České vysoké učení technické v Praze FIT

Programování v Pythonu

Jiří Znamenáček

Příprava studijního programu Informatika je podporována projektem financovaným z Evropského sociálního fondu a rozpočtu hlavního města Prahy.

Praha & EU: Investujeme do vaší budoucnosti

Python – Třídy a objekty v Python'u

Zavedení třídy

Zcela v duchu úvodu můžeme nejjednodušší třídu vyrobit jako zobecnění *pojmenované n-tice*. Narozdíl od n-tice však u třídy můžeme měnit hodnoty jejích prvků:

```
>>> class MyClass:
...     a, b, c = 1, 2, 3
...
>>> MyClass
<class '__main__.MyClass'>

>>> MyClass.a
1
>>> MyClass.a = 'ahoj'
>>> MyClass.a
'ahoj'
```

Tím jsme získali *jeden* pojmenovaný objekt, který umožňuje přístup ke svým prvkům pomocí tečkové notace (nebo jinými slovy – zavedli jsme nový jmenný prostor). To jsme si zase tak moc nepomohli. A přiřazení jiného jména to samozřejmě nijak nevylepší:

```
>>> mc = MyClass
>>> mc
<class '__main__.MyClass'>

>>> MyClass.a
1
>>> mc.a
1

# Tam..
>>> MyClass.a = 'ahoj'
>>> MyClass.a
'ahoj'
>>> mc.a
'ahoj'

# ..i zpět to samozřejmě funguje stejně.
>>> mc.b
2
>>> MyClass.b
2
>>> mc.b = 'svete'
>>> mc.b
'svete'
>>> MyClass.b
'svete'
```

→ Provedením `mc = MyClass` jsme pouze objektu v paměti „přilepili“ další identifikátor.

PS: Tímto způsobem jsme vlastně zavedli *alias* na základní třídu, což nám bude v budoucnu k užitku.

Instance třídy

I. Operování přímo s konkrétními třídami jako na předchozím slajdu je sice možné, ale dost by omezovalo jejich použití. Zajímavější je, pokud uvedenou třídu použijeme jako *konstruktor* pro jednotlivé její *instance*:

```
>>> class MyClass:
...     a, b, c = 1, 2, 3
...
>>> MyClass
<class '__main__.MyClass'>
>>> dir(MyClass)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__f

# A) Instance 'mc1' třídy 'MyClass' podědí vlastnosti svého rodiče
>>> mc1 = MyClass()
>>> mc1
<__main__.MyClass object at 0xb75dda0c>
>>> MyClass.a
1
>>> mc1.a
1

# B) Pokud na potomkovi ale něco změníme, rodič se to nedozví:
>>> mc1.a = 'ahoj'
>>> mc1.a
'ahoj'
>>> MyClass.a
1

# C) Podobně pokud nyní změníme něco na rodiči, potomek bude takté
>>> MyClass.a = 'svete'
>>> MyClass.a
'svete'
>>> mc1.a
'ahoj'

# D) Nový potomek 'mc2' ale nyní podědí i nový stav rodiče:
>>> mc2 = MyClass()
>>> mc2
<__main__.MyClass object at 0xb75ddd6c>
>>> mc2.a
'svete'
```

Všimněte si, že při použití třídy jako *konstruktoru* ji **voláme**, tedy se závorkami – `MyClass()`.

PS: Tímto způsobem jsme si vlastně ukázali rozdíl mezi atributy *instančními* a *třídními*.

II. Atributy instance tedy nemají s atributy třídy už nic společného, ke každým z nich se přistupuje odděleně. To se ale dá využít ještě o kus dál – na každé instanci si můžeme dodefinovat libovolné další atributy, které potřebujeme. Prakticky úplnou obdobu pascalovského typu *záznam* (*record*) tak získáme např. takto:

```
class Zaměstnanec:
    pass

jan = Zaměstnanec()   # vyrobení prázdného záznamu o zaměstnanci

# naplnění polí (atributů) záznamu
jan.jméno = 'Jan Novák'
jan.oddělení = 'počítačová laboratoř'
jan.plat = 15000
```

→ Upraveno podle dokumentace.

Prostě jsme vytvořili prázdný objekt, z něj vyrobili taktéž zcela prázdnou instanci a do ní jsme „naházeli“ všechna potřebná data.

Metody I

Zatím jsme naše ukázkové instance tříd pouze plnili daty. Síla tříd však tkví hlavně v tom, že spolu s daty zapouzdřují i funkce, které umí s příslušnými daty správně pracovat. Těmto funkcím, které operují na datech dané instance, se říká *metody*.

Doplňme naši ukázkovou třídu o nějakou metodu:

```
class Člověk():
    ...
    def identifikace(self):
        return 'Dobrý den! Jmenuji se ' + self.jméno + '.'
    ...
```

Přes „podivný“ parametr `self` voláme tuto metodu standardně – předáváme pouze další dodatečné parametry (což v našem příkladě znamená parametr žádný):

```
alisa.identifikace() --> "Dobrý den! Jmenuji se Алиса Селезнева."
```

Parametr `self` odkazuje na právě zavedenou instanci třídy. Při zavolání této metody ve výše uvedeném tvaru `alisa.identifikace()` se tak vlastně ve skutečnosti volá kód `Člověk.identifikace(alisa)`. V jiných jazycích je uvedený parametr předáván často „automagicky“ a nazývá se zpravidla *this*.

→ Možná to není na první pohled zřejmé, ale uvedený parametr se nemusí nutně jmenovat právě *self*. Je však *velmi dobrým* zvykem ho tak nazývat.

Upozornění: **Všechny** metody uvnitř třídy (pokud nebudou speciálně označené, jak uvidíme na dalších slajdech) dostávají **automaticky** tento

první parametr! Na začátku ho asi často zapomenete napsat, a pak se budete divit, proč na vás interpretr křičí, že voláte metodu s jiným počtem parametrů, než na jaký je stavěna.

PS: V Python'u můžete atributy třídy/instance vesele aliasovat a volat pod těmito novými jmény, např.:

```
# Po zavedení..
fn = alisa.identifikace
# ..pak můžeme volat:
fn()  -->  "Dobrý den! Jmenuji se Алиса Селезнева."
```

Atributy, inicializace

Většinou nebudeme chtít vyrobit instanci třídy (pouze) s hodnotami, jaké poskytla rodičovská třída. V takovém případě doplníme třídu o metodu `__init__()`. Ta zaručuje, že každá nově vytvořená instance bude obsahovat též takové hodnoty, které pošleme jako parametr do třídního konstruktoru:

```
# Při zavedení třídy..
class Člověk():
    """Objektová reprezentace člověka."""

    def __init__(self, jméno, datum_narození, krevní_skupina):
        self.jméno = jméno
        self.datum_narození = datum_narození
        self.krevní_skupina = krevní_skupina

# ..pro konkrétní instanci..
alisa = Člověk(
    'Алиса Селезнева',
    '2065-11-17',
    '0',
  )

# ..dostaneme:
alisa.jméno --> Алиса Селезнева
alisa.krevní_skupina --> 0
```

Všimněte si (a zapamatujte), že `__init__()` je zase pouze „obyčejná“ třídní metoda a jako taková dostává automaticky implicitní první parametr v podobě odkazu na právě zaváděnou instanci třídy. Při inicializaci se tedy vlastně volá kód `Člověk.__init__(alisa, 'Алиса Селезнева', '2065-11-17', '0')`.

PS: Na parametry konstruktoru se nepřekvapivě vztahují podobná pravidla jako na parametry funkcí. Můžete tedy používat pojmenované parametry s výchozí hodnotou apod.

Dědičnost I

Dědičnost v rámci Python'u vyjádříme jako parametr při zavádění třídy. *Školáka* jako upraveného *Člověka* tak vyrobíme třeba takto:

```
class Školák(Člověk):
  """Objektová reprezentace školáka."""

  def __init__(self, jméno, datum_narození, krevní_skupina, třída,
      Člověk.__init__(self, jméno, datum_narození, krevní_skupina)
      self.třída = třída
      self.předměty = předměty
```

A podobně i pro *Učitele*:

```
class Učitel(Člověk):
  """Objektová reprezentace učitele."""

  def __init__(self, jméno, datum_narození, krevní_skupina, třída,
      Člověk.__init__(self, jméno, datum_narození, krevní_skupina)
      self.třída = třída
      self.předměty = předměty
      self.výplata = výplata
```

`__init__()` nadřazené třídy s příslušnými parametry tedy musíme v potomkovi zavolat sami, nikdo jiný to za nás neudělá!

→ Při dědění z jednoho rodiče můžeme explicitní volání jménem rodičovské třídy nahradit voláním funkce *super()*: `super().__init__(jméno, datum_narození, krevní_skupina)`, což je ekvivalentní volání např. `super(Školák, self).__init__(jméno, datum_narození, krevní_skupina)` . Ale obecně má *super()* své mouchy.

→ Python podporuje i vícenásobnou dědičnost, „rodičů“ tudíž může být víc jak jeden. Při vyhledávání atributů jsou pak zjednodušeně řečeno prohledávány třídy v zadaném pořadí a do hloubky.

Budeme-li pokračovat v příkladu, bude upravené zavedení objektu typu *Školák* vypadat takto:

```
alisa = Školák(
      'Алиса Селезнева',
      '2065-11-17',
      '0',
      '3. A',
      {'matematika': 1, 'zeměpis': 1, }
    )

alisa.jméno --> Алиса Селезнева
alisa.třída --> 3. A
```

Zde se tedy prozměnu volá: `Školák.__init__(alisa, 'Алиса Селезнева', '2065-11-17', '0', '3. A', {'matematika': 1, 'zeměpis': 1, })`.

Dědičnost II

I. I když použijeme *super()*, pořád se na rodičovskou třídu odkazujeme jejím jménem při zavedení poděděné třídy a při případné refaktorizaci kódu nás tak čeká spousta práce s přepisováním (a nezapomenutím na nějaký výskyt).

Tento problém je ovšem možné snadno vyřešit – stačí zavést *alias* na rodičovskou třídu a v kódu používat dále pouze ten (což jsme ostatně viděli už na prvním slajdu):

```
class Člověk():
  ...

AliasČlověk = Člověk

class Školák(AliasČlověk):
  ...
```

→ Samozřejmě nezáleží na tom, jak svůj alias pojmenujete, je to normální „proměnná“.

Při refaktorizaci pak stačí zaměnit jediné místo kódu.

II. Mimochodem něco podobného se týká i rodičovských tříd definovaných v jiných *modulech*. Chcete-li z takové třídy dědit, musíte ji nejdříve do aktuálního modulu naimportovat. A jestli ji naimportujete jako součást celého modulu, nebo pouze ji, nebo pod nějakým jiným jménem, je zcela na vás.

Dědičnost III

Zatímco volání rodičovského *__init__()* je prakticky povinné*, upravujeme-li funkčnost některé rodičovské metody, nemusí to už být pravda. Pokud ale ano, je syntaxe naštěstí úplně stejná, jak ukazuje příklad:

```
class Rodič:
    ...
    def fn(self, atr1):
        ...
    ...

class Potomek(Rodič):
    ...
    # upravená metoda 'fn()'
    def fn(self, atr1, atr2):
        Rodič.fn(self, atr1)
        # následuje kód, kde využijeme i 'atr2'
        ...
    ...
```

→ * Striktně vzato bychom rodičovské *__init__()* volat nemuseli, pokud nic nedělá. Ale kdyby někdy v budoucnu něco dělat začalo, budeme mít problém :-)

Metody II

Potomek podědí z rodiče jeho atributy i metody. Obé z toho může přepsat vlastní definicí (nebo pouze nějak doplnit). Ale stejně tak si samozřejmě může zavést atributy a metody zcela vlastní.

Doplňme naši ukázkovou třídu *Školák* o metodu pro výpočet průměrné známky ze všech předmětů (ostatní kód zůstává stejný):

```
class Školák(Člověk):
  ...
  def průměrná_známka(self):
      """Vrací průměr všech známek studenta."""
      součet = 0
      for předmět in self.předměty:
          součet += self.předměty[předmět]
      return součet / len(self.předměty)
```

Nyní budou instance třídy *Školák* obsahovat oproti instancím třídy *Člověk* navíc i metodu *průměrná_známka()*.

PS: Jak už víme z dřívějška, pro volání metody platí následující ekvivalence:

```
# Volání..
alisa.průměrná_známka() --> 1
# ..je ekvivalentní volání:
Školák.průměrná_známka(alisa)
```

Introspekce I

Python ve velkém podporuje *introspekci*, tj. zjišťování, které atributy a metody jsou na objektu definované, zda je objekt potomkem nějakého rodiče (a kterého) apod. Přístup k podobným informacím je umožněn v zásadě dvěma způsoby – pomocí vestavěných funkcí a pomocí „navigace“ přes *magické atributy/metody* (to jsou ty se jménem z obou stran dvojpodtržítkovaným).

→ Ostatně vestavěné funkce často fungují právě tak, že volají některou z „magických“ metod nebo operují nad některými „magickými“ daty.

Obecně bychom introspekci objektu mohli charakterizovat jako hledání odpovědí na následující otázky:

- Jak se jmenuješ?
- Co jsi vlastně za objekt?
- Co víš?
- Co umíš?
- Kdo jsou tví rodiče?

→ Zdroj: http://www.ibm.com/developerworks/library/l-pyint.html

Introspekce II

I. Jakousi základní introspekci poskytuje už samotný interpretr Python'u. Zadání jména objektu na příkazové řádce interpretru způsobí většinou vypsání jakési jeho základní hodnoty (často reprezentované jeho magickým atributem `__str__()`), která nám už sama o objektu mnohé napoví.

Při našem předchozím příkladu s Alisou bude např. platit:

```
print(alisa)   -->   <__main__.Školák object at 0x00C72890>
```

II. Za asi nejutajenějšího kandidáta na prvek jazyka, který *velmi* využívá vlastností a schopností introspekce u Python'u, však může být pravděpodobně pokládána funkce `help()`. Ta totiž dohledává většinu svých informací v rámci dokumentačních řetězců kódu, které jsou přístupné přes magický atribut `__doc__` jednotlivých objektů.

Při pokračování příkladu s Alisou bude např. platit:

```
# dokumentační řetězec vlastní třídy i instance 'alisa' je stejný
Školák.__doc__              -->   "Objektová reprezentace školáka."
alisa.__doc__               -->   "Objektová reprezentace školáka."

# přitom atribut __class__ instance odkazuje právě na odpovídající
alisa.__class__.__doc__     -->   "Objektová reprezentace školáka."
```

III. Mezi introspekční nástroje můžeme počítat též funkci `id(OBJEKT)`. Ta vrací identifikátor objektu, který je po dobu jeho života konstantní a jedinečný (typicky odpovídá jeho adrese v paměti).

> → Ale dva různé objekty existující v různou dobu mohou dávat pochopitelně stejná *id()*. V kontrastu s tím `hash()` poskytuje pro každý objekt vždy jedinečné číslo.

V rámci stejného běhu testovacího programu jako výše vrátila funkce *id()* toto:

```
id(alisa)   -->   13052048
```

Introspekce III

I. Mnoho introspekčních vestavěných funkcí jsme už potkali. Zcela typickou je např. funkce `type(OBJEKT)`, která vrací typ daného objektu. Návratová hodnota je typový objekt a většinou odpovídá objektu v `OBJEKT.__class__`.

Při našem předchozím příkladu s Alisou bude např. platit:

```
# 'alisa' je instance třídy typu Školák a jako taková má i stejný
alisa.__class__                 -->   <class '__main__.Školák'>
type(alisa)                     -->   <class '__main__.Školák'>
type(alisa) == alisa.__class__  -->   True

# jméno třídy, do které 'alisa' patří, je právě Školák
alisa.__class__.__name__    -->   "Školák"
```

II. Pro testování typu objektu se však nejčastěji používá funkce

`isinstance(OBJEKT, TŘÍDA)`, protože bere v potaz celý řetězec dědičnosti, jak ukazují následující příklady:

```
# 'alisa' je instance třídy Školák a je to i její skutečná třída
isinstance(alisa, Školák)   -->   True
alisa.__class__ == Školák   -->   True

# 'alisa' je též instancí předka Člověk, ale její skutečná třída j
isinstance(alisa, Člověk)   -->   True
alisa.__class__ == Člověk   -->   False

# 'alisa' pochopitelně není instancí třídy Učitel
isinstance(alisa, Učitel)   -->   False
```

→ Druhým parametrem funkce *isinstance()* nemusí být nutně přímo třída, ale též například n-tice tříd. Koho zajímají podrobnosti, podívá se do dokumentace.

III. Pokud nás budou zajímat čistě „dědičné“ vlastnosti objektů, použijeme funkci `issubclass(OBJEKT, TŘÍDA)`. Například při pokračování v našem předchozím příkladu s Alisou bude pro vzájemné vztahy mezi třídami platit:

```
# Školák je potomkem Člověka, Učitel Školáka však nikoli (je to je
issubclass(Školák, Člověk)   -->   True
issubclass(Učitel, Školák)   -->   False

# možná poněkud překvapivě jsou třídy pokládány za svoje vlastní p
issubclass(Školák, Školák)   -->   True
```

→ Jako u *isinstance()* druhým parametrem funkce *issubclass()* nemusí být nutně přímo třída, ale též například n-tice tříd. Koho zajímají podrobnosti, podívá se do dokumentace.

Introspekce IV

I. Pravděpodobně zdaleka nejpoužívanějším introspekčním nástrojem (alespoň na příkazové řádce a při „od[brouk/mouch]ovávání“) bude funkce *dir()*. Ta vrací (jako řetězce) seznam všech (nedynamicky vytvořených) atributů přítomných na daném objektu.

V našem příkladu s Alisou obdržíme na dotaz `dir(alisa)` odpověď:

```
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__f
 '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '
 '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__red
 '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclassho
 '__weakref__', 'datum_narození', 'jméno', 'krevní_skupina', 'prům
 'předměty', 'přidej_školáka', 'třída', 'školáků']
```

Kromě námi vytvořených atributů se objekt *alisa* jen velmi málo liší od vlastností všech objektů, které zajišťuje jejich společný předek *object*, jak nám ukáže dotaz `dir(object)`:

```
['__class__', '__delattr__', '__doc__', '__eq__', '__format__', '_
 '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '
 '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '
 '__sizeof__', '__str__', '__subclasshook__']
```

Veškeré odlišnosti navíc přitom padají na vrub zavedení nového objektu s vlastními atributy: `__dict__` (slovník jmenného prostoru třídy; tj. atributy a jejich hodnoty), `__module__` (jméno modulu, v němž byla třída definována), `__weakref__` (signalizace přítomnosti podpory pro *weak references* u instance). Odpovídající hodnoty pro testovací příklad jsou:

```
__dict__      -->   {'jméno': 'Алиса Селезнева', 'třída': '3. A',
                     'předměty': {'zeměpis': 1, 'matematika': 1},
                     'datum_narození': '2065-11-17', 'krevní_skupi
__module__    -->   __main__
__weakref__   -->   None
```

> → Ad *weak references*: Python umožňuje odkazovat na (některé) objekty nejen klasicky „natvrdo“, ale též „slabě“ pomocí *weak references*. Objekt, na nějž odkazují už pouze *weak references*, může být při nejbližším běhu *grabage collector*u „zlikvidován“.

Objekt *__dict__* přitom můžeme použít k přímému (a tedy dost neklasickému) přístupu na atributy objektu:

```
alisa.__dict__['jméno']      -->   Алиса Селезнева
alisa.__dict__['příjmení']   -->   KeyError: 'příjmení'
```

> → PS: Jako u mnoha jiných dvojpodtržítkovaných atributů tohle normálně *nebudete* chtít dělat...

II. Funkce `dir()` vrací přehled atributů daného objektu jako obyčejný seznam řetězců bez jakéhokoliv bližšího rozlišení. Můžeme nějak snadno rozlišit mezi atributy datovými a metodami? Ano, pomocí funkce `callable()`:

```
callable(alisa.školáků)          -->   False
callable(alisa.přidej_školáka)   -->   True
```

> → Historickou shodou okolností se při přípravě Python'u 3.0 dospělo k závěru, že každý správný pythonista bude tuto informaci zjišťovat za pomoci vhodné *ABC* (tj. *abstract base class*), tedy konstrukcí `isinstance(x, collections.Callable)`. Překvapivě od Python'u 3.2 je vše zase zpátky při starém a znovu si vystačíme pouze pouze s voláním funkce `callable(x)` ^_~

III. Použití *callable()* pro zjištění „volatelnosti“ atributu (nebo obecně jakéhokoliv objektu) je sice pěkné, ale jak nám pomůže, když *dir()* poskytuje odpovědi jako řetězce, ale *callable()* očekává referenci na objekt? Se záchranou přichází funkce `getattr(OBJEKT, "atribut")` (atribut je tedy zadáván jako *řetězec*). Volání této funkce je vlastně ekvivalentní zavolání `OBJEKT.atribut`:

```
# Jelikož platí..
alisa.školáků               -->   1
getattr(alisa, 'školáků')   -->   1

# ..tak není divu, že:
alisa.školáků == getattr(alisa, 'školáků')   -->   True
```

→ Plná možnost volání uvedené funkce je `getattr(OBJEKT, "ATRIBUT", default)`, přičemž *default* je vrácen místo hodnoty `OBJEKT.ATRIBUT`, není-li tato k dispozici. Pokud *default* neuvedete a atribut uvedeného jména na objektu neexistuje, obdržíte výjimku `AttributeError`.

IV. Funkce *getattr()* má sourozence – `hasattr(OBJEKT, "atribut")`. Ta slouží ke zjištění, zda daný objekt má atribut uvedeného jména (a tak je podstatně vhodnějším kandidátem na použití v programu, než procházení návratového seznamu funkce *dir()*):

```
hasattr(alisa, 'školáků')          -->   True
hasattr(alisa, 'přidej_školáky')   -->   False
```

→ Mimochodem funkce *hasattr()* je implementována pomocí funkce *getattr()* – prostě se pokusí daný atribut získat a zachová se podle toho, zda obdrží výjimku nebo ne.

Funkce *hasattr()* a *getattr()* mají ještě třetího sourozence, který však už nemá moc společného s introspekcí. Je jím funkce `setattr(OBJEKT, "atribut", hodnota)` plnící funkci nastavení hodnoty atributu, jehož jméno je zadáno jako řetězec:

```
# výchozí hodnota
alisa.krevní_skupina   -->   0

# „klasické“ nastavení hodnoty atributu
alisa.krevní_skupina = 'A'
alisa.krevní_skupina   -->   A

# použití funkce setattr()
setattr(alisa, 'krevní_skupina', 'B')
alisa.krevní_skupina   -->   B
```

Introspekce – příklad

Použijme pěkný příklad z http://www.ibm.com/developerworks/library/l-pyint.html . Funkce *interrogate()*..

```
def interrogate(item):
    """Print useful information about item."""

    if hasattr(item, '__name__'):
        print("NAME:     ", item.__name__)

    if hasattr(item, '__class__'):
        print("CLASS:    ", item.__class__.__name__)

    print("ID:   ", id(item) )

    print("TYPE:    ", type(item) )

    print("VALUE:   ", repr(item) )

    print("CALLABLE: ", end='')
    if callable(item):
        print("Yes")
    else:
        print("No")

    if hasattr(item, '__doc__'):
        doc = getattr(item, '__doc__')
    doc = doc.strip()   # Remove leading/trailing whitespace.
    firstline = doc.split('\n')[0]
    print("DOC:  ", firstline)
```

..dává při následujícím volání..

```
from interrogate import interrogate

interrogate('řetězec')
print()
interrogate(42)
print()
interrogate(interrogate)
```

..výstup:

```
CLASS:    str
ID:       12237248
TYPE:     <class 'str'>
VALUE:    'řetězec'
CALLABLE: No
DOC:      str(string[, encoding[, errors]]) -> str

CLASS:    int
ID:       505606008
TYPE:     <class 'int'>
VALUE:    42
CALLABLE: No
DOC:      int(x[, base]) -> integer

NAME:     interrogate
CLASS:    function
ID:       12955416
TYPE:     <class 'function'>
VALUE:    <function interrogate at 0x00C5AF18>
CALLABLE: Yes
DOC:      Print useful information about item.
```

Jak poznamenává uvedený zdroj: *„Můžeme vyslýchat dokonce sami sebe – lepší už to nebude.“* ^_^

Třídní atributy

I. Někdy se nám ale může hodit operovat nikoli s atributy konkrétní instance třídy, nýbrž s atributy rodiče. Vzpomeneme-li si na první slajd, je implementace jasná – stačí vynechat slovíčko *self* a odkazovat se přímo na uvedenou rodičovskou třídu (neuvedený kód se opět zachovává):

```
class Školák(Člověk):
  ...
  školáků = 0
  ...
  def __init__(self, ...):
      ...
      Školák.školáků += 1
```

Síla *třídních atributů* tkví v tom, že jsou sdíleny všemi instancemi třídy mezi sebou navzájem (jak už jsme viděli), takže jakákoli změna v nich je viditelná ve všech jednotlivých instancích/objektech vytvořených z dané třídy.

> → Zjevně místo *Školáka* můžete dosadit libovolný jiný dostupný objekt, ale to už by daný kód dělal něco úplně jiného, než jste zamýšleli… Proto je mnohem lepší napsat to rovnou jako dále pomocí *třídní metody*.

II. Na třídních atributech pak mohou operovat komplementární *třídní metody* (`@classmethod`):

```
class Školák(Člověk):
  ...
  školáků = 0
  ...

  def __init__(self, ...):
      ...
      self.přidej_školáka()

  @classmethod
  def přidej_školáka(cls):
      cls.školáků += 1
```

→ Podobně jako metody instanční, třídní metody také dostávají první implicitní parametr. Narozdíl od instačních metod ale tímto parametrem není příslušná instance, nýbrž původní třída! Pro lepší čitelnost se proto doporučuje tento parametr značit jako *cls* (což opět není povinné, ale asi už chápete, proč je to *velmi* doporučované).

→ *@classmethod* je tzv. *dekorátor* funkce. Prozatím ho berte jako část syntaxe jazyka, která slouží k označení třídnosti dané metody.

Statické atributy

Kromě *instančních* a *třídních* metod v Python'u existují i metody *statické*, zaváděné pomocí dekorátoru `@staticmethod`:

```
@staticmethod
def pomocná_metoda(par1, par2):
    ...
```

Jak je vidět, narozdíl od běžných (instančních) metod a metod třídních neočekávají a také nedostávají žádný první implicitní parametr.

„Privátní“ atributy

I. Python nemá koncepci skutečně soukromých, privátních atributů. Ale jakýkoliv atribut, jehož *jméno začíná alespoň dvěma podtržítky a končí podtržítkem nejvýše jedním* se bude jako privátní tvářit – nebude totiž na

třídě (ani instanci) pod tímto jménem dostupný:

```
>>> class MyClass:
...     a = 1
...     __b = 2

# Na atribut 'a' se dostaneme normálně:
>>> MyClass.a
1

# S atributem '__b' už to nejde:
>>> MyClass.__b
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    MyClass.__b
AttributeError: type object 'MyClass' has no attribute '__b'
```

→ Technicky dojde k nahrazení jména atributu `__atribut` za `_classname__atribut`, kde *classname* je vytvořeno ze jména příslušné třídy (z něhož jsou nejdříve odstraněny případná začáteční podtržítka). V tomto příkladě bude tedy třída obsahovat atribut `MyClass._MyClass__b` (který se takto i bude vypisovat v rámci `dir(MyClass)`).

II. Podobně atributy (data i metody) začínající na *jedno* podtržítko jsou konvencí brány za *neveřejnou část* API (rozhraní třídy). Narozdíl od dvojpodtržítkových jsou vidět a jsou normálně přístupné, ale všichni by se jim měli vyhýbat jako čert kříži ^_~