PROGRAMOWANIE ZORIENTOWANE OBIEKTOWO

Plan szkolenia

Plan modułu

TEORIA

1 **KLASY I OBIEKTY**

- paradygmat OOP (Object-Oriented Programming)
- klasy i obiekty – podstawy
- atrybuty klasy
- metody
- instancje klasy
- atrybuty instancji

Programowanie zorientowane obiektowo

- Python jest **językiem zorientowanym obiektowo** (*OOL – Object-Oriented Language*)
- Nie wymusza stosowania tego paradygmatu na wyłączność
- Wspiera także **programowanie proceduralne** oparte na modułach i funkcjach
- Można samodzielnie wybrać właściwy paradygmat dla każdej części programu

Klasy i instancje

- Podobnie jak w innych językach zorientowanych obiektowo, **klasa** (*class*) jest właściwie definicją **typu danych** (*data type*) – wszystkie wbudowane typy danych w Pythonie są klasami
- Klasy można **instancjonować** (*instantiate*) w celu utworzenia **instancji**, czyli obiektów tego typu

- W Pythonie te idee realizuje się z pomocą **obiektów klasy** i **obiektów instancji** (*class and instance objects*)

Programowanie OOP

- Paradygmat programowania zorientowanego obiektowo (OOP) umożliwia zgrupowanie w pojedynczych jednostkach funkcjonalnych, zw. **klasami**:

stanu (*state*)	czyli danych
zachowania (*behavior*)	czyli kodu operacji na danych

- Klasę można traktować jak **szablon** (prototyp) w oparciu o który tworzone są instancje
- Klasa definiuje **zbiór atrybutów** charakteryzujących wszystkie instancje klasy

Klasy w Pythonie

- Klasa jest **obiektem Pythona**
- Jest pełnoprawnym obiektem, z wszystkimi możliwościami, jakimi dysponują inne obiekty – jest tzw. **obiektem pierwszej kategorii** (*first-class object, first-class citizen*)
- Z tego powodu:
  - klasę można przekazać jako argument w wywołaniu funkcji
  - funkcja może w wyniku wywołania zwrócić obiekt klasy
  - klasa może zostać związana ze zmienną (lokalną bądź globalną)
  - klasa może być elementem kontenera lub pełnić rolę klucza w słowniku
  - klasa może być atrybutem obiektu

Ogólna definicja klasy

Składnia definicji klasy

```
class ClassName(base-classes):
    statement(s)
```

ClassName	identyfikator reprezentujący nazwę klasy
base-classes	**opcjonalna** krotka obiektów klas, reprezentujących nadklasy w razie pominięcia nadklas, można także opuścić nawiasy (☞ więcej nt. dziedziczenia w dalszej części rozdziału. . .)
statements	**niepusta** sekwencja instrukcji, stanowiących ciało klasy

Nazwa klasy

- **Nazwa klasy** musi być poprawnym identyfikatorem
- Zgodnie z umową, w stosunku do klas stosujemy konwencję nazewniczą *CamelCase*

- Najprostsza klasa ma postać:

Definicja klasy – przykład

```
>>> class EmptyClass: pass
...
```

Atrybuty obiektów klas

Definicja klasy z atrybutami – przykład

```
>>> class A:
...     x = 123     # atrybut klasy
...     y = x + 1   # atrybut klasy
...
>>> print(A.x, A.y)
123 124
```

- Instrukcje w ciele klasy zawierające odwołania do atrybutów klasy wykorzystują **nazwy proste** (niekwalifikowane)
- Na zewnątrz klasy do odwołania się do jej atrybutu wykorzystuje się **nazwy kwalifikowane**

Atrybuty obiektów klas

- Atrybuty obiektu klasy można także tworzyć i zmieniać poza ciałem klasy:

Definicja klasy

```
>>> class EmptyClass: pass
...
>>> EmptyClass.x = 123  # atrybut
>>> print(EmptyClass.x)
123
```

- Tak utworzony atrybut nie różni się niczym od atrybutu utworzonego w ciele klasy

Niejawne atrybuty obiektów klas

- Instrukcja definicji klasy (instrukcja `class`) tworzy obiekt klasy i ustawia szereg **niejawnych atrybutów** klasy
- Wśród nich są:

ATRYBUT	ZNACZENIE
__name__	nazwa klasy
__bases__	krotka obiektów rozszerzanych klas
__doc__	opis klasy
__dict__	słownik służący klasie jako lokalna przestrzeń nazw
__module__	nazwa modułu w którym zdefiniowana jest klasa w trybie interaktywnym ten atrybut ma wartość *__main__*

Odczyt atrybutu klasy

- Odczyt atrybutu klasy, gdy nie jest on deskryptorem, przebiega następująco:

Odczyt atrybutu klasy

```
>>> class A:
...     x = 123
...
>>> # odczyt atrybutu klasy
... print(A.x)
123

>>> # równoważnie:
... print(A.__dict__['x'])
123
```

Metody

- Atrybuty klasy mogą odnosić się także do **obiektów funkcji** (instrukcja `def`)
- Atrybuty klasy związane z funkcjami nazywa się **metodami** (*methods, callable attributes*)
- Atrybuty klasy są **współdzielone** przez wszystkie instancje klasy

Metody instancyjne

- Metody, które chcą mieć dostęp do atrybutów danej instancji muszą deklarować w liście parametrów, na pierwszej pozycji, **obowiązkowy parametr**, odnoszący się do **bieżącej instancji**, tzn. tej na której metoda jest wywoływana
- Zgodnie z konwencją temu parametrowi nadaje się nazwę *self*
- Takie metody nazywamy **metodami instancyjnymi** (*instance methods*)

Metody instancyjne

Definicja klasy z metodą instancyjną – przykład

```
>>> class MyClass:
...     # metoda instancyjna
...     def hello(self):
...         print('Hello')
...
```

- Poprzez parametr *self* **metody instancyjne mają swobodny dostęp do atrybutów i innych metod tego samego obiektu**
- Metody instancyjne woła się na rzecz instancji

Instancje klasy

- Aby utworzyć **instancję klasy** (*class instance*), należy wywołać **obiekt klasy**, tak, jakby była to funkcja (funkcja konstruktorowa)
- Każde takie wywołanie zwraca nową instancję typu klasy
- Ten proces nazywamy **instancjonowaniem** (*instantiation*)

Tworzenie instancji klasy *MyClass*

```
>>> mc = MyClass()     # tworzenie instancji
>>> mc.hello()         # wywołanie metody instancyjnej
Hello
>>>
```

Instancje klasy

- Wbudowana funkcja *isinstance* może posłużyć do weryfikacji, **czy instancja jest danego typu** lub typu podklasy

Test typu instancji

```
>>> isinstance(mc, MyClass)  # test typu instancji
True
```

Metoda specjalna *__new__*

- Każda klasa posiada własną metodę specjalną *__new__* lub ją dziedziczy

Składnia metody *__new__*

```
__new__(cls[, ...])
```

- Metoda służy do **tworzenia nowych, niezainicjowanych instancji klasy**
- Pierwszy z parametrów *cls* jest obiektem klasy, której instancja jest tworzona
- Opcjonalne argumenty przekazane metodzie *__new__* są przez nią ignorowane i przekazywane metodzie specjalnej *__init__*

Metoda specjalna *__init__*

- Podczas tworzenia instancji, Python **niejawnie wywołuje specjalną metodę** *__init__* (*“dunder init”*)

Składnia metody __init__

```
__init__(self[, args...])
```

- Można tę metodę zdefiniować samodzielnie (nadpisać) lub ją dziedziczyć

Tworzenie instancji

- Przykładowa instrukcja utworzenia instancji klasy *A*:

Tworzenie instancji

```
>>> a = A(123)
```

jest równoważna:

Tworzenie instancji

```
>>> a = A.__new__(A, 123)
>>> if isinstance(a, A):
...     type(a).__init__(a, 123)
...
```

Metoda specjalna *__init__*

- Metoda *__init__* umożliwia dokonanie wszelkiej niezbędnej **inicjalizacji instancji** (najczęściej utworzenia i zainicjowania atrybutów)
- Argumenty podawane podczas tworzenia instancji są przekazywane metodzie *__init__*
- Metoda *__init__* nie może zwrócić żadnej innej wartości niż `None` – w przeciwnym razie zostanie zgłoszony wyjątek *TypeError*

Metoda specjalna *__init__* – przykład

- Po utworzeniu instancji można uzyskać dostęp do jej atrybutów (danych i metod):

Tworzenie instancji klasy

```
>>> class A:
...     def __init__(self, n):
...         print('utworzono instancję klasy A')
...         self.x = n
...

>>> a = A(123)
utworzono instancję klasy A

>>> print(a.x)      # atrybut instancyjny
123
```

Atrybuty instancji

- Atrybuty instancji można także dodać do już istniejącej instancji

Dodawanie atrybutów

```
>>> class A: pass
...
>>> a = A()
>>> a.x = 123
>>> print(a.x)
123
```

Atrybuty instancji

- Utworzenie instancji **niejawnie ustawia** dwa **atrybuty instancji**:

ATRYBUT	ZNACZENIE
__class__	obiekt klasy, do którego należy dana instancja
__dict__	słownik do przechowywania pozostałych atrybutów instancji

Atrybuty instancji

```
>>> print(a.__class__.__name__)
A

>>> print(a.__dict__)
{'x': 123}
```

Odczyt atrybutu instancji

Odczyt atrybutu instancji

```
>>> class A:
...     def __init__(self, value):
...         self.x = value   # atrybut instancji
...
>>> a = A(123)

>>> # odczyt atrybutu
>>> print(a.x)
123

>>> # równoważnie:
>>> print(a.__dict__['x'])
123
```

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**

Ćwiczenia/przykłady

ĆWICZENIA

- Ćwiczenie/przykład 1.1:
  Definiowanie klas i obiektów

Plan szkolenia

Plan modułu

TEORIA

2 HERMETYZACJA

- kontrola dostępu do atrybutów
- definiowanie i wykorzystanie właściwości (properties)

Prywatne identyfikatory

- Identyfikatory rozpoczynające się pojedynczym znakiem podkreślenia _ służą do definiowania **prywatnych** zmiennych, funkcji, metod i klas
- Niektóre instrukcje importu pomijają takie identyfikatory → prywatność na poziomie modułu
- **Zgodnie z konwencją**, są one prywatne względem zasięgu w którym zostały zdefiniowane

Prywatne identyfikatory

- Nie jest to realizacja “prawdziwej” prywatności – elementy są nadal dostępne w sposób bezpośredni z innych modułów
- Kompilator **nie wymusza i nie pilnuje w żaden sposób realizacji prywatności** – jest to **tylko umowa** respektowana przez programistów
- Rolę znaku podkreślenia dość dobrze oddaje termin *“weak internal-use indicator”*

Silnie prywatne identyfikatory

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

  $__identifier \rightarrow \_ClassName__identifier$
- Zmniejsza to ryzyko przypadkowego zdublowania nazw atrybutów, metod, czy zmiennych globalnych
- Ma to szczególne zastosowanie w podklasach

Silnie prywatne identyfikatory – przykład

Dostęp do atrybutów

```
>>> class A:
...     x = 111
...     _y = 222    # atrybut prywatny
...     __z = 333   # atrybut silnie prywatny
...
>>> print(A.x)
111
>>> print(A._y)
222
>>> print(A.__z)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'A' has no attribute '__z'
>>> print(A._A__z)
333
```

Hermetyzacja danych

- W wielu sytuacjach stajemy przed problemem **kontroli dostępu do atrybutów**
- W językach zorientowanych obiektowo można ten problem rozwiązać za pomocą **hermetyzacji** (*encapsulation*)
- Hermetyzacja polega na zablokowaniu bezpośredniego dostępu do danych obiektu z zewnątrz – dane obiektu należy zadeklarować jako prywatne
- Modyfikacja danych z zewnątrz jest możliwa tylko poprzez wywołanie **metod dostępowych**
- Przyjęło się nazywać metody służące do ustawiania wartości – **setterami**, a do odczytu danych – **getterami**

Hermetyzacja

- Trywialna realizacja tego pomysłu mogłaby wyglądać następująco:

Hermetyzacja

```
>>> class A:
...     def __init__(self, value):
...         self.__x = value
...     def get_x(self):
...         return self.__x
...     def set_x(self, value):
...         self.__x = value
...
```

Ćwiczenia/przykłady

ĆWICZENIA

- Ćwiczenie/przykład 2.1:
  Hermetyzacja

Hermetyzacja

- Takie rozwiązanie jest mało eleganckie – dostęp do danych jest skomplikowany i mało czytelny (zwł. modyfikacja danych)
- Usunięcie metod dostępowych i uczynienie atrybutu *x* publicznym, może uprościć kod, ale... to rozwiązanie **nie umożliwia kontroli nad dopuszczalnymi wartościami atrybutu**
- Taki walidujący kod można zawrzeć tylko w metodzie (najlepiej w setterze)...

Hermetyzacja

Dostęp do atrybutu

```
>>> class A:
...     def __init__(self, value):
...         self.x = value
...     def get_x(self):
...         return self.x
...     def set_x(self, value):
...         if value < 0:
                self.x = 0
            elif value > 9:
                self.x = 9
            else:
...             self.x = value
...
```

Wykorzystanie właściwości

- Niestety, ten kod nadal nie jest poprawny
- Klasa nie jest shermetyzowana i dostęp do atrybutu *x* jest możliwy zarówno poprzez getter i setter, jak i w sposób bezpośredni
- Rozwiązaniem tego problemu jest zdefiniowanie atrybutu *x* jako **właściwości** (*property*)
- Właściwości umożliwiają dostęp bezpośredni do wartości, ale faktycznie, w zależności od kontekstu, wywoływana jest odpowiednia metoda dostępowa
- Jednym ze sposobów utworzenia właściwości jest zastosowanie funkcji *property*

Definiowanie właściwości

Definiowanie właściwości

```
>>> class A:
...     def __init__(self, value):
...         self.__set_x(value)
...     def __get_x(self):
...         return self.__x
...     def __set_x(self, value):
...         if value < 0:
...             self.__x = 0
...         elif value > 9:
...             self.__x = 9
...         else:
...             self.__x = value
...     x = property(__get_x, __set_x)
...
```

Użycie właściwości

Użycie właściwości

```
>>> a = A(6)      # setter
>>> print(a.x)    # getter
6
>>> a.x *= 2      # getter + setter
>>> print(a.x)    # getter
9
>>> a.x = 5       # setter
>>> print(a.x)    # getter
5
>>> a.x = -8      # setter
>>> print(a.x)    # getter
0
```

Ćwiczenia/przykłady

ĆWICZENIA

- Ćwiczenie/przykład 2.2:
  Wykorzystanie właściwości

Plan szkolenia

Plan modułu

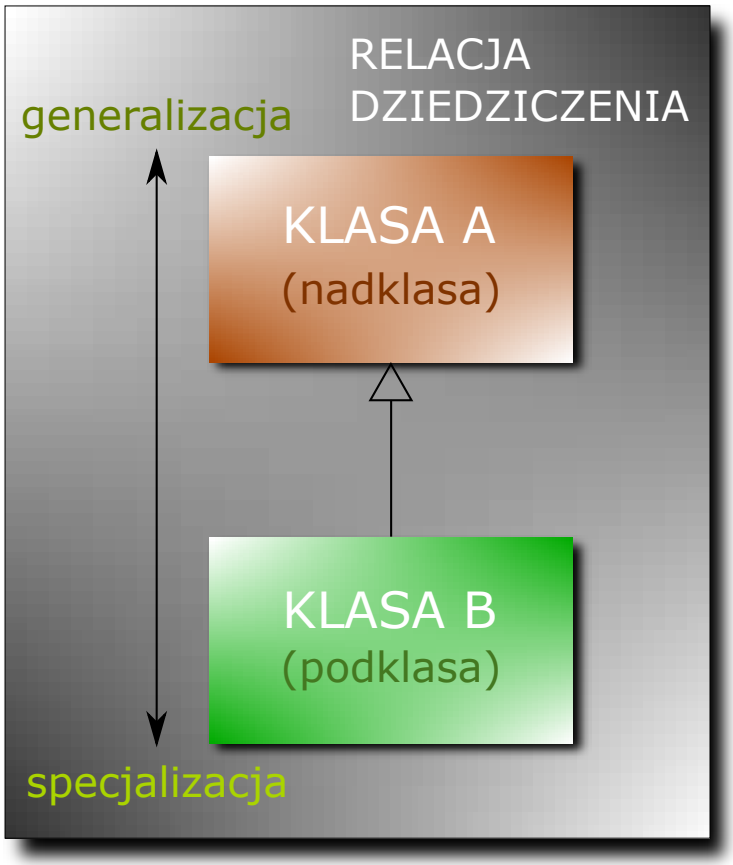
TEORIA

3 DZIEDZICZENIE

- dziedziczenie
- porządek poszukiwania atrybutów
- metody magiczne

Relacja dziedziczenia

- Jedną z zalet podejścia obiektowego jest możliwość **specjalizacji klasy** (*class specialization*), czyli wyprowadzenia nowej klasy na bazie klasy istniejącej
- Nowa klasa **dziedziczy** wszystkie **atrybuty** (dane i metody) z klasy bazowej



- Klasę bazową nazywamy **nadklasą** (*superclass*), zaś klasę potomną – **podklasą** (*subclass*)

Rodzaje dziedziczenia

- Gdy klasa może posiadać tylko jednego przodka, to mówimy o **dziedziczeniu jednobazowym** (*single inheritance*)
- Jedynym wyjątkiem od tej reguły jest klasa będąca przodkiem wszystkich innych klas – ta, jako jedyna nie posiada swojego "rodzica"
- Taka struktura zależności prowadzi do drzewa hierarchii dziedziczenia
- Jeśli dopuścimy możliwość posiadania przez klasę wielu przodków, to mamy do czynienia z **dziedziczeniem wielobazowym** (*multiple inheritance*)

Relacja dziedziczenia

- Klasa potomna może **dodać nowe** lub **przedefiniować istniejące** atrybuty
- Można **dziedziczyć**, tzn. **rozszerzać** dowolne klasy: wbudowane, ze standardowych bibliotek oraz własne
- Dziedziczenie ułatwia tworzenie kodu, gdyż można wykorzystać istniejącą i przetestowaną funkcjonalność jako bazę dla nowych klas
- Instancje klas potomnych można przekazywać do funkcji i metod napisanych dla oryginalnej klasy $\rightarrow$ polimorfizm

Dziedziczenie w Pythonie

- Dziedziczenie jest mechanizmem **współdzielenia funkcjonalności** pomiędzy klasami
- W Pythonie mamy do czynienia z **dziedziczeniem wielobazowym** – klasy mogą posiadać wielu przodków
- Definiując klasę, za jej nazwą można wyliczyć wszystkich jej bezpośrednich przodków (rodziców)
- Zawartość listy z klasami bazowymi jest dostępna poprzez atrybut *__bases__*

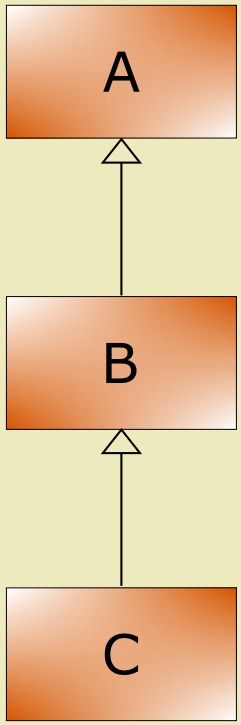
Relacja dziedziczenia

- Relacja dziedziczenia jest relacją **przechodnią** – jeśli klasa *C* rozszerza klasę *B*, a klasa *B* rozszerza klasę *A*, to klasa *C* rozszerza również klasę *A*
- Do testowania relacji dziedziczenia można wykorzystać wbudowaną funkcję *issubclass*

Relacja dziedziczenia – przykład

Relacja dziedziczenia

```
>>> class A: pass
...
>>> class B(A): pass
...
>>> class C(B): pass
...
>>> print(issubclass(C, C))
True
>>> print(issubclass(C, B))
True
>>> print(issubclass(C, A))
True
>>> print(issubclass(B, A))
True
>>> print(issubclass(A, B))
False
```



Dziedziczenie w Pythonie

- Klasy bazowe mogą mieć także swoich rodziców, itd.
- Pominięcie w definicji klasy jej klas bazowych oznacza, że tworzona klasa **niejawnie rozszerza** klasę *object*
- Klasa *object* jest **nadklasą wszystkich klas** Pythona

Klasy bazowe

```
>>> class A(object): pass
...
>>> class B: pass
...
>>> print(A.__bases__, B.__bases__)
(<class 'object'>,) (<class 'object'>,)
```

Dziedziczenie

- Klasa dziedzicząca po innej klasie uzyskuje dostęp do atrybutów klasy przodka

Dziedziczenie

```
>>> class A:
...     x = 1
...     def f(self): return 'f() called from A'
...
>>> class B(A):
...     y = 2
...     def g(self): return 'g() called from B'
...
>>> print(B.x, B.y)
1 2
>>> b = B()
>>> print(b.f())
f() called from A
>>> print(b.g())
g() called from B
```

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 funkcji *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

Nadpisywanie atrybutów – przykład

Nadpisywanie atrybutów

```
>>> class A:
...     a = 111
...     b = 222
...     def f(self): print('metoda f() z klasy A')
...     def g(self): print('metoda g() z klasy A')
...
>>> class B(A):
...     b = 333 # nadpisanie
...     c = 444
...     d = 555
...     def g(self): print('metoda g() z klasy B') # nadpisanie
...     def h(self): print('metoda h() z klasy B')
...
```

Nadpisywanie atrybutów – przykład cd.

Nadpisywanie atrybutów

```
>>> print(B.a, B.b, B.c, B.d)
111 333 444 555

>>> b = B()

>>> b.f()
metoda f() z klasy A

>>> b.g()
metoda g() z klasy B

>>> b.h()
metoda h() z klasy B
```

Delegacja do metod nadklasy

- Czasami przedefiniowująca metoda w podklasie może chcieć wykorzystać zachowanie oryginalnej metody nadpisywanej
- W takiej sytuacji można wykorzystać **delegację** (*delegation*)
- Metoda może odwołać się do obiektu metody z klasy bazowej i przekazać jej potrzebne argumenty (włącznie z *self*)

Delegacja do metod nadklasy – przykład

Delegacja do metod nadklasy

```
>>> class A:
...     def greet(self, name): print('Welcome', name)
...
>>> class B(A):
...     def greet(self, name):
...         A.greet(self, name)    # delegacja
...         print('It\'s nice to see you')
...
>>> b = B()
>>> b.greet('John')
Welcome John
It's nice to see you
```

Inicjalizacja podklasy

- Jeśli klasa potomna przedefiniuje metodę *__init__*, to zostanie ona wywołana podczas tworzenia instancji tej klasy
- Metoda automatycznie nie wywoła przedefiniowanej metody z nadklasy, w efekcie czego instancja może nie być do końca poprawnie zainicjalizowana
- Wywołanie metody *__init__* z nadklasy jest **odpowiedzialnością programisty**

Inicjalizacja podklasy – przykład

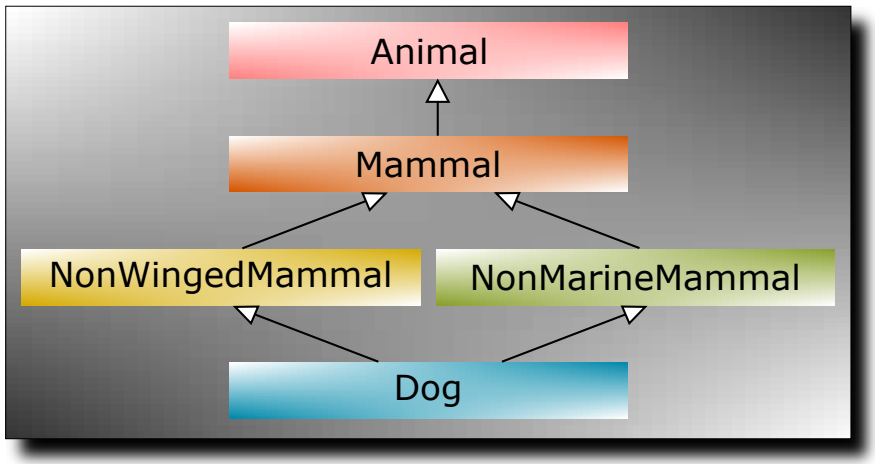
Przedefiniowanie metody *__init__*

```
>>> class Point2D:
...     def __init__(self, x_coord, y_coord):
...         self.x = x_coord
...         self.y = y_coord
...
>>> class Point3D(Point2D):
...     def __init__(self, x_coord, y_coord, z_coord):
...         Point2D.__init__(self, x_coord, y_coord)
...         self.z = z_coord
...
>>> p = Point3D(1, 2, 3)
>>> print(p.x, p.y, p.z)
1 2 3
```

Nadpisywanie metod w dziedziczeniu wielobazowym

- Przedstawiony sposób delegacji będzie działał zarówno w Pythonie 3.x, jak i 2.x

- Problematyczne może być jego zastosowanie w przypadku dziedziczenia wielobazowego, gdy grafy mają kształt diamentu (*diamond-shaped graphs*)



- Może to prowadzić do wielokrotnego wywołania tej samej metody

Nadpisywanie metod w dziedziczeniu wielobazowym

- Rozwiązaniem tego problemu może być użycie wbudowanego typu *super*
- Wywołanie *super()* zwraca obiekt proxy, który umożliwia wywoływanie metod klasy rodzica poprzez delegację
- W Pythonie 2.x wywołanie musiało mieć postać: *super(subclass, obj)*
- W Pythonie 3.x wywołanie może być bezargumentowe

Ćwiczenia/przykłady

ĆWICZENIA

- Ćwiczenie/przykład 3.1:

  Dziedziczenie – pracownicy i kierownicy zespołów

Metody specjalne

- Klasa może dziedziczyć lub przedefiniować metody specjalne (*dunder methods, magic methods*)
- Każda metoda specjalna jest związana z jakąś specyficzną operacją
- Python niejawnie wywołuje te metody podczas wykonywania operacji
- W większości przypadków wartość zwracana przez metodę specjalną jest wynikiem operacji – próba wykonania operacji, gdy brak odpowiedniej metody specjalnej skutkuje zgłoszeniem wyjątku
- Klasa, która definiuje lub dziedziczy te metody, umożliwia swoim instancjom kontrolę powiązanych operacji

Metody klasy *object*

- Do sprawdzenia jakie metody są dostępne w klasie *object* można wykorzystać poniższe wyrażenie listowe:

Metody klasy *object*

```
>>> m = [method_name for method_name in dir(object)
...                  if callable(getattr(object, method_name))]
>>> print(m)
['__class__', '__delattr__', '__dir__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__']
```

Podstawowe metody specjalne

METODA SPECJALNA	SPOSÓB UŻYCIA	ZNACZENIE
__bool__(self)	*bool(x)*	zwraca wartość logiczną dla *x*
__format__(self, fmt_spec)	*"{0}".format(x)*	umożliwia użycie *str.format()* w stosunku do własnych klas
__hash__(self)	*hash(x)*	umożliwia, aby *x* był kluczem słownika lub elementem zbioru
__init__(self, args)	*x = X(args)*	wywoływana podczas inicjalizacji obiektu
__new__(cls, args)	*x = X(args)*	wywoływana podczas tworzenia obiektu
__del__(self)	*del x*	może być wywołana (nie ma takiej gwarancji), gdy liczba referencji do obiektu spadnie do 0
__repr__(self)	*repr(x)*	zwraca reprezentację tekstową *x* taką, aby w miarę możliwości *eval(repr(x)) == x*
__repr__(self)	*ascii(x)*	zwraca reprezentację tekstową *x* zawierającą jedynie znaki ASCII
__str__(self)	*str(x)*	zwraca reprezentację tekstową *x* przeznaczoną dla człowieka

Metody specjalne związane z porównywaniem

METODA SPECJALNA	SPOSÓB UŻYCIA	ZNACZENIE
__lt__(self, other)	*x < y*	zwraca *True*, jeśli *x* jest mniejsze niż *y*
__le__(self, other)	*x <= y*	zwraca *True*, jeśli *x* jest mniejsze lub równe *y*
__eq__(self, other)	*x == y*	zwraca *True*, jeśli *x* jest równe *y*
__ne__(self, other)	*x != y*	zwraca *True*, jeśli *x* nie jest równe *y*
__gt__(self, other)	*x > y*	zwraca *True*, jeśli *x* jest większe niż *y*
__ge__(self, other)	*x >= y*	zwraca *True*, jeśli *x* jest większe lub równe *y*

Zachowanie nowych klas użytkownika

- Standardowo instancje własnych klas użytkownika **wspierają operator porównania** *==*
- Jeśli klasa nie przedefiniuje metody *__eq__* (aby zaimplementować operator *==*), to odziedziczy ją z klasy *object*
- Wtedy standardowo instancje będą porównywane na podstawie ich tożsamości (funkcja *id*)
- Z tego powodu wynikiem porównania tych obiektów zawsze będzie `False`

Zachowanie nowych klas użytkownika

- Metoda skrótu *__hash__* zwraca wartość całkowitą opartą na wartości obiektu
- Ponieważ standardowo wartością obiektu jest jego tożsamość, to z tego powodu wszystkie instancje (które nie nadpiszą metody *__hash__*) są:
  - **niemutowalne** (wartość obiektu nie ulega zmianie w całym cyklu życia)
  - **haszowalne**
- Takie obiekty mogą być elementami zbiorów oraz kluczami słowników

Zachowanie nowych klas użytkownika

Test instancji własnych klas

```
>>> class A:
...     def __init__(self, value):
...         self.x = value
...
>>> a1 = A(1)
>>> a2 = A(1)
>>> print(a1 == a2)    # porównanie instancji
False

>>> s = {a1, a2}       # instancje są haszowalne
>>> print(hash(a1))
154757017969
>>> print(hash(a2))
154757017899
```

Zachowanie nowych klas użytkownika

- Przedefiniowanie w klasie metody specjalnej *__eq__* powoduje, że instancje **nie są haszowalne**
- Python zablokuje standardową implementację funkcji skrótu, gdyż zmieniła się definicja wartości obiektu

- Aby instancje były haszowalne, trzeba dodatkowo przedefiniować metodę *__hash__*

Zachowanie nowych klas użytkownika

Test instancji własnych klas

```
>>> class A:
...     def __init__(self, value):
...         self.x = value
...     def __eq__(self, other):
...         return (self.__class__ == other.__class__
...             and self.x == other.x)
...     def __hash__(self):
...         return hash(self.x)
...
>>> a1 = A(1)
>>> a2 = A(1)
>>> print(a1 == a2)    # porównanie instancji
True
>>> s = {a1, a2}       # instancje są haszowalne
```

Numeryczne metody specjalne

METODA SPECJALNA	SPOSÓB UŻYCIA	ZNACZENIE
__abs__(self)	*abs(x)*	wartość bezwzględna
__int__(self)	*int(x)*	konwersja do typu *int*
__float__(self)	*float(x)*	konwersja do typu *float*
__complex__(self)	*complex(x)*	konwersja do typu *complex*
__index__(self)	*bin(x), oct(x), hex(x)*	zwraca wartość całkowitą
__round__(self, digits)	*round(x, digits)*	zaokrąglenie
__pos__(self)	*+x*	działanie jednoargumentowego operatora +
__neg__(self)	*-x*	działanie jednoargumentowego operatora -
__add__(self, other) **__iadd__**(self, other) **__radd__**(self, other)	*x + y* *y + x* *x += y*	operacja dodawania
__sub__(self, other) **__rsub__**(self, other) **__isub__**(self, other)	*x - y* *y - x* *x -= y*	operacja odejmowania
__mul__(self, other) **__rmul__**(self, other) **__imul__**(self, other)	*x * y* *y * x* *x *= y*	operacja mnożenia
__mod__(self, other) **__rmod__**(self, other) **__imod__**(self, other)	*x % y* *y % x* *x %= y*	operacja modulo

Numeryczne metody specjalne

METODA SPECJALNA	SPOSÓB UŻYCIA	ZNACZENIE
__truediv__(self, other) **__rtruediv__**(self, other) **__itruediv__**(self, other)	*x / y* *y / x* *x /= y*	operacja dzielenia
__floordiv__(self, other) **__rfloordiv__**(self, other) **__ifloordiv__**(self, other)	*x // y* *y // x* *x //= y*	operacja dzielenia całkowitego
__divmod__(self, other) **__rdivmod__**(self, other)	*divmod(x, y)* *divmod(y, x)*	operacja *divmod*
__pow__(self, other) **__rpow__**(self, other) **__ipow__**(self, other)	*x ** y* *y ** x* *x **= y*	operacja potęgowania

Bitowe metody specjalne

METODA SPECJALNA	SPOSÓB UŻYCIA	ZNACZENIE
__and__(self, other) **__rand__**(self, other) **__iand__**(self, other)	*x & y* *y & x* *x &= y*	operacja "bitowe i"
__or__(self, other) **__ror__**(self, other) **__ior__**(self, other)	*x | y* *y | x* *x |= y*	operacja "bitowe lub"
__xor__(self, other) **__rxor__**(self, other) **__ixor__**(self, other)	*x ^ y* *y ^ x* *x ^= y*	operacja "bitowe albo"
__lshift__(self, other) **__rlshift__**(self, other) **__ilshift__**(self, other)	*x << y* *y << x* *x <<= y*	przesunięcie bitowe w lewo
__rshift__(self, other) **__rshift__**(self, other) **__rshift__**(self, other)	*x >> y* *y >> x* *x >>= y*	przesunięcie bitowe w prawo
__invert__(self, other)	*~x*	odwrócenie bitów

Przeciążanie operatorów

- Klasy, których instancje nie są wartościami numerycznymi, mogą przedefiniować metody specjalne związane z operatorami
- W takiej sytuacji można użyć tych operatorów w stosunku do instancji
- Nazywa się to **przeciążaniem operatorów** (*operator overloading*)
- Python nie umożliwia przeciążania metod

Metody specjalne sekwencji

METODA SPECJALNA	SPOSÓB UŻYCIA	ZNACZENIE
__contains__(self, x)	*x in y*	zwraca *True*, jeśli *x* jest elementem sekwencji *y* lub kluczem słownika *y*
__delitem__(self, k)	*del y[k]*	usuwa *k*-ty element sekwencji *y* lub element spod klucza *k* słownika *y*
__getitem__(self, k)	*y[k]*	zwraca *k*-ty element sekwencji *y* lub wartość spod klucza *k* słownika *y*
__iter__(self)	*for x in y: pass*	zwraca iterator dla elementów sekwencji *y* lub kluczy słownika *y*
__len__(self)	*len(y)*	zwraca ilość elementów w *y*
__reversed__(self)	*reversed(y)*	zwraca odwrotny iterator dla elementów sekwencji *y* lub kluczy słownika *y*
__setitem__(self, k, v)	*y[k] = v*	ustawia wartość *k*-tego elementu sekwencji *y* lub wartość dla klucza *k* słownika *y* na *v*

Ćwiczenia/przykłady

ĆWICZENIA

- Ćwiczenie/przykład 3.2:
  Przeciążanie operatorów

Dziękujemy za uwagę