Programowanie obiektowe w Pythonie

Wykład 3

dr Agnieszka Zbrzezny

17 października 2022

Wybrane nowości w Pythonie 3.8

Operator morsa

- W Pythonie 3.8 wprowadzono nową składnię `:=`, która przypisuje wartości do zmiennych w ramach większego wyrażenia.
- Operator `:=` jest pieszczotliwie nazywany „operatorem morsa” ze względu na podobieństwo do oczu i kłów morsa.
- W poniższym przykładzie wyrażenie przypisania pomaga uniknąć dwukrotnego wywołania funcji `len`:

```
a = list(range(20))
if (n := len(a)) > 10:
    print("List is too long",
          f"({n} elements, expected <= 10)")
```

Mors arktyczny

Wybrane nowości w Pythonie 3.8

Operator morsa

- Operator morsa jest również przydatny w przypadku pętli while, które obliczają wartość w celu przetestowania zakończenia pętli, a następnie potrzebują tej samej wartości ponownie w treści pętli.
- Załóżmy, że plik `infile` jest otwarty do odczytu w trybie tekstowym. Poniższy fragment kodu wykorzystuje operator morsa:

```
# Loop over fixed length blocks
while (block := f.read(256)) != "":
    process(block)
```

- Należy ograniczyć użycie operatora morsa do tych przypadków, w których jego użycie zmniejsza złożoność i poprawia czytelność.

Pojęcia podstawowe

- Dane to wszystko to co może być przetwarzane umysłowo lub komputerowo.
- Obiekt to dana przechowywana i przetwarzana w programie komputerowym.
- Każdy obiekt ma trzy cechy:
  - tożsamość, czyli cechę umożliwiającą jego identyfikację i odróżnienie od innych obiektów;
  - typ, czyli opis rodzaju, struktury i zakresu wartości, jakie może przyjmować obiekt oraz operacji jakie można na nim wykonać.
  - stan, czyli aktualny stan danych składowych.

Pojęcia podstawowe

- Klasa to szablon do tworzenia nowych obiektów określonego typu i posiadających określone zachowanie.
- Obiekt utworzony na podstawie danej klasy nazywany jest jej instancją, a proces jego tworzenia instancjonowaniem.
- W klasie można definiować atrybuty, którymi są zmienne (klasowe i instancyjne) oraz metody (statyczne, klasowe i instancyjne).
- Zmienna klasowa to zmienna, która jest wspólna dla wszystkich instancji klasy. Zmienne klasowe są definiowane w klasie, ale poza metodami.
- Zmienna instancyjna (inaczej pole) to zmienna, która należy tylko do bieżącej instancji klasy. Zmienne instancyjne są definiowane wewnątrz metod.

Klasy

Definiowanie klas

- Klasy są podstawowym narzędziem do tworzenia struktur danych i nowych obiektów.
- Instrukcja `class` pozwala definiować własne klasy:

Klasy

Przykład (`punkt.py`)

Napiszmy teraz klasę Punkt reprezentującą punkt.

Klasy

Definiowanie klas

- Pierwszy argument każdej metody – `self` – jest referencją do obiektu, na rzecz którego ta metoda została wywołana.
- Metody, których nazwy zaczynają się i kończą dwoma znakami podkreślenia `__` to metody specjalne.
- Przykładowo, metoda `__init__` jest automatycznie wywoływana przy tworzeniu obiektu klasy. Ta metoda nie jest jednakże konstruktorem klasy, lecz jej inicjalizatorem.
- Argumenty tej metody, za wyjątkiem pierwszego, służą do inicjalizowania zmiennych instancyjnych.

Klasy

Definiowanie klas

- Instancje klasy powstają w wyniku wywołania funkcji, której nazwą jest nazwa klasy: `a = Punkt(3, 5)`
- Po utworzeniu nowej instancji klasy jej atrybuty i metody dostępne są za pomocą operatora kropki.
- Aby wywołać daną metodę klasy na rzecz instancji tej klasy używamy notacji z kropką: `a.move(2, -1)`
- Wewnętrznie każda instancja jest implementowana przy użyciu słownika `__dict__` zawierającego unikatowe informacje o tej instancji: `print(a.__dict__)`

Używanie klas

- Metoda `__str__` tworzy reprezentację obiektu w postaci łańcucha znaków.
- W rezultacie wyświetlenie obiektu pokazuje cokolwiek, co zwróci jego metoda `__str__`.
- Aby zamienić obiekt `ob` na łańcuch znaków można wywołać na rzecz obiektu `ob` metodę `__str__`: `ob.__str__()`.
- Jednak bardziej naturalnym sposobem jest użycie funkcji wbudowanej `str` z obiektem jako jej argumentem: `str(ob)`.
- Funkcja `str` zadziała zgodnie z oczekiwaniem, ponieważ wywoła ona metodę `__str__`.
- Metoda `__str__` wykonywana jest automatycznie za każdym razem, gdy instancja przekształcana jest na swój łańcuch wyświetlania.

Hermetyzowanie nazw w klasie

- Metodyka programowania obiektowego zaleca hermetyzować prywatne dane w obiektach danej klasy, jednakże Python nie zapewnia kontroli dostępu.
- Programując w języku Python nie możemy hermetyzować danych za pomocą mechanizmów języka, powinniśmy jednak stosować się do konwencji nazewniczej związane z przeznaczeniem danych i metod.
- Konwencja ta stanowi, iż każda nazwa rozpoczynająca się od jednego podkreślenia (_) oznacza jednostkę wewnętrzną.
- Python nie blokuje dostępu do jednostek wewnętrznych. Jednak korzystanie z nich jest uznawane za nieeleganckie i może prowadzić do powstania kodu podatnego na błędy.

Klasy

Przykład (`main.py`)

Napiszmy teraz funkcję main dla naszego punktu.

Dziedziczenie

- Dziedziczenie umożliwia nam definiowanie klasy, która przyjmuje wszystkie funkcjonalności z klasy nadrzędnej i pozwala nam dodać jeszcze więcej.
- Dziedziczenie jest potężną funkcją programowania obiektowego.
- Odnosi się do definiowania nowej klasy z niewielką lub żadną modyfikacją istniejącej klasy.

Dziedziczenie

- Dziedziczenie to mechanizm tworzenia nowych klas rozszerzających już istniejące klasy.
- Klasa po której dziedziczymy nazywa się klasą bazową lub nadklasą, zaś klasa dziedzicząca nazywana jest klasą pochodną lub podklasą.
- Podklasa dziedziczy wszystkie atrybuty oraz metody nadklasy. Powoduje to ponowną użyteczność kodu.
- Dziedziczenie zapisuje się w instrukcji `class` za pomocą listy oddzielonych przecinkami nazw klas bazowych.
- W klasie pochodnej można nadpisać wybrane metody klasy bazowej.

Klasy

Przykład (`nazwany_punkt.py`)

Napiszmy teraz klasę NazwanyPunkt oraz funkcję main w osobnym pliku.

Klasy

Przykład

Sprawdźmy teraz przynależność do klas.

Dziedziczenie

- Definicja klasy zaczynająca się nagłówka

```
class NazwaKlasy:
```

jest równoważna definicji klasy zaczynającej się od nagłówka

```
class NazwaKlasy(object):
```

- Oznacza, to że każda klasa w języku Python dziedziczy bezpośrednio lub pośrednio po klasie `object`, a tym samym dziedziczy metody klasy `object`.

```
>>> ob = object()
>>> dir(ob)
```

- Funkcja wbudowana `dir` zwraca listę atrybutów obiektu.

Deinicjalizator

- Metoda `__del__` klasy `object` jest deinicjalizatorem. Służy ona do niszczenia obiektów: `del ob`.

Przykład (`nazwany_punkt.py`)

Dopiszmy teraz tę metodę do naszego pliku punkt.py i użyjmy jej w main.

Wykład 4

24 października 2022

Dekoratory

- Aby zrozumieć dekoratory, musimy najpierw poznać kilka podstawowych faktów na temat Pythona.
- Trzeba przede wszystkim pamiętać, że wszystko w Pythonie (tak! nawet klasy) są obiektami.
- Nazwy, które definiujemy, są po prostu identyfikatorami związanymi z tymi obiektami.
- Funkcje nie są wyjątkami, też są też obiektami (z atrybutami).
- Różne nazwy mogą być powiązane z tym samym obiektem funkcji.
- Zobaczmy prosty przykład: `decorator/hello.py`.

Dekoratory

- Teraz rzeczy zaczynają być coraz dziwniejsze.
- Funkcje mogą być przekazywane jako argumenty do innej funkcji.
- Funkcje, które przyjmują inne funkcje jako argumenty, nazywają się również funkcjami wyższego rzędu. Ponadto funkcja może zwracać inną funkcję.
- Napiszmy przykładowy kod takiej funkcji: `decorator/operate.py`.

Dekoratory

- Funkcje i metody nazywane są wywoływalnymi (ang. callable), ponieważ mogą być wywoływane.
- W rzeczywistości dowolny obiekt, który implementuje specjalną metodę `__call__` jest nazywany wywoływalnym.
- Tak więc, w najbardziej podstawowym znaczeniu, dekorator jest zdolny do odwoływania, który zwraca callable.
- W prostych słowach: dekorator bierze funkcję, dodaje pewną funkcjonalność i zwraca ją.
- Zobaczmy teraz kod: `decorator/pretty.py`.

Dekoratory

- Widać, że funkcja dekoratora dodała nową funkcjonalność do oryginalnej funkcji.
- Jest to podobne do pakowania prezentu. Dekorator działa jak opakowanie.
- Natura przedmiotu zdobionego (obecny prezent wewnątrz) nie zmienia się.
- Zwykle dekorujemy funkcję i przypisujemy ją tak jak w przykładowym kodzie ale...
- Dekorowanie jest bardzo popularne i powszechnie stosowane w Pythonie, dlatego Python ma składnię upraszczającą dekorowanie.
- Możemy użyć symbolu wraz z nazwą funkcji dekoratora i umieścić go powyżej definicji funkcji, która ma być dekorowana.
- Zobaczmy teraz kod: `decorator/pretty.py`.

Dekorowanie funkcji z parametrami

- Powyższy dekorator był prosty i działał tylko z funkcjami, które nie miały żadnych parametrów.
- Co zrobić, jeśli mamy funkcje, które przyjmują więcej parametrów? (`decorator/smart_divide.py`).

Dekorowanie funkcji z parametrami

- W ten sposób możemy zdobić funkcje, które przyjmują parametry.
- Dobry obserwator zauważy, że parametry zagnieżdżonej funkcji `inner()` wewnątrz dekoratora są takie same jak parametry funkcji zdobionej.
- Biorąc to pod uwagę, możemy teraz tworzyć ogólne dekoratory, które działają z dowolną liczbą parametrów.
- W Pythonie ta sztuczka jest wykonywana jako `funkcja(*args, **kwargs)`.
- W ten sposób `args` będzie krotką argumentów pozycyjnych, a `kwargs` będzie słownikiem kluczowych argumentów. Przykładem takiego dekoratora będzie: `decorator/works_for_all.py`.

Dekorowanie funkcji z parametrami

- W Pythonie można wymieszać wiele dekoratorów.
- To znaczy, funkcja może być dekorowana wielokrotnie różnymi dekoratorami (lub tymi samymi).
- Wystarczy umieścić dekoratory powyżej pożądanej funkcji `decorator/star.py`.

Pojęcia podstawowe

- Metoda statyczna to funkcja zdefiniowana w zakresie klasy, ale poprzedzona dekoratorem `@staticmethod`.
- Metoda klasowa to funkcja zdefiniowana w zakresie klasy, ale poprzedzona dekoratorem `@classmethod`.
  - Pierwszy argument metody klasowej jest zwyczajowo nazywany `cls` i jest referencją do klasy.
- Metoda instancyjna (w skrócie: metoda) to funkcja zdefiniowana w zakresie klasy.
  - Pierwszy argument metody klasowej jest zwyczajowo nazywany `self` i jest referencją do obiektu, na rzecz którego wywołano tę metodę.

Metody statyczne

- Istnieją metody, które nie operują na konkretnej instancji klasy. Typowo nazywa się je statycznymi.
- W Pythonie nie posiadają one parametru `self`, lecz są opatrzone dekoratorem `@staticmethod`:
- Statyczną metodę można wywołać zarówno przy pomocy nazwy klasy, jak i jej obiektu, ale w obu przypadkach rezultat będzie ten sam.
- Technicznie jest to bowiem zwyczajna funkcja umieszczona po prostu w zasięgu klasy zamiast w zasięgu globalnym.
- Napiszmy zatem klasę Date.

Metody statyczne

- Wyniki działania naszego programu nie są tym czego chcielibyśmy i powinniśmy się spodziewać.
- Jest to spowodowane tym, że umieszczone w klasie `Date` metody `today` i `tomorrow` są metodami statycznymi.
- Powyższe dwie metody powinny być metodami klasowymi.
- Natomiast przydatną metodą statyczną w klasie `Date` byłaby poniższa metoda:

```
class Date:
    ...
    @staticmethod
    def seconds_per_day():
        return 24 * 60 * 60
```

Metody klasowe

- Metody klasowe są wywoływane na rzecz całej klasy (a nie jakiejś jej instancji) i przyjmują ową klasę jako swój pierwszy parametr.
- Argument ten jest często nazywany `cls`, ale jest to o wiele słabsza konwencja niż ta dotycząca `self`.
- W celu odróżnienia od innych rodzajów, metody klasowe oznaczone są dekoratorem `@classmethod`.
- Podobnie jak metody statyczne, można je wywoływać na dwa sposoby – przy pomocy klasy lub obiektu – ale w obu przypadkach do `cls` trafi wyłącznie klasa.
- W ogólności może to być także klasa pochodna.

Metody klasowe

- gooddate/date.py
- gooddate/polishdate.py
- gooddate/testdate.py

Klasy

Hermetyzacja

- Klasa `Punkt` z polami „prywatnymi” wg umowy: punkt_0/punkt.py
- Na zewnątrz klasy można modyfikować atrybuty oraz dynamicznie definiować nowe atrybuty: punkt_0/main_punkt.py
- Na zewnątrz klasy można usuwać atrybuty: punkt_1/main_punkt.py

Hermetyzacja - zmienne i metody prywatne

- Ani zmienna prywatna, ani metoda prywatna nie są widzialne poza metodami klasy, w której zostały zdefiniowane.
- Prywatne zmienne i metody są przydatne z dwóch powodów:
  1. zwiększają bezpieczeństwo i stabilność kodu dzięki selektywnemu odmawianiu dostępu do ważnych czy delikatnych części implementacji obiektu;
  2. pozwalają uniknąć konfliktów nazw wynikających z dziedziczenia.
- Klasa może definiować prywatne zmienne i dziedziczyć po klasie definiującej zmienne prywatne o tych samych nazwach, ale nie spowoduje to problemu, bo dzięki temu, że są one prywatne, każda klasa ma własne kopie.
- Prywatne zmienne służą czytelności kodu, ponieważ jasno określają, co jest używane wewnętrznie przez klasę. Wszystko inne jest interfejsem klasy.

Hermetyzacja - zmienne i metody prywatne

- Większość języków, które definiują prywatne zmienne, robi to za pomocą słowa kluczowego **private** lub podobnego.
- Konwencja w Pythonie jest prostsza i pozwala od razu dostrzec, co jest prywatne, a co nie.
- Każda metoda czy zmienna instancji, której nazwa zaczyna się od podwójnego znaku podkreślenia (ale nie kończy się nim), jest prywatna. Wszystko inne nie jest.

Hermetyzacja - zmienne i metody prywatne

- Klasa `punkt` z polami „prywatnymi”: punkt_2/punkt.py
- Teraz nie można modyfikować pól na zewnątrz klasy: punkt_2/main_punkt.py

Hermetyzacja - zmienne i metody prywatne

- Zauważmy, że mechanizm zapewniania prywatności zniekształca nazwy prywatnych zmiennych i metod, gdy kod zostanie skompilowany do kodu pośredniego (ang. bytecode).
- Operacja, jaka będzie miała wtedy miejsce, to dodanie nazwy klasy ze znakiem podkreślenia na początku nazwy klasy do początku nazwy zmiennej:

```
from punkt import Punkt
>>> dir(Punkt(3, 5))
['_Punkt__x', '_Punkt__y', ...]
>>>
```

- Operacja ta nazywa się dekorowaniem nazw (ang. name mangling, name decoration)

Hermetyzacja - zmienne i metody prywatne

- Celem operacji dekorowania nazw jest zapobiegnięcie jakimkolwiek przypadkowym udostępnieniom zmiennej.
- Gdyby ktoś sobie życzył, mógłby celowo zasymulować dekorowanie, jakie nastąpi, i w ten sposób uzyskać dostęp do zmiennej.
- Ale dokonanie dekorowania we wskazanej wyżej formie ułatwia debugowanie programu.

Właściwości

- Python posiada świetną koncepcję zwaną właściwością, która sprawia, że życie programisty obiektowego jest znacznie prostsze.
- Przed zdefiniowaniem i przejściem do szczegółów tego, co to jest property, najpierw skonstruujmy intuicję na temat tego, dlaczego to w ogóle jest potrzebne.
- Załóżmy, że zdecydujesz się na napisanie klasy, która może przechowywać temperaturę w stopniach Celsjusza.
- Implementowałaby również metodę przekształcania temperatury w stopnie Fahrenheita.
- temperature_old.py
- temperature_old_1_1.py

Właściwości

- Wielkim problemem z powyższą aktualizacją jest to, że wszyscy klienci, którzy zaimplementowali naszą poprzednią klasę w swoim programie musieli zmodyfikować swój kod
- z `obj.temperature` na `obj.get_temperature()`
- i wszystkie przypisania, takie jak `obj.temperature = val` na `obj.set_temperature(val)`.
- To refaktoryzowanie może powodować bóle głowy u klientów z setkami tysięcy linii kodu.
- W sumie nasza nowa aktualizacja nie była kompatybilna wstecznie.
- Teraz ratuje nas property.
- Pythonowym sposobem radzenia sobie z powyższym problemem jest użycie właściwości.

Właściwości

- Python pozwala programistom na bezpośredni dostęp do zmiennych instancji, bez konieczności tworzenia oprzyrządowania w postaci getterów i setterów, powszechnych w innych językach obiektowych.
- Brak metod ustawiających i odczytujących wartości czyni kod klas w Pythonie czystszym i prostszym, ale w niektórych sytuacjach użycie getterów i setterów może być wygodne.
- Załóżmy, że chcielibyśmy dokonać jakiejś operacji na wartości, zanim przypiszemy ją do zmiennej instancyjnej. Albo przydałoby się obliczanie wartości zmiennej w locie.
- W obu wypadkach metody typu `get` i `set` byłyby odpowiedzią na to zapotrzebowanie, ale ich kosztem byłaby utrata charakterystycznej dla Pythona łatwości dostępu do zmiennych instancji.

Właściwości

- Niemniej i z tym można sobie poradzić. Rozwiązaniem jest tu używanie właściwości.
- Właściwość łączy możliwość dostępu do zmiennej instancji poprzez gettery lub settery oraz przejrzystość typowej dla Pythona notacji zmiennych instancji.
- Aby stworzyć właściwość, potrzebujemy dekoratora `property` oraz metody typu `get` o nazwie takiej samej, jak właściwość.
- Bez funkcji udostępniającej ustawianie wartości taka właściwość jest dostępna jedynie do odczytu. Aby móc ją zmienić, potrzebna jest metoda typu `set`.

Klasa `Temperature` z użyciem właściwości

- temperature/temperature_2_0.py
- temperature/temperature.py
- temperature/main_temperature.py

Metody przeciążające operatory

- Klasy przechwytują i implementują działania wbudowane poprzez metody o specjalnych nazwach.
- Wszystkie one rozpoczynają się i kończą dwoma znakami podkreślenia.
- Nazwy te nie są zarezerwowane i mogą być dziedziczone z klas nadrzędnych w zwykły sposób.
- Python wyszukuje i wywołuje co najmniej jedną taką metodę w każdej operacji.
- Python automatycznie wywołuje metody przeciążające klasy w przypadku, gdy egzemplarze znajdują się w wyrażeniach oraz innych kontekstach.

Metody przeciążające operatory

- Jeżeli na przykład klasa definiuje metodę o nazwie `__getitem__`, a `X` jest egzemplarzem tej klasy, to wyrażenie `X[j]` jest równoważne wywołaniu metody `X.__getitem__(j)`.
- Nazwy przeciążanych metod mogą być dowolne — metoda `__add__` klasy nie musi realizować dodawania (lub konkatenacji).
- Co więcej, w klasach dopuszczalne jest mieszanie metod przetwarzających liczby i kolekcje oraz działań mutowalnych i niemutowalnych.
- Większość nazw przeciążających operatory nie ma wartości domyślnych, a odpowiadające im działania zgłaszają wyjątek, jeśli określona metoda nie jest zdefiniowana.

Metody przeciążające operatory

- Metody do porównywania obiektów:
  `__lt__(self, inny)`
  `__le__(self, inny)`
  `__eq__(self, inny)`
  `__ne__(self, inny)`
  `__gt__(self, inny)`
  `__ge__(self, inny)`
- Metody te są używane odpowiednio podczas porównań:
  `self < inny`, `self <= inny`, `self == inny`, `self != inny`, `self > inny` oraz `self >= inny`.
- Są to tzw. metody bogatych porównań, które są wywoływane w odniesieniu do wszystkich wyrażeń wykorzystujących porównania.
- Przykładowo, porównanie `X < Y` wywołuje metodę `X.__lt__(Y)`, o ile ją zdefiniowano.

Metody przeciążające operatory

- Powyższe metody mogą zwrócić dowolną wartość, ale jeżeli operator porównania zostanie użyty w kontekście operacji logicznych, to zwrócona wartość będzie zinterpretowana jako logiczny wynik (typu Boolean) działania operatora.
- Metody te mogą również zwracać (choć nie zgłaszają wyjątku) specjalny obiekt `NotImplemented` w przypadku, gdy operandy ich nie obsługują.
- Efekt jest taki, jakby metoda w ogóle nie została zdefiniowana.

Metody przeciążające operatory

- Nie istnieją domniemane relacje pomiędzy operatorami porównań. Na przykład z faktu, że wyrażenie `x == y` ma wartość `True`, nie wynika, że wyrażenie `x != y` ma wartość `False`.
- Aby operatory działały symetrycznie, należy zdefiniować metodę `__ne__` razem z metodą `__eq__`.
- Nie istnieją również prawostronne (o zamienionych argumentach) wersje tych metod do wykorzystania w sytuacji, kiedy lewy argument nie obsługuje określonego działania, a prawy je obsługuje.
- Metody `__lt__` i `__gt__`, `__le__` i `__ge__` oraz `__eq__` i `__ne__` są swoimi wzajemnymi odbiciami.
- W Pythonie 3 dla potrzeb operacji sortowania należy używać metody `__lt__`.

Metody przeciążające operatory dwuargumentowe

- Jeśli któraś z poniższych metod nie obsługuje działania dla przekazanych argumentów, to powinna zwrócić (nie zgłosić) wbudowany obiekt `NotImplemented`, który działa tak, jakby metoda w ogóle nie była zdefiniowana.
- Podstawowe metody działań dwuargumentowych:
  - `__add__(self, inny)`
    Wywoływana podczas odwołań `self + inny` – realizuje dodawanie liczb lub konkatenację sekwencji.
  - `__sub__(self, inny)`
    Wywoływana przy odwołaniach `self - inny`.
  - `__mul__(self, inny)`
    Wywoływana podczas odwołań `self * inny` – realizuje mnożenie liczb lub powtarzanie (repetycję) sekwencji.
  - `__truediv__(self, inny)`
    Wywoływana podczas odwołań `self / inny` w celu przeprowadzenia dzielenia (z uwzględnieniem reszty).

Metody przeciążające operatory dwuargumentowe

- Podstawowe metody działań dwuargumentowych:
  - `__floordiv__(self, inny)`
    Wywoływana przy odwołaniach `self // inny` w celu realizacji dzielenia z obcinaniem.
  - `__mod__(self, inny)`
    Wywoływana przy odwołaniach `self % inny`.
  - `__divmod__(self, inny)`
    Wywoływana przy odwołaniach `divmod(self, inny)`.
  - `__pow__(self, inny [, modulo])`
    Wywoływana przy odwołaniach `pow(self, inny [, modulo])` oraz `self ** inny`.
  - `__lshift__(self, inny)`
    Wywoływana przy odwołaniach `self << inny`.
  - `__rshift__(self, inny)`
    Wywoływana przy odwołaniach `self >> inny`.

Metody przeciążające operatory dwuargumentowe

- Podstawowe metody działań dwuargumentowych:
  - `__and__(self, inny)`
    Wywoływana przy odwołaniach `self & inny`.
  - `__xor__(self, inny)`
    Wywoływana przy odwołaniach `self ^ inny`.
  - `__or__(self, inny)`
    Wywoływana przy odwołaniach `self | inny`.

Prawostronne metody działań dwuargumentowych

- Podstawowe metody prawostronnych działań dwuargumentowych:

```
__radd__(self, inny)
__rsub__(self, inny)
__rmul__(self, inny)
__rtruediv__(self, inny)
__rfloordiv__(self, inny)
__rmod__(self, inny)
__rdivmod__(self, inny)
__rpow__(self, inny)
__rlshift__(self, inny)
__rrshift__(self, inny)
__rand__(self, inny)
__rxor__(self, inny)
__ror__(self, inny)
```

Prawostronne metody działań dwuargumentowych

- Są to prawostronne odpowiedniki operatorów dwuargumentowych, opisanych na poprzednich slajdach.
- Ich nazwy rozpoczynają się od prefiksu `r` (np. `__add__` oraz `__radd__`). Odmiany prawostronne mają takie same listy argumentów, ale argument `inny` występuje po prawej stronie operatora.
- Przykładowo, działanie `self + inny` wywołuje metodę `self.__add__(inny)`, natomiast `inny + self` wywołuje metodę `self.__radd__(inny)`.

Prawostronne metody działań dwuargumentowych

- Metody prawostronne (r) są wywoływane tylko wtedy, kiedy egzemplarz klasy znajduje się po prawej stronie, a lewy operand nie jest egzemplarzem klasy, która implementuje działanie:
  `egzemplarz + innyobiekt` uruchamia metodę `__add__`
  `egzemplarz + egzemplarz` uruchamia metodę `__add__`
  `innyobiekt + egzemplarz` uruchamia metodę `__radd__`.
- Jeśli w działaniu występują obiekty dwóch klas przeciążających działanie, to preferowana jest klasa argumentu występującego po lewej stronie.
- Metoda `__radd__` często jest implementowana w ten sposób, że zamienia kolejność operandów i wywołuje metodę `__add__`.

Metody działań dwuargumentowych

- Metody działań dwuargumentowych z aktualizacją w miejscu:

```
__iadd__(self, inny)
__isub__(self, inny)
__imul__(self, inny)
__itruediv__(self, inny)
__ifloordiv__(self, inny)
__imod__(self, inny)
__ipow__(self, inny [, modulo])
__ilshift__(self, inny)
__irshift__(self, inny)
__iand__(self, inny)
__ixor__(self, inny)
__ior__(self, inny)
```

- Są to metody przypisania z aktualizacją (w miejscu).

Metody działań dwuargumentowych

- Są one wywoływane odpowiednio dla następujących formatów instrukcji przypisania: `+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=`, `<<=`, `>>=`, `&=`, `^=` oraz `|=`.
- Metody te powinny podejmować próbę wykonania działania w miejscu (z modyfikacją egzemplarza self) i zwracać wynik, którym może być egzemplarz `self`.
- Jeśli metoda jest niezdefiniowana, to działanie z aktualizacją sięga do zwykłych metod.
- W celu obliczenia wartości wyrażenia `X += Y`, gdzie `X` jest egzemplarzem klasy ze zdefiniowaną metodą `__iadd__`, wywoływana jest metoda `x.__iadd__(y)`.
- W przeciwnym razie wykorzystywane są metody `__add__` oraz `__radd__`.

Przykład klasy Temperature z metodami przeciążającymi

```
class Temperature:
    def __init__(self, temp = 0):
        self.__temp = temp


    ...

    def __add__(self, ile_stopni):
        return Temperature(self.__temp + ile_stopni)

    def __iadd__(self, ile_stopni):
        self.__temp += ile_stopni
        return self
```

Atrybut `__slots__`

- Niekiedy programy tworzy dużą liczbę obiektów (liczoną w milionach) i zużywają dużo pamięci.
- W klasach, które przede wszystkim pełnią funkcję prostych struktur danych, często można znacznie zmniejszyć ilość pamięci zajmowaną przez obiekty, dodając do definicji klasy atrybut `__slots__`. Przykładowo:
- date_slots/date.py

Atrybut `__slots__`

- Gdy zdefiniujemy atrybut `__slots__`, Python będzie stosował znacznie zwięźlejszą reprezentację obiektów.
- Zamiast dodawać słownik do każdego obiektu, Python tworzy wtedy obiekty oparte na małej tablicy o stałym rozmiarze (przypominającej krotkę lub listę).
- Nazwy atrybutów wymienione w specyfikatorze `__slots__` są wewnętrznie odwzorowywane na konkretne indeksy tablicy.
- Efektem ubocznym stosowania tej techniki jest to, że do obiektów nie można dodawać nowych atrybutów.
- Dozwolone są tylko atrybuty wymienione w specyfikatorze `__slots__`.

Atrybut `__slots__`

- Choć może się wydawać, że przedstawione rozwiązanie jest przydatne w wielu sytuacjach, nie należy go nadużywać.
- W wielu miejscach Pythona stosuje się standardowy kod oparty na słownikach. Ponadto klasy utworzone za pomocą opisanej techniki nie obsługują niektórych mechanizmów, np. wielodziedziczenia.
- Dlatego technikę tę należy stosować tylko w tych klasach, które są często używane w programie (np. gdy program tworzy miliony obiektów danej klasy).
- Często technika slotów jest traktowana jako narzędzie zapewniające hermetyzację, które uniemożliwia użytkownikom dodawanie nowych atrybutów do obiektów.

Atrybut `__slots__`

- Do klasy `Date` możemy dodać właściwości dla poszczególnych atrybutów.
- date_slots/date_prop.py

Atrybut `__slots__`

- Metoda `__lt__`
- Metody `__eq__` oraz `__le__`
- Metoda `__str__` oraz `__repr__`

Serializowanie obiektów Pythona

- Zdarza się, że programista chce zserializować obiekt Pythona na strumień bajtów, aby móc zapisać dany obiekt do pliku, zachować go w bazie danych lub przesłać przez sieć.
- Najczęściej stosowanym narzędziem do serializowania danych jest moduł `pickle`. Aby zapisać obiekt w pliku, należy użyć poniższego kodu:

```
import pickle
data = ... # obiekt Pythona
f = open("somefile", "wb")
pickle.dump(data, f)
```

- Do zapisywania obiektu w łańcuchu znaków służy funkcja `pickle.dumps`:

```
s = pickle.dumps(data)
```

Serializowanie obiektów Pythona

- Aby odtworzyć obiekt ze strumienia bajtów, można zastosować funkcję `pickle.load` lub `pickle.loads`:
- Odtwarzanie z pliku

```
f = open('somefile', 'rb')
data = pickle.load(f)
f.close()
```

- Odtwarzanie z łańcucha znaków

```
data = pickle.loads(s)
```

Serializowanie obiektów Pythona

- W większości programów funkcje `dump` i `load` wystarczą do skutecznego korzystania z modułu `pickle`.
- Rozwiązanie to działa dla większości typów danych Pythona i klas zdefiniowanych przez użytkowników.
- Jeżeli korzystamy z biblioteki, która umożliwia zapisywanie i odtwarzanie obiektów Pythona w bazach danych lub przesyłanie obiektów przez sieć, bardzo możliwe, że używa ona modułu `pickle`.
- Moduł `pickle` odpowiada za charakterystyczne dla Pythona samoopisowe kodowanie danych. Dzięki temu, że jest samoopisowe, serializowane dane zawierają informacje o początku i końcu każdego obiektu oraz o jego typie.
- Dlatego nie trzeba martwić się o definiowanie rekordów – kod działa i bez tego.

Serializowanie obiektów Pythona

- Przykładowo, do serializacji grupy obiektów możemy zastosować następujący kod:

```
>>> import pickle
>>> f = open("somedata", "wb")
>>> pickle.dump([1, 2, 3, 4], f)
>>> pickle.dump("Witaj", f)
>>> pickle.dump({"Jabłko", "Gruszka", "Banan"}, f)
>>> f.close()
>>> f = open("somedata", "rb")
>>> pickle.load(f)
[1, 2, 3, 4]
>>> pickle.load(f)
'Witaj'
>>> pickle.load(f)
{'Jabłko', 'Gruszka', 'Banan'}
>>>
```

Serializowanie obiektów Pythona

- W ten sposób można serializować funkcje, klasy i obiekty, przy czym w wygenerowanych danych zakodowane są tylko referencje do powiązanych obiektów z kodu. Oto przykład:

```
import math
import pickle
print(pickle.dumps(math.log))
print(pickle.dumps([math.sin, math.cos]))
```

- W momencie deserializacji program przyjmuje, że cały potrzebny kod źródłowy jest dostępny. Moduły, klasy i funkcje są w razie potrzeby automatycznie importowane.
- Gdy dane Pythona są współużytkowane przez interpretery z różnych komputerów, może to utrudniać konserwację kodu, ponieważ wszystkie komputery muszą mieć dostęp do tego samego kodu źródłowego.

Serializowanie obiektów Pythona

- Funkcji `pickle.load` nigdy nie należy używać do niezaufanych danych.
- W ramach wczytywania kodu moduł `pickle` automatycznie pobiera moduły i na ich podstawie tworzy obiekty.
- Napastnik, który wie, jak działa moduł `pickle`, może przygotować specjalnie spreparowane dane powodujące, że Python wykona określone polecenia systemowe.
- Dlatego moduł `pickle` należy stosować tylko wewnętrznie w interpreterach, które potrafią uwierzytelniać siebie nawzajem.

Serializowanie obiektów Pythona

- Niektórych obiektów nie można zserializować w ten sposób.
- Są to zwykle obiekty mające zewnętrzny stan w systemie, takie jak otwarte pliki, otwarte połączenia sieciowe, wątki, procesy, ramki stosu itd.
- W klasach zdefiniowanych przez użytkownika można czasem obejść to ograniczenie, udostępniając metody `__getstate__` oraz `__setstate__`.
- Wtedy funkcja `pickle.dump` wywołuje metodę `__getstate__`, aby pobrać serializowany obiekt, a przy deserializacji wywoływana jest metoda `__setstate__`.
- Aby zilustrować możliwości tego podejścia, poniżej przedstawiono klasę ze zdefiniowanym wewnętrznie wątkiem, którą jednak można zarówno serializować, jak i deserializować:

Serializowanie obiektów Pythona

```
import time
import threading

class Countdown:
    def __init__(self, n):
        self.n = n
        self.thr = threading.Thread(target=self.run)
        self.thr.daemon = True
        self.thr.start()

    ...
```

Serializowanie obiektów Pythona

```
class Countdown:

    ...

    def run(self):
        while self.n > 0:
            print('T-minus', self.n)
            self.n -= 1
            time.sleep(5)

    def __getstate__(self):
        return self.n

    def __setstate__(self, n):
        self.__init__(n)
```

Serializowanie obiektów Pythona

- Teraz możemy przeprowadzić następujący eksperyment:

```
>>> import pickle
>>> import countdown
>>> c = countdown.Countdown(30)
>>> T-minus 30
T-minus 29
T-minus 28
...
>>> # Po pewnym czasie
>>> f = open("cstate.p", "wb")
>>> pickle.dump(c, f)
>>> f.close()
```

Serializowanie obiektów Pythona

- Teraz możemy wyjść z interpretera Pythona i po ponownym jego uruchomieniu wywołać następujący kod:

```
>>> import pickle
>>> f = open("cstate.p", "rb")
>>> pickle.load(f)
countdown.Countdown object at 0x10069e2d0>
T-minus 19
T-minus 18
...
```

- Widzimy, jak wątek w magiczny sposób ponownie zaczyna działać i wznawia pracę od miejsca, w którym zakończył ją w momencie serializowania.

Operacje wejścia-wyjścia

Serializowanie obiektów Pythona

- Moduł `pickle` nie zapewnia wysokiej wydajności kodowania dużych struktur danych, np. tablic binarnych tworzonych przez takie biblioteki jak moduł `array` lub `numpy`.
- Jeżeli chcemy przenosić duże ilości danych tablicowych, lepszym rozwiązaniem może być zapisanie ich w pliku lub zastosowanie standardowego kodowania, np. `HDF5` (obsługiwanego przez niestandardowe biblioteki).
- Ponieważ moduł `pickle` działa tylko w Pythonie i wymaga kodu źródłowego, zwykle nie należy go używać do długoterminowego przechowywania danych.
- Jeżeli kod źródłowy zostanie zmodyfikowany, wszystkie przechowywane dane mogą stać się nieczytelne.

Operacje wejścia-wyjścia

Serializowanie obiektów Pythona

- Przy przechowywaniu danych w bazach danych lub archiwach zwykle lepiej jest stosować bardziej standardowe kodowania, np. `XML`, `CSV` lub `JSON`.
- Są one w większym stopniu ustandaryzowane, obsługuje je wiele języków i są lepiej dostosowane do zmian w kodzie źródłowym. Ponadto warto pamiętać, że moduł `pickle` udostępnia wiele różnych opcji i ma skomplikowane przypadki brzegowe.
- Przy wykonywaniu typowych zadań nie trzeba się nimi przejmować.
- Jeżeli jednak pracujemy nad rozbudowaną aplikacją, która do serializacji używa modułu `pickle`, należy zapoznać się z jego oficjalną dokumentacją: `https://docs.python.org/3/library/pickle.html`

```
from datetime import date


class Court:
    __width: float
    __length: float
    __address: str
    __year_built: int

    def __init__(self, length: float = 150, width: float = 68, address: str = '', year_built: int = 0) -> None:
        if year_built < 2008:
            if (width >= 45 and width <= 90) and (length >= 90 and length <= 120):
                self.__length = length
                self.__width = width
            else:
                self.__length = 150
                self.__width = 68
        else:
            self.__width = width
            self.__length = length
        self.__address = address
        self.__year_built = year_built

    @property
    def width(self) -> float:
        return self.__width

    @width.setter
    def width(self, value: float) -> None:
        self.__width = value
```

```
@property
def length(self) -> float:
    return self.__length

@length.setter
def length(self, value: float) -> None:
    self.__length = value

@property
def address(self) -> str:
    return self.__address

@address.setter
def address(self, value: str) -> None:
    self.__address = value

@property
def year_built(self) -> int:
    return self.__year_built

@year_built.setter
def year_built(self, value: int) -> None:
    self.__year_built = value

def area(self) -> float:
    return self.width * self.length

@staticmethod
def validate(obj: 'Court') -> None:
    current_year = date.today().year
    if obj.year_built < 0 or obj.year_built > current_year:
```

```
            obj.year_built = current_year

    def __repr__(self) -> str:
        return f'Boisko wybudowane w roku {self.year_built}, ' \
               f'o długości {self.length} i szerokości {self.width}\n' \
               f'Pole powierzchni: {self.area()} mkw.\n' \
               f'Adres: {self.address}\n'

    def __eq__(self, other: 'Court') -> bool:
        if self.area() == other.area():
            return True
        return False

    def __ne__(self, other: 'Court') -> bool:
        if self.area() != other.area():
            return True
        return False
```

czesc 2

```
from main import Court
from typing import Optional


class Stadium(Court):
    __name: str
    __common_name: Optional[str]
    __capacity: int

    def __init__(self, width: float = 68, length: float = 150, address: str = '',
                 year_built: int = 0, name: str = '', common_name: Optional[str] = '',
```

```
            capacity: int = 0) -> None:
    super().__init__(width, length, address, year_built)
    self.__name = name
    self.__common_name = common_name
    if capacity >= 0:
        self.__capacity = capacity

@property
def name(self) -> str:
    return self.__name

@name.setter
def name(self, value: str) -> None:
    self.__name = value

@property
def common_name(self) -> str:
    return self.__common_name

@common_name.setter
def common_name(self, value: str) -> None:
    self.common_name = value

@property
def capacity(self) -> int:
    return self.__capacity

@capacity.setter
def capacity(self, value: int) -> None:
    if value < 0:
        print(f'niepoprawne dane')
```

```
        self.__capacity = value

    def __eq__(self, other: 'Stadium') -> bool:
        if self.area() == other.area() and self.capacity == other.capacity:
            return True
        return False

    def __ne__(self, other: 'Stadium') -> bool:
        if self.area() != other.area() or self.capacity != other.capacity:
            return True
        return False

    def __repr__(self) -> str:
        text = f"Boisko wybudowane w {self.year_built}, " \
               f"o długości {self.length} i szerokości {self.width} metrów.\n" \
               f"Pole powierzchni: {self.area()} mkw.\n" \
               f"Adres: {self.address}.\n" \
               f"Nazwa: {self.name}.\n"
        if self.common_name is not None and self.common_name != '':
            text += f"Nazwa zwyczajowa: {self.common_name}.\n"
        text += f"Pojemność stadionu {self.capacity}.\n"
        return text
```

czesc 3

__lt__(self, other)	<
__le__(self, other)	<=
__eq__(self, other)	==
__ne__(self, other)	!=
__gt__(self, other)	>
__ge__(self, other)	>=
__add__(self, other)	+

```
__sub__(self, other)      –
__mul__(self, other)      *
# Metoda hasattr()zwraca true, jeśli obiekt ma podany atrybut nazwany, a false, jeśli go nie ma.
#
# Przykład
# class Person:
#     age = 23
#     name = "Adam"
#
# person = Person()
#
# print("Person's age:", hasattr(person, "age"))
# print("Person's salary:", hasattr(person, "salary"))
#
# # Output:
# # Person's age: True
# # Person's salary: False

# dowolna liczbe argumentów nazwanych **kwargs
# dowolna liczbe argumentów nienazwanych *args

# # mnozenie
#     def __mul__(self, other):
#         return Wymierna(self.licznik * other.licznik, self.mianownik * other.mianownik)
# # dzielenie
#     def __div__(self, other):
#         return Wymierna(self.licznik * other.mianownik, self.mianownik * other.licznik)

#przestrzen nazw __dict__
```

Testy

```
from main import Court

from Stadium import Stadium

court_1 = Court(address='Słoneczna 10, 10-100 Olsztyn', year_built=1999)#1.    Stwórz obiekt o nazwie court_1 i zainicjalizuj go domyślną szerokością i długością.

# Rok budowy i adres jak w punkcie 1.3.1.

print(court_1)

court_2 = Court(500, 500, 'Słoneczna 10, 10-100 Olsztyn', 1999)

print(court_2)

court_3 = Court(100, 50, 'Słoneczna 10, 10-100 Olsztyn', 1999)

court_3.width = 50

court_3.length = 100

print(court_3)

print(court_1.length)#Wypisz na ekran wartość atrybutu length obiektu court_1.

court_1.year_built = 1990          #Zmień wartość atrybutu year_built obiektu court_1 na 1990 i wypisz na ekran reprezentację obiektu court_1 po zmianie stanu.

print(court_1)

print(court_1.area())

court_1.year_built = 2030

print(court_1)

Court.validate(court_1)#Użyj metody statycznej aby zwalidować rok budowy boiska court_1

print(court_1)

print("---------------------------------")

stadium_1 = Stadium(address='Słoneczna 10, 10-100 Olsztyn', year_built=1999,

                    name='Słoneczny stadion', common_name='Słoneczko', capacity=10000)

print(stadium_1)

stadium_2 = Stadium(address='Słoneczna 10, 10-100 Olsztyn', year_built=1999,

                    name='Słoneczny stadion', capacity=10000)

print(stadium_2)

stadium_1.year_built = 2030

print(stadium_1)

Stadium.validate(stadium_1)
```

```
print(stadium_1)
print(stadium_1 == stadium_2)
stadium_1.width = 50
stadium_1.length = 100
print(stadium_1 == stadium_2)
print(stadium_1 != stadium_2)
stadium_1.capacity = 500
print(stadium_1 == stadium_2)
```