Zaawansowane programowanie w Pythonie

Wykład 1 - klasy dokończenie

dr Agnieszka Zbrzezny

20 października 2024

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:

```
class Date:
    __slots__ = ('__year', '__month', '__day')

    def __init__(self, year, month, day):
        self.__year = year
        self.__month = month
        self.__day = day
```

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.

```
class Date:
    __slots__ = ('__year', '__month', '__day')

    @property
    def year(self):
        return self.__year

    @year.deleter
    def year(self):
        raise AttributeError(
            "Nie można usunąć atrybutu year")

    ...
```

Atrybut `__slots__`

- Atrybut `month`:

```
class Date:
    __slots__ = ('__year', '__month', '__day')

    ...

    @property
    def month(self):
        return self.__month

    @month.deleter
    def month(self):
        raise AttributeError(
            "Nie można usunąć atrybutu month")

    ...
```

Atrybut `__slots__`

- Atrybut `day`:

```
class Date:
    __slots__ = ('__year', '__month', '__day')

    ...

    @property
    def day(self):
        return self.__day

    @day.deleter
    def day(self):
        raise AttributeError(
            "Nie można usunąć atrybutu day")

    ...
```

Atrybut `__slots__`

- Metoda `replace`:

```
class Date:
    __slots__ = ('__year', '__month', '__day')

    ...

    def replace(self, year=None, month=None, day=None):
        """Zwraca nową datę z nowymi wartościami
        wybranych atrybutów."""
        if year is None:
            year = self.__year
        if month is None:
            month = self.__month
        if day is None:
            day = self.__day
        return type(self)(year, month, day)
```

Atrybut `__slots__`

- Metoda `__str__` oraz `__repr__`:

```
class Date:
    __slots__ = ('__year', '__month', '__day')

    ...

    def __lt__(self, other):
        return self.year < other.year or \
            self.year == other.year \
                and self.month < other.month or \
            self.year == other.year \
                and self.month == other.month \
                and self.day < other.day

    ...
```

Atrybut `__slots__`

- Metody `__eq__` oraz `__le__`:

```
class Date:
    __slots__ = ('__year', '__month', '__day')

    ...

    def __eq__(self, other):
        return self.year == other.year \
            and self.month == other.month \
            and self.day == other.day

    def __le__(self, other):
        return self < other or self == other

    ...
```

Atrybut `__slots__`

- Metody `__str__` oraz `__repr__`:

```
class Date:
    __slots__ = ('__year', '__month', '__day')

    ...

    def __str__(self):
        return "{0:04d}-{1:02d}-{2:02d}".format\
             (self.year, self.month, self.day)

    def __repr__(self):
         return (f"{self.__module__}."
                 f"{self.__class__.__name__}"
                 f"({self.year}, {self.month}, {self.day})")

    ...
```

Atrybut `__slots__`

- Przykładowa sesja interaktywna:

```
>>> from date import Date
>>> d = Date(2022, 3, 26)
>>> e = Date(2022, 3, 27)
>>> d == e
False
>>> d != e
True
>>> d < e
True
>>> d > e
False
>>> d <= e
True
>>> d >= e
False
```

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)
```

Operacje wejścia-wyjścia

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)
```

Operacje wejścia-wyjścia

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.

Operacje wejścia-wyjścia

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.

Operacje wejścia-wyjścia

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.

Operacje wejścia-wyjścia

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:

Operacje wejścia-wyjścia

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()

    ...
```

Operacje wejścia-wyjścia

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`