# Metatřídy v Pythonu a jejich využití

Petr Zemek, xzemek02@stud.fit.vutbr.cz

19. dubna 2009

**Abstrakt**

Zatímco v mnoha běžných a rozšířených objektově orientovaných jazycích má třída speciální postavení a není s ní tedy povolena běžná manipulace jako s ostatními prvky (objekty) daného jazyka, tak v některých objektově orientovaných programovacích jazycích má třída stejné postavení jako jiné entity. Takže mimo jiné platí, že každá třída má svou třídu, ze které byla vytvořena. Taková třída jiné třídy se nazývá metatřídou a tento článek se zabývá popisem metatříd v jazyce Python a jejich využitím v praxi.

**Klíčová slova:** Python, metatřída, praktické využití

## 1 Úvod

V třídně orientovaném (class-based) [8] objektovém paradigma [11] obecně platí, že každý vytvořený objekt má svou třídu, ze které byl vytvořen (je to instance dané třídy). Zatímco v některých objektově orientovaných jazycích (z těch nejznámějších jsou to například Java a C++) má třída speciální postavení a není považována za objekt první třídy (first-class object) [9], tedy s ní nelze pracovat jako s každým jiným objektem (uložit třídu do proměnné, předávat třídu jako argument metodě a podobně), tak některé jazyky (například Smalltalk, Python a Ruby) toto omezení nemají a se třídami se pracuje jako s objekty. Jelikož každý objekt má svou třídu, ze které byl vytvořen, tak i každá třída má svou třídu, ze které je vytvořena. Tato třída se označuje jako metatřída (metaclass) [10] dané třídy[1].

Tak, jako třída je zodpovědná za vytváření svých instancí (objektů), je metatřída zodpovědná za vytváření svých instancí (tříd, které jsou ale zároveň objekty). U jazyků, které nepodporují metatřídy, musí být způsob vytváření a konfigurace tříd specifikován v dokumentaci (standardu) k těmto jazykům a je neměnný. V případě podpory metatříd je ovšem situace jednodušší a mnohem flexibilnější – pro změnu způsobu vytváření a konfigurace tříd stačí implementovat vlastní metatřídu (typicky zděděním originální metatřídy a předefinováním potřebných metod, podle toho, co chceme při vytváření třídy změnit). Mezi vlastnosti, které metatřída řeší, může patřit [10]:

- samotné vytvoření třídy
- informace potřebné pro vytvoření třídy a instancí dané třídy (způsob přidělení paměti, uložení atributů a metod v paměti, inicializace, ...)
- změny ve třídě před jejím vytvoření (například přidání metod a atributů, které nejsou specifikovány uživatelem při vytváření třídy)
- zajištění provedení určitých akcí při jiné akci (například to, že při vytváření instance dané třídy se vždy zavolá konstruktor pro inicializaci)

[1] Přísně technicky vzato, třída `MetaA`, která je třídou třídy `A`, může být nazvána metatřídou pouze pro instance třídy `A`, nikoliv pro třídu `A` samotnou. Tato terminologická skutečnost ale nebývá v praxi příliš rozšířena [14].

V tomto článku se budu zabývat pouze využitím metatříd pro změny ve vytvářené třídě, takže budu vždy dědit z metatřídy, která již přidělování paměti a podobné záležitosti řeší. Pokud potřebujete změnit některé aspekty na nižší úrovni (práce z pamětí), tak prostudujte dokumentaci (standard) k danému jazyku (není zaručeno, že to daný jazyk bude podporovat, i když podporuje metatřídy).

Sekce 2 popisuje syntaxi a sémantiku metatříd v jazyce Python a ukazuje, jak si vytvořit vlastní jednoduchou metatřídu a co obnáší její použití. V sekci 3 jsou ukázány komplexnější příklady využití metatříd v praxi. Sekce 4 diskutuje některé možné alternativy k použití metatříd. V závěru článku je shodnocena praktická využitelnost metatříd.

# 2 Syntaxe a sémantika metatříd v Pythonu

V následujícím textu se budu zabývat pouze třídami označovanými jako "třídy nového typu" (new-style classes), zavedenými od verze 2.2 k unifikaci tříd a typů a Pythonem verze 2. Klasické třídy ("třídy staršího typu") sice obsahují také podporu pro metatřídy, ale ve verzi 3 už by vůbec neměly existovat a jejich použití je v nových aplikacích považováno za zastaralé [5]. Třída nového typu se pozná tak, že vždy explicitně dědí od nějaké třídy. Pokud třída nemá žádné logické předky, pak je jejím předkem třída `object`, která se tak stává bázovou třídou pro všechny třídy v Pythonu.

## 2.1 Všechno je objekt a existuje standardní metatřída

V Pythonu platí, že všechno je objekt (dokonce i moduly jsou objekty) a že všechno má svůj typ [16]. Přesvědčit se můžeme na následujícím jednoduchém příkladu:

```
class A(object):
    pass

>>> print(type(A()))
<class '__main__.A'>

>>> print(type(A))
<type 'type'>

>>> print(type(type))
<type 'type'>
```

Vestavěná funkce `type()` zde zjišťuje typ daného objektu. Jak vidíme, tak typ instance třídy `A` je `A`. Typem třídy `A` je `type`, což je její metatřída. Typ této metatřídy je opět `type`, tedy třída `type` má už jako svojí metatřídu sama sebe. Metatřída `type` je standardní metatřídou v jazyce Python a při vytváření vlastních metatříd z ní dědíme a pouze předefinujeme potřebné metody.

## 2.2 Dynamické vytvoření třídy

Třídu lze vytvořit nejen staticky (ve smyslu její definice pomocí klíčového slova `class`), ale také dynamicky pomocí funkce `type()`, tentokrát se třemi parametry. Následující příklad je převzat z [2].

```
class Test(object):
    def __init__(self, x):
        self.x = x

    def printX(self):
        print self.x
```

Interpreter jazyka Python při výskytu takovéto definice třídy převede tuto definici do tvaru, který lze předat funkci `type()` a pomocí ní provede samotné vytvoření třídy [4]. Jednotlivé parametry funkce `type(name, bases, dict)` znamenají:

- `name` – jméno vytvářené třídy (řetězec)
- `bases` – třídy, ze kterých vytvářená třída dědí (ntice)
- `dict` – slovník, který obsahuje mapování atributu třídy na definici tohoto atributu (například název metody a definici metody v podobě objektu)

Takže třída vytvořená pomocí klíčového slova `class` (uvedená výše), je v základu ekvivaletní vytvoření třídy pomocí funkce `type()`, volané následovně [2]:

```
def __init__(self, x):
    self.x = x

def printX(self):
    print self.x

Test = type('Test', (object,),
    {'__init__': __init__, 'printX': printX})
```

Vidíme zde vytvořené dvě funkce (které jsou také objekty), které ve slovníku předaném funkci `type()` reprezentují metody třídy (každá z nich má jako první parametr `self`). Jméno vytvářené třídy je `Test` a dědí pouze ze třídy `object`.

### 2.3 Specifikace metatřídy pro danou třídu

Ještě předtím, než ukážu, jak si vytvořit vlastní metatřídu, tak se podíváme na to, jak u dané třídy specifikovat, že tato třída má mít jako svou metatřídu právě zvolenou metatřídu. Tato specifikace se provádí v atributu `__metaclass__` vytvářené třídy. Hodnota tohoto atributu je jakýkoliv objekt, který lze zavolat se třemi parametry, které jsou stejné, jako má funkce `type()`. Jak už víme, tak standardní metatřída v Pythonu je `type`, takže explicitně můžeme specifikovat metatřídu takto:

```
class MyClass(object):
    __metaclass__ = type
```

V praxi tato explicitní specifikace metatřídy `type`) samozřejmě nemá smysl, uvádím to pouze jako příklad. Pokud ve třídě nespecifikujeme žádnou metatřídu, tak se vezme metatřída první třídy, ze které naše třída dědí (třída `object` metatřídu specifikovanou má a je to právě `type`) [14].

### 2.4 Vytvoření vlastní metatřídy

Naše první ukázková metatřída bude velice jednoduchá a nebude dělat nic nového oproti standardní metatřídě.

```
class MyMetaclass(type):
    def __new__(cls, name, bases, dict):
        # Zde lze provést modifikace třídy před jejím vytvořením
        return type.__new__(cls, name, bases, dict)
```

Dědíme z metatřídy `type` a předefinováváme metodu `__new__()`, která se volá při požadavku o vytvoření třídy. Na rozdíl od běžných tříd nemá první parametr pojmenován `self`, ale `cls`, protože operujeme nad vytvářenou třídou, nikoliv nad metatřídou [15]. Zbývající parametry jsou totožné s těmi, které přijímá funkce `type`. V těle této metody můžeme provést manipulaci s originální třídou (bude ukázáno dále) a poté zavoláme metodu `__new__()`[2] bázové třídy, která zajistí skutečné vytvoření zmodifikované třídy.

Naši metatřídu použijeme takto:

```
class MyClass(object):
    __metaclass__ = MyMetaclass
```

Z následujícího volání vidíme, že naše třída má místo metatřídy `type` naši vlastní metatřídu:

```
>>> type(MyClass)
<class '__main__.MyMetaclass'>
```

## 2.5 Jednoduchý příklad použití metatřídy

Nyní uvedu velmi jednoduchý příklad, který vede na použití metatřídy. Komplexnější příklady použití budou uvedeny v následující sekci. Řekněme, že nám nevyhovuje standardní výpis, který dává funkce `str()` při zavolání nad nějakou třídou. Například, mějme následující třídu `MyClass`:

```
class MyClass(object):
    pass
```

Při zvolání `str(Foo)` dostaneme následující výsledek:

```
>>> print(str(MyClass))
<class '__main__.MyClass'>
```

My bychom chtěli dosáhnout toho, aby tento výpis obsahoval jen název třídy. Funkce `str()` používá metodu `__str__()` objektu, který jí byl předán jako parametr. Protože nám jde o výpis jména třídy, nikoliv instance dané třídy, tak nelze předefinovat metodu `__str__()` ve třídě `Foo`, ale musíme tuto metodu předefinovat v její metatřídě. K tomu účelu si vytvoříme metatřídu `PrettyPrint`:

```
class PrettyPrint(type):
    def __str__(cls):
        return '<class ' + cls.__name__ + '>'
```

Protože nám nejde o modifikaci atributů vytvářené třídy, tak nám stačí předefinovat metodu `__str__`. Nyní bude definice naší třídy `MyClass` vypadat takto:

```
class MyClass(object):
    __metaclass__ = PrettyPrint
```

Pokud nyní zavoláme funkci `str()` nad naší třídou, dostaneme kýžený modifikovaný výpis. Místo metody `__str__()` originální metatřídy `type` se zavolá naše metoda, která dostane jako první parametr třídu `Foo` a vrátí řetězec, který nám funkce `str()` přepošle jako výsledek.

```
>>> print(str(MyClass))
<class 'MyClass'>
```

[2] Pokud bychom místo toho zavolali pouze funkci `type()`, tak by výsledná třída neměla jako metatřídu naši třídu `MyMetaclass`, ale standardní metatřídu `type` [4].

# 3 Využití metatříd v praxi

V této sekci uvedu příklady využití metatříd v praxi. Dva příklady uvedu s podrobnějším popisem, u ostatních pouze naznačím možnost užití a případně odkážu na články, kde je diskutována implementace nebo myšlenka.

## 3.1 Aspektově orientované programování

Metatřídy v Pythonu lze využít k aspektově orientovanému programování [7, 17]. Principiálně jde o to, že někdy potřebujeme, aby se před každou (nebo některými zvolenými) metodou volala nějaká funkce, případně aby se volala po dokončení volání původní metody. Například různé kontroly, jako je kontrola práv uživatele vykonávat určitou činnost, logování volání metody do záznamů, profilování a také ladění (kontrolní výpisy). Obvykle se to řeší tak, že se explicitně do daných metod přidá na začátek (a případně na konec) volání nějaké funkce. Tento způsob řešení je ale velice neflexibilní a je nutné editovat kód na mnoha místech. A co když někdo přidá další metodu a zapomene přidat volání těchto funkcí? Jak vidíte, tento způsob je také náchylný k chybám.

Využitím metatříd lze zařídit, že se před vytvořením třídy všechny (či zvolené) metody obalí jinou funkcí, která zajistí zavolání zvolených funkcí před zavoláním metody, poté zavolá původně volanou metodu a nakonec zavolá zvolené funkce, které se mají zavolat po dokončení volání metody. Pak už jen stačí, aby daná cílová třída měla nastavenou jako metatřídu naši metatřídu. Pro programátora cílová třídy je vše transparentní a nemůže se stát, že se někdy zapomene přidat volání funkce na začátek nové metody.

Pro ukázku implementace nahlédněte do [17].

## 3.2 Modifikace atributů tříd

Metatřídy lze využít pro modifikaci atributů (atributem jsou míněny i metody) dané třídy před jejím vytvořením. Například lze do tříd automaticky přidat atributy, změnit atributy (nahradit metodu jinou metodou) či odstranit atributy. U metod lze dokonce měnit (přidávat či odebírat) její parametry.

Jako příklad uvedu implementaci tzv. "Selfless" Pythonu [2]. V klasickém Pythonu musí mít všechny (nestatické) metody explicitně jako první parametr `self`, čili objekt, ve kterém je daná metoda volána. Využitím metatříd lze zajistit, že parametr `self` nebude třeba explicitně v metodě uvádět a před vytvořením třídy se do všech metod automaticky přidá.

Následující implementace je převzata z [2] a využívá nestandardní modul `byteplay`[3].

```
from types import FunctionType
from byteplay import Code, opmap

def MetaClassFactory(function):
    class MetaClass(type):
        def __new__(meta, classname, bases, classDict):
            for attributeName, attribute in classDict.items():
                if type(attribute) == FunctionType:
                    attribute = function(attribute)

                newClassDict[attributeName] = attribute
            return  type.__new__(meta, classname, bases, classDict)
    return MetaClass
```

[3] http://wiki.python.org/moin/ByteplayDoc

```
    def _transmute(opcode, arg):
        if ((opcode == opmap['LOAD_GLOBAL']) and
            (arg == 'self')):
            return opmap['LOAD_FAST'], arg
        return opcode, arg

    def selfless(function):
        code = Code.from_code(function.func_code)
        code.args = tuple(['self'] + list(code.args))
        code.code = [_transmute(op, arg) for op, arg in code.code]
        function.func_code = code.to_code()
        return function

    Selfless = MetaClassFactory(selfless)
```

Funkce `MetaClassFactory()` vrací metatřídu, která na každou metodu při vytváření třídy aplikuje předanou funkci `function()`. Pracuje tak, že předefinovává metodu `__new__()` standardní metatřídy `type` a prochází všechny atributy třídy. V případě, že je daný atribut metoda, tak na ni zavolá funkci `function()` a změněnou metodou přepíše tu původní. Funkce `selfless()` a `_transmute()` pak provedou samotné přidání parametru `self` do předané metody.

Použití je jednoduché:

```
    class Test(object):
        __metaclass__ = Selfless

        def __init__(x=None):
            self.x = x

        def getX():
            print self.x

        def setX(x):
            self.x = x

    test = Test()
    test.getX()
```

### 3.3 Provádění akcí při vytváření tříd a jejich instancí

Využitím metatříd lze při vytváření tříd (a jejich instancí) zajistit, aby byla zavolána zvolená funkce. Lze tak například zajistit automatické registrace tříd (použitelné například pro pluginy) nebo třeba počítání, kolik tříd bylo vytvořeno.

Jako příklad uvedu implementaci návrhového vzoru Jedináček (Singleton) [13] převzatou z [3].

```
class SingletonMetaclass(type):
        def __call__(cls):
            if cls.instance is None:
                cls.instance = type.__call__(cls)

            return cls.instance
```

Dosažení existence jediné instance je zajištěno přeedfinováním metody `__call__()` v metatřídě, která se volá v případě "zavolání" cílové třídy, čili v případě instanciace cílové třídy. V této metodě je zajištěno, že pokud dosud nebyla žádná instance vytvořena, vytvoří se nová instance. Pokud už vytvořená instance existuje, pak se pouze vrátí.

Ukázka použití a toho, že to funguje:

```
class Singleton(object):
    __metaclass__ = SingletonMetaclass

    instance = None

    def __init__(self):
        print 'Initialized'


>>> s1 = Singleton()
Initialized
>>> s2 = Singleton()
>>> s2 is s1
True
```

Tato implementace trpí určitými neduhy, například není použitelná ve vícevláknových aplikacích (thread-safe) a třída, která má jako metatřídu třídu `SingletonMetaclass` musí mít atribut `instance` (toto druhé omezení lze vyřešit tak, že atribut `instance` do třídy přidáme dynamicky před vytvořením třídy).

### 3.4 Další možná použití

Kromě výše uvedených nejčastějších použití metatříd existuje celá řada dalších možných použití [6, 4]. Lze například kontrolovat, zda všechny metody tříd jsou dokumentované (všimněte si, dokonce dokumentační řetězec k metodě je objekt), vytvořit automaticky databázové schéma pro danou třídu či trasovat podtřídy dané třídy pro účely registrace (například pro implementaci pluginů a zvolení toho správného v případě požadavku o nějakou akci). Pygments[4] využívá metatřídy k vytvoření syntaktického analyzátoru z dodaných definicí ve třídě. Interně využívají metatřídy i mnohé frameworky, například Django[5]. u dynamických modelů [1].

## 4 Alternativy k metatřídám

V mnoha případech může být použití metatříd zbytečně složité a implementačně náročné. Pokud jde pouze o přidání či změnu funkcionality metod a tříd, tak existují dvě alternativy:

- Použití klasické dědičnosti a agregace [6]. Vytvoříme novou třídu, která buď dědí z původní třídy a předefinovává potřebné metody, nebo tato nová třída obsahuje instanci původní třídy jako atribut a ve svých metodách volá její metody.
- Použití dekorátorů[6] metoda a tříd [18]. Dekorátory lze v Pythonu chápat jako makra [12]. Máme existující metodu či třídu, kterou pomocí dekorátoru určitým způsobem modifikujeme. Takto lze

[4] http://www.pygments.org/

[5] http://www.djangoproject.com/

[6] Dekorátory v Pythonu lze použít k implementaci návrhového vzoru dekorátor (decorator) [13], ale jedná se pouze o jejich omezené použití, protože dekorátory v Pythonu toho zvládnou více [12].

například k metodě přidat kód, který se má provést před provedením a po provedení dané metody (aspektově orientované programování, viz sekce 3). Nevýhodou je to, že dekorátor musíme uvést před každou metodou, kterou chceme tímto dekorátorem modifikovat. Výhodou je jednodušší implementace, než v případě metatříd, takže dekorátory v Pythonu lze považovat za jednodušší alternativou k metatřídám [12].

# 5 Závěr

Metatřídy představují velice silný a užitečný koncept, pomocí kterého lze řešit některé problémy efektivněji (ve smyslu návrhu, nikoliv výkonu) než užitím jiných prostředků. Jejich nevýhodou je relativně vyšší složitost při reálném použití v komplexních situacích. Na tomto místě bych chtěl citovat Tima Peterse, který jednou prohlásil následující [15] (volně přeloženo):

> Metatřídy jsou mnohem hlubší magie než to, o co se bude muset 99% uživatelů kdy starat. Jestli přemýšlíte o tom, zda je potřebujete, tak odpověď je, že nepotřebujete (lidé, kteří je opravdu potřebují, s naprostou jistotou ví, že je potřebují, a nemusí tedy vysvětlovat, proč je potřebují).

Pokud tedy lze problém řešit pomocí jednodušších prostředků, jako je dědění a agregace či dekorátory (sekce 3), tak byste toto řešení měli použít. V některých případech ale může být použití metatříd opodstatněné.

# Reference


[1] Django – Dynamic models. [online], poslední aktualizace 2009-01-21. [cit. 2009-04-19]. Dostupné na URL: `<http://code.djangoproject.com/wiki/DynamicModels>`.

[2] Meta-classes made easy. [online], poslední aktualizace 2008-02-15. [cit. 2009-04-19]. Dostupné na URL: `<http://www.voidspace.org.uk/python/articles/metaclasses.shtml>`.

[3] Meta-classes made easy. [online], poslední aktualizace 2007-04-23. [cit. 2009-04-19]. Dostupné na URL: `<http://www.voidspace.org.uk/python/weblog/arch_d7_2007_04_21.shtml#e691>`.

[4] Metaclasses in five minutes. [online], poslední aktualizace 2009-04-10. [cit. 2009-04-19]. Dostupné na URL: `<http://www.voidspace.org.uk/python/articles/five-minutes.shtml>`.

[5] Python 2.5.2 documentation. [online], poslední aktualizace 2008-02-21. [cit. 2009-04-18]. Dostupné na URL: `<http://www.python.org/doc/2.5.2/>`.

[6] Stack Overflow – What are your (concrete) use-cases for metaclasses in Python? [online], poslední aktualizace 2008-12-28. [cit. 2009-04-19]. Dostupné na URL: `<http://stackoverflow.com/questions/392160/what-are-your-concrete-use-cases-for-metaclasses-in-python>`.

[7] Wikipedia, the free encyclopedia – Aspect-oriented programming. [online], poslední aktualizace 2009-03-18. [cit. 2009-04-18]. Dostupné na URL: `<http://en.wikipedia.org/wiki/Aspect-oriented_programming>`.

[8] Wikipedia, the free encyclopedia – Class-based programming. [online], poslední aktualizace 2008-12-21. [cit. 2009-04-18]. Dostupné na URL: `<http://en.wikipedia.org/wiki/Class-based_programming>`.

[9] Wikipedia, the free encyclopedia – First-class object. [online], poslední aktualizace 2009-03-18. [cit. 2009-04-18]. Dostupné na URL: `<http://en.wikipedia.org/wiki/First-class_object>`.

[10] Wikipedia, the free encyclopedia – Metaclass. [online], poslední aktualizace 2009-03-18. [cit. 2009-04-18]. Dostupné na URL: <http://en.wikipedia.org/wiki/Metaclass>.

[11] Wikipedia, the free encyclopedia – Object-oriented programming. [online], poslední aktualizace 2009-04-17. [cit. 2009-04-18]. Dostupné na URL: <http://en.wikipedia.org/wiki/Object-oriented_programming>.

[12] Eckel, B.: Decorators I: Introduction to Python decorators. [online], poslední aktualizace 2008-10-18. [cit. 2009-04-19]. Dostupné na URL: <http://www.artima.com/weblogs/viewpost.jsp?thread=240808>.

[13] Gamma, E.; Helm, R.; Johnson, R.; aj.: *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Professional, 1994, ISBN 978-0201633610.

[14] Martelli, A.: *Python in a nutshell*. O'Reilly, 2003, ISBN 0-596-00188-6.

[15] Mertz, D.; Simionato, M.: Metaclass programming in Python: Pushing object-oriented programming to the next level. [online], poslední aktualizace 2003-02-26. [cit. 2009-04-18]. Dostupné na URL: <http://www.ibm.com/developerworks/linux/library/l-pymeta.html>.

[16] Pilgrim, M.: *Dive Into Python*. Apress, 2004, ISBN 978-1590593561.

[17] Ribic, A.: Aspect-oriented Python and metaclasses. [online], poslední aktualizace 2009-02-25. [cit. 2009-04-19]. Dostupné na URL: <http://www.hackersinshape.net/archives/67>.

[18] Smith, K. D.; Jewett, J. J.; Montanaro, S.; aj.: Decorators for Functions and Methods. [online], poslední aktualizace 2003-05-05. [cit. 2009-04-19]. Dostupné na URL: <http://www.python.org/dev/peps/pep-0318/>.