

knihovna programátora

- Učebnice jazyka i referenční příručka v jednom
- Podrobný výklad konstrukcí objektově orientovaného programování
- Přehledné vysvětlení základních principů, na nichž je jazyk postaven a jejichž znalost umožní lépe využívat jeho možnosti
- Dekorátory, deskriptory, generátory, metatřídy
- Vlákna, procesy, asynchronní konstrukce, subinterpret
- Navazuje na titul věnovaný algoritmickým konstrukcím



Rudolf Pecinovský

Python 3.14

OBJEKTOVÉ KONSTRUKCE



*Mé ženě Jarušce a dětem
Štěpánce, Pavlínce, Ivance a Michalovi*

knihovna programátora

RUDOLF PECINOVSKÝ

Python 3.14

OBJEKTOVÉ KONSTRUKCE

GRADA
Publishing

Rudolf Pecinovský

Python 3.14

Objektové konstrukce

Vydala Grada Publishing, a.s.
U Průhonu 22, Praha 7
obchod@grada.cz, www.grada.cz
tel.: +420 234 264 401
jako svou 10352. publikaci

Odpovědný redaktor Petr Somogyi
Grafická úprava a sazba Rudolf Pecinovský
Počet stran 600
První vydání, Praha 2025
Vytiskly Tiskárny Havlíčkův Brod, a. s.

© Grada Publishing, a.s., 2025
Cover Design © Grada Publishing, a.s., 2025
Cover Photo © Depositphotos/cristi180884

Upozornění pro čtenáře a uživatele této knihy
Všechna práva vyhrazena. Žádná část této tištěné či elektronické knihy nesmí být reprodukována a šířena v papírové, elektronické či jiné podobě bez předchozího písemného souhlasu nakladatele. Neoprávněné užití této knihy bude trestně stíháno. Názvy produktů, firem apod. použité v knize mohou být ochrannými známkami nebo registrovanými ochrannými známkami příslušných vlastníků. Automatizovaná analýza textů nebo dat ve smyslu čl. 4 směrnice 2019/790/EU a použití této knihy k trénování AI jsou bez souhlasu nositele práv zakázány.

ISBN 978-80-271-8362-3 (pdf)

ISBN 978-80-271-6062-4 (print)

Stručný obsah

Úvod	28
------------	----

Část A Objektově orientované programování	37
--	-----------

Kapitola 1 Základy OOP	38
Kapitola 2 Třídy a jejich instance	59
Kapitola 3 Jednoduché dědění	90
Kapitola 4 Násobné dědění	105
Kapitola 5 Vlastnosti, abstraktní třídy a protokoly	123
Kapitola 6 Další objektové konstrukce	146

Část B Prohloubení probraných témat	173
--	------------

Kapitola 7 Moduly, balíčky, virtuální prostředí	174
Kapitola 8 Volatelné objekty a jejich vlastnosti	200
Kapitola 9 Dekorátory	215
Kapitola 10 Třídní metody	229
Kapitola 11 Iterátory a generátory	241
Kapitola 12 Výčtové typy	260
Kapitola 13 Datové třídy	292
Kapitola 14 Anotace a koncept typového systému	308
Kapitola 15 Generické datové typy a generické funkce	338
Kapitola 16 Výjimky a varování	358

Část C Pokročilejší objektové konstrukce	385
Kapitola 17 Přetěžování operátorů	386
Kapitola 18 Ovlivnění přístupu k atributům	416
Kapitola 19 Správa paměti a pokročilé techniky práce s objekty	445
Kapitola 20 Ovlivnění tvorby tříd, metatřídy	456
 Část D Programování souběžných úloh	 477
Kapitola 21 Principy souběžného programování	478
Kapitola 22 Vlákna a modul threading	497
Kapitola 23 Podprocesy a modul multiprocessing	520
Kapitola 24 Korutiny a modul asyncio	546
Kapitola 25 Subinterprety a modul concurrent.interpreters	567
 Literatura.....	 591
Rejstřík	593

Podrobný obsah

Úvod	28
Komu kniha není určena	28
Komu je kniha určena	29
Struktura příručky	29
Koncepte výkladu	30
Jazyk identifikátorů	31
Potřebné vybavení	31
Operační systém	32
Doprovodné programy a jejich organizace	32
Odkazy do prvního dílu a modul <code>dbg</code>	33
Použité typografické konvence	33
Odbočka – podšeděný blok	35
Zpětná vazba	36
 Část A Objektově orientované programování	37
Kapitola 1 Základy OOP	38
1.1 Přehled	38
1.1.1 Kdy se OOP začíná vyplácet	39
1.1.2 Objektově orientované paradigma	39
Postup dle procedurálního paradigmatu	40
Postup dle objektově orientovaného paradigmatu	40
1.1.3 Výhody postupu podle OOP	40
1.1.4 Různé pohledy na OOP	41
1.2 Základní princip OOP	42
1.3 Objekty a jejich atributy	42
1.4 Třídy a jejich instance	43
Terminologická vsuvka	44
1.4.1 Třída versus datový typ	45
1.4.2 Instance	45
1.5 Objekt třídy versus instance třídy	46
1.6 Atributy třídy versus atributy instancí	46
1.7 Zprávy	47
1.8 Metody	47
1.8.1 Metody versus funkce	48
1.9 Dědění	48
1.9.1 Terminologie	48
Dědění versus dědičnost	49
1.9.2 LSP – substituční princip Liskové	49
1.9.3 Virtuální metody a jejich přebíjení	49
1.9.4 Zakrývání versus přebíjení	50
1.9.5 Polymorfismus	50
1.9.6 Rodičovský podobjekt	51

1.9.7	Násobné dědění a diamantový problém.....	51
1.9.8	Zobecňování.....	52
1.10	Obecnější pohled na dědění.....	53
1.10.1	Rozhraní versus implementace.....	53
1.10.2	Signatura versus kontrakt.....	54
1.10.3	Formalizace popisu rozhraní – API.....	54
1.10.4	Dva způsoby dědění.....	55
	Jmenovité dědění (nominal subtyping).....	55
	Strukturální dědění.....	55
1.10.5	Tři druhy dědění.....	56
	Přirozené (nativní) dědění.....	56
	Dědění rozhraní.....	56
	Dědění implementace.....	57
	Shrnutí.....	57
1.11	Shrnutí kapitoly a doprovodné soubory.....	58
1.11.1	Doprovodné soubory.....	58
Kapitola 2	Třídy a jejich instance.....	59
2.1	Definice třídy a jejích atributů.....	59
2.1.1	Definice třídy je příkaz.....	61
2.2	Práce s atributy objektu.....	61
2.2.1	Získání a modifikace hodnoty atributu.....	61
2.2.2	Přidání a odebrání atributu.....	62
2.2.3	Omezení zabudovaných typů.....	64
2.2.4	Neveřejné atributy.....	64
2.2.5	Třída jako parametr a návratová hodnota.....	64
2.3	Instance a práce s nimi.....	65
2.3.1	Textový podpis instance.....	66
2.3.2	Kvalifikace atributu třídy instancí.....	66
	Datové atributy.....	66
	Metody.....	67
2.3.3	Tři druhy metod a použití příslušných dekorátorů.....	68
2.3.4	Metody instancí a parametr <code>self</code>	71
2.3.5	Vytváření instancí – konstruktor, alokátor, initor.....	73
2.3.6	Instanční metody třídy <code>C2</code>	74
	Terminologická vsuvka: co je konstruktor.....	75
2.3.7	Použití instančních a třídních atributů.....	76
2.3.8	Změny instančních a třídních datových atributů.....	77
2.3.9	Atribut třídy jako výchozí hodnota instančního atributu.....	78
2.3.10	Převod metod na funkce a funkcí na metody.....	80
2.3.11	Zavádění nových třídních funkčních atributů.....	81
2.3.12	Zavádění nových instančních funkčních atributů.....	82
2.4	Vestavěné třídy jsou nemodifikovatelné.....	83
2.5	Nutnost kvalifikace atributů.....	83
2.6	Některé speciální atributy – dunderý.....	85
	<code>__bases__</code>	85
	<code>__class__</code>	85
	<code>__dict__</code>	85
	<code>__dir__(self)</code>	85
	<code>__doc__</code>	85
	<code>__module__</code>	85
	<code>__mro__</code>	85
	<code>__name__</code>	86
	<code>__qualname__</code>	86
	<code>__subclasses__()</code>	86
	Ukázky použití.....	86
2.7	Alternativní definice třídy.....	87

2.8 Odložené vyhodnocování anotací.....	88
2.9 Shrnutí kapitoly a doprovodné soubory.....	88
2.9.1 Doprovodné soubory.....	89
Kapitola 3 Jednoduché dědění.....	90
3.1 Vytváření potomka a jeho vlastnosti	90
3.2 Příklad.....	91
3.2.1 Třída LA.....	91
3.2.2 Třídy LB a LC	93
3.3 Jmenné prostory.....	93
3.3.1 Atributy a jmenné prostory tříd.....	94
3.3.2 Atributy a jmenné prostory instancí.....	95
3.4 Volání metod v hierarchii dědění	96
3.4.1 Jak potvrdit anebo zakázat přebíání metody.....	97
3.5 Initory se dědí	97
3.6 Hierarchie výjimek.....	98
3.6.1 Dopad hierarchie výjimek na jejich zpracování	99
Povinnost řadit rodiče za potomky	100
To byl jenom začátek.....	100
3.7 Definice vlastní výjimky	101
3.8 Varování a jeho hierarchie.....	102
3.9 Vestavěné funkce pracující s objekty	102
callable(object).....	102
classmethod(method, *args, **kwargs).....	102
delattr(object, name)	102
getattr(object, name[, default])	102
hasattr(object, name)	103
isinstance(object, classinfo)	103
issubclass(class, classinfo)	103
setattr(object, name, value)	103
staticmethod(method, *args, **kwargs).....	103
super([type[, object-or-type]]).....	103
class type(object) class type(name:str, bases:tuple[type], dict:dict[str,Any], **kwargs) -> type.....	103
3.10 Shrnutí kapitoly a doprovodné soubory.....	104
3.10.1 Doprovodné soubory	104
Kapitola 4 Násobné dědění.....	105
4.1 Python a násobné dědění	105
Vytváření posloupnosti MRO	106
4.2 Příklad: jednoduchý diamant.....	107
4.3 Analýza chování.....	108
4.3.1 Přímé zadání volaného initoru	110
4.3.2 Problémy s parametry a argumenty – hubnoucí parametr	111
4.3.3 Virtuální metody	113
4.4 Pořadí prohledávání – MRO	113
4.5 Složitější hierarchie dědění	116
4.5.1 Nerealizovatelná a následně opravená hierarchie	116
4.5.2 Zásady specifikace MRO	117
4.6 Komolení jmen a pseudosoukromé atributy.....	118
4.7 Šablonová metoda.....	119
4.8 Shrnutí kapitoly a doprovodné soubory.....	121
4.8.1 Doprovodné soubory	122

Kapitola 5 Vlastnosti, abstraktní třídy a protokoly	123
5.1 Atributy × vlastnosti	123
5.2 Přímá definice vlastnosti	124
5.2.1 Využití lambda-výrazů, atributy jen pro čtení	126
5.2.2 Vztah vlastnosti k třídě a instancím	128
5.3 Zadání vlastnosti pomocí dekorátoru	129
5.3.1 Použití	130
5.3.2 Shrnutí	132
5.4 Definice vlastností třídy	132
5.5 Abstraktní × konkrétní třídy a metody	132
5.5.1 Princip abstraktních metod	133
5.5.2 Omezení abstraktních tříd	133
5.5.3 Abstraktní a konkrétní třídy v mainstreamových jazycích	134
5.5.4 Formálně a skutečně abstraktní třídy a metody v jazyce Python	134
5.6 Definice skutečně abstraktní třídy	135
5.6.1 Přebíjet musí i ti, kteří abstraktní metodu nepotřebují	136
5.6.2 Definice potomka abstraktní třídy	136
5.6.3 Proč definovat metodu použitou v šabloně	137
5.6.4 Proč dekorovat metodu jako abstraktní	138
5.6.5 Proč v hlavičce deklarovat předka ABC	138
5.6.6 Shrnutí	138
5.7 Abstraktní statické a třídí metody a vlastnosti	138
5.8 Statické, dynamické a kachní typování	140
5.8.1 Statické typování	140
5.8.2 Dynamické typování	140
5.8.3 Kachní typování	140
5.8.4 Statické kachní typování a jeho kontrola	141
5.9 Protokoly	142
5.9.1 Deklarace rozhraní v mainstreamových jazycích	142
5.9.2 Třída <code>Protocol</code> a dekorátor <code>runtime_checkable</code>	142
5.9.3 Protokoly a statické kachní typování	145
5.10 Shrnutí kapitoly a doprovodné soubory	145
5.10.1 Doprovodné soubory	145
Kapitola 6 Další objektové konstrukce	146
6.1 Výčtové typy – základy	146
6.1.1 Změny mezi verzemi	147
6.1.2 Trocha terminologie	147
Implicitní podoba podpisů	147
6.1.3 Výčtový typ definovaný voláním funkce	147
6.1.4 Vlastnosti výčtového typu a jeho instancí	149
Tři způsoby získání instance výčtového typu	149
Další zajímavé informace	149
6.1.5 Výčtový typ definovaný jako třída	150
6.1.6 Neměnné a proměnné části	151
6.2 Datové třídy – základy	152
6.3 Generické datové typy	154
6.4 Srovnávání objektů se vzory v příkazu <code>match</code>	154
6.4.1 Opakování terminologie	154
6.4.2 Dosazování hodnot	155
6.4.3 Prověřování posloupností	156
6.4.4 Vzory s výběrem hodnot	158
6.5 Definice vlastního správce kontextu	159
<code>__exit__(self, exc_type, exc_value, traceback)</code>	160
6.5.1 Uschopnění robota Karla vystupovat jako správce kontextu	160
6.5.2 Příklad využívající robota Karla jako správce kontextu	162
6.5.3 Sdružování	163

6.5.4	Příklad.....	163
6.5.5	Jak to funguje – emulace příkazu with	163
6.6	T-stringy – bezpečné šablony	166
6.6.1	Syntaxe a vytvoření šablony	166
6.6.2	Princip fungování t-stringů	167
6.6.3	API t-stringů	167
	Atributy instancí třídy <code>string.Template</code>	167
	Atributy instancí třídy <code>string.Template.Interpolation</code>	168
6.6.4	Zobrazení výsledného stringu	168
	Aplikace na HTML	170
6.6.5	Dosazování nových hodnot	170
6.6.6	Porovnání t-stringů s f-stringy a třídou <code>string.Template</code>	171
6.7	Shrnutí kapitoly a doprovodné soubory	172
6.7.1	Doprovodné soubory	172

Část B Prohloubení probraných témat 173

Kapitola 7	Moduly, balíčky, virtuální prostředí	174
7.1	Zprostředkovaný import	174
7.2	Oprava načteného modulu	175
7.2.1	Korektní opětovné načtení	177
7.2.2	Opětovné načtení modulu s běhovou chybou	178
7.2.3	Opětovné načtení modulu s běhovou chybou	178
7.3	Získání odkazu na modul, atributy modulu	178
7.3.1	Získání odkazu na existující modul z názvu modulu	179
7.3.2	Načtení (import) modulu se zadaným názvem	179
	<code>importlib.import_module(name, package=None)</code>	179
	<code>importlib.invalidate_caches()</code>	179
7.3.3	Systémové atributy modulu	180
7.4	Koncepce balíčků a její důsledky	180
7.4.1	Fyzická a logická pozice modulu	180
7.4.2	Absolutní a relativní import	180
7.4.3	Modul nemusí znát svoji logickou lokaci	181
7.5	Virtuální prostředí Pythonu	181
7.5.1	Vytvoření virtuálního prostředí	182
7.5.2	Spuštění virtuálního prostředí	185
7.5.3	Přímé spuštění interpretu ve virtuálním prostředí	187
7.6	Kde hledat moduly	188
7.6.1	Přidání stromu složek při startu Pythonu – soubory <code>.pth</code>	188
	Umístění souborů <code>.pth</code>	188
	Obsah souborů <code>.pth</code>	189
	Ukázka použití	190
7.7	Neinicializované balíčky – NS-balíčky	192
7.7.1	Teoretický úvod	192
	NS-balíčky jsou pouze zobečněním pravidel pro kořenový balíček	193
	NS-balíček smí mít pouze NS rodiče	193
7.7.2	Postup hledání modulu	193
7.7.3	Důsledky	194
7.7.4	Standardní balíček v NS-balíčku	194
7.7.5	Příklad	195
	Modul <code>mA_importing</code>	195
	Importované moduly	195
	Spuštění	197
7.8	Shrnutí kapitoly a doprovodné soubory	198
7.8.1	Doprovodné soubory	199

Kapitola 8 Volatelné objekty a jejich vlastnosti	200
8.1 Co jsou volatelné objekty	200
8.1.1 Dosud probrané volatelné objekty	200
8.1.2 Jak udělat objekt volatelným	201
8.1.3 Jak definovat volatelný modul	202
8.2 ZNA a jeho rámce	203
8.3 Požadavky funkcionálního programování	205
8.4 Funkce jako plnohodnotné objekty, funkce vyššího řádu	206
8.4.1 Funkce jako návratová hodnota	206
8.4.2 Funkce jako parametr a argument	207
8.4.3 Použití modulu operator při předávání operátorů v parametru	208
8.5 Lambda-výrazy jako anonymní funkce	209
8.6 Uzávěry	209
8.6.1 Definice vnější a vnitřní funkce	210
8.6.2 Nutnost použití uzávěru	211
8.6.3 Prověrka	211
8.6.4 Jak to celé funguje	211
8.7 Částečně vyhodnocené funkce	212
8.7.1 Použití lambda výrazů	213
8.7.2 Použití funkce functools.partial()	213
8.8 Shrnutí kapitoly a doprovodné soubory	214
8.8.1 Doprovodné soubory	214
Kapitola 9 Dekorátory	215
9.1 Co jsou dekorátory	215
9.1.1 Terminologie	216
9.1.2 „Operátor“ @	216
Dosavadní použití dekorátorů	217
9.1.3 Prefixové a postfixové použití dekorátoru	218
Dekorace externího objektu	218
Vytažení některých atributů do obálky	218
9.2 Tvorba jednoduchého dekorátoru	219
9.3 Dekorátor definovaný jako třída	220
9.4 Dekorátory s parametry	223
9.5 Současné použití více dekorátorů	223
9.6 Dekorátory třídy	224
9.6.1 Jedináček – singleton	224
9.7 Ukládání výsledků do vyrovnávací paměti	226
9.8 Shrnutí kapitoly a doprovodné soubory	228
9.8.1 Doprovodné soubory	228
Kapitola 10 Třídní metody	229
10.1 K čemu je dekorátor classmethod	229
10.2 Tovární metody	231
10.3 Ovlivnění tvorby dceřiných tříd	232
Definice třídy C04a	233
Definice potomků	235
10.3.1 Zákaz definice potomků dané třídy	235
10.4 Jedináček definovaný pomocí __new__()	235
10.5 Komplexnější použití třídních metod	236
10.5.2 Návrhový vzor Multiton	236
10.5.3 Definice společné rodičovské třídy	237
10.5.4 Prověrka	239
10.6 Shrnutí kapitoly a doprovodné soubory	240
10.6.1 Doprovodné soubory	240

Kapitola 11	Iterátory a generátory	241
11.1	Iterovatelné objekty – iterables	241
11.2	Iterátory	242
	__iter__(self)	242
	__next__(self)	242
11.2.1	Příklad	243
11.3	Generátorový výraz	244
11.4	Nekorektní použití generátorového výrazu	245
11.5	Generátorová funkce a generátorový iterátor	245
11.6	Definice generátorové funkce jako zdroje	247
11.7	Použití více příkazů yield v těle funkce	248
11.8	Vestavěné funkce související s iteracemi	249
	iter(object[, sentinel])	249
	map(function, iterable, ...)	249
	next(iterator[, default])	250
11.9	Jak to celé funguje	250
11.10	Operátor yield – yield jako výraz	250
	Analogie – AlzaBox	251
11.10.1	Demonstrace ruční aktivace generátoru s výrazem yield	251
	Popis funkce counter	252
	Použití funkce counter	253
11.10.2	Demonstrační AHA-příklad použití výrazu yield	254
	Generátorová funkce ord_gen()	254
	Funkce orders()	255
11.10.3	Metoda send() nemusí zadávat jen čísla	257
11.11	Výraz yield from	257
11.11.1	Trochu složitější příklad	258
11.12	Shrnutí kapitoly a doprovodné soubory	259
11.12.1	Doprovodné soubory	259
Kapitola 12	Výčtové typy	260
12.1	Výčtové typy nejsou součástí definice jazyka	260
12.2	Další možnosti výčtových typů	261
12.2.1	Definice „obyčejných“ atributů	261
12.2.2	Přezdívky hodnot	262
12.2.3	Automatické přiřazení obalovaných hodnot	264
12.2.4	Poloautomatické přiřazení obalovaných hodnot	265
	Poznámka k vývoji Pythonu	266
12.3	Zveřejnění prvků – dekorátor global_enum	266
12.4	Výměna systémových metod	268
12.5	Útroby výčtového typu	270
12.5.1	Výčtové typy jako programový incest	270
	Shrnutí	271
12.5.2	Interní atributy výčtových typů a jejich instancí	271
12.5.3	Užitečné interní metody	273
	_generate_next_value(name, start, count, last_values)	273
	_new_member(cls, *args)	275
	_missing(cls, value)	275
	_new__(cls, *args)	275
12.5.4	Využití definice vlastního alokátoru __new__()	275
	Definice alokátoru – metody __new__()	277
	Definice zbylých metod	277
	Vytvoření objektu třídy D4	278
	Prověrka funkcionality	278

Proč nestačilo definovat <code>__init__()</code>	279
Definice třídy D4x	279
Vytvoření objektu třídy D4x	279
Porovnání atributů tříd D4 a D4x	279
Závěr	281
12.6 Výčtové typy s více předky	281
12.7 Odvozené výčtové typy	281
Co to jsou mixiny a k čemu jsou dobré	282
12.7.1 Prázdné výčtové typy	283
12.7.2 ReprEnum	284
12.7.3 IntEnum	284
12.7.4 StrEnum	285
12.7.5 Flag	286
12.7.6 Reakce na hodnoty mimo rozsah – třída FlagBoundary	287
Barevné systémy	288
STRICT	288
CONFORM	289
EJECT	289
KEEP	290
12.7.7 IntFlag	290
12.8 Shrnutí kapitoly a doprovodné soubory	291
12.8.1 Doprovodné soubory	291
Kapitola 13 Datové třídy	292
13.1 Představení	292
13.2 Definice	292
<code>dataclass(*, init=True, repr=True, eq=True, order=False,</code> <code>unsafe_hash=False, frozen=False, match_args=True, kw_only=False,</code> <code>slots=False, weakref_slot=False)</code>	293
13.2.1 Omezení implicitních počátečních hodnot	295
Pořadí	295
Proměnnost	295
13.2.2 Užitečné datové atributy	296
KW_ONLY	296
MISSING	297
13.3 Funkce field()	297
<code>field(*, default=MISSING, default_factory=MISSING, init=True,</code> <code>repr=True, hash=None, compare=True, metadata=None, kw_only=MISSING)</code>	297
Příklady použití funkce field() s argumentem default_factory	298
13.4 Vylepšujeme vytváření instancí	300
13.4.1 Třídní datové atributy	300
Vliv třídního atributu na implicitní hodnotu	300
Zdůraznění třídnosti atributu v anotaci	301
13.4.2 Pole v hierarchii dědění	301
13.4.3 Přerovnávání zděděných polí při povinném pojmenování	302
13.4.4 Metoda <code>__post_init__()</code>	303
Doplnění volání initoru předka	303
13.4.5 Inicializační proměnné – anotace InitVar	304
13.5 Další užitečné funkce	305
<code>asdict(obj, *, dict_factory=dict)</code>	306
<code>astuple(obj, *, tuple_factory=tuple)</code>	306
<code>replace(obj, /, **changes)</code>	306
13.6 Shrnutí kapitoly a doprovodné soubory	307
13.6.1 Doprovodné soubory	307

Kapitola 14	Anotace a koncept typového systému	308
14.1	Trocha historie	308
14.1.1	Výhody používání typových anotací	309
14.2	Statické a dynamické testování	310
	Dynamické testování	310
	Statické testování	310
14.2.1	Nejpoužívanější statické analyzátoru a testery	311
	MyPy	311
	Pyright	311
	Pylance	311
	MonkeyType / Pyannotate	312
	Shrnutí	312
14.2.2	Nejpoužívanější knihovny pro tvorbu dynamických testů	312
14.3	Zápis anotace	312
14.4	Deklarace běžných typů	314
14.4.1	Jednoduché typy	314
14.4.2	Kontejnery	314
14.4.3	Volatelné objekty	315
14.4.4	Zabudované datové typy a jejich přezdívky („aliasy“)	316
14.5	Atribut <code>__annotations__</code>	317
14.5.1	Geneze odloženého vyhodnocování anotací	318
	Okamžité vyhodnocování	318
	Stringová sebeobrana	318
	Postponed evaluation	318
	Deferred evaluation	318
14.5.2	Starší řešení – postponed evaluation	319
	Modul <code>__future__</code>	320
14.5.3	Současné implicitní řešení (deferred evaluation)	322
	Přepnutí do režimu postponed	322
14.6	Modul <code>typing</code>	324
14.6.1	Starší definice přezdívek typů – funkce <code>TypeAlias</code>	324
14.6.2	Moderní definice přezdívek typů – příkaz <code>type</code> (Python 3.12+)	325
	Příkaz <code>type</code> a generické typy	325
14.6.3	Sjednocení datových typů – třída <code>Union</code>	326
	Datový typ <code>Optional[x]</code>	327
14.6.4	Třída <code>TypeVar</code> a její použití	327
	Konstruktor	327
	<code>TypeVar(name, *constraints, bound=None, covariant=False, contravariant=False)</code>	328
	Použití	328
14.6.5	Zavedení nového datového typu – třída <code>NewType</code>	329
14.6.6	Několik užitečných anotací	331
	<code>final</code>	331
	<code>override</code>	331
	<code>LiteralString</code>	332
	<code>Never</code> <code>NoReturn</code>	332
	<code>Self</code>	332
14.7	Postupné typování (gradual typing)	332
14.7.1	Datový typ <code>Any</code>	333
14.7.2	Funkce <code>cast()</code>	334
14.8	Zdánlivé přetěžování funkcí a metod	334
14.9	Shrnutí kapitoly a doprovodné soubory	337
14.9.1	Doprovodné soubory	337
Kapitola 15	Generické datové typy a generické funkce	338
15.1	Představení	338

15.1.1	Používání generických datových typů.....	339
15.2	Definice vlastních generických datových typů.....	339
15.2.1	Starší podoba definice generického typu (Python 3.5+)	340
15.2.2	Současná podoba definice generického typu (Python 3.12+)	341
15.2.3	Definice generického typu s více typovými proměnnými	342
15.2.4	Generické přezdívky typů – příkaz <code>type[T]</code>	342
	Pojmenování tvaru místo rozepisování detailů	343
	Přehlednost API a snadná změna na jednom místě.....	343
	Neplést s anotačním zápisem <code>type[T]</code>	344
15.3	Kladení požadavků na typové proměnné	344
15.3.1	Omezení typových parametrů na potomky	344
15.3.2	Výčet možných typů	345
15.3.3	Výchozí hodnoty typových parametrů (Python 3.13+)	347
15.3.4	Další možnosti dosažitelné pomocí <code>TypeVar</code>	347
	Kovariance	348
	Kontravariance	348
	Odvození variance	349
15.4	Generické funkce a metody	349
15.5	Pomocné funkce pro výstižnější anotace	349
	<code>Final</code>	350
	<code>Literal</code>	350
	<code>Optional</code>	351
	<code>TypeGuard</code>	351
15.6	Funkce <code>typing.Annotated</code>	352
15.6.1	Motivace k zavedení funkce <code>Annotated</code>	353
15.6.2	K čemu se třída <code>Annotated</code> používá.....	354
15.6.3	Příklad zadání metadat pro datovou třídu	354
15.6.4	Demo běhové verifikace objektu pomocí metadat z <code>Annotated</code>	355
15.7	Shrnutí kapitoly a doprovodné soubory.....	357
15.7.1	Doprovodné soubory.....	357
Kapitola 16	Výjimky a varování.....	358
16.1	Anatomie výjimky	358
16.1.1	Proč <code>BaseException</code>	358
16.1.2	Důležité atributy	359
	<code>args</code>	359
	<code>__cause__</code>	359
	<code>__context__</code>	359
	<code>__notes__</code>	359
	<code>__suppress_context__</code>	359
	<code>__traceback__</code>	359
	Zřetězený seznam	360
16.1.3	Důležité metody	360
	<code>__str__()</code>	361
	<code>__repr__()</code>	361
	<code>add_note(note:str) -> None</code>	361
	<code>with_traceback(tb)</code>	361
16.2	Doplnění výjimky o poznámku	361
16.2.1	Kód chyby.....	362
16.3	Příkaz <code>raise...from</code> a kontext výjimky	363
16.3.1	Příkaz <code>raise ... from None</code> a vyhození ošetřované výjimky	365
16.4	Definice vlastní reakce na neošetřené výjimky	366
16.4.1	Kdy definovat vlastní <code>excepthook</code>	367
16.4.2	Kdy vlastní <code>excepthook</code> nedefinovat	367
16.4.3	Závěrečná doporučení a upozornění.....	367

Uložení a volání původního hooku	367
Robustnost samotného excepthook	368
Zabalení hlavní funkce programu	368
Frameworky	368
Služby pro reportování chyb	368
16.5 Skupinové výjimky a větve except*	368
16.5.1 Zavedené třídy a větve except*	369
16.5.2 Definice skupinové výjimky	369
16.5.3 Atributy a metody třídy BaseExceptionGroup	371
message	371
exceptions	371
subgroup(condition) -> typing.Self	371
split(condition) -> tuple[typing.Self, typing.Self]	371
derive(excs) -> typing.Self	371
__traceback__	373
Poznámka o konstruktorech a metodě __new__()	373
16.5.4 Příklad: použití v definici funkce řešící komplexní problém	373
16.5.5 Závěrečná doporučení	373
16.6 Test, zda se právě neošetřuje výjimka	374
Funkce sys.exception()	376
Závěrečné doporučení	376
16.7 Získání informací o volajících	376
16.7.1 ZNA a jeho rámce	376
Objekty typu types.FrameType	376
Objekty typu types.CodeType	377
16.7.2 Třída traceback	377
16.7.3 Využití modulu inspect – nahlédnutí pod kapotu Pythonu	378
Funkce stack() a instance třídy inspect.FrameInfo	378
Příklad využití modulu inspect při tvorbě dekorátoru:	379
Prověrka vytvořeného dekorátoru	382
16.8 Shrnutí kapitoly a doprovodné soubory	383
16.8.1 Doprovodné soubory	384

Část C Pokročilejší objektové konstrukce

385

Kapitola 17 Přetěžování operátorů	386
17.1 Přetěžování metod versus přetěžování operátorů	386
17.2 Základy	387
17.2.1 Neimplementované operace – NotImplemented	387
17.3 Základní sada	388
17.3.1 Reprezentace objektu a jeho logická hodnota	388
__bool__(self)	388
__repr__(self), __str__(self)	388
__format__(self, format_spec)	388
17.3.2 Objekty reprezentující hodnotu (ORV) a objekty reprezentující entitu (ORE)	389
17.3.3 Neměnnost ORV objektů	390
17.3.4 Porovnávání objektů	390
__lt__(self, other) __le__(self, other) __eq__(self, other)	391
__ne__(self, other) __gt__(self, other) __ge__(self, other)	391
Není třeba definovat všechny	391
Demonstrační příklad	391
Interpret chápe metody obecněji	393
Obecnější mohou být i návratové hodnoty	393
17.3.5 Hešování objektů	394
__hash__()	394

17.3.6 Finalizace objektu	394
__del__()	394
Destruktor versus finalizér	395
17.4 Operátory + - * @ / // % ** << >> & ^ a emulace numerických typů	395
17.4.1 Binární operátory	396
Pravostranné (reflexivní) operátory	396
17.4.2 Unární operátory	397
17.4.3 Zbylé emulační funkce	397
17.4.4 Konverzní funkce	398
17.4.5 Souhrnný příklad: třída Zlomek	398
Přezdívky typů	403
Initor	403
Reprezentační metody	403
Konverzní metody	403
Unární operátory	404
Porovnávací operátory	404
Binární aritmetické operátory	404
Rozšířené přiřazení	404
17.4.6 Prověra správnosti definice třídy Zlomek	405
17.5 Operátor [], emulace indexovatelných kontejnerů	406
__contains__(self, item)	406
__delitem__(self, key)	407
__getitem__(self, key)	407
__iter__(self)	407
__len__(self)	407
__missing__(self, key)	407
__reversed__(self)	407
__setitem__(self, key, value)	407
17.5.1 Příklad	407
Prověra třídy PosloupnostMocnin	408
17.6 Operátor () – volatelné objekty	409
17.6.1 Předehra	410
17.6.2 Jednoduchý příklad	410
17.6.3 Něco trochu složitějšího	410
17.7 Další užitečné dunderly	412
17.7.1 Metody pro kopírování objektů	413
__copy__(self)	413
__deepcopy__(self, memo)	413
17.7.2 Metoda pro konverzi na index	413
__index__(self)	413
17.7.3 Metody pro zaokrouhlování	413
__round__(self, ndigits=None)	413
__trunc__(self)	414
__floor__(self)	414
__ceil__(self)	414
Závěr	414
17.8 Shrnutí kapitoly a doprovodné soubory	414
17.8.1 Doprovodné soubory	415
Kapitola 18 Ovlivnění přístupu k atributům	416
18.1 Předehra	416
18.2 Co jsou deskriptory	417
__get__(self, instance, owner=None)	417
__set__(self, instance, value)	417
__delete__(self, instance)	417
__set_name__(self, owner, name)	417

18.2.1	Demonstrační příklad	418
	Použití definovaných tříd	419
18.3	Dělení deskriptorů	419
18.3.1	Datové deskriptory	420
18.3.2	Nedatové deskriptory	420
18.4	Definice ekvivalentu třídy property	420
18.5	Deskriptor spravující hodnotu	423
18.5.1	Test vytvořeného deskriptoru	424
18.6	Pořadí vyhodnocování	426
	Proč XXX a ne None	426
	1. Datový deskriptor	427
	2. Atribut instance	427
	3. Nedatový deskriptor	427
	4. Řadový atribut třídy	428
	5. Chyba	428
18.7	Příklad: deskriptor pro odložené vyhodnocení	428
18.8	Jak definovat vlastnosti třídy	430
18.8.1	Kudy cesta nevede	430
18.8.2	Definice třídy classproperty	431
18.8.3	Demonstrace užití třídy classproperty	431
	Robustnost	433
	Překvapivé chování del	433
18.9	Operátor . (tečka) – přístup k atributům	434
	__getattr__(self, name)	434
	__getattr__(self, name)	434
	__setattr__(self, name, value)	434
	__delattr__(self, name)	435
	__dir__(self)	435
18.9.1	AHA-příklad – definice	435
18.9.2	AHA-příklad – prověrka	435
	Aktivita IDLE	436
	IDLE versus jiná prostředí	438
	Další příkazy	439
18.9.3	Lepší řešení	439
18.10	Jaký způsob správy přístupu k atributu zvolit	440
18.11	Ovlivnění přístupu k atributům modulu	441
18.12	Shrnutí kapitoly a doprovodné soubory	444
18.12.1	Doprovodné soubory	444
Kapitola 19	Správa paměti a pokročilé techniky práce s objekty	445
19.1	Co už víme z prvního dílu	445
	Vše je objekt	445
	Analogie: proměnná je telefon a odkaz je telefonní číslo	446
	Správa paměti	446
19.1.1	Proč řešit paměť a životní cyklus objektů	446
19.2	Základy správy paměti	446
19.2.1	Počítání odkazů	447
	Analogie s telefonními čísly	447
19.2.2	Architektura správy paměti v Pythonu	447
19.2.3	Správce paměti (Memory Manager)	447
19.2.4	Uklízeč (Garbage Collector)	447
19.2.5	Spolupráce uklízeče a správce paměti	448
19.3	Slots	448
19.3.1	Výhody a nevýhody slotů	449
19.3.2	Kdy __slots__ použít a kdy je nepoužívat	450
19.4	Slabé odkazy	450

19.4.1	Problém s nechtěným udržováním objektů	451
19.4.2	Řešení: weakref – odkaz bez závazků	452
19.5	Propojení slotů a slabých odkazů	454
19.6	Shrnutí kapitoly a doprovodné soubory	455
19.6.1	Doprovodné soubory	455
Kapitola 20	Ovlivnění tvorby tříd, metatřídy	456
20.1	Přehled k výkladu metatříd	456
20.2	Co jsou to metatřídy	457
20.3	Postup při vytváření třídy	458
20.4	Různé definice metatříd	460
20.4.1	Vzorový kód pro demonstraci funkce metatříd	460
20.4.2	Jednoduchý úvod s metatřídou definovanou jako funkce	462
	Import modulu <code>modules.m20b_MetaSimple</code>	463
20.4.3	Kompletní příklad s metatřídou definovanou jako funkce	464
	Import modulu <code>modules.m20c_MetaFun</code>	465
20.4.4	Metatřída definovaná jako třída	466
	Import modulu <code>modules.m20d_MetaCls</code>	467
20.4.5	Metatřída definovaná jako objekt	469
	Import modulu <code>modules.m20e_MetaObj</code>	470
20.4.6	Závěr	471
20.5	Jedináček definovaný prostřednictvím metatříd	472
20.6	Potřeba postfixové dekorace třídy	473
20.6.1	Import modulu	475
20.7	Shrnutí kapitoly a doprovodné soubory	476
20.7.1	Doprovodné soubory	476

Část D Programování souběžných úloh

477

Kapitola 21	Principy souběžného programování	478
21.1	Souběžné provádění více úloh	478
21.1.1	Souběžnost versus paralelismus	479
	Souběžnost (concurrency)	479
	Paralelismus (parallelism)	479
	Rozdíly	479
21.2	Plánovač, vlákna a strategie plánování	479
21.2.1	Kooperativní plánování	480
21.2.2	Preemptivní plánování	480
21.3	Procesy – vlákna – koprogramy – subinterprety	481
21.3.1	Procesy	481
21.3.2	Vlákna	482
21.3.3	Koprogramy a korutiny	482
	Historická vsuvka o vývoji terminologie	483
21.3.4	Subinterprety (novinka Pythonu 3.14)	483
21.4	Kategorizace řešených problémů	484
21.4.1	Úlohy vytěžující CPU – CPU-bound	484
21.4.2	Úlohy vytěžující vstup a/nebo výstup – IO-bound	485
21.5	Komunikace mezi programovými jednotkami	485
21.5.1	Předávání zpráv	485
21.5.2	Sdílená paměť	485
21.5.3	Kritická sekce a souběh	486
	Hazardy	487
21.5.4	Hybridní řešení	487
21.6	Synchronizace	488
21.6.1	Explicitní zámky	488
	Zámek pro čtenáře a zapisovatele	488

21.6.2	Monitory a implicitní zamykání	489
21.6.3	Rendezvous	489
21.6.4	Co dál?	489
21.7	Stavový diagram chování vláken	490
21.7.1	Analogie	491
21.7.2	Skutečnost je maličko komplexnější	492
21.8	Přístup jazyka Python	492
21.8.1	Další vývoj GIL	493
	Projekt nogil	493
	Subinterprety	493
21.9	Srovnání s jazyky Java a C#	494
21.10	Otevření dveří pro knihovny v dalších jazycích	494
21.10.1	Projekt HPy: nové, univerzálnější API	495
21.11	Shrnutí kapitoly a doprovodné soubory	495
21.11.1	Doprovodné soubory	496
Kapitola 22	Vlákna a modul threading	497
22.1	Předehra	497
22.2	Znovu Karel	498
22.2.1	Modul <code>util.karelcz_lib</code>	498
22.2.2	Funkce <code>multithread()</code>	498
	Proč to děláme tak složitě	499
22.3	Třída Thread	499
22.3.1	Konstruktor instancí třídy Thread	500
	<code>Thread(group=None, target=None, name=None, args=(), kwargs={}, *,</code> <code>daemon=None, context=None)</code>	500
22.3.2	Důležité atributy a metody třídy Thread	500
	<code>start()</code>	500
	<code>run()</code>	501
	<code>join(timeout=None)</code>	501
	<code>name</code>	501
	<code>is_alive()</code>	501
	<code>daemon</code>	501
22.4	Nezávislá vlákna	502
22.4.1	Vytvoření světa se čtyřmi dvorky	502
22.4.2	Příprava na úkol	503
22.4.3	Zabalení a spuštění úkolu současně pracujících robotů	504
22.5	Nezávislé čekání do doběh vlákna	505
22.5.1	Modifikované zadání	505
22.5.2	Zřetězení činností vláken	506
	Závěr	508
22.6	Nesynchronizované sdílení prostředků	508
22.6.1	Scénář se sdíleným zdrojem	509
22.6.2	Výchozí funkce pro dodavatele a odběratele	509
	Dodavatel	509
	Odběratel	509
22.6.3	Propojení nesynchronizovaných činností	510
22.6.4	Problémy nesynchronizovaných činností	512
22.7	Kritické sekce a zámky	513
22.7.1	Třída <code>threading.Lock</code>	514
	<code>acquire(blocking=True, timeout=-1)</code>	514
	<code>release()</code>	514
	Příklad	515
22.7.2	Skrytý pozůstatek démona	516
22.7.3	Správce kontextu a příkaz <code>with</code>	517
22.7.4	Úprava architektury	517

22.8 Shrnutí kapitoly a doprovodné soubory.....	518
22.8.1 Doprovodné soubory.....	519
Kapitola 23 Podprocesy a modul multiprocessing.....	520
23.1 Základní kameny modulu multiprocessing.....	520
23.1.1 Procesy a podprocesy.....	520
23.1.2 Oddělená paměť.....	521
23.1.3 Komunikace mezi procesy.....	521
23.1.4 Serializace.....	522
23.1.5 Třída multiprocessing.Process.....	522
23.1.6 Ochrana hlavního modulu: if __name__ == "__main__":.....	522
23.2 Klíčové objekty.....	523
23.2.1 Třída multiprocessing.Process.....	523
Process(group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None).....	523
run().....	523
start().....	523
join(timeout=None).....	523
name.....	524
is_alive().....	524
daemon.....	524
pid.....	524
terminate().....	524
close().....	524
23.2.2 Třída multiprocessing.Queue.....	524
Queue(maxsize).....	525
qsize().....	525
empty().....	525
put(obj, block=True, timeout=None).....	525
put_nowait(obj).....	525
get(block=True, timeout=None).....	525
get_nowait().....	525
close().....	525
join_thread().....	525
cancel_join_thread().....	525
23.2.3 Tovární funkce multiprocessing.Pipe().....	526
23.2.4 Třída multiprocessing.connection.Connection.....	526
send(obj).....	526
recv().....	526
close().....	526
23.2.5 Globální atributy modulu multiprocessing.....	526
active_children() -> list[Process].....	526
cpu_count() -> int.....	526
current_process() -> Process.....	526
parent_process() -> Process.....	527
23.2.6 Mrtvý proces je ještě potřeba „pohřbit“.....	527
23.3 Demonstrační příklad.....	527
23.3.1 Demonstrační programy budou moduly.....	527
23.3.2 Nefunkční přístup ke sdílení paměti.....	527
Kontrolní tisky.....	529
Hlavní proces.....	529
Nešlo by to jednodušeji?.....	530
Podproces – funkce worker().....	530
Spouštěcí blok.....	530
Spuštění programu.....	530

Závěr.....	532
23.3.3 Funkční řešení pomocí Queue.....	532
Tělo modulu.....	532
Hlavní proces – funkce <code>main()</code>	533
Podproces – funkce <code>worker()</code>	534
Spuštění.....	534
Možná překvapení.....	534
23.4 Knihovna <code>KarelCz7475</code> a její GUI.....	535
23.4.1 Návrhový vzor Posluchač.....	535
23.4.2 Základní architektura knihovny <code>KarelCz7475</code>	536
23.4.3 Jak implementovat posluchače v odděleném procesu.....	536
23.4.4 Návrh zástupce a jeho partnera.....	537
23.4.5 Demonstrace funkce zástupce.....	537
23.4.6 Návrh komunikace.....	537
23.4.7 GUI server.....	539
23.4.8 Testovací aplikace.....	541
Spuštění.....	542
23.5 Vícevláknová aplikace se vzdáleným GUI.....	543
23.6 Shrnutí kapitoly a doprovodné soubory.....	545
23.6.1 Doprovodné soubory.....	545
Kapitola 24 Korutiny a modul <code>asyncio</code>	546
24.1 Knihovna/modul <code>asyncio</code>	546
24.2 Koprogramy, korutiny a očekávatelné objekty.....	547
24.3 Definice a použití korutin.....	548
24.3.1 Vytvoření a spuštění korutiny.....	549
24.3.2 Čekání na jinou korutinu.....	549
24.3.3 Využití objektů typu <code>Task</code>	551
24.3.4 Správa skupiny úloh – <code>gather()</code> a <code>TaskGroup</code>	552
24.3.5 Třída <code>TaskGroup</code> a příkaz <code>async with</code>	553
Příkaz <code>async with</code>	553
Třída <code>asyncio.TaskGroup</code>	553
24.3.6 Nejčastější chyba – zapomenutý <code>await</code>	554
Zlaté pravidlo <code>asyncio</code>	555
24.4 Úprava knihovny pro asynchronní provoz.....	555
Podrobněji o <code>asyncio</code> a jeho korutinách.....	556
Role <code>await</code> a smyčky událostí.....	556
Proč to nelze volat synchronně?.....	556
24.4.1 Postup doplnění podpory <code>asyncio</code>	557
24.4.2 Aktivační modul <code>util.karelCz_async</code>	558
Zákaz řetězení asynchronních volání.....	559
Synchronní metody v asynchronním světě.....	560
24.4.3 Prověrka funkce.....	560
24.5 Asynchronní cyklus – příkaz <code>async for</code>	561
24.6 Asynchronní zámky a synchronizace.....	562
24.6.1 Třída <code>asyncio.Lock</code> a příkaz <code>async with</code>	563
24.7 Shrnutí kapitoly a doprovodné soubory.....	565
24.7.1 Doprovodné soubory.....	566
Kapitola 25 Subinterpretů a modul <code>concurrent.interpreters</code>	567
25.1 Předehra.....	567
25.1.1 Dva režimy běhu subinterpretů.....	568
Režim A – lokálně v témže vlákne.....	568
Režim B – v samostatném vlákne.....	568
Analogie.....	568
25.2 Základní operace: vytvoření a ukončení.....	568

25.2.1	Pomocná metoda counter()	569
25.2.2	Demonstrační program	570
25.2.3	Společná konzole (stdin, stdout, stderr)	571
	Výstup	572
	Vstup	572
	Proč jsme to nevyužívali v procesech	572
25.3	Předávání dat – opět Karel	572
25.3.1	Architektura řešení	573
	Místo datovodu použijeme frontu	574
	Komunikace prostřednictvím zástupce	574
	Modul IO	574
25.3.2	Zaváděč knihovny Karelcz7475 pro subinterprety	574
	Hlavní funkce modulu	575
	Zaváděcí funkce	575
	Tvorba náhradního modulu karelcz.IO	577
	Začlenění vytvořeného modulu	577
25.3.3	Lokální zástupce	577
	Funkce v roli zavaděče	579
	Spuštění metody	580
25.3.4	Vzdálený zástupce – dispečer	580
25.3.5	Koordinátor	582
25.3.6	Jednoduchý test	582
	Nastavení podrobnosti ladících informací	582
	Import funkce run_in_si()	585
	Import funkce simple_demo()	585
	Příkaz run_in_si(simple_demo)	585
	Činnost funkce run_callable_in_si()	586
25.4	Subinterpret a vícevláknový režim	587
25.4.1	Třída threading.Event	587
	is_set()	587
	set()	587
	clear()	587
	wait(timeout:float=None)	587
25.4.2	Úprava testovacích funkcí – modul m25f_multithread	588
25.5	Shrnutí kapitoly a doprovodné soubory	590
25.5.1	Doprovodné soubory	590
	Literatura	591
	Rejstřík	593

0 autorovi

Ing. Rudolf Pecinovský, CSc. studoval teoretickou kybernetiku na *Fakultě jaderné a fyzikálně inženýrské ČVUT* a poté technickou kybernetiku na *Fakultě Elektrotechnické ČVUT*, kterou ukončil v roce 1979. Titul CSc. získal v Ústavu teorie informace a automatizace ČSAV v roce 1983. Od počátku 80. let učí a publikuje, přičemž svůj výzkum soustředí především na oblast vstupních kurzů moderního programování a výuku objektově orientovaného paradigmatu. V současné době učí na *Fakultě jaderné a fyzikálně inženýrské ČVUT* a na *Fakultě informatiky a statistiky Vysoké školy ekonomické v Praze*. Vedle toho ještě vede řadu kurzů pro začátečníky i profesionální programátory. Doposud mu vyšlo přes 70 knih, které byly přeloženy do pěti jazyků. Většina jeho knih je zaměřena na výuku moderního programování a návrh objektově orientované architektury.



Poděkování

Chtěl bych především nesmírně poděkovat své ženě Jarušce, která byla po celou dobu mojí největší oporou a jejíž nekonečná trpělivost a vstřícnost mi pomohla dokončit knihu v termínu, který se příliš nelišil od toho, jež jsme původně s nakladatelem dohodli, a ne až někdy za rok po něm. Stále marně přemýšlím, kde má schovanou tu svatozář.

Původně jsem se domníval, že s dalším vydáním nebude moc práce. Šeredně jsem se zmýlil, protože veškeré úpravy, jež jsem do knihy zanesl, jsou vykoupeny hodinami studia a experimentování, které rodina s neuvěřitelnou trpělivostí snášela.

Na vylepšování textu nového vydání se podílela řada dalších lidí. Mezi nimi musím poděkovat především těm, kteří si dali práci s odhalováním případných chyb ve vznikajícím rukopisu. Mezi nimi pak především Luďkovi Šťastnému, který po celou přípravu rukopis pročetl a odhaloval v něm pasáže, jež by si zasloužily vylepšit. Neméně velkou zásluhu na současné podobě má i Jirka Kofránek, který mne průběžně upozorňoval na některé problémy s výukou podle běžně používaných postupů. Řadu podnětných myšlenek přinesl i Michal Palas, jenž pracuje v oblasti analýzy a zpracování dat a přispěl tak řadou poznatků z praxe.

V neposlední řadě patří můj dík redakci, především redaktoru Petru Somogyimu, který trpělivě snášel mé neustálé modifikace již zkorigovaného textu, a Radku Matulíkovi, který mne k napsání jednotlivých knih z posledních let vyhecoval a byl pak ochoten týden či dva počkat, když se mi nepodařilo přesně dodržet původně dohodnutý termín.

Úvod

Python je moderní programovací jazyk, který umožňuje velmi jednoduše navrhovat jednoduché programy, ale na druhou stranu nabízí mocné prostředky k tomu, abyste mohli s přiměřeným úsilím navrhovat i programy poměrně rozsáhlé. Je pro něj vyvinuto obrovské množství knihoven, které uživatelům umožňují soustředit se na řešení úkol a nerozptylovat se vývojem nejrůznějších pomocných podprogramů.

Python je v současné době nejlepším jazykem pro ty, kteří se nechtějí živit jako programátoři, ale jejich profese či zájem je nutí jednou za čas něco naprogramovat. Postupně však sílí i skupina programátorů, kteří v něm vyvíjejí rozsáhlejší programy a ocení jeho obrovské možnosti, jež běžné jazyky neposkytují. Znalost těchto možností ocení i ti, kteří řeší složitější problémy a potřebují proto znát jazyk do větší hloubky.

Pro takový druh potenciálních uživatelů je určena i tato učebnice. Najdou zde výklad řady konstrukcí, na něž v běžných učebnicích nezbylo místo, ale které na druhou stranu dokážou výrazně zefektivnit řešení některých úkolů.

Komu kniha není určena

Dopředu musím upozornit, že tato kniha není koncipována pro začátečníky, kteří se s programováním teprve seznamují. Ti potřebují poněkud jiný postup výkladu a jiný výběr příkladů. Pro ně je určena kniha *Začínáme programovat v jazyku Python* ([\[9\]](#)), která vedle používání základních konstrukcí učí své čtenáře také řadu zásad moderního programování, jejichž zvládnutí je nutnou podmínkou pro všechny, kteří nehodlají zůstat u malých žákovských programů, ale chtějí se naučit efektivně vyvíjet robustní středně rozsáhlé aplikace, jejichž údržba nepovede uživatele k chrlení nepublikovatelných výroků na adresu autora.

Kniha, kterou právě čtete, předpokládá alespoň minimální programátorské znalosti a zkušenosti, což jí umožní soustředit se na výklad konstrukcí a rysů jazyka včetně těch, na které v učebnicích pro začátečníky již nezbyvá místo.

Komu je kniha určena

Je určena pro tři typy uživatelů:

- Autory, jejichž programátorské zkušenosti nejsou příliš hluboké, protože je zatím využívali pro tvorbu jednoduchých programů řešících každodenní problémy a na řešení složitějších problémů se teprve chystají. Pro ty jsem výklad konstrukcí jazyka občas proložil výklady základů programování, které mohou ti pokročilejší přeskočit.

U těchto čtenářů bych byl rád, kdyby jim kniha pomohla v postupném prohlubování jejich znalostí a dovedností. Narazí-li někde na obtížnější pasáž, jejíž znalost právě nepotřebují, mohou ji klidně přeskočit a vrátit se k ní, až se jejich zkušenosti prohloubí a budou potřebovat probíranou vlastnost použít.

- Programátory, kteří doposud vyvíjeli v jiném programovacím jazyce a chtějí rozšířit své dovednosti o programování v jazyce *Python*. Pro ty jsou určena různá upozornění na rysy, jimiž se *Python* může odlišovat od toho, na co jsou ze své dosavadní praxe zvyklí.

Z ohlasů na předchozí verze publikace odhaduji, že těchto čtenářů je 60 až 80 %, takže se jim budu snažit vyjít vstříc a vždy včas upozornit na potřebu odlišného přístupu k probírané konstrukci při návrhu programu.

Předpokládám navíc, že tito čtenáři budou chtít tuto knihu používat spíše jako referenční příručku pro situace, kdy si nebudou zcela jisti, jak se některá konstrukce používá nebo jak přesně funguje.

- Uživatelé, kteří již v *Pythonu* programují, ale potřebují znát jazyk do větší hloubky, než jim poskytly absolvované kurzy a prostudované učebnice, nebo se potřebují seznámit s novými konstrukcemi a rysy, které se mezi tím v *Pythonu* objevily.

K dokonalému využití platformy *Python* je však potřeba i znalost základních knihoven. Přestože je tato dvoudílná příručka poměrně rozsáhlá, mohu vám v ní představit pouze ty nejzákladnější funkce a třídy. Těm, kteří touží po podrobnějším seznámení s knihovnami určenými pro práci s daty, poslouží publikace [\[12\]](#). Ty, kteří se zabývají jinými druhy knihoven, musím odkázat na dokumentaci anebo učebnice jiných autorů.

Struktura příručky

Tato kniha je druhým dílem dvoudílné příručky. První díl nazvaný [Python 3.14 – Algoritmické konstrukce](#) čtenáře seznamuje s vestavěnými jednoduchými datovými typy a základními algoritmickými konstrukcemi jazyka. Poskytne základní informace o práci s objekty a naučí ho pracovat s nejdůležitějšími strukturovanými datovými typy.

Tento díl je pak zaměřen na konstrukce podporující objektově orientované programování a na hlubší pochopení některých pokročilých objektových programových konstrukcí.

Soustřeďuje se na výklad jazyka. Doprovodným knihovnám se až na pár výjimek nevěnuje. Umožňuje to probrat jazyk do větší hloubky, takže se dostane i na oblasti, které běžné učebnice opomíjejí a očekávají, že si některá témata čtenáři osvojí metodou pokus-omyl.

Knihy je rozdělena do čtyř částí doplněných přílohami:

- Část A seznamuje se základními vlastnostmi jazyka podporujícími objektově orientované programování. Nejprve zopakuje jeho principy a pak postupně představí základní konstrukce, jakými jsou třídy a jejich dědění včetně násobného, představí vlastnosti, abstraktní třídy a protokoly. V závěrečné kapitole projde základy několika dalších objektových konstrukcí.
- Část B prohlubuje probraná témata. Začne moduly a balíčky a představí princip virtuálních prostředí. Poté probere volatelné objekty, dekorátory, třídní metody a iterátory a generátory. Po nich se podrobně věnuje výčtovým typům, datovým třídám, anotacím a generickým datovým typům.
- Část C se věnuje pokročilejším objektovým konstrukcím. Postupně probere přetěžování operátorů, speciální způsoby přístupu k atributům, slabé odkazy a slooty. Tato část končí výkladem metatříd a příklady jejich použití.
- Část D se specializuje na souběžné programování. Nejprve probere teoretické základy, aby se pak v následujících kapitolách postupně věnovala použití vláken, podprocesů a asynchronního programování.
- Jak už bylo řečeno, samostatnou část knihy tvoří přílohy. Ty nejsou součástí tištěné verze, ale najdete je na webových stránkách příručky. Elektronické verze jsou distribuovány včetně těchto příloh a včetně seznamů s odkazy na jednotlivé výpisy, obrázky, tabulky a podšeděné bloky.

Koncepce výkladu

Snažil jsem se, aby tato kniha mohla sloužit současně jako učebnice jazyka *Python* i jako referenční příručka. Tyto dva druhy publikací si ve svých požadavcích poněkud odporují.

- Dobrá učebnice vyžaduje, aby se výklad obešel bez dopředných odkazů. Musí proto maximálně omezit (nejlépe zcela zrušit) používání čehokoliv, co ještě nebylo vysvětleno, a to i za cenu toho, že výklad mnoha témat bude třeba rozdělit do několika částí, aby v něm mohly být vždy použity jenom doposud probrané rysy jazyka.

V opačném případě čtenáři vnímají danou konstrukci jako obrázek, který okopírují a používají, aniž by znali význam jednotlivých jeho součástí. (To je ostatně nečinnost řady učebnic začínajících oblíbeným příkladem „Hello World“.) Při takovémto přístupu totiž čtenáři občas nechápou důsledky některých obrátů a akcí a příčiny ohlášených chyb, jejichž odstranění je pak stojí nemalé úsilí.

- Naproti tomu referenční příručky by měly umožňovat co nejsnazší a nejrychlejší dohledání potřebných informací. Mohou být proto koncipovány tak, že každé téma je v nich plně probráno na jednom místě se všemi detaily, a to i za cenu dopředných odkazů, protože při výkladu často potřebují použít konstrukci, která bude vysvětlena až v rámci některého dalšího tématu. Mohou si to dovolit, protože jsou určeny především zkušenějším čtenářům, u nichž se předpokládají základní znalosti a danou referenční příručku používají většinou k tomu, aby si ozřejmili některé detaily.

Jak jsem řekl, chtěl jsem, aby kniha mohla sloužit k oběma účelům, tedy jak jako základní příručka, podle které se dá učit, tak jako referenční příručka, kde lze rychle nalézt pasáž zabývající se diskutovaným tématem. Její struktura je proto navržena tak, aby jednotlivé kapitoly probraly dané téma pokud možno komplexně a bez potřeby se k němu vracet, ale aby se v nich na druhou stranu pokud možno neobjevovaly dopředné odkazy i za cenu toho, že bude třeba výklad některých pasáží rozdělit.

Veškerý výklad je prostoupen hojnými AHA-příklady, tedy příklady, jejichž hlavním účelem není vytvoření nějakého zdánlivě užitečného programu, ale kódu, jehož cílem je co nejjednodušeji a co nejsrozumitelněji demonstrovat probíranou konstrukci, aby čtenář pochopil její princip – aby u něj nastal kýžený AHA-efekt.

Vedle nich se občas objeví i nějaký příklad, který by byl v praxi užitečný. Protože se ale nejedná o učebnici programování, ale především o učebnici jazyka, bude těchto praktických příkladů poskrovnu.

Jazyk identifikátorů

Ve většině programů budou použity anglické identifikátory. V několika jednoduchých AHA-příkladech vysvětlujících základy některých probíraných konstrukcí však budou pro větší názornost použity identifikátory české.

Potřebné vybavení

Pro úspěšné studium této knihy je vhodné mít instalovanou platformu *Python*. Tu lze stáhnout na adrese <https://www.python.org/downloads/>.

Kniha je psána pro verzi 3.14, ale drtivá většina příkladů poběží i na verzích nižších. Protože ale vím, že řada čtenářů pracuje se staršími verzemi (např. proto, že to jejich zaměstnavatel požaduje z důvodu kompatibility), snažím se upozorňovat na všechna vylepšení, která se objevila v novějších verzích počínaje verzí 3.5, abyste byli včas upozorněni na to, co by vám nemuselo na počítači chodit bez vašeho zavinění.

Operační systém

Při práci v *Pythonu* vám může být většinou zcela jedno, nad jakým operačním systémem je instalovaný. V některých výjimečných situacích na tom ale záleží. Protože 80 % mých studentů i účastníků kurzů používá operační systém *Windows*, jsou mé knihy primárně zaměřeny na ně. Pokusím se ale nezapomínat ani na ten zbytek.

Doprovodné programy a jejich organizace

Text knihy je prostoupen řadou doprovodných programů. Budete-li si je chtít spustit a ověřit jejich funkci, potřebujete je nejprve stáhnout. Najdete je na stránce knihy na adrese¹ http://knihy.pecinovsky.cz/75_python314obj.

Soubory s doprovodnými programy jsou uloženy v ZIP-souboru, jehož název začíná `75_Python_314obj_PGM_`. Za tímto začátkem následuje číslo verze a datum vytvoření.

Když tento ZIP-soubor rozbalíte, vytvoří se následující složky:

75_ILB Složka se spustitelným souborem `Karecz7475.pyz` obsahujícím knihovnu, se kterou budeme v některých kapitolách pracovat.

75_ILS Složka se zdrojovými kódy knihovny `Karel7475`

75_INP Složka s doprovodnými programy. K její vnitřní struktuře se za chvíli ještě vrátím.

75_IWD Texty výpisů programů v příručce vysazené spolu s čísly řádků, abyste si mohli daný soubor zobrazit vedle knihy např. v textovém editoru a při pročítání popisu daného programu nemuseli listovat v knize, abyste zjistili, jak vypadá řádek s daným číslem, o němž se v textu hovoří. Soubory odpovídají stejnojmenným souborům ze složky `75_INP/lessons`.

Pojďme se nyní podívat na vnitřní strukturu složky `75_INP`. Složka představuje doprovodné programy kořenový balíček. V něm je klíčový modul `dbg`, který používám při ladění a k názornému zobrazení průběhu některých procesů.

Kromě toho jsou tu tři podsložky představující balíčky. Každý z nich je schránkou na jeden druh souborů:

lessons Balíček s doprovodnými programy obsahujícími příkazy zadávané v průběhu kapitol většinou řádkovému interpretu, občas ale operačnímu systému. Zadáání příkazu i odpověď systému jsou pak zobrazeny ve výpisech v knize. Názvy těchto souborů začínají vždy písmenem `m` následovaným dvojmístným číslem kapitoly, dvěma podtržítky a textem naznačujícím téma kapitoly – např. `m02__Třída_a_jejich_instance.py`.

¹ Číslem 75 na počátku poslední složky se nevzrušujte, to je pouze moje interní označení pořadí vytvářené knihy, protože bych v nich jinak bloudil.

modules Balíček se samostatnými moduly, které se vztahují ke konkrétním kapitolám. Názvy zde umístěných souborů začínají vždy písmenem **m** následovaným dvojmístným číslem kapitoly, písmenem určujícím pořadí vzniku souboru v průběhu kapitoly, podtržítkem a textem naznačujícím obsah souboru.

util Balíček s užitečnými programy, které se občas použijí.

NS Pomocný balíček pro výklad neinicializovaných balíčků.

Stažený ZIP-soubor si rozbalte, kam uznáte za vhodné. Jak napovídají adresy souborů v jejich úvodních komentářích, já mám celý ZIP rozbalený do kořenové složky substitutovaného disku **S:**, takže složka **S:/75_INP** je kořenovou složkou doprovodných programů.

Odkazy do prvního dílu a modul **dbg**

Jak jsem již řekl, tato kniha je určena uživatelům, kteří již mají s programováním jisté zkušenosti a chtějí by se dozvědět podrobnosti o implementaci objektově orientovaného paradigmatu v jazyku *Python*. Ze zkušenosti ale vím, že každý občas zapomene nějaké detaily z dříve probrané látky. Když se mi proto zdá, že by čtenář mohl při čtení narazit na nutnost osvěžení látky, kterou jsme probírali v prvním dílu, tak vždy doplním odkaz na příslušnou pasáž, kde se dané téma probíralo.

Dopředu pak upozorňuji, že v doprovodných příkladech budu často používat modul **dbg**, který je součástí doprovodných programů a jehož funkčnost jsem relativně podrobně postupně probíral v prvním dílu, konkrétně v pasážích:

- [10.2 Modul **dbg** a kontrolní tisky](#),
- [11.1.2 Zobrazování začátku a konce inicializace balíčků a modulů](#),
- [13.9 Pomocné funkce a dekorátory z modulu **dbg**](#) a
- [23.7 Funkce modulu **dbg** pracující s kontejnery](#).

Všechny funkce v modulu mají sice poměrně podrobné dokumentační komentáře, ale chápu, že jejich dohledávání v modulu o přibližně 1000 řádcích je obtížné a obdobně obtížné je i pročítání příslušné nápovědy.

Použité typografické konvence

K tomu, abyste se v textu lépe vyznali (a také abyste si vykládanou látku lépe zapamatovali), používám několik prostředků pro odlišení a zvýraznění textu.

Termíny První výskyt nějakého termínu a další texty, které chci zvýraznit, vysazuji **tučně**.

Název Názvy firem a jejich produktů vysazuji *kurzivou*. Kurzivou vysazuji také názvy kapitol, podkapitol a oddílů, na které v textu odkazuji.

Citace	Texty, které si můžete přečíst na displeji, například názvy polí v dialogových oknech či názvy příkazů v nabídkách, vysazují tučným bezpatkovým písmem .
<u>Odkaz</u>	Celá kniha je prošpikovaná křížovými odkazy na související pasáže. Ne-li odkazovaný objekt (kapitola, obrázek, výpis programu, ...) na stejné stránce nebo na některé ze sousedních stránek, je v tištěné verzi doplněn o číslo stránky, na níž se nachází. Čtenářům elektronické verze stačí, když na něj klepnou, a použitý čtecí program by je měl na odkazovaný objekt ihned přenést.
Adresa	Internetové adresy vysazují obyčejným bezpatkovým písmem.
Program	Identifikátory a další části programů zmíněné v běžném textu vysazují neproporcionálním písmem , které je v elektronických verzích pro zvýraznění tmavě červené.
Zadání	Ve výpisech konverzace s počítačem budou zadání uživatele vysazena tučně (v elektronických verzích pro zvýraznění modře).
Keyword	Klíčová slova jazyka (anglicky keywords) budou vysezena také tučně i v běžných výpisech programu (v elektronických verzích pro zvýraznění tmavě červeně).
Chyba	Obdobně budou zvýrazněna chybová hlášení. Nebudou podbarvena a v elektronických verzích budou vysazena červeně. Na několika místech v knize je ukázka komunikace v příkazovém panelu operačního systému. V těchto výpisech zobrazují pro přehlednost zprávy systému stejně jako chybová hlášení, aby byly jasně odlišeny od následné komunikace s <i>Pythonem</i> .
Komentář	Svůj styl budou mít ve výpisech programů i komentáře, které budou vysazeny kurzivou a v elektronických verzích zeleně podbarveny.
Výzva	Výzvy operačního příkazu k zadání příkazu či jeho části budou podbarvené tak, aby byly snadno odlišitelné od zadávaného kódu.

Kromě výše zmíněných částí textu, které považuji za důležité zvýraznit nebo alespoň odlišit od okolního textu, najdete v knize ještě řadu doplňujících poznámek a vysvětlivek. Všechny budou v jednotném rámečku, jenž bude označen ikonou charakterizující druh informace, kterou vám chce poznámka či vysvětlivka předat.



Symbol jin-jang bude uvozovat poznámky, s nimiž se setkáte na počátku každé kapitoly. Zde vám vždy prozradím, co se v dané kapitole naučíte.



Píšící ruka označuje obyčejnou poznámku, která pouze doplňuje informace z hlavního proudu výkladu o nějakou zajímavost.



Symbol znamení raka označuje poznámky upozorňující na odchylky od *mainstreamových jazyků*, tj. od jazyků *Java*, *C/C++*, *C#*, *Delphi*, *Visual Basic* apod. V této poznámce občas opakuji informace, které jsou v hlavním textu, ale bojím se, že by mohly být přehlédnuty.



Obrázek knihy označuje poznámku týkající se používané terminologie. Tato poznámka většinou upozorňuje na další používané termíny označující stejnou skutečnost nebo na konvence, které se k probírané problematice vztahují.



Ruka s hrozcícím prstem upozorňuje na věci, které byste měli určitě vědět a na něž byste si měli dát pozor, protože jejich zanedbání vás většinou dostane do problémů.



Usměváček vás bude upozorňovat na různé tipy, jimiž můžete vylepšit svůj program nebo zefektivnit svoji práci.



Mračoun vás naopak bude upozorňovat na některé nepříjemnosti a bude vám radit, jak se těmto nástrahám vyhnout (či jak to zařídit, aby vám alespoň pokud možno nevadily).



Brýle označují tzv. „poznámky pro šfouraly“, ve kterých se vás snažím seznámit s některými zajímavými vlastnostmi probírané konstrukce nebo upozorňuji na některé souvislosti, které však nejsou k pochopení látky nezbytné.

Odbočka – podšeděný blok

Občas je potřeba vysvětlit něco, co nezapadá přímo do okolního textu. V takových případech používám podšeděný blok se silnou čarou po straně. Tento podšeděný blok je takovou drobnou odbočkou od výkladu. Nadpis podšeděného bloku pak najdete i v podrobném obsahu mezi nečíslovanými nadpisy.

Při prvním čtení můžete tyto bloky přeskakovat a vracet se k nim až ve chvíli, kdy v textu narazíte na téma probírané v bloku a budete si chtít osvěžit či doplnit svoje znalosti.

Zpětná vazba

Předem přiznávám, že tato kniha je sice mou sedmdesátou první publikovanou učebnicí, ale protože obsahuje řadu novinek, tak přestože knihu četlo několik lektorů, mohou se v ní objevit přehlédnutí, která nemá redaktor šanci zachytit a opravit.

Pokud vám proto bude někde připadat text nepříliš srozumitelný nebo budete mít nějaký dotaz (ať už k vykládané látce či použitému vývojovému prostředí), anebo pokud v knize objevíte nějakou chybu nebo budete mít návrh na nějaké její vylepšení, neostýchejte se poslat svůj dotaz či připomínku na adresu rudolf@pecinovsky.cz jako e-mail s předmětem [75 Python 314obj DOTAZ](#).

Bude-li se dotaz týkat něčeho obecnějšího nebo to bude upozornění na chybu, pokusím se co nejdříve zveřejnit na stránce knihy (tam, odkud jste stahovali doprovodné programy) http://knihy.pecinovsky.cz/75_Python314obj odpověď i pro ostatní čtenáře, kteří by mohli o danou chybu zakopnout, nebo by je mohl obdobný dotaz napadnout za pár dní, anebo jsou natolik ostýchaví, že se netroufnou sami zeptat.

Jenom se musím dopředu omluvit, že na dotazy neodpovídám okamžitě, protože při mém pracovním vytížení zapínám poštu jen jednou za čas.

Část A

Objektově orientované programování

Tato část vás seznámí s vlastnostmi jazyka podporujícími objektově orientované programování. Nejprve zopakuje jeho principy a pak postupně předvede, jak se definují třídy, jak se vytvářejí jejich instance, a probere základní vlastnosti objektů. Vysvětlí základní pravidla dědění včetně násobného a ukáže jejich realizaci v *Pythonu*. Následně představí koncepci vlastností, abstraktních tříd a protokolů a na závěr stručně probere základy série dalších objektových konstrukcí jazyka.

Kapitola 1

Základy OOP



Co se v kapitole naučíte

Kapitola seznamuje se základy objektově orientovaného paradigmatu, které tvoří teoretický základ pro objektově orientované programování, označované většinou zkratkou OOP. Tento popis jsem umístil do samostatné kapitoly, abych s ním nemusel prokládat následný výklad syntaxe a pravidel použití popisovaných konstrukcí.

Tato kapitola by měla být spíše opakovací, protože předpokládám, že čtenáři se již s informacemi o OOP setkali a základní termíny znají. Zařadil jsem do ní ale i výklad základů určený pro ty čtenáře, jejichž zkušenosti s objektově orientovaným programováním jsou minimální nebo dokonce nulové.

Její alespoň zběžné přečtení bych ale doporučoval i těm, kteří se domnívají, že principy OOP znají, protože má sloužit především ke sjednocení terminologie a upozornění na některá specifika implementace OOP v *Pythonu*.

1.1 Předehra

Na počátku úvodu jsem říkal, že: *„Python je moderní programovací jazyk, který umožňuje velmi jednoduše navrhovat jednoduché programy, ale na druhou stranu nabízí mocné prostředky k tomu, abyste mohli s přiměřeným úsilím navrhovat i programy poměrně rozsáhlé.“* Hlavní součástí oněch mocných prostředků je právě podpora objektově orientovaného programování.



V dalším textu budu místo zdlouhavého „objektově orientovaný/álé“ používat zkratku OO a termín *objektově orientované programování* budu nahrazovat zkratkou OOP.

1.1.1 Kdy se OOP začíná vyplácet

Když některým programátorům řeknete, že budou muset ve svých programech pracovat s objekty, tak se hrůzou osypou. Řada programátorů používajících *Python* tvrdí, že OOP ve svých programech nepoužívají, takže jeho znalost nepotřebují. Podobně hovoří i řada vyučujících probírajících *Python* na střední škole.

Příznějme si, že zejména v *Pythonu*, kde je všechno objekt, je nemožné napsat program nepoužívající objekty. Takže to, že se někdo tváří, že objekty nepoužívá, ještě neznamená, že tomu tak doopravdy je.

Naštěstí je poměrně velký rozdíl mezi tím, když používáte hotové objekty, které pro vás připravil někdo jiný, a tím, když takové objekty máte sami navrhovat a vytvářet. U jednoduchých programů si bohatě vystačíme s používáním hotových objektů a návrhem programů v duchu procedurálního paradigmatu.

Proto jsem se také v prvním dílu snažil probrat z objektových vlastností *Pythonu* jen to, co nezbytně potřebujete vědět, i když neprogramujete objektově, ale programujete podle klasického procedurálního paradigmatu a objekty jenom používáte.

Chcete-li si zopakovat, co jsme si v prvním dílu o objektech řekli, tak vám připomenou, že jsem vás se základními informacemi o objektech seznámil v podkapitole [2.1 Objekty a objektové programování](#), a když jsme potřebovali pracovat s objekty intenzivněji, tak jsem na tyto informace navázal v podkapitole [9.2 Další trocha teorie OOP](#).

To, že v *Pythonu* musíte pracovat s objekty, ale ještě neznamená, že musíte používat objektově orientované paradigma, že musíte „přemýšlet objektově“. Dokonce si troufám tvrdit, že kratičké programy v rozsahu do 200 až 500 příkazů dokáže začátečník napsat neobjektově efektivněji.²

Obávám se však, že ti, kterým kratší programy stačí, tuto knihu nepotřebují a spokojí se s prvním dílem doplněným případně o příručku [\[12\]](#). Budu proto předpokládat, že se chystáte vyvíjet programy takové velikosti, při níž se již používání OOP vyplatí. A počítejte s tím, že s růstem vašich programů bude růst i efektivita práce, kterou OOP přináší.

Jakmile váš program bude za něco stát, budete jej co chvíli chtít v nějaké drobnosti vylepšit. Program začne narůstat a neobjektově postavená architektura by vám další rozšiřování komplikovala. Práce z přelomu 80. a 90 let ukazovaly, že od jisté velikosti programu již není v lidských silách vytvořit takový program s konkurenceschopnou trojicí (cena, doba vývoje, spolehlivost) jinak než s využitím OOP.

Pojďme tedy na to. Začneme znovu „od podlahy“.

1.1.2 Objektově orientované paradigma

O objektově orientovaném paradigmatu jsem v prvním dílu hovořil několikrát. Připomenou jen pasáž [2.1.1 Koncepce Pythonu](#), kde jsme si řekli, že termínem *programové*

² Zkušený objektový programátor však s výhodou používá objekty i u relativně krátkých programů.

paradigma označujeme způsob nazírání na problém a jeho následné řešení, způsob uvažování při řešení problémů. Charakterizujeme jím celkový přístup programátora k návrhu programu, jeho programovací styl a s tím spojenou sadu preferovaných programových konstrukcí.

Různé programovací jazyky jsou vyvíjeny tak, aby podporovaly různá paradigma. Používané paradigma však není definováno používaným programovacím jazykem, ale způsobem přemýšlení při návrhu programu. Jazyk může pouze nabídnout nástroje, které používání daného paradigmatu usnadní.

Postup dle procedurálního paradigmatu

Při řešení problému podle klasického procedurálního paradigmatu programátor rozdělí řešený problém na sadu jednodušších podproblémů, přičemž řešení každého z nich definuje jako podprogram – proceduru.

Je-li některý z oněch podproblémů složitější, opět ho rozdělí, a tak rekurzivně postupuje, dokud se nedostane k dostatečně jednoduchým problémům, které není obtížné vyřešit.

Např. při řešení programu pro účtárnu se postupně zpracovává příjem požadavků na fakturaci či naopak zaplacení, jejich zpracování, zanesení do databáze a následná realizace.

Postup dle objektově orientovaného paradigmatu

Při řešení problému podle objektově orientovaného paradigmatu se nejprve definují klíčoví účastníci – objekty. U každého z nich je pak třeba specifikovat jeho vlastnosti a schopnosti.

Ukáže-li se, že některý z objektů je stále velmi komplexní, může se analýza postupně aplikovat na jeho atributy, a to jak datové, tak funkční.

Když bychom měli např. řešit program pro účtárnu, objeví se postupně jako „účastník“ samotná účtárna, faktury, účty, pohledávky, termíny splatnosti, upomínky, dodavatelé a odběratelé a další. Tito účastníci pak spolu nějak komunikují, čímž program simuluje provoz v účtárně.

1.1.3 Výhody postupu podle OOP

Připomeňme si hlavní výhody objektově orientovaného přístupu k řešení složitých problémů. Nechci vás přesvědčovat (to už asi jste, když jste si poříдили tuto knihu), jenom nabízím argumenty pro přesvědčování vašich spolupracovníků a oponentů.

- **Abstraktní modelování**

Objektový přístup pomáhá modelovat složitý svět pomocí jednoduchých abstrakcí. Lze například vytvořit třídu `Auto`, která skrývá složitost motoru a převodovky a nabízí jen metody jako `jet()` nebo `brzdit()`. Usnadňuje to přemýšlení o problému a snižuje jeho komplexitu.

S tím souvisí i další výhoda: zákazník může delší dobu kvalifikovaně sledovat analýzu a vstupovat do ní s připomínkami, že plánované chování některého objektu neodpovídá realitě. To má velký vliv na spolehlivost a bezchybnost vyvíjeného programu.

- **Znovupoužitelnost**

Jednou napsaný kód se při tomto přístupu dá daleko snadněji použít v různých částech programu nebo v jiných programech.

- **Modulárnost**

OOP umožňuje rozdělit velký program na menší, nezávislé třídy s jasně definovaným rozhraním. Díky tomu může na velkém projektu pracovat více programátorů najednou, protože každý z nich může psát svůj modul, aniž by se navzájem rušili.

- **Snadnější správa a udržitelnost**

Při správném návrhu je kód organizovaný tak, že každý objekt se stará sám o sebe a ostatní části programu nemusí vědět, jak přesně uvnitř funguje. Když je třeba něco změnit nebo opravit, stačí se zaměřit jen na jednu třídu, aniž bychom museli přepisovat velkou část programu.

- **Jednodušší vývoj a řešení problémů**

Je-li program správně navržený, dá se většinou rychle zjistit, který objekt je za vzniklý problém zodpovědný. Pak se lze zaměřit jen na daný objekt a opravit chybu bez prohledávání záplavy řádků kódu.

1.1.4 Různé pohledy na OOP

Řada programovacích jazyků je označována jako objektově orientované. Autoři každého z nich ale implementovali svoji představu o tom, co to znamená, že programovací jazyk je objektově orientovaný.

Osobně se domnívám, že z opravdu rozšířených objektově orientovaných jazyků je *Python* klasickému objektově orientovanému paradigmatu zavedenému autory jazyka *Smalltalk* nejbližší. To ale neznamená, že jsou na tom ostatní jazyky hůře. Jejich autoři modifikovali implementované paradigma tak, aby ho podle svého názoru optimalizovali pro třídu typických úloh, pro jejichž řešení byl daný jazyk navržen.

Bude-li vám proto někdo tvrdit, že v OOP je něco jinak, řekněte mu: „*V paradigmatu implementovaném tvým jazykem je to možná trochu jinak. Nicméně v Pythonu je to tak, jak tvrdím já.*“

1.2 Základní princip OOP

OOP vychází z myšlenky, že **všechny programy, které vytváříme, jsou simulací skutečného, nebo nějakého námi vymyšleného virtuálního světa**. Čím bude tato simulace přesnější, tím bude výsledný program lepší a navíc se nám bude i snáze navrhovat.

Všechny tyto simulované světy můžeme chápat jako světy objektů, které mají různé vlastnosti a schopnosti a které spolu nějakým způsobem komunikují (říkáme, že **si navzájem posílají zprávy**). Touto komunikací se přitom nemyslí jen klasická komunikace mezi lidmi či zvířaty, ale i mnohem obecnější vzájemné interakce (např. židle a podlaha spolu komunikují tak, že židle stojící na podlaze posílá podlaze informaci o své váze a naopak podlaha podpírá židli a informuje ji o její svislé pozici – když se podlaha proboří, svislá pozice židle se změní).

Budeme-li chtít, aby naše programy modelovaly tento svět co nejvěrněji, měly by být schopny modelovat obecné objekty spolu s jejich specifickými vlastnostmi a schopnostmi a současně modelovat i jejich vzájemnou komunikaci (tj. vzájemné posílání zpráv). Čím bude tento model věrnější, tím budou naše programy přesnějším modelem skutečnosti a budou i použitelnější, spolehlivější a snáze udržitelné.

Optimální programový model simulovaného světa bude založen na programových objektech, které reprezentují odpovídající objekty onoho simulovaného světa. Pro co nejefektivnější vytváření programů potřebujeme jazyk umožňující snadno vytvářet programové objekty, které budou co nejlépe reprezentovat odpovídající objekty simulovaného světa.

Máme-li být schopni rychle vytvářet kvalitní programy, měli bychom mít k dispozici jazyk, jehož jazykové konstrukce nám umožní se vyjadřovat co nejpřirozeněji a co nejméně se nechat znásilňovat omezeními počítače, na kterém má program běžet. Musí umožnit co nejvyšší míru abstrakce, při níž se můžeme vyjadřovat tak, abychom mohli přirozeně popsat modelovanou skutečnost.

1.3 Objekty a jejich atributy

Před chvílí jsem říkal, že svět simulovaný naším programem je světem objektů. V běžném životě jsme za objekty ochotni považovat osoby, zvířata a věci. Při vývoji programů však programátoři záhy zjistili, že musí takovéto chápání objektů zobecnit. Za objekty jsou proto považovány i vlastnosti (barva, chuť, vůně, ...), události (připomení, přerušení, ...), stavy (klid, pohyb, vztek, ...) a další zdánlivě abstraktní vlastnosti (krása, pohoda, ...). Obecně bychom tedy mohli prohlásit, že **v OOP je za objekt považováno vše, co můžeme nazvat podstatným jménem**.

Možná vám takovéto zobecnění bude zpočátku připadat poněkud podivné. Rychle si však na ně zvyknete, protože s takovýmito objekty budeme často pracovat. Stačí si jen uvědomit, že mám-li s něčím v programu pracovat, musím si to nějak charakterizovat. **Z hlediska programu je objektem souhrn dat charakterizujících ono „něco“, co je pak daným objektem v programu reprezentováno.**

Informace charakterizující daný objekt, jeho vlastnosti, schopnosti a aktuální stav ukládáme do proměnných, které jsou ve vlastnictví tohoto objektu a které označujeme jako **atributy daného objektu**.



V některých situacích se pro zjednodušení nepoužívá termín *atribut* pro označení příslušných proměnných, ale označují se jím přímo objekty odkazované těmito proměnnými. Pokud si budu myslet, že nemůže dojít k nedorozumění, budu tuto „zkratku“ používat i já.

Atributy jsou tedy v *Pythonu* proměnné odkazující na objekty, z nichž se daný objekt skládá, nebo které jinak ovlivňují možnosti jeho použití. Objekt může mít obecně tři druhy atributů:

- **datové atributy** definují vlastnosti daného objektu (např. odkazují na objekty, z nichž se skládá, ale mohou udržovat informace o velikosti, barvě apod.),
- **funkční atributy** označované většinou jako **metody** definují jeho schopnosti (odkazují na kód definující reakci objektu na různé podněty),
- **typové atributy** definují pomocné datové typy (v *Pythonu* třídy), které se při práci s daným objektem mohou hodit. Ty bývají občas označovány jako *vnorené třídy* (anglicky *embedded classes*) nebo *vnitřní třídy* (anglicky *inner classes*).



Python má oproti mainstreamovým jazykům jednu zvláštnost: v *Pythonu* můžete atributy objektů přidávat a odebírat za chodu programu. Na rozdíl od mainstreamových jazyků se v *Pythonu* nedefinují atributy objektu předem, ale vytvářejí se až v průběhu konstrukce objektu a následné práce s ním.

Když se nad tím zamyslíte, rychle si uvědomíte, že každý objekt je svým způsobem kontejner, přičemž uložená data jsou hodnoty jeho atributů. K těmto datům přistupujeme tak, že oslovíme objekt, jehož atribut nás zajímá, napíšeme tečku a za ní název požadovaného atributu.

Než se s atributy rozloučíme, tak bych vás jen upozornil, že vedle atributů mívají objekty i takzvané *vlastnosti*. Výkladu vztahu atributů a vlastností se budu věnovat v kapitole [5 Vlastnosti, abstraktní třídy a protokoly](#) na straně .

1.4 Třídy a jejich instance

Ve větších programech se vyskytují tisíce a desettisíce objektů. Abychom s nimi mohli rozumně pracovat, musíme je nějak roztřídit. Budete-li mít na stole všechny papíry na jedné hromadě, asi se vám s nimi také nebude dobře pracovat a nejprve si je nějak uspořádáte do tematických skupin.

Terminologická vsuvka

Jedním z nepříjemných rysů (nejen) současného programování je, že mnozí z autorů programovacích jazyků zavádějí vlastní terminologii, která je v rozporu se stávající zaužívanou terminologií. Připomenu zde proto terminologii označování různých druhů atributů v nejrozšířenějších objektově orientovaných jazycích a její odchylky od terminologie jazyka *Python*.

Relativní jednotnost existuje v označování funkcí – ty jsou téměř všude označovány jako *metody* (anglicky *methods*), i když v C++ se občas setkáte také s termínem *členská funkce* (anglicky *member function*).

V jazyku C++ se datové atributy většinou označují jako *členské proměnné* (anglicky *member variables*), anebo jako *členská data* (anglicky *member data*).

V jazyku *Java* se v originální dokumentaci označují datové atributy jako *pole* (anglicky *field*). V mých českých učebnicích zůstávám u termínu *atribut* (termín *pole* používám pro objekty označované anglicky termínem *array*), setkal jsem se ale s učebnicemi, které je po vzoru C++ označovaly jako *členské proměnné*.

Jazyk C# převzal terminologii od jazyka *Java* a v originále označuje datové atributy jako *pole* (anglicky *field*). Obdobně tuto terminologii převzaly i české texty.

Jazyk *JavaScript* je označuje jako *vlastnosti* (anglicky *property*), aby nekolidoval s termínem *atribut* používaným v jazyce HTML. Řada ostatních jazyků však používá termín *vlastnost* pro speciální podmnožinu atributů. Podrobněji si o tom povíme v podkapitole [5.3 Zadání vlastnosti pomocí dekorátoru](#) na straně .

Autoři jazyka *Python* nám v tomto směru život také neulehčují, protože zavádějí terminologii vlastní. V prvním přiblížení je sice vše atribut, ale pro funkční atributy používají střídavě termíny *metoda* a *funkce*. Oficiální dokumentace deklaruje metody jako funkce definované v těle třídy. Avšak vzhledem k možnosti přidávat a odebírat atributy za chodu, a to včetně těch funkčních, je tato charakteristika zavádějící.

V této učebnici používám označení *funkce* pro ty objekty vytvářené pomocí klíčového slova **def**, které nemusíte kvalifikovat. Platí to pro vestavěné funkce a pro funkce, jež jsou atributy modulu. Ty, které kvalifikovat musíme, tj. atributy jiných objektů než moduly, pak označuji jako metody. Tohoto pravidla se snažím držet, ale předem se omlouvám, pokud mi to někde ujede.

Podobně je tomu i s objekty. I ty můžeme většinou rozdělit do skupin s velmi podobnými vlastnostmi. Tyto skupiny označujeme jako **třídy**. Objekty patřící do dané třídy pak označujeme jako **instance** této třídy – např. židle, na které sedíte, je instancí třídy židlí. Mohli bychom tedy říci, že instance (objekt) je nějakou konkrétní realizací obecného vzoru definovaného v příslušné třídě (židle, na které sedím, je konkrétní realizací obecné židle).

Třída (přesněji objekt reprezentující danou třídu) je vytvořena tak, aby uměla vytvořit své instance. Tím současně definuje jejich společné vlastnosti. Přirovnáme-li

programování k hraní si na pískovišti, pak je třída něco jako formička a její instance jsou bábovičky, které za pomoci této formičky vytváříme.

Třídy jsou jediné objekty, které umějí vytvářet jiné objekty. Kdykoliv nějaký objekt tvrdí, že umí vytvořit jiný objekt, tak je to buď třída, anebo nějakou třídu k vytvoření tohoto objektu využívá. Mohli bychom tedy prohlásit, že třídy jsou vlastně továrny na objekty – jejich instance.

1.4.1 Třída versus datový typ

V mainstreamových jazycích je termín *třída* (po rozšíření o konstrukce označované jako interfejs [Java, C#], protokol [Objective C, Swift] apod.) pokládán za ekvivalent termínu *datový typ*. V *Pythonu* je datový typ něco obecnějšího než třída. K danému tématu se ještě vrátím v kapitole [14. Anotace a koncept typového systému](#) na straně [308](#).

Python nezavádí speciální syntaktické konstrukce reprezentující další druhy datových typů,³ protože si pro konstrukci takovýchto typů vystačí se stávajícími prostředky jazyka – vzpomeňte si např. na pojmenované *n-tice*. V dalším textu se proto soustředíme převážně na třídy.

Jedinou výjimkou bude situace, kdy budu hovořit o typových attributech (tj. o třídách definovaných jako atributy objektu), protože termín *třídní atribut* je již vyhrazen pro atributy patřící třídě, na rozdíl od *instančních atributů*, které patří instancím (všechny zmíněné termíny za chvíli probereme).

1.4.2 Instance

Každý objekt je instancí nějaké třídy. Třída, která danou instanci vytvořila, je **mateřská třída** dané instance a instance vytvořená danou třídou je **vlastní instance** dané třídy.

Třída může mít obecně libovolný počet instancí. Existují však i třídy, které dovoli vytvoření pouze omezeného počtu instancí, někdy dokonce povolí jen jedinou instanci – jedináčka, a někdy dokonce vůbec žádnou. Takové třídy se sice nevyskytují příliš často, ale na druhou stranou nejsou žádnou exotickou výjimkou. S některými z nich se v dalším textu seznámíme.



V mainstreamových jazycích definuje třída atributy svých instancí a všechny její vlastní instance mají stejné atributy. Různé instance mohou mít různé hodnoty těchto atributů, ale sada atributů je pro všechny společná. Pokud se sada atributů v průběhu programu mění, mění se všem instancím dané třídy společně.

V *Pythonu* se mohou sady atributů jednotlivých instancí dané třídy lišit. Instance může za chodu programu své atributy přidávat a odebírat.

³ Např. **interface**, **enum** a **record** v jazyce *Java*, **protocol** v jazyce *Swift*, **trait** v jazyce *Scala* apod.

1.5 Objekt třídy versus instance třídy

Některé programovací jazyky nezavádí třídu jako běžný objekt, ale buď s ní pracují jako s čistou abstrakcí (např. C++), nebo je v nich definována jako nestandardní objekt, který není instancí žádné třídy, a není jej proto možné uložit do proměnné (*Java*, *C#*).

Programovací jazyk *Python* třídy takto neomezuje. Všechny třídy jsou v něm definovány jako standardní objekty uložitelné do proměnné. Každá třída je v *Pythonu* instancí třídy `type` a třída `type` je sama svojí instancí.

V prvním dílu jsme si ukázali, jak se používá funkce `type()`. Teď vám prozradím, že tato funkce vrací třídu, jejíž instancí je zadaný argument. Proto podpisy vrácených objektů začínají slovem `class` – třída.

Třídy mají mezi objekty zvláštní postavení – jak už jsem řekl, jsou to jediné objekty, které umějí vytvořit nové objekty – své instance. Bude-li vám libovolný jiný objekt tvrdit, že pro vás vytvoří instanci, zjistíte nakonec, že pouze zprostředkoval získání instance vyrobené nějakou třídou.

Aby vám bylo v dalším textu vždy jasné, o čem hovořím, dohodněme se na následující terminologii:

- Termín **instance třídy** budu používat tehdy, budu-li jej chtít doplnit o informaci, čím je daný objekt instancí.
- Termín **objekt třídy** použiju tehdy, budu-li hovořit o objektu reprezentujícím danou třídu.
- Samotný termín **objekt** budu používat tehdy, budu-li hovořit o obecných objektech bez nutnosti zdůraznit, čím jsou instancí.

1.6 Atributy třídy versus atributy instancí

V mnoha učebnicích objektově orientovaných jazyků se často setkáte s tím, že text hovoří o metodách a atributech třídy, ale autoři mají přitom na mysli metody a atributy jejích instancí. K tomuto nedorozumění dochází většinou kvůli syntaktickým pravidlům, která vyžadují definovat metody instancí uvnitř definice třídy.

Python se v tomto od mainstreamových jazyků liší. V *Pythonu* se totiž ve třídě definují pouze atributy dané třídy. Budu-li proto v dalším textu hovořit o atributu či metodě třídy anebo o tom, že „třída definuje XYZ“, budu tím mít opravdu na mysli, že se jedná o něco, co patří objektu třídy. Bude-li se jednat o definice čehokoliv, co je „v majetku instance“, vždy to explicitně uvedu.

1.7 Zprávy

Řekli jsme si, že OO programy musí umět simulovat i vzájemnou komunikaci objektů. Tu v programech řešíme zasíláním zpráv. Objekty si navzájem posílají zprávy, ve kterých se žádají o různé služby, přičemž onou službou může často být pouhá informace. Můžeme se např. zeptat židle, je-li čalouněná. V praxi bychom to realizovali nejspíše tak, že bychom k ní poslali upřený pohled, v programu k ní vyšleme zprávu.

V mainstreamových jazycích je běžné, že všechny instance dané třídy rozumějí stejné sadě zpráv. Tuto sadu definuje jejich třída, která vybavuje své instance schopností reagovat na tyto zprávy, když je (instance) vypouští „do života“. Jak ale později uvidíte, v *Pythonu* může většina objektů během svého života množinu zpráv, na něž jsou schopny reagovat, rozšiřovat nebo zužovat.

Ve většině nejpoužívanějších programovacích jazyků se zaslání zprávy zapisuje tak, že se za název osloveného objektu (přesněji za název proměnné, v níž je uložen objekt či odkaz na objekt) napíše tečka a za ni zpráva, kterou objektu posíláme. Tak je tomu i v *Pythonu*, přičemž onou zprávou je buď název atributu, jehož hodnota nás zajímá, anebo volání funkčního atributu – metody.



Už jsem vám prozradil, že v *Pythonu* může objekt za chodu programu přidávat a odebírat své atributy. Tím pádem může měnit i množinu zpráv, na které je schopen reagovat.

1.8 Metody

Metoda je část programu, která definuje, jak bude objekt reagovat na obdrženou zprávu. Každá zpráva, na kterou umí objekt zareagovat, má v programu přiřazenu svoji vlastní metodu (případně několik metod). Metody jsou tak definovány jako funkce, na něž odkazují funkční atributy daného objektu.

Zde se objevuje drobný terminologický guláš. Analytici totiž dávají přednost termínu „poslat zprávu“, programátoři hovoří spíše o volání metod. Praktikum je jedno, jestli řeknete, že objektu pošlete zprávu nebo že zavoláte jeho metodu. Teoreticky v tom ale rozdíl je, protože to, jakou zprávu posíláte, víte, ale to, jaká metoda se v reakci na tuto zprávu zavolá (tj. jaký konkrétní kód se spustí), není vždy zcela jasné (viz pasáž [1.9.5 Polymorfismus](#) na straně).

Python tento terminologický guláš ještě trochu prohlubuje, protože rozlišuje metody a funkce. Jako metody označuje funkce, které jsou definovány jako atributy třídy. Jsou to sice stále obyčejné funkce, ale protože jsou atributy třídy, zachází s nimi interpret maličko jinak.

Jako funkce jsou tak označovány pouze funkční atributy modulů. Funkční atributy tříd definovaných v rámci těchto modulů jsou označovány jako metody. Vzhledem k tomu, že se přístup *Pythonu* k atributům a metodám opravdu liší, budu se snažit tuto terminologii dodržovat.

1.8.1 Metody versus funkce

Metody objektů se definují jako funkce. V prvním dílu jsme pracovali převážně s funkcemi. Při práci s kontejnery je však často potřeba volat jejich metody, a tak jsem vám je stručně představil a prozradil, že funkční atributy modulů označujeme jako funkce, kdežto funkční atributy tříd označujeme jako metody. Metody jsou sice definovány stejně jako funkce, ale protože jsou atributy třídy, zachází s nimi interpret malíčko jinak. Rozdíl si vysvětlíme za chvíli v pasáži [2.3.10 Převod metod na funkce a funkcí na metody](#) na straně .

1.9 Dědění

Jedním z důležitých rysů OO jazyků je podpora dědění (anglicky inheritance) tříd a z něj vyplývajících funkcionalit. Jeho výklad by si zasloužil samostatnou kapitolu, ale nebudu teoretický výklad dále dělit. Rozdělil jsem ho pouze do dvou částí:

- Tato podkapitola je trochu konkrétnější a soustředí se na problematiku blíže související s konkrétními konstrukcemi jazyka.
- Následující podkapitola je poněkud abstraktnější a probírá problematiku, jejíž důležitost roste s velikostí vytvářených aplikací.

1.9.1 Terminologie

Mezi instancemi nějaké třídy můžeme velmi často najít skupinu instancí, které mají nějaké společné speciální vlastnosti. Tato skupina bývá často natolik významná, že se vyplatí pro ni definovat vlastní **podtřídu** (subclass), která tuto specializovanou skupinu objektů charakterizuje.

Tato podtřída bývá označována jako **potomek** (child) původní třídy. Z toho logicky vyplývá, že původní třída bývá pro změnu označována jako **rodičovská třída** (parent class), případně **nadtřída** (superclass) dané podtřídy. A když už jsem u té terminologie, tak někdy bývá rodič označován jako **základní třída** (base class) a potomek jako **odvozená třída** (derived class). A aby vyučující studenti zmátli, tak hovoří-li o třídě, která je potomkem, tak neříkají, že je to „potomčí“ třída, ale **dceřiná třída**.

Rodičovská třída může být sama potomkem své vlastní rodičovské třídy. Třída, která je potomkem nějaké třídy, je současně potomkem všech jejích předků. Abychom odlišili jednotlivé úrovně rodičovství, tak třídy, k nimž se naše třída přihlásí ve své hlavičce jako ke svým rodičům, označíme jako **bezprostřední rodiče** (direct parents), a naopak třídy, které se k dané třídě hlásí ve svých hlavičkách, jako **bezprostřední potomky** (direct children/subclasses).

Základní vlastností dědění je, že dceřiná třída zdědí všechny atributy své rodičovské třídy. Protože však o dceřiné třídě víme něco navíc (známe její specializaci), můžeme její instance vybavit i dalšími vlastnostmi a schopnostmi, které z této specializace vyplývají – přidat další atributy a/nebo modifikovat její metody.



Neplete si právě vysvětlený termín *rodičovská třída* s termínem *mateřská třída*, který jsme před chvílí probrali v pasáži [1.4.2 Instance](#) na straně .

Dědění versus dědičnost

Většina českých učebnic překládá anglický termín *inheritance* (dědění, dědictví, dědičnost, ...) termínem **dědičnost**, což je podle [\[22\]](#) *schopnost přebírat vlastnosti předků a předávat je potomkům*. Protože při výkladu nevysvětlují žádnou schopnost, ale proces tohoto přebírání a předávání, dávám přednost termínu **dědění**, který tento proces označuje výstižněji.

1.9.2 LSP – substituční princip Liskové

V květnu 1988 měla Barbara Liskov na konferenci SIGPLAN úvodní přednášku [\[5\]](#), v níž vysvětlila povinnost dceřiných tříd vytvářet instance, které budou současně plnohodnotnými instancemi předků těchto tříd. Tato zásada označovaná zkratkou LSP (*Liskov substitution principle* – substituční princip Liskové) je jednou z nejdůležitějších zásad OOP.

Toto pravidlo je logickým důsledkem skutečnosti, že dceřiná třída specifikuje dodatečné charakteristiky, jež platí pro její instance, které tak tvoří podmnožinu instancí jejich rodičovské třídy. Jako prvky podmnožiny však nepřestávají být prvky původní množiny a jsou tak zákonitě i instancemi rodičovské třídy.

Rada programátorů (a dokonce i lektorů OO jazyků) tuto povinnost ignoruje. Při návrhu programu se soustředí především na to, jak maximálně zjednodušit kódování, a definují zaváděnou třídu jako dceřinou třídu jiné třídy jenom proto, aby mohli od tohoto rodiče zdědit nějaký kód. Tím ale vnesou do programu nekonzistence, které se jim později mohou vymstít, a takto navržený program se od jisté chvíle začne „hroutit vlastní vahou“.

V praxi např. může být třída **Pes** podtřídou třídy **Savec**, protože psi vykazují všechny rysy savců. Nemůžeme ale definovat kruhovou výseč jako potomka kruhu, u něž ke zděděným atributům se souřadnicemi středu a poloměrem přidáme úhel výseče. Kruhová výseč není speciálním případem kruhu, a proto třída kruhových výsečí nemůže být definována jako potomek třídy kruhů.

1.9.3 Virtuální metody a jejich přebíjení

Když rodičovská třída definuje metodu, která se nám v dceřiné třídě nehodí, můžeme v dceřiné třídě definovat metodu se stejnou signaturou, jež bude vyhovovat požadavkům dceřiné třídy. (Nechceme-li porušovat LSP, nesmí současně narušovat požadavky rodičovské třídy.) Příslušný atribut pak bude pro instance dceřiné třídy odkazovat na nově definovanou metodu.

O této nové definici říkáme, že *přebije* (anglicky *overrides*) zděděnou metodu. Je to stejné jako u karet. Původní (tj. přebitá) metoda přebitím nezmizí, pouze se v daném kontextu už neuplatní.

Není tedy pravda, že se metoda přepíše (anglicky *overwrite*) nebo předefinuje (anglicky *redefine*), jak se někde dočtete – to jsou operace provádějící něco jiného. Při přebití zůstává původní metoda netknutá (nepřepsaná, nepředefinovaná). Vlastní instance třídy, v níž byla definovaná, a instance všech potomků této třídy, které metodu nepřebily, ji stále používají v její původní podobě.

V některých jazycích, např. v *Javě*, již není možné zavolat přebitou metodu pro instance třídy, která ji přebila. V *Pythonu* to není problém – stačí metodu zavolat prostřednictvím její definiční třídy a danou instanci předat jako argument (viz výpis [3.4](#) na straně).

V některých jazycích se musí předem uvést, které metody se smějí (C++, C#), resp. nesmějí (*Java*) přebít. Přebitelné metody označujeme jako *virtuální*. V *Pythonu* jsou primárně virtuální všechny metody. Metody, které nemají být přebity, však můžete označit dekorátorem `final` z modulu `typing`, a informovat program použitý na kontrolu dodržování anotací, a prostřednictvím něj požádal statický analyzátor, aby ověřil, že se nikdo nesnaží danou metodu přebít. Překladač vám sice jejich přebití nezakáže, ale statický analyzátor vás na takovýto pokus upozorní.⁴

1.9.4 Zakrývání versus přebíjení

Některé staticky typované programovací jazyky rozlišují zakrytí (*hiding*) a přebití (*overriding*) metody, které se liší v tom, zda to, která metoda je spuštěna, závisí nebo nezávisí na předpokládaném typu osloveného objektu v době překladu, kdy překladač vybírá, kterou metodu zavolá. U *Pythonu* se v době překladu typ objektu nezjišťuje, takže toto dělení postrádá smysl.

V souladu s učebnicemi a výukovými texty, které mi prošly rukama, proto budu v dalším textu hovořit o přebíjejících (*overriding*) a přebitých (*overridden*) metodách.

Termín *zakrytí metody* nebudu v dalším textu používat v souvislosti s děděním, ale pouze v souvislosti s oblastí platnosti.

1.9.5 Polymorfismus

Termín *polymorfismus* označuje skutečnost, že se oslovený objekt rozhodne, jak zareaguje na zaslou zprávu. Jednou z cest, jak zadat různé reakce různých objektů na stejnou zprávu, je právě definice virtuálních metod a jejich přebíjení. To, jaký kód se v reakci na zaslou zprávu spustí, tj. jaká konkrétní metoda se zavolá, záleží na osloveném objektu a na jeho mateřské třídě.

⁴ Statické analyzátor jsou samostatné programy, o nichž si budeme podrobněji povídat v pasáži [14.2.1 Nejpoužívanější statické analyzátoru a testery](#) na straně [311](#).

Jak už jsme řekli, novou verzi metody definuje primárně odvozená třída prostřednictvím přebití zděděné metody. Nicméně, jak si ukážeme v pasáži [2.3.12 Zavádění nových instančních funkčních atributů](#) na straně , v *Pythonu* můžeme s jistými omezeními definovat novou verzi metody i pro jedinou instanci.

1.9.6 Rodičovský podobjekt

Aby dceřiné třídy (a především pak jejich instance) zdědily opravdu všechny schopnosti rodičovské třídy a jejích instancí, jsou instance dceřiných tříd konstruovány tak, že jejich integrální součástí jsou instance příslušných rodičovských tříd. Říkáme, že každá instance dceřiné třídy obsahuje svůj **rodičovský podobjekt**, který s sebou přinese veškerou funkcionalitu instancí rodičovské třídy. V *Pythonu* je přitom asi nejdůležitější to, že se sloučí jmenné prostory dané instance a oněch rodičovských podobjektů.

1.9.7 Násobné dědění a diamantový problém

V jazycích *Simula 67* a *Smalltalk*, které stály u zrodu OOP, měla třída povoleného pouze jediného bezprostředního předka. Takovýto způsob dědění bývá označován jako *jednoduché*.

Problém jednoduchého dědění spočívá v tom, že daný objekt může být speciálním případem několika obecnějších množin objektů:

- Speciálními podmnožinami dopravních prostředků jsou kola, auta, vlaky, lodě a další. Paralelně je ale můžeme rozdělovat na prostředky poháněné silou člověka, zvířete či nějakého motoru, který může být elektrický, benzinový, dieselový apod. Když se pak bavíme o nějaké konkrétní třídě vozidel, tak např. řekneme, že se jedná o osobní auto (jeden předek) s elektrickým motorem (druhý předek).

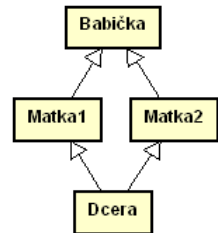
Tady bych jen připomněl oblíbenou studentskou chybu, že definují jako předka motor. Vozidlo ale není speciálním případem motoru, takže podle LSP nemůže být jeho potomkem. V našem případě by druhým předkem našeho vozidla mohl být „dopravní prostředek poháněný elektrickým motorem“.

- Živočichy dělí biologie do různých řádů, druhů, čeledí atd., ale paralelně je můžeme dělit na živočichy žijící na suchu, ve vodě či v obojím prostředí současně. Dále je můžeme dělit na býložravé, masožravé, hmyzožravé atd. Delfín pak bude patřit do třídy vodních (jeden předek) masožravých (druhý předek) savců (třetí předek).

Obdobných příkladů najdeme v praxi celou řadu, a proto autoři programovacích jazyků začali vymýšlet mechanismus, jak tuto vlastnost modelovaného světa podchytit v programu.

V polovině osmdesátých let přišel jazyk C++ s možností definovat více bezprostředních předků jedné třídy – tato vlastnost bývá označována jako *násobné* (případně *vícenásobné* nebo *mnohonásobné*) *dědění* (*multiple inheritance*).

Zkušenosti s používáním více bezprostředních rodičů v jazyce C++ ukázaly řadu možných problémů, které byly hromadně označeny jako *diamantový problém* (*diamond problem*) podle charakteristického tvaru zobrazení architektury,⁵ na níž se problém demonstruje (viz obrázek 1.1). Při použití této architektury se objeví několik úloh, které je třeba řešit:



Obrázek 1.1:
*Diamantový
problém*

- Chce-li potomek použít zděděný atribut (většinou metodu), avšak několik předků definuje atribut se stejným názvem, jak určit, který z nich bude použit?

V literatuře je často jako příklad uváděna situace, kdy některý z předků přebíje virtuální metodu prapředka a jiný ne. Dejme tomu, že by v našem příkladu **Babička** definovala metodu `virtual()`, kterou by **Matka1** přebila, ale **Matka2** ne. Instance třídy **Dcera** by se pak musela rozhodnout, zda bude používat přebíjející metodu zděděnou od třídy **Matka1**, anebo nepřebitou verzi zděděnou od třídy **Matka2**, a tím vlastně od třídy **Babička**.

- U datových atributů zděděných od prapředka je třeba se rozhodnout, zda bude mít každý z předků vlastní rodičovský podobjekt prapředka a tím i vlastní kopii všech jeho datových atributů, anebo zda budou předci, kteří jsou všichni potomky společného prapředka, rodičovský podobjekt tohoto prapředka sdílet.

Jazyk C++ přišel se svým řešením, které se však autorům některých jiných jazyků nelíbilo a rozhodli se, že ve svém jazyku povolí třídám pouze jediného třídního předka (*Java*, *C#*, *Delphi*, ...). K tomu, aby instance dané třídy mohly v různých situacích vystupovat v různých rolích, použijí alternativní řešení – nový druh datového typu označovaný jako *interface* nebo *protocol*.



Strom × Svaz

Při jednoduchém dědění, při němž má každá třída jediného předka, tvoří třídy v dědické hierarchii datovou strukturu označovanou *strom* (anglicky *tree*). Při násobném dědění se struktura v uzlech nejen košatí, ale i slučuje. Dědická hierarchie pak tvoří strukturu označovanou jako *svaz* (anglicky *lattice*).

1.9.8 Zobecňování

Řekli jsme si, že definicí potomka zavádíme specializaci, protože definujeme třídu, jejíž instance jsou speciální podmnožinou instancí předka. Při návrhu programu ale občas postupujeme i obráceně: když vidíme, že instance skupiny tříd mají společné vlastnosti

⁵ Při pohledu na architekturu je zřejmé, že by se termín *diamond* měl do češtiny překládat spíše jako *kosočtvercový*, ale mezi programátory se vžil spíše onen nepřesný překlad *diamantový* (ono to lépe zní).

a schopnosti, můžeme definovat zobecňující třídu, do níž vytkneme to, co mají všechny třídy společné, a tuto třídu pak prohlásíme za jejich společného předka.

Zavedení mechanismu dědění umožňuje i obrácený postup. Místo abychom hledali specializované podmnožiny instancí dané třídy a definovali podtřídy, které je charakterizují a jejichž instancemi pak budou, tak můžeme také naopak hledat, co mají instance různých tříd společného, a definovat jejich společného předka. Do něj pak můžeme vytknout společné atributy a definovat je tak na jednom místě.

Tím vyjdeme vstříc důležité programátorské zásadě označované zkratkou DRY z anglického *Don't Repeat Yourself*, česky *Neopakuj se*. Ta nabádá, abychom na více místech neopakovali stejný nebo podobný kód, ale umístili jej na jediné místo a z ostatních jej už jenom volali.

1.10 Obecnější pohled na dědění

Jak jsem již zmínil na počátku předchozí podkapitoly, tak kapitole je poněkud abstraktnější a probírá problematiku, jejíž důležitost roste s velikostí vytvářených aplikací. Opět začneme představením důležitých termínů.

1.10.1 Rozhraní versus implementace

Na počátku části o OOP jsme si říkali, že „*všechny programy, které vytváříme, jsou simulací skutečného, nebo nějakého námi vymyšleného virtuálního světa.*“ Ke správnému naprogramování výše zmíněné simulace potřebujeme vědět, že každá entita programu má dvě složky – rozhraní a implementaci:

- **Rozhraní** (anglicky interface) dané entity specifikuje, co daná entita umí a jak s ní můžeme komunikovat. V případě objektu pak říká, jaké zprávy je mu možné posílat (jakým zprávám objekt rozumí) a jak na ně reaguje. Mohli bychom říci, že **rozhraní shrnuje, co by měl okolní program o dané entitě vědět.**

Rozhraní objektu představuje to, co o sobě objekt zveřejňuje a co se např. můžete dozvědět z jeho nápovědy, tj. zavoláte-li funkci `help()`, jíž předáte daný objekt jako argument.

Zapamatujte si, že rozhraní objektu není nic reálného, jsou to jenom sliby, že objekt bude umět to či ono, požádáte-li o to zadaným způsobem, přičemž i tento způsob je součástí rozhraní (např. chci-li po funkci `maximum` ze sady hodnot, musím jí v argumentech tyto hodnoty předat).

- **Implementace** (anglicky implementation) pak představuje to, jak je zařízeno, že objekt umí to, co slibuje, že umí (pamatuje si údaje, provádí akce pod.). Jinými slovy: implementace má na starosti, aby daná entita dělala přesně to, co její rozhraní slibuje.

U mainstreamových jazyků (a obecně u jazyků se statickým typováním) je rozhraní používaných objektů dáno jejich typem. Tento typ jasně předepisuje, co budou objekty daného typu umět a jak s nimi bude možné komunikovat. Překladač to pak vše může zkontrolovat.

V *Pythonu* (a obecněji u dynamicky typovaných jazyků) je to maličko složitější, protože se to, zda bude oslovený objekt umět zareagovat na zaslanou zprávu (zavolanou metodu), se zjišťuje až za chodu programu. Výhodou tohoto přístupu je, že k tomu, abych mohl danému objektu bezpečně poslat zprávu, stačí, abych věděl, že daný objekt má definovanou potřebnou metodu.

Jinými slovy: znám-li rozhraní objektu (tj. vím-li, jaké služby nabízí a jak o ně požádat), nemusím se starat o jeho typ.

1.10.2 Signatura versus kontrakt

Samotné rozhraní objektu má také dvě tváře: signaturu a kontrakt.

- Termínem *signatura* označujeme souhrn vlastností, které může zkontrolovat překladač, případně nějaký pomocný program. U funkcí a u tříd bychom sem zařadili informace z hlavičky, u všech objektů (včetně tříd a funkcí) pak seznam jejich atributů.
- Termínem *kontrakt* pak označujeme souhrn informací definujících požadavky objektu na způsob použití (např. že argumentem musí být název existujícího souboru) a co se stane po použití objektu (např. že přidá zadaný prvek do zadaného kontejneru, nebo že zadaný soubor smaže apod.).

Kontrakt funkcí, modulů a tříd bývá uveden v jejich dokumentačních komentářích, kontrakt jiných objektů (pokud je třeba jej zveřejnit) bývá v běžných komentářích.

1.10.3 Formalizace popisu rozhraní – API

V souvislosti s rozhraním se často setkáte s termínem API, což je zkratka z anglického *Application Programming Interface* – aplikační programové rozhraní. Jedná se o sadu pravidel, definic a nástrojů specifikujících podporované způsoby komunikace s daným programem.

Na API se můžete dívat ze dvou pohledů:

- Pro uživatele programu s daným API je to sbírka slibů, co je vám daný program (knihovna, framework, ...) schopen poskytnout, pokud o to požádáte způsobem specifikovaným v daném API.
- Pro poskytovatele programu s daným API je to vedle výše zmíněné nabídky také sbírka požadavků, co musí jeho program umět, aby s ním ostatní mohli podle dané specifikace API spolupracovat.

Pro specifikaci požadavků daného rozhraní v kódu zavedly některé programovací jazyky zvláštní datové konstrukce, které označují jako **interface** (*Java*, *C#*) nebo jako **protocol** (*Objective C*, *Swift*). Tyto konstrukce se však uplatní pouze u *jmenovitého dědění* (vysvětlíme si je za chvíli). *Python* proto ve verzi 3.8 zavedl své vlastní řešení, které si podrobněji vysvětlíme v pasáži [5.9 Protokoly](#) na straně .

1.10.4 Dva způsoby dědění

Při práci v *Pythonu* se uplatní dva způsoby dědění:

Jmenovité dědění (nominal subtyping)

Většina programátorů v třídních objektové orientovaných jazycích chápe pod pojmem dědění pouze jmenovité dědění, protože jazyky, v nichž pracují, ani jiné dědění nepodporují. Při jmenovitém dědění se v definici dceřiné třídy vyjmenují její bezprostřední rodičovské třídy. Překladač pak zařídí, že dcera zdědí od svých rodičů kompletní implementaci.

Jmenovité dědění je tedy primárně děděním implementace. Dokud nezačneme dceřinou třídu upravovat, funguje v případě jediného rodiče i jako dědění rozhraní. My se pak musíme postarat, aby naše úpravy nebo nešikovně zadaná kombinace rodičů nenarušily povinnost nenabourat zděděné rozhraní.

Protože o dceřiné třídě víme něco navíc (známe její specializaci), můžeme její instance vybavit i dalšími vlastnostmi a schopnostmi, které z této specializace vyplývají – přidat další atributy a/nebo modifikovat její metody. Proto se také o dceřiné třídě někdy říká, že *rozšiřuje* (anglicky *extends*) svoji rodičovskou třídu.

Toto rozšíření ale v žádném případě neznamená, že by se rozšířil počet objektů, které jsou jejími instancemi. Naopak, ten se většinou zmenší. Mohli bychom říci, že dceřiná třída rozšíří požadavky, kterým musejí vyhovovat její instance, a tím logicky zmenší počet instancí, jež jim budou vyhovovat.

Strukturální dědění

Python patří mezi jazyky, které vedle jmenovitého dědění podporují i strukturální dědění (anglicky *structural subtyping*), při kterém instance dceřiné třídy implementuje všechny atributy z rozhraní rodičovské třídy. Je přitom jedno, zda tyto atributy obdrží od své mateřské třídy nebo zda si je vytvoří sama.

Mohli bychom tak říci, že strukturální dědění je děděním rozhraní. Protože za nás při tomto způsobu dědění překladač nic neudělá, musíme o to více dbát na dodržování LSP.

Tento způsob práce se v *Pythonu* používá velmi často. V řadě případů nehledíme na to, komunikujeme-li s instancí konkrétní třídy, ale pouze na to, má-li oslovaný objekt požadované vlastnosti a schopnosti – má-li požadované rozhraní, resp. podporuje-li požadovaný protokol, a může se proto vydávat za požadovaný objekt.

Na rozdíl od jmenovitého dědění nepotřebuje strukturální dědění žádnou podporu v syntaxi jazyka. Proto se v této části soustředím především na dědění jmenovité a ke strukturálnímu dědění se vrátím až v podkapitole [5.8 Statické, dynamické a kachní typování](#) na straně .

1.10.5 Tři druhy dědění

Podíváme-li se na dědění trochu z nadhledu, můžeme rozlišit tři různé druhy.

Přirozené (nativní) dědění

Jak jsme si řekli, dědění používáme tehdy, můžeme-li v instancích předka vymezit jistou podmnožinu objektů se speciálními vlastnostmi, kvůli nimž má smysl pro ně definovat zvláštní třídu.

Přirozené dědění hovoří o tom, jak specializaci objektů cítíme bez ohledu na to, jak ji naprogramujeme. Ve škole vás např. učili, že čtverec je zvláštní druh obdélníku, který má všechny strany stejně dlouhé. Čtverec bychom tedy v programu mohli teoreticky definovat jako potomka obdélníku. Jenomže to by platilo pouze do chvíle, než by naše aplikace vyžadovala, aby všechny obdélníky uměly libovolně měnit svůj rozměr. To čtverec nedokáže, protože musí mít stále všechny strany stejně dlouhé. Čtverce tedy mohou v programech vystupovat jako potomci obdélníků pouze tehdy, pokud je přípustná změna velikosti pouze jako násobek velikosti původní a není potřeba měnit nezávisle výšku a šířku objektu. Obdobné by to bylo s kruhem jako potomkem elipsy.

Jiným příkladem mohou být např. slovníky. Jak jsme si řekli, slovník bychom mohli považovat za množinu dvojic (klíč: hodnota). Jinými slovy, slovník bychom mohli chápat jako speciální případ množiny. V knihovně ale není naprogramován jako potomek množiny a v některých knihovnách je naopak množina naprogramována jako speciální druh slovníku. Dokud je to interní záležitost implementace, o které se uživatel v podstatě nedozví, tak to příliš nevadí. Problém by byl, kdyby spolu přirozené chápání a způsob implementace kolidovaly.

Závěr: Nepřenášejte své znalosti o tom, že něco je speciálním případem něčeho, do svých programů mechanicky. Přemýšlejte nejprve o tom, jestli bude dané dědění v programu přirozené a nebude si vynucovat nějaké zvláštní zacházení.

Dědění rozhraní

Dědění rozhraní je v programování klíčové. **Zdělím-li rozhraní svého rodiče, budu nabízet totéž co rodič** (možná i něco navíc, ale to teď pomineme) a **budu se proto moci vydávat za instanci rodiče**.

Zdůrazňuji, že **dědění rozhraní je o slibech**. Když programátor prohlásí, že daný datový typ dědí od nějakého jiného typu, tak tím prohlašuje, že daný typ bude umět vše, co jeho předek. Na zodpovědnosti programátora ale je, aby tento slib dodržel. Nedodržení tohoto slibu je příčinou celé řady chyb.



Upozornění pro ty, kteří se setkali s jazykem *Java* nebo *C#*: **Nezaměňujte dědění rozhraní s děděním interfejsů**⁶ definovaném v těchto jazycích. Mají sice leccos společného, ale jejich dědění interfejsů je standardní jmenovité dědění a nemusí se omezovat pouze na dědění rozhraní.

Dědění implementace

To, že se uplatní dědění implementace, znamená, že potomek zdědí veškerý rodičem definovaný kód a může jej používat jako svůj. O to, aby to fungovalo, se v OO jazycích postará překladač. Programátor pak nemusí definovat v potomkovi pro všechno vlastní metody, ale v řadě případů může použít metodu zděděnou od jeho rodiče či od některého z prarodičů.

V dědění implementace jsou skryté mnohé záludnosti. Nezkušený programátor je často tak nadšen tím, že by mohl zdědit implementaci nějaké metody a ušetřit si tak nějaké to psaní, že zcela zapomene na nutnost dodržet také rozhraní. Rodičovské rozhraní pak může kolidovat s naprogramovaným, takže instance potomka již není schopna plnohodnotně vystupovat v roli instance předka, a program pak v některých situacích vykazuje podivné chování.

Se špatnými návrhy dědění ignorujícími výše popsaná pravidla se setkáte v řadě učebnic a kurzů včetně univerzitních. Ukázkově nevhodný návrh pak předvedl jeden lektor, který vysvětloval výhody dědění následovně:

Představte si, že máte cyklistu, který umí jezdit na kole. Přijede k potoku s řadou kamenů, po nichž by jej mohl přeskákat. Cyklista to ale neumí. Víme ale, že to umí žába. Definujeme proto cyklistu jako potomka žáby, čímž zdědíme schopnost přeskákat potok po kamenech. Cyklista jej přeskáče a může pokračovat dál.

Uvědomíte-li si, že potomek je vždy pouze speciálním případem předka, je vám jasné, že příslušný lektor nenaprogramoval cyklistu, který umí skákat po kamenech, ale žabu, která umí jezdit na kole. Se schopností skákat po kamenech cyklista automaticky zdědil i schopnost plodit pulce, pořádat večerní žabí koncerty a řadu dalších „užitečných“ žabích schopností a dovedností.

Shrnutí

Předchozí dlouhé povídání bychom mohli shrnout do dvou bodů:

- Zopakuji závěr nativního dědění: Nepřenášejte své znalosti o tom, že něco je speciálním případem něčeho, do svých programů mechanicky. Přemýšlejte nejprve o tom, jestli bude dané dědění v programu přirozené a nebude si vynucovat nějaké zvláštní zacházení.
- Při jmenovitém dědění hlídejte, abyste nevhodnou modifikací zděděných atributů (a mezi nimi především metod) nenarušili rozhraní předka. Přísně dodržujte LSP.

Toto pravidlo platí ještě přísněji při používání strukturálního dědění probíraného před chvílí.

⁶ Interfejs je v těchto jazycích definován jako datová struktura podobná třídě.

1.11 Shrnutí kapitoly a doprovodné soubory

V této kapitole jsme položili teoretické základy objektově orientovaného paradigmatu, které budeme v dalších částech knihy aplikovat v *Pythonu*. Nejdůležitější probrané principy jsou:

- OOP je přístup, který vnímá software jako simulaci světa složeného z komunikujících objektů.
- Třída slouží jako vzor či továrna pro objekty, které nazýváme její instance. Vlastnosti a schopnosti objektů jsou definovány jejich atributy a metodami.
- Klíčovým mechanismem je dědění, které umožňuje vytvářet specializované podtřídy. Polymorfismus pak zařizuje, aby různé objekty reagovaly na stejnou zprávu odlišným způsobem v závislosti na jejich skutečném typu.
- Substituční princip Liskové (LSP) je zásadní pravidlo dědění, které říká, že instance potomka musí být vždy schopna plnohodnotně vystupovat v roli instance svého předka.

1.11.1 Doprovodné soubory

Tato kapitola byla čistě teoretická, takže se v ní nepoužívaly žádné doprovodné programy.

Kapitola 2

Třídy a jejich instance



Co se v kapitole naučíte

Tato kapitola vás nejprve naučí definovat třídu a její atributy. Ukáže vám, jak je možné atributy dodatečně přidávat, odebírat a měnit. Poté vám vysvětlí, jak zařídit, aby třída správně vytvářela své instance, jak ve třídě definovat vlastnosti a schopnosti jejích instancí a jak s vytvořenými instancemi následně pracovat. Na závěr představí některé speciální atributy a probere zvláštnosti vestavěných tříd.

2.1 Definice třídy a jejích atributů

Definice třídy je v *Pythonu* složený příkaz, jehož hlavička začíná klíčovým slovem **class**. Tělo třídy pak může obsahovat inicializace atributů třídy, mezi něž zahrnujeme i definice jejích metod. Ve výpisu [2.1](#) jsou připraveny dvě definice tříd.



Abych mohl interpretu spuštěnému z příkazového řádku zkopírovat definice ve výpisu [2.1](#) ze schránky, vložil jsem na začátek oddělovacích řádků uvnitř definice třídy **C1** (ve výpisu řádky [9](#), [11](#), [15](#), [22](#), [26](#) a [28](#)) znak **#** uzavírací komentář, protože by jinak interpret považoval tyto řádky za ukončení definice a následující řádek by vyvolal chybu **IndentationError: unexpected indent**.

V běžném zdrojovém kódu by tento problém nenastal stejně, jako nenastane při používání IDLE, protože IDLE zpracovává vstup až po stisku ENTER, takže řádky hromadně zadané ze schránky bere jako zadání jednoho příkazu a rozčlňuje se pro změnu tehdy, zadáte-li dva příkazy najednou.

Tělo třídy **C0** definované na řádku [1](#) tvoří pouze dokumentační komentář.

Definice třídy **C1** na řádcích [3-29](#) je trochu složitější. Třída **C1** definuje datový atribut **ca** (řádek [10](#)), metodu (tj. funkční atribut) **cm()** (řádky [12-14](#)) a vnořenou třídu (typový atribut) **C11** (řádky [16-27](#)), která pak definuje svůj atribut **ca** (řádek [21](#)) a metodu **cm()** (řádky [23-25](#)). Jak jistě odhadnete, takovýchto atributů, metod a vnořených tříd můžeme definovat tolik, kolik bude potřeba.

Výpis 2.1: Definice třídy a jejích atributů

```

1  >>> class C0: """Třída s prázdným tělem"""
2  ...
3  >>> class C1:
4  ...     """Třída definující svůj datový, funkční i typový atribut,
5  ...     tj. datovou proměnnou, metodu a třídu.
6  ...     Současně demonstruje spouštění příkazů v těle při definici.
7  ...     """
8  ...     print('START definice třídy C1')
9  ...     #
10 ...     ca = 'Atribut ca třídy C1'
11 ...     #
12 ...     def cm():
13 ...         """Statická metoda třídy C1 -- volá se "C1.cm()"."""
14 ...         print('Byla zavolána metoda C1.cm')
15 ...     #
16 ...     class C11:
17 ...         """Vnitřní třída třídy C1 - oslovuje se "C1.C11".
18 ...         Má vlastní datový atribut a funkční atribut (metodu).
19 ...         """
20 ...         print('    START definice třídy C1.C11')
21 ...         ca = 'Atribut ca třídy C1.C11'
22 ...     #
23 ...     def cm():
24 ...         """Statická metoda třídy C11 - volá se "C1.C11.cm()"."""
25 ...         print('Byla zavolána metoda C1.C11.cm')
26 ...     #
27 ...     print('    KONEC definice třídy C1.C11')
28 ...     #
29 ...     print('KONEC definice třídy C1')
30 ...
31 START definice třídy C1
32     START definice třídy C1.C11
33     KONEC definice třídy C1.C11
34 KONEC definice třídy C1
35 >>> f'{type(C0)=}, {type(C1)=}, {type(C1.C11)=}'
36 "type(C0)=<class 'type'>, type(C1)=<class 'type'>, type(C1.C11)=<class 'type'>"
37 >>>

```

Kód těla třídy `C1` i její vnořené třídy `C11` navíc obsahuje příkazy tisku, které mají demonstrovat, že při definici třídy se příkazy v jejím těle ihned provádějí. Jimi realizované tisky jsou na řádcích 31–34.

Všechny tři definované třídy i obě definované metody jsou vybaveny dokumentačními komentáři. Ty nám mimo jiné připomínají, že metoda je vlastně funkční atribut a že vnořená třída je typový atribut.

Jak jsme si řekli, v *Pythonu* je vše objekt, takže i třídy jsou objekty. Na řádce 36 zjišťujeme, jakého jsou typu. Podle očekávání se dozvíme, že všechny tři jsou (stejně jako všechny ostatní třídy) instancemi třídy `type`.