” wątków
stack_size	rozmiar stosu w bajtach wykorzystywany przez nowe wątki ustawienie *size* na 0 oznacza wybór wartości domyślnej podanie wartości nieakceptowanej przez system, powoduje zgłoszenie wyjątku *ValueError*

Uruchamianie wątków

- Instancje klasy *Thread* reprezentują wątki

Tworzenie instancji wątków

```
Thread(name=None, target=None, args=(), kwargs={})
```

- Zaleca się przekazywanie danych w formie argumentów nazwanych
- Zadanie realizowane przez wątek można podać na dwa sposoby:
  - jako funkcję przekazaną jako wartość parametru *target*
  - poprzez rozszerzenie klasy *Thread* i nadpisanie metody *run*
- Wątek może rozpocząć działanie dopiero po wywołaniu metody *start* na jego instancji

Uruchamianie wątków

Tworzenie i startowanie wątków – sposób 1

```
from threading import Thread, current_thread

def zadanie():
    nazwa_watka = current_thread().name
    for nr in range(1, 11):
        print(f'[{nazwa_watka}] hello #{nr:02d}')

w1 = Thread(target=zadanie, name='wątek1')
w2 = Thread(target=zadanie, name='wątek2')
w1.start()
w2.start()
```

Uruchamianie wątków

Tworzenie i startowanie wątków – sposób 2

```
from threading import Thread

class Watek(Thread):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def run(self):
        nazwa_watka = self.name
        for nr in range(1, 11):
            print(f'[{nazwa_watka}] hello #{nr:02d}')


w1 = Watek(name='wątek1')
w2 = Watek(name='wątek2')
w1.start()
w2.start()
```

Uruchamianie wątków

- Nie można zakładać żadnej kolejności w jakiej wątki będą wykonywały swoje zadania – nie ma znaczenia kolejność startowania wątków
- Zawartość klasy *Thread*:

WŁAŚCIWOŚĆ	ZNACZENIE
daemon	właściwość określająca, czy wątek jest demoniczny (proces może się zakończyć, nawet, gdy wątki demoniczne są żywe – to kończy działanie wątków)
name	właściwość definiująca nazwę wątka

Uruchamianie wątków

- Klasa *Thread* definiuje także metody:

METODA	ZNACZENIE
is_alive	metoda testuje, czy wątek jest jeszcze “żywy” (został wystartowany, ale nie zakończył działania)
join	wstrzymuje działanie wątka wołającego, do momentu zakończenia działania wątka na którym metoda została wywołana
run	metoda definiująca zadanie wątka standardowo wywołuje metodę podaną przez parametr *target* nie należy metody wywoływać samodzielnie
start	powoduje, że wątek staje się aktywny i umożliwia wykonanie metody *run*

Synchronizacja wątków

- **Synchronizacja wątków** – mechanizm, który zapewnia, że dwa lub więcej współbieżnych wątków nie wykonuje jednocześnie określonego segmentu programu zwanego **sekcją krytyczną**
- Sekcja krytyczna odnosi się do części programu, w których uzyskuje się dostęp do **współdzielonego zasobu**
- Dostęp do sekcji krytycznej powinien odbywać się ze **wzajemnym wykluczeniem**

Synchronizacja wątków

- Jednoczesny dostęp do współdzielonych zasobów może prowadzić do zjawiska **wyścigu** (*race condition*)
- Ma to miejsce, gdy dwa lub więcej wątków może uzyskać dostęp do współdzielonych danych i próbują je zmienić w tym samym czasie
- W rezultacie wartości zmiennych mogą być nieprzewidywalne i różnić się w zależności od czasów przełączania kontekstu procesów

Synchronizacja wątków

Zjawisko wyścigu

```
from threading import Thread

class Licznik:
    def __init__(self):
        self.stan = 0

    def zwieksz(self):
        for _ in range(1_000_000):
            self.stan += 1

    def zmniejsz(self):
        for _ in range(1_000_000):
            self.stan -= 1

...
```

Synchronizacja wątków

Zjawisko wyścigu cd.

```
licznik = Licznik()
print(licznik.stan)  # 0

w1 = Thread(target=licznik.zwieksz)
w2 = Thread(target=licznik.zmniejsz)

w1.start()
w2.start()

# czekamy na zakończenie pracy wątków
w1.join()
w2.join()

print(licznik.stan)  # może być wartość inna niż 0
```

Blokady

- Moduł *threading* dostarcza kilku mechanizmów synchronizacji, umożliwiających wątkom komunikację i koordynację działania

Metody blokady *Lock*

```
acquire(blocking=True, timeout=-1)
release()
locked()
```

acquire	założenie i wejście w posiadanie blokady operacja może być blokująca lub nie
release	zwolnienie blokady metodę może wywołać wątek niebędący właścicielem blokady
locked	sprawdzenie, czy blokada jest założona

Blokady

- Obiekty blokady *RLock* posiadają identyczny zestaw metod jak blokada *Lock*
- Blokada *RLock* to **blokada wielowejściowa** (*re-entrant lock*), w której zaimplementowany jest mechanizm własności
- Tylko ten wątek, który założył blokadę może ją zwolnić
- Wątek posiadający blokadę może wielokrotnie wywołać metodę *acquire*, bez konieczności blokowania
- Blokada zostaje zwolniona, gdy metoda *release* zostanie wywołana tyle samo razy, co *acquire*
- Blokady implementują **protokół menedżera kontekstu**

Synchronizacja wątków

Użycie blokady do synchronizacji

```
from threading import Thread, Lock

class Licznik:
    def __init__(self):
        self.stan = 0

    def zwieksz(self, blokada):
        for _ in range(1_000_000):
            with blokada:          # blokada.acquire()
                self.stan += 1     # self.stan += 1
                                   # blokada.release()

    def zmniejsz(self, blokada):
        for _ in range(1_000_000):
            with blokada:
                self.stan -= 1

...
```

Synchronizacja wątków

Użycie blokady do synchronizacji cd.

```
licznik = Licznik()
print(licznik.stan)  # 0

blokada = Lock()

w1 = Thread(target=licznik.zwieksz, args=(blokada,))
w2 = Thread(target=licznik.zmniejsz, args=(blokada,))

w1.start()
w2.start()

# czekamy na zakończenie pracy wątków
w1.join()
w2.join()

print(licznik.stan)  # 0
```

Semafory

- **Semafory** (*semaphores*) to uogólnienie blokad
- Stanem blokady jest wartość logiczna (`True` lub `False`)
- Stanem semafora jest licznik (wartość pomiędzy 0, a podaną liczbą “przepustek”)
- Semafory mogą być użyteczne przy implementacji puli zasobów o określonym rozmiarze (można także do tego celu użyć kolejki *Queue*)
- Semafor reprezentują klasy *Semaphore* oraz *BoundedSemaphore*

Semafory

Metody semaforów

```
acquire(blocking=True)
release()
```

acquire	jeśli stan licznika jest dodatni, licznik jest dekrementowany i zwracana jest wartość `True` jeśli stan licznika ma wartość 0, a argument *blocking* – `True`, to wątek jest blokowany do momentu, aż inny wątek wywoła metodę *release* jeśli stan licznika ma wartość 0, a argument *blocking* – `False`, metoda od razu zwraca wartość `False`
release	jeśli stan licznika jest dodatni, lub licznik ma wartość 0 i nie ma żadnych wątków wstrzymanych, to licznik jest inkrementowany jeśli licznik semafora ma wartość 0 i są wątki oczekujące, to stan licznika nie ulega zmianie i jest wznawiany jeden z wątków wstrzymanych

Semafory

- Podczas tworzenia instancji semafora można zainicjować stan licznika (domyślnie licznik ma wartość 1)
- Jeśli licznik semafora *BoundedSemaphore* przekroczy wartość początkową, to zostanie zgłoszony wyjątek *ValueError*

Wielowątkowość w Pythonie – czy warto?

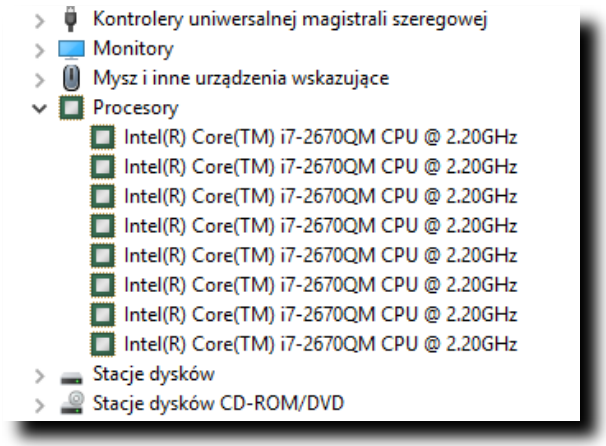
- Podstawowym pytaniem przed którym stajemy jest:
  Czy i kiedy warto wykorzystywać wielowątkowość?

- Spróbujmy przeprowadzić następujący eksperyment...
  - naszym zadaniem jest znalezienie wartości maksymalnej w dużej liście (kilka milionów elementów) zawierającej losowe wartości liczbowe
  - zadanie zostanie zrealizowane na dwa sposoby:
    - w “klasyczny sposób” wykorzystując iterację po całej liście
    - z użyciem dwóch wątków do których delegujemy zadanie znalezienia maksimum w połówce listy, następnie sprawdzimy, która wartość jest większa

Wielowątkowość w Pythonie – czy warto?

- W obu wariantach zmierzymy czas wykonania operacji (w drugim przypadku uwzględniamy także czas potrzebny na utworzenie wątków, ich zadań, wystartowanie, zaczekanie na wynik i wybór większej wartości)

- Aplikacja zostanie uruchomiona na maszynie 4-procesorowej (każdy procesor ma 2 rdzenie)
- Aplikacja działa na referencyjnej implementacji Pythona (*CPython*)
- Czego można się spodziewać?



Wielowątkowość w Pythonie – czy warto?

- Na pierwszy rzut oka wyniki mogą być zaskakujące. . .
- Wariant w którym zadanie zostało zdekomponowane na dwa niezależne podzadania jest znacząco wolniejszy
- Jest to spowodowane realizacją wielowątkowości w wybranych implementacjach Pythona, np. *CPython* (implementacja w C), *PyPy* (implementacja w Pythonie)
- Problem nie dotyczy takich implementacji Pythona jak *Jython* (implementacja w Javie), czy *IronPython* (implementacja w .NET)

Global Interpreter Lock

- “Winowajcą” jest tzw. **GIL** (*Global Interpreter Lock*)
- GIL jest muteksem, który pozwala tylko jednemu wątkowi na kontrolowanie interpretera Pythona
- W konsekwencji wątki mogą co prawda działać na różnych procesorach, ale w danej chwili będzie działać tylko jeden
- GIL ogranicza programowanie równoległe w Pythonie, nawet w architekturze wielowątkowej z więcej niż jednym rdzeniem procesora

Global Interpreter Lock

- Co ustaloną liczbę instrukcji kodu bajtowego GIL jest zwalniany, co pozwala działać wątkom pracującym poza interpreterem, czyli nieodwołującym się do API Pythona
- GIL jest zwalniany także w przypadku blokujących operacji wejścia/wyjścia, czyli np. przy odczycie czy zapisie do pliku
- Wątki w Pythonie nadają się do programów mocno obarczonych operacjami I/O
- Brak zysku przy operacjach czasochłonnych
- Potrzeba użycia rozwiązań opartych o system operacyjny

Procesy

- **Proces** to instancja uruchomionego programu
- Pakiet *multiprocessing* to pakiet, który umożliwia tworzenie i zarządzanie procesami przy użyciu interfejsu API podobnego do modułu *threading*
- Pakiet oferuje współbieżność lokalną i zdalną, przy użyciu podprocesów zamiast wątków
- W ten sposób można ominąć problemy związane z działaniem *Global Interpreter Lock* i w pełni wykorzystać wiele procesorów na danej maszynie
- Działa na systemach Unix i Windows
- Pełna dokumentacja tego pakietu jest dostępna pod tym linkiem

Procesy

- Procesy reprezentuje klasa *Process*
- Po utworzeniu instancji i przekazaniu zadania, proces należy wystartować (podobnie jak wątki)
- Należy zadbać o to, aby kod modułu głównego mógł być zaimportowany przez interpreter Pythona bez niezamierzonych efektów ubocznych (np. wystartowania nowego procesu)

Procesy

Tworzenie i startowanie procesów

```
from multiprocessing import Process, current_process
import os

def info():
    print('{:12} [id: {:5d}, rodzic: {:5d}]'.format(
        current_process().name,
        os.getpid(),
        os.getppid()))

if __name__ == "__main__":
    info()
    for i in range(4):
        Process(target=info).start()
```

```
MainProcess  [id:   824, rodzic:  8788]
Process-1    [id:  2856, rodzic:   824]
Process-2    [id:  2520, rodzic:   824]
Process-3    [id:  8880, rodzic:   824]
Process-4    [id:  8696, rodzic:   824]
```

Kolejki

- System operacyjny **izoluje** procesy między sobą
- Efektywne wykorzystanie wielu procesów zwykle wymaga pewnej **komunikacji** między nimi, tak aby można było równoważyć obciążenie i agregować wyniki
- Prostym sposobem komunikacji między procesami jest użycie **kolejki** do przekazywania komunikatów w obie strony
- Każdy obiekt serializowalny (z użyciem *pickle*) można przesłać przez kolejkę
- W ten sposób można uniknąć synchronizacji (np. z użyciem blokad)

Kolejki

- Kolejkę reprezentuje klasa *Queue*
- Zestaw metod jest podobny do metod klasy *Queue* z modułu *queue*
- Oprócz klasy *Queue* można także użyć klasy:

SimpleQueue	uproszczonej wersji kolejki
JoinableQueue	podklasy *Queue* oferującej dodatkowo metody: *task_done* oraz *join*

Kolejki

Przykład użycia kolejki

```
from multiprocessing import Process, Queue, current_process
import time

class Faks:
    def __init__(self, tresc):
        self.tresc = tresc

    def __str__(self):
        return self.tresc


def zadanie_serwera(kolejka):
    while True:
        dokument = kolejka.get()
        if not dokument:
            break
        time.sleep(1)
        print('Wysyłam faks', dokument)
```

Kolejki

Przykład użycia kolejki cd.

```
if __name__ == '__main__':
    kolejka_faksow = Queue()

    proces_serwera = Process(target=zadanie_serwera,
                             args=(kolejka_faksow,))
    proces_serwera.start()

    for i in range(3):
        faks = Faks('#{:d} od {}'.format(i,
                                         current_process().name))
        kolejka_faksow.put(faks)
    kolejka_faksow.put(None)

    kolejka_faksow.close()
    proces_serwera.join()
```

Blokady

- Innym sposobem synchronizacji procesów jest użycie blokad
- Moduł *multiprocessing* definiuje klasy będące odpowiednikami klas z modułu *threading*:
  - *BoundedSemaphore*
  - *Lock*
  - *RLock*
  - *Semaphore*

Pule

- W prawdziwym życiu trzeba uważać na tworzenie nieograniczonej liczby procesów roboczych
- Korzystanie z wielu procesów może przynieść korzyści w wydajności tylko wtedy, gdy liczba procesów jest równa lub bliska liczbie rdzeni w komputerze (p. metoda *cpu_count*)
- Wykonywanie większej liczby procesów roboczych niż ta optymalna wiąże się ze znacznymi dodatkowymi kosztami
- W konsekwencji typowym wzorcem projektowym jest **tworzenie puli z ograniczoną liczbą procesów roboczych** i przydzielanie im pracy
- Klasa *Pool* umożliwia realizację tego wzorca

Pule

Tworzenie instancji puli

```
Pool(processes=None, initializer=None, initargs=(),
     maxtasksperchild=None)
```

processes	liczba procesów w puli
initializer	opcjonalna funkcja wywoływana przy starcie każdego nowego procesu
initargs	argumenty przekazywane do funkcji initializującej procesy
maxtasksperchild	maksymalna liczba zadań wykonywanych przez każdy proces puli

- Instancje puli oferują szereg metod
- Mogą być one wywołane tylko przez ten proces w którym została utworzona pula

Pule

Metody puli

```
apply(func, args=(), kwds={})
apply_async(func, args=(), kwds={}, callback=None)
```

apply
: w dowolnym z procesów roboczych wywołuje funkcję z podanymi argumentami, w sposób synchroniczny i zwraca wynik

apply_async
: w dowolnym z procesów roboczych wywołuje funkcję z podanymi argumentami, w sposób asynchroniczny i nie czekając na wynik zwraca instancję *AsyncResult*

  jeśli podano funkcję *callback*, to przekazywany jest jej wynik, gdy jest on gotowy

  funkcja nie powinna być czasochłonna, gdyż może zablokować proces

Pule

Metody puli

```
close()
imap(func, iterable, chunksize=1)
imap_unordered(func, iterable, chunksize=1)
```

close	nie można przesłać więcej zadań do puli procesy robocze kończą się, gdy zakończą wszystkie zaległe zadania
imap	zwraca iterator po wynikach wywołania podanej funkcji na kolejnych elementach obiektu iterowalnego *chunksize* określa, ile kolejnych elementów jest wysyłanych do każdego procesu
imap_unordered	podobnie do *imap*, ale kolejność nie jest ustalona

Pule

Metody puli

```
join()
map(func, iterable, chunksize=1)
map_async(func, iterable, chunksize=1, callback=None)
terminate()
```

join	czeka na zakończenie wszystkich procesów wcześniej należy wywołać *close* lub *terminate*
map	działa podobnie do *imap* ale zwraca listę wyników, a nie iterator
map_async	asynchroniczny wariant metody *imap*
terminate	kończy wszystkie procesy robocze natychmiast, bez czekania na zakończenie pracy

Pule

Przykład

```
from multiprocessing import Pool, current_process
import os
import time
import random

def jaka_dlugosc(tekst):
    time.sleep(random.random() * 2)   # symulacja czasochłonnej
                                      # operacji
    return len(tekst)

def zadanie(slowo):
    print('proces {:6}, {}'.format(os.getpid(),
                                   current_process().name))
    return jaka_dlugosc(slowo)

def dlugosci(tekst):
    with Pool() as pula:
        krotka = tuple(pula.imap(zadanie, tekst.split()))
    return krotka
```

Pule

Przykład cd.

```
if __name__ == '__main__':
    d = dlugosci('How I wish I could calculate pi')
    print('\nwynik: ', *d, sep='')
```

```
-> proces   2244, SpawnPoolWorker-1
-> proces  12416, SpawnPoolWorker-2
-> proces   7960, SpawnPoolWorker-4
-> proces  11588, SpawnPoolWorker-3
-> proces  12360, SpawnPoolWorker-5
-> proces   6664, SpawnPoolWorker-6
-> proces   6488, SpawnPoolWorker-7
->
-> wynik: 3141592
```

Procesy demoniczne

- Procesy, podobnie jak wątki, posiadają flagę *daemon*
- Wartość tej flagi (`True` lub `False`) można ustawić zanim proces zostanie wystartowany (metoda *start*)
- Kiedy zwykły (niedemoniczny) proces kończy pracę, próbuje zakończyć wszystkie swoje demoniczne procesy potomne – procesy demoniczne nie mają wpływu na całkowity czas działania aplikacji
- Proces demoniczny nie może tworzyć procesów potomnych
- W przeciwnym razie mógłby pozostawić swoje procesy potomne “osierocone”, gdyby sam został zakończony po zakończeniu swojego procesu-rodzica

Wymiana danych pomiędzy procesami

- Zazwyczaj najlepiej jest unikać udostępniania stanu między procesami – zamiast tego można użyć kolejek do jawnego przekazywania między nimi komunikatów
- Jednak w sytuacjach, w których trzeba współdzielić stan, moduł *multiprocessing* dostarcza klas dostępu do **pamięci współużytkowanej** (*shared memory*)

Wymiana danych pomiędzy procesami

Pamięć współdzielona

```
Value(typecode, *args, lock=True)
Array(typecode, size_or_initializer, lock=True)
```

Value	klasa do przechowywania pojedynczej wartości wspólnej dla dwóch lub więcej procesów
Array	klasa do przechowywania ustalonej ilości wartości prostych (tego samego typu)

- Blokadę można uzyskać poprzez wywołanie metody *get_lock*
- Do odczytu/ustawienia wartości służy atrybut *value*

Wymiana danych pomiędzy procesami

- Bardziej elastycznym rozwiązaniem, umożliwiającym m.in. koordynację między różnymi komputerami w sieci (nie współdzielącymi pamięci) jest użycie klasy *Manager*
- Jest ona podklasą klasy *Process* (z tymi samymi metodami i atrybutami)
- Instancja klasy steruje procesem serwera, który zarządza obiektami współużytkowanymi
- Inne procesy mogą uzyskiwać dostęp do udostępnionych obiektów za pośrednictwem obiektów proxy $\rightarrow$ większy narzut

Wymiana danych pomiędzy procesami

Przykład

```
from multiprocessing import Manager, Process, current_process
import os
import time
import random

def jaka_dlugosc(s):
    time.sleep(random.random() * 2)  # symulacja czasochłonnej operacji
    return len(s)

def zadanie(nr, slowo, slownik):
    print('proces {:6}, {}'.format(os.getpid(), current_process().name))
    slownik[nr] = jaka_dlugosc(slowo)

def dlugosci(tekst):
    mgr = Manager()
    slownik = mgr.dict()
    procesy = []
    for nr, s in enumerate(tekst.split()):
        p = Process(target=zadanie, args=(nr, s, slownik))
        p.start()
        procesy.append(p)
    for p in procesy:
        p.join()
    return [dlugosc for _, dlugosc in sorted(slownik.items())]
```

Wymiana danych pomiędzy procesami

Przykład cd.

```
if __name__ == '__main__':
    d = dlugosci('How I wish I could calculate pi')
    print('\nwynik: ', *d, sep='')
```

```
-> proces  12404, Process-2
-> proces  11368, Process-3
-> proces   9472, Process-4
-> proces   7748, Process-5
-> proces   8952, Process-6
-> proces  10104, Process-7
-> proces   8180, Process-8
->
-> wynik: 3141592
```

Plan szkolenia

1 PROGRAMOWANIE FUNKCYJNE

2 PROGRAMOWANIE OOP

3 POMOCNE NARZĘDZIA

4 KOLEKCJE

5 WYRAŻENIA REGULARNE

6 PRZETWARZANIE DANYCH

7 BAZY DANYCH

8 WĄTKI I PROCESY

9 **ASYNCHRONICZNY PYTHON**

10 WSTĘP DO TESTÓW

Plan modułu

9 ASYNCHRONICZNY PYTHON

- moduł ASYNCIO – podstawowe zagadnienia

Bibliotek asyncio

- Biblioteka do programowania asynchroniczego
  - opiera się na coroutines (współprogram)
  - “funkcje” współpracujące, które dobrowolnie oddają kontrolę
  - możliwe jest zawieszenie wykonywania programu i przeniesienie wykonywania do innego współprogramu
  - w przeciwieństwie do subroutines (podprogram, funkcja) które muszą być wywłaszczone
  - wykonywanie naprzemiennie (*concurrent*), ale nie równolegle (*parallel*)

Coroutines w Pythonie

- Zaimplementowane na bazie generatorów
- Dobrowolnie oddają wykonywanie
- W momencie oddania kontroli zapamiętują swój stan (patrz *yield*)
- Mogą kontynuować swój program od tego momentu (patrz *next* oraz *send*)
- Zminimalizowany problem wyścigu (*race conditions*)

async def, await, event_loop

- asyncio dostarcza nam *event_loop*, który zarządza wykonywaniem kodu asynchronicznego (coroutynami)
- Słowo kluczowe `async def` definiuje nam coroutynę (jako funkcję):
  async def coroutine(args): ...
- Wyrażenie *await* w bloku coroutyny może wstrzymać jej wykonywanie dopóki wartość argumentu jest gotowa:
  val = await awaitable

Awaitable

- *Awaitable* to podstawowy typ w *asyncio* na którego wynik można oczekiwać (w sposób asynchroniczny)
- *Coroutine*, *Future* oraz *Task* to podklasy *awaitable*
- Jest ona “awaitable" – można na niej wykonać:
  val = await awaitable
- Klasa definująca metodę *__await__*

Future

- Reprezentuje jednostkę, która jest w trakcie wykonywania
- Wynik jej działania może nie być dostępny natychmiast
- Oczekiwanie na *future* oddaje wykonywanie do *event_loop* dopóki wynik jest gotowy
- Posiadają również synchroniczny interfejs:
  - f.done()
  - f.result()
  - f.exception()

Task

- Podstawowa jednostka wykonywana przez *event_loop*
- *event_loop* może w danym momencie wykonywać tylko jeden task, pozostałe taski w tym momencie oczekują
- Jest podklasą *Future*
- Jest wykorzystywana do wykonywania corutyn
- Jest gotowy kiedy corutyna zakończyła działanie

Dodatkowe konstrukcje

- *asyncio.run(future)* powoduje wykonanie *future* (w tym tasku) w domyślnym *event_loop* do momentu, aż wynik *future* jest dostępny
  - w przypadku coroutine zostanie ona opakowana w task
- *await asyncio.sleep(0)* oddaje wykonywanie
- Dodatkowe wyrażenia:
  - *async for* dla (async) iteratorów, które zwracają *awaitables*
  - *async with* dla (async) contextmanagerów, które oczekują na zajęcie lub zwolnienie zasobu

sync in async

- Nie powinniśmy wykonywać synchronicznych operacji blokujących w kodzie asynchronicznym
- Kod blokujący wykonujemy za pomocą: *await loop.run_in_executor(blocking_func)*
- Zostanie ona wykonana w *threadpool* albo *processpool*

Plan szkolenia

1 PROGRAMOWANIE FUNKCYJNE

2 PROGRAMOWANIE OOP

3 POMOCNE NARZĘDZIA

4 KOLEKCJE

5 WYRAŻENIA REGULARNE

6 PRZETWARZANIE DANYCH

7 BAZY DANYCH

8 WĄTKI I PROCESY

9 ASYNCHRONICZNY PYTHON

10 WSTĘP DO TESTÓW

Plan modułu

10 WSTĘP DO TESTÓW

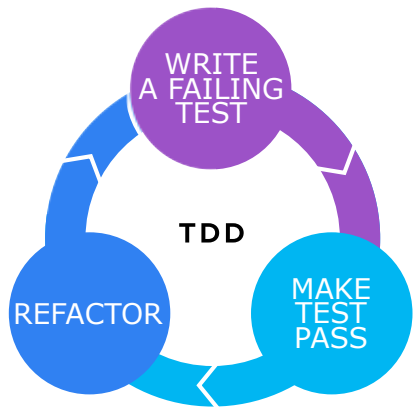
- testy jednostkowe
- wstęp do TDD
- biblioteka Unittest
- przegląd innych bibliotek

Testy jednostkowe

- Za pomocą testów jednostkowych staramy się zweryfikować funkcjonalność aplikacji na najbardziej podstawowym poziomie
- Testujemy każdą jednostkę kodu, zazwyczaj metodę, w izolacji od innych, aby sprawdzić, czy w określonych warunkach reaguje w oczekiwany sposób
- Przeniesienie testowania na ten poziom daje pewność, że każda część aplikacji będzie zachowywać się zgodnie z oczekiwaniami i umożliwia wykrycie przypadków brzegowych, w których aplikacja może działać w niestandardowy sposób i odpowiednio radzić sobie z nimi

TDD

- **Programowanie sterowane testami** (*TDD – Test-Driven Development*) to paradygmat, w ramach którego wdraża się nową funkcję lub wymaganie:
  - najpierw pisząc testy
  - obserwując, jak kończą się niepowodzeniem
  - a następnie pisząc kod, aby testy które zakończyły się niepowodzeniem – teraz przeszły



TDD

- Gdy podstawowy szkielet funkcji zostanie zaimplementowany w ten sposób, można go dalej rozbudowywać, modyfikując testy, a następnie zmieniając kod programu tak, aby uwzględnić dodaną funkcjonalność
- Ten proces powtarza się tak długo, aż nowe wymagania zostaną wypełnione i nowe funkcje zostaną dodane do istniejącego kodu
- Pisząc testy automatyczne przed kodem programu, musimy najpierw zastanowić się nad problemem
- Rozpoczynając tworzenie testów, trzeba pomyśleć o sposobie pisania kodu, który musi przejść już napisane testy automatyczne, aby został zaakceptowany

Biblioteka *unittest*

- Biblioteka *unittest* – wbudowana biblioteka służąca do automatyzacji testów jednostkowych w Pythonie, wzorowana na *JUnit*, o podobnych możliwościach jak frameworki testów jednostkowych w innych językach
- Biblioteka umożliwia:
  - automatyzację testów
  - współdzielenie kodu konfiguracji i kończenia testów
  - grupowanie testów w zestawy
  - niezależność testów od frameworka raportowania
- Pełna dokumentacja jest dostępna od tym linkiem

Podstawowe pojęcia

klasa przypadków testowych (*test case class*)	klasa bazowa dla wszystkich klas w modułach testowych wszystkie klasy testowe są wyprowadzane z tej klasy
środowisko testowe (*test fixture*)	funkcje lub metody wykonywane przed i po blokach kodu testowego, konieczne do ich wykonania oraz wszelkie powiązane z nimi działania "czyszczące" (przywracające stan pierwotny)
asercje (*assertions*)	funkcje lub metody używane do weryfikacji zachowania testowanego komponentu
zestaw testów (*test suite*)	zbiór powiązanych przypadków testowych lub zestawów testów służy do grupowania testów, które powinny być wykonywane razem

Podstawowe pojęcia

przypadek testowy (*test case*)	indywidualna jednostka testowa – w *unittest* jest nim pojedyncza metoda sprawdza odpowiedź na określony zestaw danych wejściowych
moduł uruchamiający (*test runner*)	program lub fragment kodu, który wykonuje zestaw testów i dostarcza wyniki użytkownikowi
formater wyników testów (*test result formatter*)	formatuje wyniki wykonanych testów do wybranego, czytelnego dla człowieka formatu (np. zwykły tekst, HTML, XML, . . .)

Klasa testowa

- Podstawowymi elementami składowymi testów jednostkowych są przypadki testowe – pojedyncze scenariusze, które należy skonfigurować i sprawdzić pod kątem poprawności
- Przypadki testowe są reprezentowane przez instancje *TestCase*
- Aby tworzyć własne przypadki testowe, należy rozszerzyć klasę *TestCase* (tworząc **klasę testową**) lub użyć *FunctionTestCase*
- Nazwy metod testowych mają przedrostek *test_*
- Metody testowe są wykonywane w porządku alfabetycznym, niezależnie od kolejności ich umieszczenia w kodzie

Moduły testowe

- W jednym pliku można umieścić wiele klas testowych
- Taki plik nosi nazwę **modułu testowego**
- Wszystkie klasy testowe w module są wykonywane w porządku alfabetycznym

Środowisko testowe

- **Środowisko testowe** (*test fixture*) to zestaw czynności wykonywanych **przed** i **po** testach
- Są one implementowane jako metody klasy *TestCase* i mogą być nadpisane do własnych celów

ŚRODOWISKO TESTOWE NA POZIOMIE...	METODA	JEST WYKONYWANA...
modułu	*setUpModule* *tearDownModule*	**przed** jakąkolwiek metodą w module testów **po** wszystkich metodach w module testów
klasy	*setUpClass* *tearDownClass*	**przed** jakąkolwiek metodą w klasie testów **po** wszystkich metodach w klasie testów
metody	*setUp* *tearDown*	**przed** każdą metodą w klasie testów **po** każdej metodzie w klasie testów

Klasa testowa

Przykład

```
import unittest

def setUpModule():
    print('-> setUpModule')

def tearDownModule():
    print('-> tearDownModule')

class KlasaTestowa(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        print('-> setUpClass')

    @classmethod
    def tearDownClass(cls):
        print('\n-> tearDownClass')
```

Klasa testowa

Przykład cd.

```
    # cd...
    def setUp(self):
        print('-> \nsetUp')

    def tearDown(self):
        print('-> tearDown')

    def test_if_uppercase(self):
        self.assertTrue("TEST".isupper())
        print('-> test_if_uppercase')

    def test_if_not_uppercase(self):
        self.assertFalse("test".isupper())
        print('-> test_if_not_uppercase')

if __name__ == '__main__':
    unittest.main()
```

Klasa testowa

Wynik testów

```
-> setUpModule
-> setUpClass

-> setUp
-> test_if_not_uppercase
-> tearDown
.
-> setUp
-> test_if_uppercase
-> tearDown
.
-> tearDownClass
-> tearDownModule

----------------------------------------------------------------------
Ran 2 tests in 0.018s

OK
```

Przypadki testowe

- Do identyfikacji przypadków testowych mogą być przydatne metody:

Użyteczne metody

```
id()
shortDescription()
```

id	zwraca tekst specyficzny dla danego przypadku testowego zwykle jest to pełna nazwę przypadku testowego, zawierający nazwę modułu i metody testowej
shortDescription	opis przypadku testowego zwykle to pierwsza linia dokumentacji (docstring'a)

Asercje

- W metodach testowych wykorzystuje się **asercje**
- Jeżeli argument spełnia warunek asercji, to test kończy się pomyślnie, w przeciwnym razie zawodzi
- Można też wymusić fiasko testów lub je pominąć

Wybrane metody

```
fail(msg)
expectedFailure()

skip(reason)
skipIf(condition, reason)
skipUnless(condition, reason)
```

Asercje

METODA	SPRAWDZA CZY...
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) == True
assertFalse(x)	bool(x) == False
assertIs(a, b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)
assertNotIsInstance(a, b)	not isinstance(a, b)

Asercje

METODA	SPRAWDZA CZY. . .
assertAlmostEqual(a, b)	round(a-b, 7) == 0
assertNotAlmostEqual(a, b)	round(a-b, 7) != 0
assertGreater(a, b)	a > b
assertGreaterEqual(a, b)	a >= b
assertLess(a, b)	a < b
assertLessEqual(a, b)	a <= b
assertRegexpMatches(s, r)	r.search(s)
assertNotRegexpMatches(s, r)	not r.search(s)
assertItemsEqual(a, b)	sorted(a) == sorted(b)
assertDictContainsSubset(a, b)	wszystkie pary klucz/wartość z *a* są też w *b*

Asercje

METODA	SŁUŻY DO PORÓWNANIA...
assertMultiLineEqual(a, b)	tekstów
assertSequenceEqual(a, b)	sekwencji
assertListEqual(a, b)	list
assertTupleEqual(a, b)	krotek
assertSetEqual(a, b)	zbiorów (*set* lub *frozenset*)
assertDictEqual(a, b)	słowników

Wyjątki w przypadkach testowych

- Wystąpienie nieobsłużonego wyjątku w przypadku testowym, powoduje zakończenie testu fiaskiem
- Można też za pomocą asercji sprawdzać możliwość wystąpienia wyjątku

Oczekiwanie wystąpienia wyjątku

```
assertRaises(exception, callable, *args, **kwds)
assertRaises(exception, *, msg=None)
```

Wykonywanie testów

- Do uruchomienia modułu testowego wykorzystuje się metodę *main*
- Można jej przekazać argument *verbosity* kontrolujący “szczegółowość” prezentowanych wyników testów

Inne biblioteki testowe

doctest	opis dostępny jest pod tym linkiem
pytest	opis dostępny jest pod tym linkiem
nose	opis dostępny jest pod tym linkiem
nose2	opis dostępny jest pod tym linkiem