Wykład 3

Zaawansowane konstrukcje języka
Programowanie obiektowe

dr inż. Robert Kazała

Programowanie obiektowe

- Tworzenie klasy

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)
```

- Utworzenie obiektu i wywołanie metody

```
p1 = Person("John", "Doe")
p1.printname()
```

Programowanie obiektowe

- Funkcja **__init __ ()** jest wywoływana automatycznie za każdym razem, gdy klasa jest używana do utworzenia nowego obiektu.
- Parametr **self** jest odniesieniem do bieżącej instancji klasy i służy do uzyskiwania dostępu do zmiennych należących do klasy. Parametr ten nie musi być nazwany self, możesz go nazwać dowolnie, ale musi to być pierwszy parametr dowolnej funkcji w klasie.
- Modyfikacja właściwości obiektu

```
p1.age = 40
```

- Usuwanie właściwości obiektu

```
del p1.age
```

- Usuwanie obiektu

```
del p1
```

Instrukcja pass

- Instrukcja pass

```
class Person:
    pass
```

Dziedziczenie

- Tworzenie klasy potomnej

```
class Student(Person):
    pass
```

- Utworzenie obiektu i wywołanie metody dla klasy potomnej

```
x = Student("Mike", "Olsen")
x.printname()
```

Dziedziczenie

- Dodanie metody __init__ dla klasy potomnej

```
class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)
```

- Python ma również funkcję super(), która sprawi, że klasa potomna odziedziczy wszystkie metody i właściwości od swojego rodzica.

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
```

Python – tworzenia odpowiednika struktury języka C

- Czasami warto mieć typ danych podobny do „rekordu” lub „struktury” Pascala, łącząc kilka nazwanych elementów danych.
- Można wykorzystać do tego definicję pustej klasy.

```
class Employee:
    pass

john = Employee()  # Utworzenie pustego rekordu

# Wypelnienie pol recordu
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

Zmienne prywatne

- „Prywatne” zmienne instancji, do których dostęp jest możliwy tylko z wnętrza obiektu, nie istnieją w Pythonie.
- Istnieje jednak konwencja, do której stosuje się większość kodu w języku Python: nazwa poprzedzona znakiem podkreślenia (np. **_spam**) powinna być traktowana jako niepubliczna część interfejsu API (niezależnie od tego, czy jest to funkcja, metoda czy element danych) . Należy to uznać za szczegół implementacji i może ulec zmianie bez powiadomienia.
- Ponieważ istnieje uzasadniony przypadek użycia dla członków klasy prywatnej (a mianowicie w celu uniknięcia kolizji nazw z nazwami zdefiniowanymi przez podklasy), istnieje ograniczone wsparcie dla takiego mechanizmu, zwanego przekręcaniem nazw (**name mangling**).
- Każdy identyfikator postaci __spam (co najmniej dwa wiodące podkreślenia, co najwyżejj jeden końcowy podkreślnik) jest tekstowo zastąpiony przez **_classname__spam**, gdzie nazwa klasy jest bieżącą nazwą klasy z usuniętymi wiodącymi podkreśleniem (podkreśleniami). To zniekształcanie odbywa się bez względu na składniową pozycję identyfikatora, o ile występuje w definicji klasy.

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update   # kopia prywatna oryginalnej metody update()
```

Funkcja object()

- Funkcja object() pozwala na uworzenie pustego obiektu.

- Przykład

```
x = object()
```

- Nie można dodawać nowych właściwości ani metod do tego obiektu.
- Ten obiekt jest podstawą wszystkich klas, zawiera wbudowane właściwości i metody, które są domyślne dla wszystkich klas.

Funkcja object()

- Bazową klasą w języku Python jest object

- Przykład

```
dir(object)
```

- Funkcja object() pozwala na uworzenie pustego obiektu.

- Przykład

```
x = object()
dir(x)
```

- Nie można dodawać nowych właściwości ani metod do tego obiektu.

- Przykład

```
x.a=10
```

- Ten obiekt jest podstawą wszystkich klas, zawiera wbudowane właściwości i metody, które są domyślne dla wszystkich klas.

Klasa object - dziedziczenie

- W przypadku klasy, która z tego objektu dziedziczy można dodawać nowe właściwości i metody do tego obiektu.

- Przykład

```
class test(object):
    pass


t=test()
t.a=10
t.a
dir(test)
```

Klasy w Python 2

- W Pythonie 2.x (od wersji 2.2) istnieją dwa style klas w zależności od obecności lub braku obiektu jako klasy podstawowej:

- klasy „klasyczne”: nie mają obiektu jako klasy podstawowej:

```
>>> class ClassicSpam:      # no base class
...     pass

>>> ClassicSpam.__bases__
()
```

Nowe klasy w Python 2

- „nowe” klasy stylów: mają, bezpośrednio lub pośrednio (np. dziedziczą po typie wbudowanym), obiekt jako klasę podstawową:

```
>>> class NewSpam(object):                  # directly inherit from object
...     pass
>>> NewSpam.__bases__
(<type 'object'>,)
>>> class IntSpam(int):                     # indirectly inherit from
object...
...     pass
>>> IntSpam.__bases__
(<type 'int'>,)
>>> IntSpam.__bases__[0].__bases__          # ... because int inherits from
object
(<type 'object'>,)
```

Nowe klasy w Python 2 i Python 3 - korzyści

- Korzyści z wykorzystywania klas nowego typu są następujące:
  - Obsługa deskryptorów. W szczególności możliwe są następujące konstrukcje za pomocą deskryptorów:
    - classmethod: Metoda, która odbiera klasę jako domyślny argument zamiast instancji.
    - staticmethod: Metoda, która nie odbiera domyślnego argumentu self jako pierwszego argumentu.
  - właściwości z property: Twórz funkcje do zarządzania uzyskiwaniem, ustawianiem i usuwaniem atrybutu.
  - __slots__: Oszczędza zużycie pamięci przez klasę, a także zapewnia szybszy dostęp do atrybutów. Oczywiście nakłada ograniczenia.

Nowe klasy w Python 2 i Python 3 - korzyści

- Korzyści z wykorzystywania klas nowego typu są następujące:
  - *Metoda statyczna __new__: pozwala dostosować sposób tworzenia nowych instancji klas.*
  - *Kolejność wywoływania metod (MRO): w jakiej kolejności będą wyszukiwane klasy podstawowe klasy podczas próby rozstrzygnięcia, którą metodę wywołać.*
  - *Związane z MRO, wywołania super.*

- Jeśli nie dziedziczy się po object, nie można tych mechanizmów wykorzystywać.

- Jedną z wad klas nowego stylu jest to, że sama klasa wymaga więcej pamięci. Jednak w wiekszości przypadków nie jest to problemem, a wykorzystanie klas w nowym stylu daję więcej korzyści.

Klasy w Pythonie 3

- W Pythonie 3 tworzenie klas jest uproszczone. Istnieją tylko klasy w nowym stylu (zwane po prostu klasami) i nie jest wymagane jawne wpisanie dziedziczenia po object. Dziedziczenie to zawsze jest realizowane.
- Definicja klasy

```
class ClassicSpam:
    Pass
```

- jest całkowicie równoważna

```
class NewSpam(object):
    pass
```

- oraz

```
class Spam():
    pass
```

Atrybuty specjalne klas

- Implementacja dodaje kilka specjalnych atrybutów tylko do odczytu do kilku typów obiektów, tam gdzie są one istotne. Niektóre z nich nie są zgłaszane przez wbudowaną funkcję dir ().
  - *object **.__ dict__** - Słownik lub inny obiekt odwzorowujący używany do przechowywania atrybutów obiektu (zapisywalnych).*
  - *instance.**__class__** - Klasa, do której należy instancja klasy.*
  - *class.**__bases__** - Krotka klas podstawowych obiektu klasy.*
  - *definition.**__name__** - Nazwa klasy, funkcji, metody, deskryptora lub instancji generatora.*
  - *definition.**__qualname__** - Kwalifikowana nazwa klasy, funkcji, metody, deskryptora lub instancji generatora.*
- Nowości w wersji 3.3.
  - *class.**__mro__** - Ten atrybut jest krotką klas, które są brane pod uwagę przy poszukiwaniu klas podstawowych podczas rozwiązywania metody.*
  - *class.**mro ()** - Metodę tę można zastąpić tę metodę, aby dostosować kolejność rozwiązywania metod dla jej instancji. Jest wywoływany przy tworzeniu instancji klasy, a jego wynik jest przechowywany w __mro__.*
  - *class.**__subclasses__()** - Każda klasa przechowuje listę słabych odniesień do swoich bezpośrednich podklas. Ta metoda zwraca listę wszystkich wciąż istniejących referencji.*

Dynamiczne definiowanie klas

- Wbudowana funkcja type() po przekazaniu jednego argumentu zwraca typ obiektu. W przypadku klas w nowym stylu jest to na ogół taki sam jak atrybut **__class__** obiektu:

```
>>> type(3)
<class 'int'>

>>> type(['foo', 'bar', 'baz'])
<class 'list'>

>>> t = (1, 2, 3, 4, 5)
>>> type(t)
<class 'tuple'>

>>> class Foo:
...     pass
...
>>> type(Foo())
<class '__main__.Foo'>
```

Dynamiczne definiowanie klas

- Można także wywołać type() z trzema argumentami — typ (<name>, <bases>, <dct>):
  - <name> określa nazwę klasy. Staje się to atrybutem __name__ klasy.
  - <bases> określa krotkę klas podstawowych, z których dziedziczy klasa. Staje się to atrybutem __bases__ klasy.
  - <dct> określa słownik przestrzeni nazw zawierający definicje treści klasy. Staje się to atrybutem __dict__ klasy.
- Wywołanie type() w ten sposób tworzy nowe wystąpienie metaklasy type. Innymi słowy, dynamicznie tworzy nową klasę.

Dynamiczne definiowanie klas – przykład 1

- W poniższym przykładzie górny fragment dynamicznie definiuje klasę za pomocą type (), podczas gdy dolny fragment definiuje klasę w zwykły sposób, za pomocą instrukcji class. W każdym przypadku dwa fragmenty są funkcjonalnie równoważne.
-
- Przykład 1

```
>>> Foo = type('Foo', (), {})

>>> x = Foo()
>>> x
<__main__.Foo object at 0x04CFAD50>



>>> class Foo:
...     pass
...
>>> x = Foo()
>>> x
<__main__.Foo object at 0x0370AD50>
```

Dynamiczne definiowanie klas – przykład 2

- W tym przykładzie <bases> jest krotką z jednym elementem Foo, określającą klasę nadrzędną, z której dziedziczy Bar. Atrybut, attr, jest początkowo umieszczany w słowniku przestrzeni nazw
- Przykład 1

```
>>> Bar = type('Bar', (Foo,), dict(attr=100))

>>> x = Bar()
>>> x.attr
100
>>> x.__class__
<class '__main__.Bar'>
>>> x.__class__.__bases__
(<class '__main__.Foo'>,)


>>> class Bar(Foo):
...     attr = 100
...

>>> x = Bar()
>>> x.attr
100
>>> x.__class__
<class '__main__.Bar'>
>>> x.__class__.__bases__
(<class '__main__.Foo'>,)
```

Python super()

- W Pythonie super() ma dwa główne przypadki użycia:
  - *Pozwala nam uniknąć jawnego używania nazwy klasy podstawowej*
  - *Ułatwia pracę z wielokrotnym dziedziczeniem*

- W przypadku pojedynczego dziedziczenia pozwala nam odwoływać się do klasy bazowej przez super().

```
class Mammal(object):
  def __init__(self, mammalName):
    print(mammalName, 'is a warm-blooded animal.')


class Dog(Mammal):
  def __init__(self):
    print('Dog has four legs.')
    super().__init__('Dog')


d1 = Dog()
```

Python super()

- W przedstawionym przypadku wywołanie

```
super().__init__('Dog')
```

- jest równoważne

```
Mammal.__init__(self, 'Dog')
```

- Ponieważ nie musimy podawać nazwy klasy bazowej, gdy wywołujemy jej członków, możemy łatwo zmienić nazwę klasy bazowej.

```
# zmiana nazwy klasy na CanidaeFamily
class Dog(CanidaeFamily):
  def __init__(self):
    print('Dog has four legs.')

    # nie trzeba zmieniać tej linii
    super().__init__('Dog')
```

Python super()

- Powyższa składnia jest dopuszczalna w Pythonie 3 gdzie można napisać super () .__ init __ (), natomiast w Python 2 wywołanie miałoby postać super(Dog, self) .__ init __ ().
- Kiedy piszemy klasę, chcemy aby inne klasy mogły z niej korzystać. super() ułatwia innym klasom korzystanie z klasy, którą napisaliśmy.
- Gdy inna klasa podklasuje napisaną przez nas klasę, może ona również dziedziczyć po innych klasach. I te klasy mogą mieć __init__, który pojawia się po tym __init__ na podstawie kolejności klas w algorytmie rozpoznawania metod.
- Bez super prawdopodobnie wywoływany byłby rodzic klasy, która jest nadpisywana (tak jak w przykładzie). Oznaczałoby to, że nie wywołano następnego __init__ w MRO, a zatem nie będzie można ponownie użyć kodu w nim.
- Jeśli piszemy własny kod do użytku osobistego, możemy nie dbać o to rozróżnienie. Ale jeśli chcemy, aby inni używali naszego kodu, użycie super jest jedną rzeczą, która pozwala użytkownikom na większą elastyczność.

Dziedziczenie wielokrotne - MRO

- Mechanizm „method resolution order“ (MRO) określa w języku Python sposób szukania odziedziczonych metod.
- Jest to przydatne, gdy wykorzystywana jest metoda super (), ponieważ MRO mówi dokładnie, gdzie Python będzie szukał metody, którą wywołuje za pomocą super () i w jakiej kolejności.
- Każda klasa ma atrybut .__ mro__, który pozwala nam sprawdzić kolejność wywoływania.

Dziedziczenie wielokrotne – funkcja super

- Funkcja super zachowuje się w szczególny sposób w sytuacji wielokrotnego dziedziczenia (czyli w sytuacji, gdy dana klasa ma więcej niż jednego rodzica). O ile każda z metod w hierarchii wywołuje metodę "nadrzędną" przez super, to wszystkie metody w hierarchii zostaną wywołane w pewnym ściśle określonym porządku.
- Zauważmy, że A.__init__ wywołuje B.__init__, mimo że klasa A zupełnie nic o klasie B nie wie. Niemniej, z punktu widzenia klasy C, to by obydwoje jej rodzice zostali poprawnie zainicjalizowani, jest absolutnie kluczowe.
- Po trzecie, nie musimy przekazywać self jako pierwszego argumentu wywołania metody. Oczywiście niewiele na tym zyskujemy, bo musimy przekazać self jako argument funkcji super.
- Możnaby pomyśleć, że takie przekazywanie oczywistych informacji, np. tego wewnątrz jakiej klasy metoda jest zdefiniowana, jest niepotrzebne i interpreter Pythona mógłby sam to stwierdzić. Tak się dzieje, ale dopiero w Pythonie 3. Piszemy po prostu super().

```
class A(object):
   def __init__(self):
     super(A, self).__init__()
     print( 'init A' )

class B(object):
   def __init__(self):
     super(B, self).__init__()
     print( 'init B' )

class C(A, B):
   def __init__(self):
     super(C, self).__init__()
     print( 'init C' )

c = C()
```

- Wynik wywołania

```
>>> init B
>>> init A
>>> init C
```

Dziedziczenie wielokrotne - przykład

- Przykład

```
class A(object):
       def foo(self, call_from):
           print ("foo from A, call from %s" % call_from)
           super().foo("A")

class B(A):
       def foo(self, call_from):
           print ("foo from B, call from %s" % call_from)
           super().foo("B")

class C(A):
       def foo(self, call_from):
           print ("foo from C, call from %s" % call_from)
           super().foo("C")

class F(object):
       def foo(self, call_from):
           print ("foo from F, call from %s" % call_from)

class D(B, C, F):
       def foo(self):
           print ("foo from D")
           super().foo("D")

d = D()
d.foo()

print( D.__mro__ )
```

Klasy abstrakcyjne

- Klasy abstrakcyjne to klasy zawierające jedną lub więcej metod abstrakcyjnych. Metoda abstrakcyjna to metoda zadeklarowana, ale nie zawierająca implementacji.
- Instancje klas abstrakcyjnych nie mogą być tworzone i wymagają podklas w celu zapewnienia implementacji metod abstrakcyjnych.
- Interfejs służy do określania zachowania, które klasy dziedziczące muszą wdrożyć.
- Klasy abstrakcyjne służą do implementacji interfejsów i są klasami, które zawierają jedną lub więcej metod abstrakcyjnych.
- Metoda abstrakcyjna to metoda zadeklarowana, ale nie zawierająca implementacji.
- W rzeczywistości Python nie zapewnia klas abstrakcyjnych.
- Jednak Python jest wyposażony w moduł, który zapewnia infrastrukturę do definiowania abstrakcyjnych klas podstawowych (ABC).
- Ten moduł nazywa się **abc**.

Klasa pseudoabstrakcyjna

- Przykładowa implementacja klasy

```
class AbstractClass:

    def do_something(self):
        pass


class B(AbstractClass):
    pass

a = AbstractClass()
b = B()
```

- Jeśli uruchomimy ten program, zobaczymy, że nie jest to klasa abstrakcyjna, ponieważ:
  - *możemy utworzyć instancję*
  - *nie jesteśmy zobowiązani do implementowania do_something w definicji klasy B*

Klasa abstrakcyjna

- Przykładowa implementacja klasy abstrakcyjnej

```
from abc import ABC, abstractmethod

class AbstractClassExample(ABC):

    def __init__(self, value):
        self.value = value
        super().__init__()

    @abstractmethod
    def do_something(self):
        pass
```

Klasa abstrakcyjna – błędne dziedziczenie

- Zdefiniujemy teraz podklasę przy użyciu wcześniej zdefiniowanej klasy abstrakcyjnej.
- Można zauważyć, że nie jest wdrożona metoda **do_something**, mimo że należy ją zaimplementować, ponieważ metoda ta jest dekorowana jako metoda abstrakcyjna za pomocą dekoratora „abstractmethod”.
- Otrzymujemy wyjątek, że DoAdd42 nie może zostać utworzony:

```
class DoAdd42(AbstractClassExample):
    pass

x = DoAdd42(4)

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-2bcc42ab0b46> in <module>
      2     pass
      3
----> 4 x = DoAdd42(4)

TypeError: Can't instantiate abstract class DoAdd42 with abstract methods do_something
```

Klasa abstrakcyjna – poprawne dziedziczenie

- Nie można utworzyć instancji klasy pochodnej z klasy abstrakcyjnej, chyba że wszystkie jej metody abstrakcyjne zostaną zastąpione.
- Zrobimy to poprawnie w poniższym przykładzie, w którym zdefiniujemy dwie klasy dziedziczące po naszej klasie abstrakcyjnej:

```
class DoAdd42(AbstractClassExample):

    def do_something(self):
        return self.value + 42

class DoMul42(AbstractClassExample):

    def do_something(self):
        return self.value * 42

x = DoAdd42(10)
y = DoMul42(10)

print(x.do_something())
print(y.do_something())
```

Klasa abstrakcyjna – metody abstrakcyjne

- Może się wydawać, że metod abstrakcyjnych nie można zaimplementować w abstrakcyjnej klasie bazowej.
- Wrażenie to jest błędne: metoda abstrakcyjna może mieć implementację w klasie abstrakcyjnej!
- Nawet jeśli zostaną zaimplementowane, projektanci podklas będą zmuszeni zastąpić implementację.
- Podobnie jak w innych przypadkach „normalnego” dziedziczenia, metodę abstrakcyjną można wywołać za pomocą mechanizmu wywołania super ().
- Umożliwia to zapewnienie podstawowych funkcji w metodzie abstrakcyjnej, które można wzbogacić o implementację podklasy.

Klasa abstrakcyjna – przykład metody abstrakcyjnej

- Przykład implementacji metody abstrakcyjnej

```
from abc import ABC, abstractmethod

class AbstractClassExample(ABC):

    @abstractmethod
    def do_something(self):
        print("Implementacja metody abstrakcyjnej!")

class AnotherSubclass(AbstractClassExample):

    def do_something(self):
        super().do_something()
        print("Metoda dziedzicząca")

x = AnotherSubclass()
x.do_something()
```

Deskryptory

- Deskryptor to klasa, której można użyć do wywołania metody z prostym dostępem do atrybutów,
- Deskryptor implementuje co najmniej jedną z tych trzech metod:
  - **__get __ ()**,
  - **__set __ ()**
  - **__delete __ ()**.
- Każda z tych metod ma listę niezbędnych parametrów i każdy z nich jest wywoływany przez inny rodzaj dostępu do atrybutu reprezentowanego przez deskryptor.
- Wykonanie prostego dostępu **a.x** wywoła metodę **__get __ ()** obiektu x; ustawienie atrybut używający **a.x = wartość** wywoła metodę **__set __ ()** obiektu x; i użycie **del a.x** wywoła, zgodnie z oczekiwaniami, metodę **__delete __ ()** z obiektu x.
- Od wersji 3.6 istnieje inna metoda o nazwie __set_name __ (), ale użycie tej metody nie tworzy deskryptora w sposób taki jak dla pozostałych metod. Ta metoda nie odgrywa tak dużej roli w działaniu deskryptorów.
- Jak wspomniano, tylko jedna z metod musi zostać zaimplementowana w aby być uważanym za deskryptor, ale może być dowolna ich liczba zaimplementowano.
- W zależności od typu deskryptora i metod, które są zaimplementowane, brak implementacji niektórych metod może ograniczyć pewne typy dostępu do atrybutów lub zapewnić im alternatywne zachowania.
- Istnieją dwa typy deskryptorów, na podstawie których wdrażane są zestawy tych metod: **data** i **non-data**.

Data Descriptors i Non-Data Descriptors

- Deskryptor danych implementuje co najmniej __set __ () lub __delete __ (), ale może zawierać jedno i drugie.
- Deskryptory danych często obejmują także __get __ (), ponieważ rzadko jest potrzebne ustawienie czegoś bez możliwości jego uzyskania.
- Możesz jednak uzyskać tę wartość, nawet jeśli deskryptor nie zawiera __get __ (), stosujac inne metody dostepu.
- Deskryptor niebędący danymi implementuje tylko __get __ (). Jeśli doda metodę __set __ () lub __delete __ (), stanie się deskryptorem danych.
- Warto zauważyć, że deskryptory są nieodłączną częścią tego, w jaki sposób Python działa.

Deskryptory w Pythonie

- Warto zauważyć, że deskryptory są nieodłączną częścią tego, w jaki sposób Python działa.
- Deskryptory są używane niejawnie w Pythonie dla mechanizmów obiektowych języka.
- Dzięki deskryptorom w Pythonie możliwe jest programowanie obiektowe.
- Deskryptory są bardzo zaawansowane i użyteczne.
- Deskryptory zajmują dużą część języka Python, ponieważ mogą zastępować dostęp do atrybutów wywołaniami metod, a nawet ograniczać, które typy dostępu do atrybutów są dozwolone.

Deskryptory w Pythonie cont.

- Domyślnym zachowaniem dostępu do atrybutu jest uzyskanie, ustawienie lub usunięcie atrybutu ze słownika obiektu.
- Na przykład a.x ma łańcuch odnośników rozpoczynający się od .__ dict __ ['x'], a następnie type(a).__ dict __ ['x'] i kontynuuje przez bazowe klasy type(a) z wyłączeniem metaklas.
- Jeśli szukana wartość jest obiektem definiującym jedną z metod deskryptora, Python może zastąpić domyślne zachowanie i zamiast tego wywołać metodę deskryptora.
- To, co dzieje się w łańcuchu pierwszeństwa, zależy od tego, które metody deskryptora zostały zdefiniowane.
- Deskryptory to protokół ogólnego przeznaczenia. Są mechanizmem stojacym za właściwościami, metodami, metodami statycznymi, metodami klas i super().

Descriptor Protocol

- descr.__get__(self, obj, type=None) -> value
- descr.__set__(self, obj, value) -> None
- descr.__delete__(self, obj) -> None
- Deskryptory danych i nie-danych różnią się sposobem obliczania zastąpień w odniesieniu do pozycji w słowniku instancji.
- Jeśli słownik instancji ma pozycję o tej samej nazwie co deskryptor danych, pierwszeństwo ma deskryptor danych.
- Jeśli słownik instancji ma pozycję o takiej samej nazwie jak deskryptor nie-danych, pozycja słownika ma pierwszeństwo.

Dostęp do deskryptorów

- Deskryptor można wywołać bezpośrednio po nazwie jego metody. Na przykład d .__ get __ (obj).
- Alternatywnie częściej deskryptor jest wywoływany automatycznie po uzyskaniu dostępu do atrybutu.
- Na przykład obj.d wyszukuje d w słowniku obj. Jeśli d definiuje metodę __get __ (), to d .__ get __ (obj) jest wywoływane zgodnie z regułami pierwszeństwa wymienionymi poniżej.
- Szczegóły wywołania zależą od tego, czy obj jest obiektem, czy klasą.
- W przypadku obiektów mechanizm znajduje się w object.__ getattribute __ (), który przekształca b.x w typ (b) .__ dict __ ['x'] .__ get __ (b, typ (b)).
- Implementacja działa poprzez łańcuch pierwszeństwa, który daje deskryptorom danych pierwszeństwo przed zmiennymi instancji, priorytetem zmiennych instancji nad deskryptorami nie-danymi i przypisuje najniższy priorytet __getattr __ (), jeśli jest dostępny. Pełną implementację C można znaleźć w PyObject_GenericGetAttr () w Objects / object.c.

Przykład z deskryptorem

- Poniższy kod tworzy klasę, której obiektami są deskryptory danych, które wypisują komunikat dla każdego pobrania lub ustawienia wartości.
- Przesłanianie __getattribute __ () to alternatywne podejście, które może to zrobić dla każdego atrybutu. Jednak jest to mniej elastyczne podejście:

```
class Descriptor:


    def init(self):
        self.name = ''


    def get(self, instance, owner):
        print "Getting: %s" % self.name
        return self.name


    def _set(self, instance, name):
        print "Setting: %s" % name
        self._name = name.title()


    def __delete(self, instance):
        print "Deleting: %s" %self._name
        del self._name


class Person:
    name = Descriptor()
```

```
>>> user = Person()

>>> user.name = 'john smith'

Setting: john smith

>>> user.name

Getting: John Smith

'John Smith'

>>> del user.name

Deleting: John Smith
```

Przykład z deskryptorem

```
class Descriptor:

    def __init__(self):

        self.__fuel_cap = 0

    def __get__(self, instance, owner):

        return self.__fuel_cap

    def __set__(self, instance, value):

        if isinstance(value, int):

            print(value)

        else:

            raise TypeError("Fuel Capacity can only be
an integer")



        if value < 0:

            raise ValueError("Fuel Capacity can never
be less than zero")



        self.__fuel_cap = value



    def __delete__(self, instance):

        del self.__fuel_cap
```

```
class Car:

    fuel_cap = Descriptor()

    def __init__(self,make,model,fuel_cap):

        self.make = make

        self.model = model

        self.fuel_cap = fuel_cap



    def __str__(self):

        return "{0} model {1} with a fuel capacity of
{2} ltr.".format(self.make,self.model,self.fuel_cap)



car2 = Car("BMW","X7",40)

print(car2)



40

BMW model X7 with a fuel capacity of 40 ltr.
```

Przykład z deskryptorem

```
class RevealAccess:
    """A data descriptor that sets and returns values
       normally and prints a message logging their access.
    """
    def __init__(self, initval=None, name='var'):
        self.val = initval
        self.name = name

    def __get__(self, obj, objtype):
        print('Retrieving', self.name)
        return self.val

    def __set__(self, obj, val):
        print('Updating', self.name)
        self.val = val
```

Przykład z deskryptorem

```
>>> class MyClass:
...     x = RevealAccess(10, 'var "x"')
...     y = 5
...
>>> m = MyClass()
>>> m.x
Retrieving var "x"
10
>>> m.x = 20
Updating var "x"
>>> m.x
Retrieving var "x"
20
>>> m.y
5
```