edice
začínáme s ...

Začínáme programovat v jazyku PYTHON

- Nepředpokládá žádné předchozí znalosti programování
- Výklad je postaven na vybudování jednoduché aplikace a průběžném seznamování s potřebnými konstrukcemi
- Neomezuje se na výuku kódování v Pythonu, ale učí, jak program navrhnout a postupně vyvinout a rozchodit
- Učí čtenáře programovat podle moderních zásad a metodik

soubory ke stažení na www.grada.cz

RUDOLF PECINOVSKÝ

GRADA®

edice
začínáme s ...

Začínáme programovat v jazyku PYTHON

RUDOLF PECINOVSKÝ

GRADA Publishing

Rudolf Pecinovský

Začínáme programovat v jazyku Python

Vydala Grada Publishing, a.s.
U Průhonu 22, Praha 7
obchod@grada.cz, www.grada.cz
tel.: +420 234 264 401
jako svou 7731. publikaci

Odpovědný redaktor: Petr Somogyi
Fotografie na obálce Depositphotos/novotnyfi
Grafická úprava a sazba Rudolf Pecinovský
Počet stran 272
První vydání, Praha 2020
Vytiskly Tiskárny Havlíčkův Brod, a. s.

Dotisk 2021

ISBN 978-80-271-1828-1 (ePub)
ISBN 978-80-271-1827-4 (pdf)
ISBN 978-80-271-1237-1 (print)

Všem, kteří se chtějí něco naučit

Stručný obsah

Podrobný obsah

Úvod

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

Popularita jazyka *Python* nepřetržitě roste. Postupně se stává klíčovým jazykem v řadě oblastí, především v těch, které souvisejí s výukou a výzkumem. Je to nejčastěji vyučovaný první jazyk na univerzitách i středních školách, je nejpoužívanějším jazykem ve statistice, programování umělé inteligence, v aplikacích využívajících strojové učení, je hlavním jazykem v oblasti analýzy dat a postupně proniká do dalších oblastí tvorby softwaru. Hraje důležitou roli i v oblasti webového programování a různých vědeckých výpočtů. Často se k němu obracejí odborníci, kteří potřebují na počítači vyřešit nějaký problém a jiné jazyky jim připadají buď příliš těžkopádné, anebo pro ně neexistují potřebné knihovny.

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. Potřebují proto jazyk, který se mohou rychle naučit a v němž budou moci rychle vytvářet jednoduché programy řešící (nebo pomáhající řešit) jejich problém. Na druhou stranu ale sílí i jeho využití profesionálními vývojáři pro rozsáhlé podnikové a webové aplikace.

Komu je kniha určena

Tato kniha je určena především těm, kteří ještě nikdy neprogramovali, anebo se je to sice někdo snažil naučit, ale oni už vše zase zapomněli. Nepředpokládá žádné předběžné znalosti a dovednosti kromě základů práce s počítačem. Jejím cílem je předat čtenáři základní znalosti a naučit ho dovednosti potřebné k vytváření jednoduchých aplikací. Osvojené základy jim pak umožní, aby v případě hlubšího zájmu o programování v jazyku *Python* pokračovali některou z učebnic určených pro mírně pokročilé programátory – nejlépe samozřejmě mojí učebnicí [11] a doplňkovou příručkou [12].

Zkušenost však ukázala, že v této učebnici najdou řadu cenných informací i programátoři, kteří již mají jisté zkušenosti, ale kurzy, jimiž doposud prošli, se soustředily především na odpověď na otázku **jak**, a oni by se nyní rádi dozvěděli také odpověď na otázku **proč**.

Kniha je učebnicí programování. Učí své čtenáře navrhovat programy a dále je vylepšovat. Není učebnicí jazyka *Python* (tou je učebnice [11]), a proto se nesnaží probrat všechny jeho konstrukce, ale omezuje se při výkladu pouze na ty, jejichž zvládnutí je pro návrh jednoduché aplikace nezbytné.

Vedle konstrukcí jazyka ale učí čtenáře také řadu zásad moderního programování, jejichž zvládnutí je nutnou podmínkou pro všechny, kdo nechtějí zůstat u malých žákovských programů, ale chtějí se naučit efektivně vyvíjet robustní středně rozsáhlé aplikace, jejichž údržba nebude vést uživatele k chrlení nepublikovatelných výroků na adresu autora.

Dopředu se omlouvám, že se obě knihy částečně překrývají, ale cítil jsem potřebu některé věci vysvětlit jak začátečníkům, pro něž je určena tato učebnice, tak pokročilejším čtenářům, pro něž jsem psal knihu [11]). Nepočítal jsem to, ale odhaduji, že asi 30 stránek mají obě knihy společných.

V případných příštích vydáních se pokusím tento překryv zmenšit, ale tentokrát jsem považoval za důležitější rychlost. Zájem o začátečnickou učebnici, který vyvolalo vydání příručky [10], mne i nakladatele zaskočil, že jsme se rozhodli upřednostnit rychlost vydání před případnou dokonalostí obsahu.

Koncepce výkladu a jeho uspořádání

Kniha je koncipována tak, aby mohla sloužit jako středoškolská učebnice programování i jako učebnice pro samouky zajímající se o programování. Probírá vše potřebné od naprostých základů až po některé rysy, které se v začátečnických příručkách běžně neprobírají, ale jejichž znalost považuji za velmi užitečnou, protože pomáhá rychleji odhalit příčiny chyb. Kniha je rozdělena do čtyř částí.

První část

První část probírá naprosté základy, bez jejichž znalosti nelze vytvořit ani velice jednoduchou aplikaci. Naučíte se v ní pracovat s čísly a texty a dozvíte se, jak vytvářet vlastní funkce. Pak představí základy objektově orientovaného programování, na němž stojí celý *Python* a které nevědomky používají i ti, kteří tvrdí, že objektově nepracují. V závěru vás pak naučí ukládat vytvořené části programu a v případě potřeby je opět spouštět.

Doprovodné programy v první části se nesnaží nic řešit. Jsou to vesměs AHA-příklady, tj. příklady, jejichž jediným cílem je, aby si čtenář řekl: „Aha, takto to funguje.“

Druhá část

Ve druhé části začínáme vytvářet jednoduchou aplikaci s tím, že když narazíme na problém, který ještě neumíte řešit, tak vás seznámím s programovými konstrukcemi, jež se k řešení takovýchto problémů používají.

Zkušenost ukazuje, že studenti, kteří se nejprve naučí kódovat a soustředit se především na detail, mívají později velké problémy s návrhem architektury svých programů. Proto vás nejprve seznámím se základy návrhu architektury programu a principem návrhových vzorů.

Poté navrhneme základní architekturu naší aplikace a začneme tuto architekturu realizovat. Paralelně vám přitom představím datové struktury umožňující efektivně pracovat se skupinami dat.

Seznámíte se s metodikou programování řízeného testy a v závěru této části se nejprve naučíte zabudovávat do programu rozhodování a opakování. Poté navrhneme test chystané hry, který nám v další části bude ukazovat, kudy postupovat.

Třetí část

Ve třetí části postupně vytváříme plánovanou aplikaci. Vždy spustíme test, a ten nám řekne, co máme v dalším kroku opravit a rozchodit. Na konci této části budete mít rozchozenou jednoduchou aplikaci realizující textovou konverzační hru.

Jakmile bude aplikace funkční, doplníme k ní jednoduché uživatelské rozhraní a ukážeme si, jak aplikaci uložit, abychom ji pak mohli snadno distribuovat těm, kteří ji potřebují nebo po ní alespoň touží.

Čtvrtá část

Čtvrtá část je poměrně krátká a představuji v ní několik námětů, jak je možné aplikaci zdokonalit. V prvních dvou kapitolách ještě ukazuji některé doposud nepoužité programátorské obraty a opravdu v nich něco naprogramujeme. Poslední kapitola této části reaguje na to, že cílem učebnice není vytvoření dokonalé aplikace, ale výuka základních znalostí a dovedností moderního programování. Náměty v ní uvedené jsou proto opravdu pouhé náměty pro vaše další experimenty.

Přílohy

Kniha obsahuje tři přílohy. V první seznamuji uživatele *Windows* s možnostmi definice substituovaných disků. Druhá obsahuje přehled metod ze standardní knihovny, které jsme v průběhu výuky použili. Třetí obsahuje výtah z oficiálních konvencí pro psaní kódu a dokumentačních komentářů na stránkách *Pythonu*.

Jazyk identifikátorů

Jak jsem řekl, doprovodné programy v první části jsou vesměs AHA-příklady vysvětlující probíranou konstrukci. V nich budu pro větší názornost používat české identifikátory. V AHA-příkladech budu češtinu používat i v dalších částech knihy.

Při návrhu aplikace ale budu používat identifikátory anglické, jak je ostatně v programování zvykem (a jak doporučují konvence). Komentáře a tištěné texty však zůstanou nadále české.

Potřebné vybavení

Pro úspěšné studium této knihy je vhodné mít instalovanou platformu *Python* ve verzi nejméně 3.9. Tu lze stáhnout na adrese https://www.python.org/downloads/. Pokud jste si však pořídili knihu před 5. říjnem 2020, na kdy je naplánováno vydání ostré verze, tak zaběhněte na https://www.python.org/ftp/python/3.9.0/, kde najdete poslední betu či kandidáta vydání (release candidate). Na její rodičovské stránce najdete odkazy na stránky pro všechny doposud vydané verze od 2.0 i rozpracované verze, které se teprve připravují.

Doprovodné programy

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 adrese[1] http://knihy.pecinovsky.cz/65_pythonvb.

Názvy zdrojových souborů s doprovodnými programy (moduly) začínají vždy písmenem `m` následovaným dvojmístným číslem kapitoly. Následují-li dvě podtržítka a text (např. `m01__prolog`), jedná se o soubor příkazů zadaných v průběhu kapitoly. Je-li za číslem malé písmeno následované jedním podtržítkem (např. `m01a_script`), jedná se o samostatný modul.

Většina těchto souborů se vyskytuje ve třech podobách se stejným názvem, ale odlišnou příponou. Soubory s příponou `py` jsou zdrojové soubory *Pythonu*, soubory s příponou `pyrec` obsahují záznam konverzace se systémem a soubory s příponou `pydoc` obsahují výpisy programů v knize i s uvedenými čísly řádků. Tyto soubory vám mají usnadnit sledování rozboru některých programů, abyste při něm nemuseli neustále listovat mezi rozborem a rozebíraným výpisem.

[1] Číslicí šedesát pět v adrese se nevzrušujte, jedná se pouze o moje interní označení pořadí vytvářené knihy, protože bych v nich jinak bloudil.

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 jejích produktů vysazuji *kurzivou*. Kurzivou vysazuji také názvy kapitol, podkapitol a oddílů, na něž v textu odkazuji.

Citace Texty, které si můžete přečíst na displeji, např. názvy polí v dialogových oknech či názvy příkazů v nabídkách, vysazuji **tučným bezpatkovým písmem**.

Odkaz Celá kniha je prošpikovaná křížovými odkazy na související pasáže. Není-li odkazovaný objekt (kapitola, obrázek, výpis programu, …) na stejné stránce nebo na některé ze sousedních stránek, je pro čtenáře tištěné verze doplněn o číslo stránky, na níž se nachází. Čtenářům elektronické verze stačí, když na něj klepnou. Použitý prohlížeč by je měl na odkazovaný objekt ihned přenést.

Adresa Názvy souborů a internetové adresy vysazuji obyčejným bezpatkovým písmem.

`Program` Identifikátory a další části programů zmíněné v běžném textu vysazuji `neproporcionálním písmem`, které je v elektronických verzích pro zvýraznění tmavě červené.

zadání V záznamech komunikace se systémem budou texty, které zadává uživatel, vysazeny tučně (v elektronických verzích pro zvýraznění tmavě modře).

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.

Otevřená schránka s dopisy označuje informace o doprovodných programech týkajících se dané kapitoly.

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

Ruka s hrozícím prstem upozorňuje na věci, které byste měli určitě vědět a na které 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 různá úskalí programovacího jazyka nebo programů, s nimiž budeme pracovat, a bude vám radit, jak se těmto nástrahám vyhnout či jak to zařídit, aby vám alespoň pokud možno nevadily.

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. Seznam všech terminologických poznámek najdete v rejstříku pod heslem „terminologie“.

Brýle označují tzv. „poznámky pro šťouraly“, ve kterých se vás snažím seznámit s některými zajímavými vlastnostmi probírané konstrukce nebo upozorňuji na některé souvislosti, avšak které 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 ostatního výkladu. Nadpis podšeděného bloku pak najdete i v podrobném obsahu mezi nečíslovanými nadpisy.

Zpětná vazba

Předem přiznávám, že tato kniha je sice mou šedesátou pátou učebnicí, ale současně je to moje první učebnice programování v jazyku *Python*. Nelze proto vyloučit, že 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 či budete mít návrh na nějaké její vylepšení, neostýchejte se poslat na adresu rudolf@pecinovsky.cz e-mail s předmětem 65_PYTHON_NZ_DOTAZ.

Bude-li se dotaz týkat něčeho obecnějšího nebo to bude upozornění na chybu, pokusím se co nejdříve zveřejnit na stránce knihy http://knihy.pecinovsky.cz/65_python39vb 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 si netroufnou se sami zeptat.

Část A
Základy

První část knihy vás seznámí se základními konstrukcemi jazyka *Python*, bez jejichž znalosti se při sestavování programu neobejdete. Nejprve poskytne základní informace o počítačích, programech a programovacích jazycích. Pak vás seznámí se zadáváním čísel a textů, naučí vás definovat funkce a třídy a na závěr vám ukáže, jak ukládat a načítat vaše programy.

Kapitola 1
Předehra

Co se v kapitole naučíte
Tato kapitola bude spíše teoretická. Má vás především seznámit s termíny, které pak budeme v textu knihy používat. Zároveň vám představí vývojové prostředí IDLE, které budeme v dalším textu používat k zadávání a spouštění doprovodných programů.

1.1 Hardware a software

První počítače

První samočinný počítač řízený programem byl mechanický a navrhl jej v roce 1837 Charles Babbage. Tou dobou byl také publikován první program. Napsala jej Ada Lovelace a vyšel v jejích poznámkách překladatele ke knize o Babbageově stroji.

První fungující elektromechanický samočinný počítač zprovoznil v roce 1938 Konrad Zuse v Německu. V průběhu druhé světové války vznikly další počítače v Anglii a ve Spojených státech.

Po válce vznikl poměrně rychle trh s počítači. Počítače konstruovali převážně muži a programovali je převážně ženy, protože řada mužů považovala programování za něco přízemního, co není hodno jejich intelektu. Situace se začala měnit až v šedesátých letech, protože se postupně ukazovalo, že podniky utratí mnohem více za programové vybavení než za vlastní počítač.

Navíc většina důvtipných řešení a patentů v oblasti hardwaru rychle zastarala a byla překonána mnohem důvtipnějšími řešeními někoho jiného, kdežto objevy na poli softwaru přežívaly celá desetiletí. V současné době se už proto hlavní vývoj neodehrává v oblasti hardwaru, ale softwaru.

Co je to program

Program bychom mohli charakterizovat jako v nějakém programovacím jazyku zapsaný předpis popisující, jak má procesor, pro nějž je program určen (v našem případě počítač), splnit zadanou úlohu.

Cílovým procesorem nemusí být vždy počítač. Oblíbeným příkladem programů jsou např. kuchařské předpisy. V předpočítačových dobách zaměstnávaly některé instituce (např. armáda) velké skupiny počtářů[2] řešících na mechanických kalkulačkách výpočty podle algoritmů, které dnes zakódujeme v nějakém programovacím jazyku a předhodíme počítači.

Na programu je důležité, že musí být napsán v nějakém programovacím jazyku, kterému rozumí programátor. Napsaný program je pak jiným programem (překladačem) převeden do „jazyka“, kterému rozumí počítač.

Vybrat jazyk pro kuchařský předpis je jednoduché, vybrat jazyk pro počítač je mnohem složitější. Vlastnosti použitého jazyka totiž naprosto zásadně ovlivňují jak rychlost vývoje programu, tak i rychlost a kvalitu výsledných programů. Proto také prošly programovací jazyky i metodiky jejich používání celou řadou revolučních změn.

Změny přístupu k tvorbě programů

Počítače byly postupně nasazovány v dalších a dalších oblastech a programátoři pro ně vytvářeli stále dokonalejší programy. Programy byly čím dál rafinovanější a složitější, a to začalo vyvolávat velké problémy. Programátoři totiž přestávali být schopni své programy rozchodit, a když je vítězně rozchodili, nedokázali z nich v rozumném čase odstranit chyby, které uživatelé v programu objevili.

Tato krize vedla k zavádění nejrůznějších metodik, které měly jediný cíl: pomoci programátorům psát spolehlivé a snadno upravovatelné programy. V padesátých letech minulého století se tak prosadily vyšší programovací jazyky, v šedesátých letech modulární programování, v sedmdesátých letech na ně navázalo strukturované programování a v průběhu osmdesátých a zejména pak devadesátých let ovládlo programátorský svět objektově orientované programování, jehož vláda pokračuje dodnes. (Jednotlivé termíny si za chvíli probereme podrobněji.)

Hlavním cílem programátorů v počátcích programování bylo, aby jejich programy spotřebovaly co nejméně paměti a byly co nejrychlejší. Tehdejší počítače totiž měly paměti málo, byly z dnešního hlediska velice pomalé a jejich strojový čas byl drahý. Se stoupající složitostí programů však byly takto psané programy stále méně stabilní a stále hůře udržovatelné. Současně s tím, jak klesala cena počítačů, jejich strojového času i paměti, začínal být nejdražším článkem v celém vývoji člověk.

---

[2] Přiznejme si, že to tehdy byly většinou počtářky, protože muži dělají při práci podobného druhu příliš mnoho chyb. Takovéto „lidské počítače“ pomáhaly např. v Sovětském svazu s vývojem atomové pumy či s prvními lety do vesmíru.

Cena strojového času a dalších prostředků spotřebovaných za dobu života programu začínala být pouze zlomkem ceny, kterou bylo nutno zaplatit za jeho návrh, zakódování, odladění a následnou údržbu. Začal být proto kladen stále větší důraz na produktivitu programátorů i za cenu snížení efektivity výsledného programu.

Prakticky každý program zaznamená během svého života řadu změn. Požadavky zákazníka na to, co má program umět, se většinou průběžně mění, a program je proto třeba průběžně upravovat, rozšiřovat a vylepšovat. Celé současné programování je proto vedeno snahou psát programy nejenom tak, aby pracovaly efektivně, tj. rychle a s minimální spotřebou různých zdrojů (operační paměť, prostor na disku, kapacita sítě atd.), ale aby je také bylo možné kdykoliv jednoduše upravit a vylepšit. A k tomu je třeba, aby byl program čitelný a srozumitelný, protože programátoři věnují mnohem více času čtení programu než jeho psaní.

Předchozí zásady krásně shrnul Martin Fowler ve své knize Refactoring ([4], český překlad [5])

> **„Napsat program, kterému porozumí počítač, umí každý trouba.**
> **Dobrý programátor píše programy, kterým porozumí i člověk.“**[3]

K tomu se vás budu snažit vést v průběhu celého výkladu. Neustále se vám budu snažit vštěpovat zásadu CRIDP, což je zkratka z anglického *Code Readability Increases Development Productivity* – čitelnost kódu zvyšuje efektivitu vývoje.

1.2 Překladače, interprety, platformy

Než se pustíme do dalšího výkladu, měli bychom si ujasnit pár termínů, které budu v průběhu dalšího textu často používat.

Operační systém

Operační systém je sada programů, jejímž úkolem je zařídit, aby počítač co nejlépe sloužil zadanému účelu. Operační systémy osobních počítačů se snaží poskytnout co největší komfort a funkčnost jak lidským uživatelům, tak programům, které operační systém nebo tito uživatelé spouští. (Teď nehodnotím, jak se jim to daří.)

Operační systém se snaží uživatele (člověka nebo program) odstínit od hardwaru použitého počítače. Uživatel může střídat počítače, avšak dokud bude na všech stejný operační systém, bude si se všemi rozumět.

Při obsluze lidského uživatele to má operační systém jednoduché: člověk komunikuje s počítačem pomocí klávesnice, obrazovky, myši a případně několika dalších

[3] *“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”*

zařízení. Ty všechny může operační systém převzít do své správy a zabezpečit, aby se nejrůznější počítače chovaly vůči uživateli stejně.

U programů to má ale složitější. Programy totiž potřebují komunikovat nejenom s operačním systémem (např. když chtějí něco přečíst z disku nebo na něj něco zapsat), ale také přímo s procesorem, kterému potřebují předat své instrukce k vykonání. Problémem ale je, že různé procesory rozumí různým sadám instrukcí.

Platforma

Abychom věděli, že náš program na počítači správně poběží, musíme vědět, že počítač bude rozumět té správné sadě instrukcí a že na něm poběží ten správný operační systém. Kombinaci *použitý hardware + operační systém* budu v dalším textu označovat termínem *platforma HWOS*.

Platforma HWOS je speciálním případem obecnější počítačové platformy, kterou bychom definovali jako *pracovní prostředí umožňující bezproblémovou činnost programů*. Takováto obecnější platforma odstíní program, který na ní běží, od všeho (nebo alespoň téměř všeho), co se děje „pod ní" – např. od HWOS.

V současné době se dává často přednost oněm obecnějším platformám běžícím nad HWOS. Nejznámější takovou platformou je např. *Java* nebo *JavaScript* (ten ale sám běží často nad platformou definovanou příslušným prohlížečem). Takovouto obecnější platformou je i *Python*.

Programovací jazyky

Jak asi všichni víte, pro zápis programů používáme nejrůznější programovací jazyky. Ty jsou vymýšleny tak, aby v nich mohl člověk co nejlépe popsat svoji představu o tom, jak má počítač splnit požadovanou úlohu.

Program zapsaný v programovacím jazyku pak musíme nějakým způsobem převést do podoby, které porozumí počítač. Podle způsobu, jakým postupujeme, dělíme programy na překládané a interpretované.

Překládaný program

U překládaných programů se musí napsaný program nejprve předat zvláštnímu programu nazývanému překladač (někdo dává přednost termínu kompilátor), který jej přeloží (zkompiluje), tj. převede do podoby, s níž si již daná platforma ví rady. Jinými slovy: musí jej přeložit do kódu příslušného procesoru a používat instrukce, kterým rozumí použitý operační systém. Přeložený program pak můžeme kdykoliv na požádání spustit.

Interpretovaný program

Naproti tomu interpretovaný program předáváme v podobě, v jaké jej programátor vytvořil, programu označovanému jako interpret. Ten obdržený program prochází a ihned jej také provádí – interpretuje jeho příkazy.

Porovnání

Výhodou překládaných programů je, že většinou běží výrazně rychleji, protože u interpretovaných programů musí interpret vždy nejprve přečíst kus programu, zjistit, co má udělat, a teprve pak může tento požadavek vykonat.

Výhodou interpretovaných programů bývá na druhou stranu to, že jim většinou tak moc nezáleží na tom, na jaké platformě běží. Stačí, když na daném počítači běží potřebný interpret. Mohli bychom říci, že platformou těchto programů je právě onen interpret.

Vytvoříte-li pro nový počítač interpret, můžete na něj vzápětí přenést i všechny programy schopné běhu nad tímto interpretem. Kdykoliv tyto programy po systému něco chtějí, požádají o to svoji platformu (interpret), a ta jim příslušné služby zprostředkuje. Takovým programům pak může být jedno, na jakém procesoru a pod jakým operačním systémem běží, protože se beztak „baví“ pouze se svou platformou.

Naproti tomu překládané programy se většinou musí pro každou platformu trochu (nebo také hodně) upravit a znovu přeložit. Při implementaci programu pro více platforem bývá pracnost přizpůsobení programu jednotlivým platformám srovnatelná s pracností vývoje jeho první verze.

Hybridně zpracovávaný program

Vedle těchto základních druhů zpracování programů existuje ještě hybridní způsob zpracování, který se snaží sloučit výhody obou skupin. Při něm se program nejprve přeloží do jakéhosi mezijazyka, který je vymyšlen tak, aby jej bylo možno co nejrychleji interpretovat. Takto přeložený program je potom interpretován speciálním interpretem označovaným často jako **virtuální stroj**.[4] Ten je zodpovědný za maximálně efektivní interpretaci kódu připraveného překladačem.

Hybridní zpracování spojuje výhody obou kategorií. K tomu, aby v nich napsané programy mohly běžet na různých platformách, stačí pro každou platformu vyvinout potřebný virtuální stroj. Ten pak spolu s knihovnou vytváří vyšší, mnohem univerzálnější platformu. Je-li tento virtuální stroj dostatečně „chytrý“ (a to jsou v současné době prakticky všechny), dokáže odhalit často se opakující části kódu a někam stranou je přeložit, aby je nemusel pořád kolem dokola interpretovat.

---

[4] Virtuální stroj se mu říká proto, že se vůči programu v mezijazyku chová obdobně, jako se chová procesor vůči programu v čistém strojovém kódu.

Jazyk versus způsob zpracování

Na překládané, interpretované a hybridní bychom měli dělit způsoby zpracování vytvořených programů, avšak často se takto dělí i programovací jazyky. Je sice pravda, že to, zda bude program překládaný, interpretovaný nebo hybridní, není závislé na použitém jazyku, ale je to především záležitostí implementace daného jazyka, nicméně každý z jazyků má svoji typickou implementaci, podle které je pak zařazován.

Všechny jazyky mohou být implementovány všemi třemi způsoby a pro mnohé opravdu existují všechny tři druhy implementace. U většiny však typická implementace natolik výrazně převažuje, že se o těch ostatních prakticky nemluví. Klasický *Basic* je považován za interpretovaný jazyk, *Java* a *Python* za hybridní a jazyky *C* a *Pascal* za překládané.

Hybridní implementace jazyků se v posledních letech výrazně prosadila a hybridně zpracovávané jazyky jsou dnes králi programátorského světa. Vyvíjí v nich převážná většina programátorů a procento implementací v těchto jazycích neustále vzrůstá.

1.3 Platforma Python

Kniha má sice v názvu, že je učebnicí programování v jazyku *Python*, ale samotný jazyk vám není k ničemu do té doby, než získáte nástroje k tomu, abyste jej mohli používat, tj. než si stáhnete jeho platformu.

Především potřebujete knihovnu, v níž jsou připravené podprogramy realizující nejčastější operace, abyste je nemuseli všechny vytvářet znovu sami. Pak potřebujete překladač, který váš program přeloží do interního jazyka označovaného jako bajtkód (bytecode) a propojí jej s použitými knihovnami. Nakonec potřebujete virtuální stroj, kterému přeložený program předáte, aby jej spustil.

To vše je součástí standardní instalace. Patří do ní i poměrně podrobná dokumentace s definicí jazyka, popisem instalovaných knihoven, jednoduchým tutoriálem a řadou dalších užitečných informací.

Součástí instalace je i nastavení operačního systému, aby textové soubory s některou z definovaných přípon začínajících `py` správně zpracoval. Spustíte-li *Python* z příkazového řádku v konzolovém okně zadáním příkazu `python`, spustíte základní interpret, v němž můžete hned zapisovat jednotlivé příkazy i celé programy. Jako vývojové prostředí je však základní interpret v konzolovém okně příliš těžkopádný a je vhodný maximálně pro spouštění jednotlivých skriptů.

Skripty

Termínem *skript* (anglicky *script*) označujeme spustitelný textový soubor obsahující kód v nějakém skriptovacím jazyce. Klíčové je, že tento soubor musí být spustitelný, tj. že na počítači musí existovat program, který umí tento soubor načíst a zadaný kód provést. Operační systém by pak měl umět poznat, kterému programu má zadaný soubor předat ke zpracování.

Python patří mezi skriptovací jazyky, tj. mezi jazyky, v nichž je možné zapisovat skripty. Soubory se skripty v jazyku *Python* by měly mít příponu `py`, jejich přeložené verze pak příponu `pyc`. Máte-li správě instalovaný *Python*, tak pokud takový soubor zadáte ke spuštění, měl by být spuštěn program `python` a kód v daném souboru provést.

Máte-li již instalovaný *Python* a stažené doprovodné soubory zmiňované v pasáži *Potřebné vybavení* na straně 21, můžete si vše vyzkoušet. Otevřete složku, do níž jste umisťovali doprovodné programy, najděte v ní soubor `m01a_script.py` a poklepejte na něj. Mělo by se otevřít okno konzoly s obsahem:

```
    Toto je první spuštěný skript v Pythonu.
    Až si text přečtete, stiskněte Enter.

Stiskněte Enter
```

Dokumentace

Říkali jsme si, že součástí standardní instalace je i poměrně podrobná dokumentace. Přítomnost dokumentace je nesmírně důležitá (a troufám si tvrdit, že je jednou z příčin úspěchu jazyka), protože *Python* přichází (ostatně jako většina současných platforem) s obrovskou nabídkou možností, které si průměrný lidský mozek nedokáže zapamatovat. Proto jistě oceníte možnost rychlého získání potřebných informací jak o samotném jazyku, tak o instalovaných knihovnách. Navíc zde najdete i několik rad a doporučení.

Soubor s doprovodnou dokumentací má formát `CHM` (zkratka odvozená z Compiled HTML), což je starší formát *Microsoftu* pro tvorbu nápovědy, jehož popularita vedla k tomu, že byly vyvinuty programy umožňující jej číst i v jiných operačních systémech.

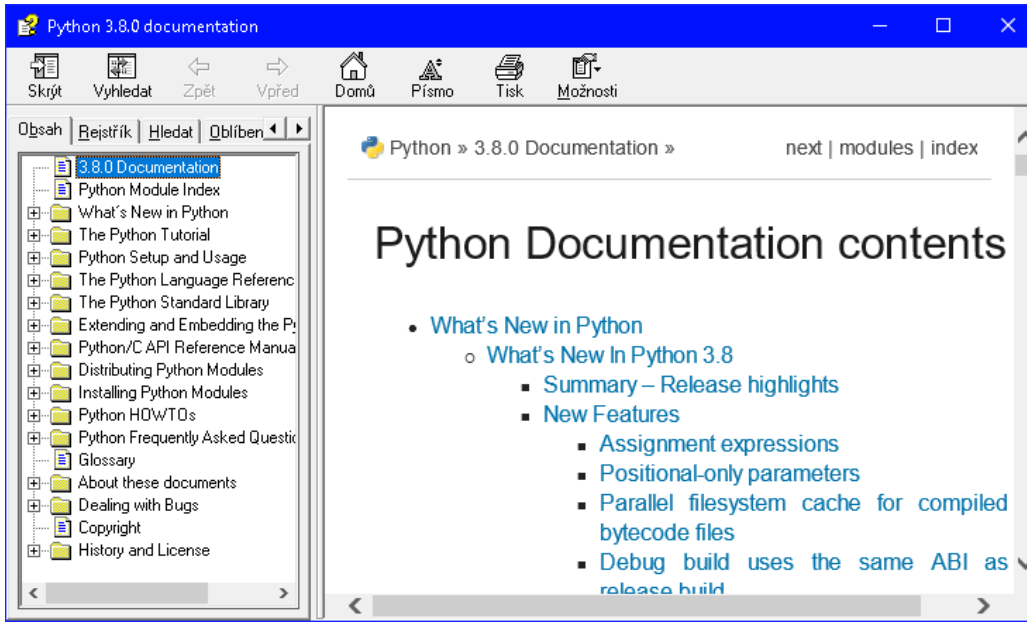
Soubor s dokumentací najdete v podsložce `Doc` složky, do níž jste instalovali *Python*. Po svém spuštění otevře okno, jehož podobu pod operačním systémem *Windows* si můžete prohlédnout na obrázku 1.1. Prohlížeče těchto souborů však existují i pro ostatní platformy.

Dokumentace existuje i v jiných podobách. Na stránce https://docs.python.org/3/ najdete online dokumentaci a na stránce https://docs.python.org/3/download.html odkazy na soubory s dokumentací v různých formátech, které si můžete stáhnout.

Dokumentace je sice pouze anglicky, ale přiznejme si, že všichni, kdo se v současné době zajímají o informatiku, musejí umět anglicky alespoň pasivně, anebo si musejí hledat jiné zaměstnání.

1.4 Vývojové prostředí

Pro psaní skriptů můžete teoreticky využít jakýkoliv textový editor. Většina programátorů však dává přednost speciálním vývojovým prostředím označovaným **IDE**, což je zkratka z anglického *Integrated Development Environment* – integrované vývojové prostředí.



Obrázek 1.1:
Okno s dokumentací platformy

IDE tvoří sada programů, která nabízí řadu funkcí, jež vám pomohou vývoj programů výrazně zefektivnit. Především vám nabídne přiměřeně komfortní editor, prostřednictvím nějž program napíšete, zprostředkuje překlad programu do interního mezijazyka i jeho následné spuštění a zobrazení případných výsledků. Užitečnou součástí jsou i programy, které vám usnadní hledání případných chyb.

Součástí základní instalace *Pythonu* je vývojové prostředí označované zkratkou IDLE, což je jméno jednoho z členů skupiny *Monty Python*, po níž se jazyk jmenuje. Nicméně později byl název vyložen jako zkratka z anglického *Integrated Development and Learning Environment*.

Toto prostředí budu používat v následujícím výkladu, protože je předinstalované a protože se s ním velice jednoduše pracuje. Díky své jednoduchosti neodpoutává pozornost čtenáře od probírané látky k vlastnostem použitého vývojového prostředí.

Pro ty z vás, kteří překonají počáteční nástrahy jazyka a budou chtít začít používat nějaký komfortnější nástroj, bych měl dvě doporučení:

IDE *Thonny*, které je k dispozici na adrese https://thonny.org/, bylo vyvinuté na univerzitě v estonském městě Tartu. Nemám s ním vlastní zkušenosti, ale autoři i recenze tvrdí, že je optimalizované pro začínající programátory, takže je navržené tak, aby práce s ním a jeho ovládání byly co nejjednodušší.

Těm, kteří touží po něčem maximálně dokonalém, bych doporučil vývojové prostředí *PyCharm* vyvinuté firmou *JetBrains* sídlící (mimo jiné) i v Praze. Jeho základní verze je sice placená, nicméně existuje i verze označená *Community Edition*, která je zdarma. Zvládnutí tohoto prostředí je sice poněkud náročnější, ale zejména při vývoji rozsáhlejších projektů se vám tato námaha bohatě vrátí.

1.5 Prostředí IDLE

Jak už jsem ale řekl: my budeme v této učebnici používat prostředí IDLE, protože je za prvé zabudované a za druhé je ze všech nejjednodušší. Pojďme si je představit. Začneme jeho spuštěním.

Spuštění

Spouštěcí skript prostředí IDLE je v souboru `#/Lib/idlelib/IDLE.pyw`, kde symbol `#` zastupuje složku, kam jste instalovali *Python*. Přípona `pyw` znamená, že skript nepotřebuje spouštět okno konzole, protože komunikuje s uživatelem prostřednictvím grafického uživatelského rozhraní (GUI) využívajícího (mimo jiné) oken a myši. Standardní systémové vstupy a výstupy budou proto od programu odpojeny a vše, co by na ně program poslal, se ztratí.

Při práci se soubory se otevírací, resp. ukládací okno otevře ve složce, z níž byl skript spuštěn. Spusťte jej proto ze složky, v níž budete mít uloženy svoje programy, protože jinak se přístup prostředí k těmto programům zkomplikuje.

Optimální je definovat zástupce, kterého umístíte do složky se svými skripty a který IDLE spustí. Používáte-li *Linux*, budete si nejspíš umět nastavit potřebného zástupce otevírajícího IDLE a nastavujícího složku `65_PYT` se zdrojovými soubory jako aktuální. Používáte-li *Windows*, můžete využít mého zástupce v souboru !IDLE_Python_39.lnk. Tento zástupce je ale nastaven pro můj počítač, takže si jej budete muset otevřít a upravit podle uspořádání na vašem počítači. Případný návod najdete v příloze *A.2 Nastavování zástupce spouštějícího IDLE* na straně 257.

Základní popis

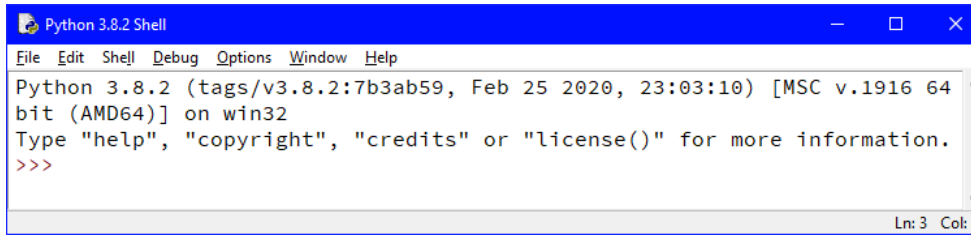
Prostředí IDLE je multiokenní prostředí, jehož okna mohou pracovat v jednom ze dvou režimů:

- Jedno z oken může pracovat v příkazovém režimu, který označujeme jako **interaktivní režim**. V něm uživatel komunikuje přímo se systémem, tj. s interpretem jazyka *Python*. Uživatel v něm zadává příkazy a systém na každý zadaný příkaz ihned zareaguje a provede jej.

  V dalším textu budu toto okno označovat jako **příkazové okno**. Příkazové okno může mít IDLE otevřené pouze jedno. Budete-li však chtít provádět několik věcí paralelně, můžete spustit několik instancí programu IDLE a v každé z nich pracovat nezávisle na těch druhých.

- Ostatní okna mohou pracovat pouze v **editačním režimu**, v němž uživatel edituje zdrojové či datové soubory, které pak uloží a v interaktivním režimu použije. Budu je proto označovat jako **editační okna**.

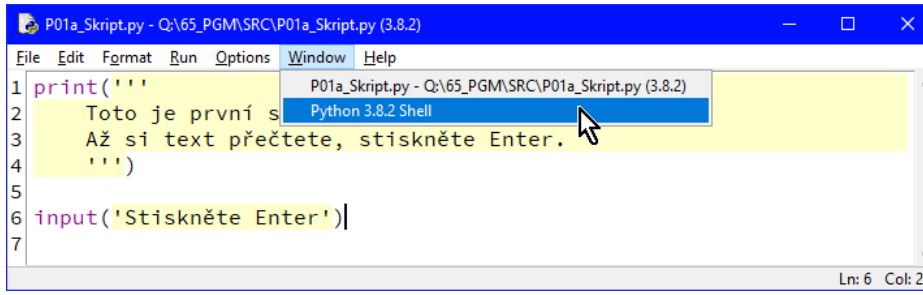
Režim aktuálního okna poznáte podle titulkové lišty. Příkazové okno v ní má uvedenu verzi programu následovanou slovem **Shell** (viz obrázek 1.2). Editační okno v ní uvádí název otevřeného souboru následovaný pomlčkou, úplnou cestou k danému souboru a v kulatých závorkách uvedenou verzi *Pythonu* (viz obrázek 1.3).



Obrázek 1.2:
Okno právě spuštěného vývojového prostředí IDLE

Režim otevřeného okna ovlivňuje i hlavní nabídku (nabídkovou lištu). Oba typy oken mají vlevo nabídku **File** a **Edit**. Příkazové okno pokračuje nabídkami **Shell** a **Debug**, zatímco editační ono má místo nich nabídky **Format** a **Run**. Poslední tři nabídky jsou opět shodné.

Mezi okny se přepínáte buď jejich aktivací na příkazovém panelu operačního systému (jednotlivá okna se zde vydávají za samostatné aplikace), anebo výběrem požadovaného cílového okna v nabídce **Window** (viz obrázek 1.3).



Obrázek 1.3:
Okno vývojového prostředí IDLE

Při prvním otevření bude mít okno takovou velikost, aby se v něm zobrazilo 40 řádků po 80 znacích. Zadáním příkazu **Zoom Height** nebo stiskem klávesové zkratky ALT+2 zvětšíte výšku okna na velikost displeje, opětovným zadáním tohoto příkazu je vrátíte do výchozí velikosti, a to i v případě, kdy mělo před zvětšením jinou velikost.

Dáváte-li přednost jiné výchozí velikosti okna, můžete ji nastavit v dialogovém okně **Settings** vyvolaném zadáním příkazu **Option → Configure IDLE**. Podrobněji již možnosti nastavení tohoto programu prozatím vysvětlovat nebudu; zájemci si jistě poradí sami, i když třeba s trochou experimentování. Jenom bych upozornil, že některá nastavení (mezi nimi právě velikost okna) se uplatní až po zavření a následném spuštění programu.

Příkazové okno

Nenastavíte-li ve výše zmíněném okně **Settings** něco jiného, tak se po spuštění aplikace otevře příkazové okno (viz obrázek 1.2), v němž se zobrazí úvodní řádek následovaný řádkem s výzvou k zadání příkazů, jejichž prostřednictvím získáte další informace, a řádkem s výzvou k zadání příkazu tvořenou trojicí znaků `>>>`.

Na obrázku 1.2 to sice vypadá, že se příkazové okno otevře se čtyřmi řádky textu, ale to je jenom proto, že se první řádek do okna nevešel, a tak jej program zalomil a pokračoval na druhém řádku. Pokud byste uchopili svislý okraj okna a okno roztáhli, přetékající konec prvního řádku by se vrátil na své místo.

Jak jsem již řekl, trojice většítek tvoří výzvu (anglicky prompt) k zadání příkazu. Když za ní něco napíšete a stisknete ENTER, interpret zadaný řádek zpracuje a zdá-li se mu, že jste zadali celý příkaz, vypíše na další řádek výsledek a za ním opět vypíše řádek s výzvou – viz výpis 1.1.

Výpis 1.1: *Zadání jednoduchého příkazu*

```
1 >>> 123 + 456
2 579
3 >>>
```

Restart interaktivního systému

V interaktivním režimu je občas výhodné smazat všechny výsledky pokusů a začít znovu. K tomu slouží příkaz **Restart Shell** v nabídce **Shell**, anebo stisk klávesové zkratky CTRL+F6.

Po zadání tohoto příkazu se za text vloží řádek s čárou z rovnítek přerušenou uprostřed textem `RESTART: Shell`. Od tohoto okamžiku se interpret chová stejně, jak kdybyste jej právě spustili.

Ukázky ve výpisech jsou většinou koncipovány tak, že žádný restart nevyžadují a mnohé na sebe dokonce navazují. Pokud by někdy byl potřeba restart, vždy na to výslovně upozorním.

Uložení záznamu seance

IDLE umožňuje uložit záznam aktuální seance do souboru. To se může hodit v případech, kdy se dostanete do potíží a potřebujete se s někým poradit, kde děláte chybu, nebo když naopak chcete někomu ukázat, jak má postupovat: co má zadávat a co může po svých zadáních od systému očekávat.

Záznam seance uložíte příkazem **Save** nebo **Save As...** v nabídce **File**. IDLE otevře dialogové okno a nabídne vám uložit soubor s příponou souboru `py`. Pro záznam seance bych ale tuto příponu určitě nepoužíval, protože je to přípona vyhrazená pro zdrojové soubory *Pythonu*, a tím záznam seance není, protože vedle vašich příkazů obsahuje i odpovědi systému a jeho následné výzvy (`>>>`) k zadání dalšího příkazu. Zvolte proto raději příponu `txt` nebo nějakou jinou vhodnou příponu, která nebude uživatele ani systém mást (já používám příponu `pyrec`).

Po uložení svého obsahu se příkazové okno s daným souborem sdruží a název souboru se zobrazí v titulkové liště okna. Při příštím zadání příkazu **Save** se soubor aktualizuje. Příkazem **Save As...** uložíte soubor pod novým jménem a ihned s ním okno sdružíte. Chcete-li průběžně ukládat mezistavy do jiných souborů, aniž byste měnili aktuální sdružení okna se souborem, použijte příkaz **Save Copy As...**.

Editační okno

Editační okno je sice primárně určeno k tvorbě zdrojových souborů modulů (o modulech začneme hovořit v kapitole *5 Moduly a práce s nimi* na straně 78), ale můžete v něm zobrazovat a/nebo upravovat libovolný textový soubor.

V nabídce **Options** můžete příkazem **Hide Line Numbers** vypnout číslování řádků. Příkaz se tím automaticky změní na **Show Line Numbers**, jehož zadáním číslování řádků opět zapnete.

Teď trochu předběhnu, protože bych zde chtěl uvést pohromadě všechny důležité příkazy, které by se vám mohly hodit. Prozradím vám proto, že v téže nabídce je také příkaz **Show Code Context**, po jehož zadání se nahoře objeví podšeděný řádek, v němž se budou zobrazovat hlavičky otevřených složených příkazů, které mají příkaz svého těla zobrazen v horním řádku editačního okna. Bude-li daný příkaz uvnitř vnořeného složeného příkazu, tak se počet řádků tohoto pole zvětší, aby se v něm mohly zobrazit hlavičky všech otevřených složených příkazů.

S prvním složeným příkazem se sice seznámíte až v podkapitole *3.1 Definice funkce* na straně 54, ale příkaz **Show Code Context** začne být poprvé užitečný, až budeme ve výpisu 5.4 na straně 85 analyzovat první vlastní modul. Pak vám jej připomenu.

Umístění editovaných souborů

Jak už jsem naznačil v pasáži *Spuštění* na straně 34, chcete-li otevřít zdrojový soubor, IDLE jej implicitně hledá v aktuální složce (adresáři), tj. ve složce, z níž jste prostředí spustili. Doporučuji vám proto zařídit (například vhodným nastavením zástupce), aby se prostředí spouštělo ve složce, v níž máte umístěné zdrojové soubory.

Já se nakonec rozhodl umístit text knihy i všechny její doprovodné programy na substituovaný disk `Q:`.[5] Pro doprovodné programy je na něm vyhrazena složka `65_PGM` a v ní jsou zdrojové soubory doprovodných programů umístěny ve složce `65_PYT`, soubory se záznamy seancí ve složce `65_REC` a soubory s kopiemi výpisů z knihy včetně čísel řádků ve složce `65_WRD`.

[5] Pracujete-li ve *Windows*, můžete popřemýšlet, jestli by se vám také nehodilo využít substituovaný disk. Podrobněji si o této možnosti můžete přečíst v příloze *A Konfigurace ve Windows* na straně 256.

Barevné zvýraznění textu

Jednou z výhod IDLE oproti používání příkazového řádku je i to, že barevně zvýrazňuje zadávaný i vypisovaný text, aby byl pro uživatele přehlednější a srozumitelnější. Způsob tohoto zvýraznění, tj. přiřazení barev popředí a pozadí jednotlivým druhům textu, si můžete nastavit. O toto zvýraznění vás v následujících výpisech doprovodných programů bohužel ochudím. Zadáte-li ale tyto texty do okna IDLE, budou barevně zvýrazněny.

Použité písmo

Při psaní kódu je nesmírně důležité zapisovat přesně to, co chcete počítači sdělit. Mnohá písma ale neumožňují zkontrolovat, zda jsem napsal přesně to, co jsem chtěl, protože zobrazují některé znaky stejně. Při programování je důležité používat písmo, které jednoznačně odliší nulu a velké O. Stejně tak je důležité odlišit znak velké I od malého L a jedničky.

Velmi často používané písmo Courier dobré rozlišení těchto znaků nenabízí – snadno se v něm zamění nejenom nula s O, ale také jednička s malým L (`O0 - Il1 # MOJE×M0JE; 321×32l`).

O něco lépe je na tom písmo Consolas definované v *Microsoftu*. To již nulu od O výrazně odlišuje, ale s odlišením jedničky od malého L máme stále problém (`O0 - Il1 # MOJE×M0JE; 321×32l`).

Z písem, která jsem měl možnost poznat, dopadlo zatím nejlépe písmo *Source Code Pro* (`O0 - Il1 # MOJE×M0JE; 321×32l`). To je navíc volně k dispozici, takže nemáte-li na svém počítači písmo s vhodnými vlastnostmi, můžete si je stáhnout a instalovat. Toto písmo je použito i v textu této knihy (tedy pokud nepoužíváte e-knihu ve formátu EPUB, který s tím má občas problémy).

Doporučuji vám proto, abyste si ve svých editorech, které používáte k psaní kódu, nastavili právě toto písmo. V programu IDLE toho dosáhnete zadáním příkazu **Options** → **Configure IDLE**. V následně otevřeném dialogovém okně pak na kartě **Fonts/Tabs** vyberete v seznamu **Font Face** požadované písmo a stiskem **OK** své zadání potvrdíte.

1.6 Shrnutí

V této kapitole jsme spustili pouze jednoduchý demonstrační skript uložený v souboru `m01a_script.py`.

Kapitola 2
Superzáklady

Co se v kapitole naučíte
Tato kapitola je částečně teoretická a částečně praktická. Seznámí vás s těmi nejzákladnějšími programovými konstrukcemi, které budeme používat: naučí vás pracovat s čísly a texty, vytvářet proměnné a volat funkce.

2.1 Počáteční mezery

V jazyku *Python* mají mezery na počátku řádku speciální význam. Když proto zadáváte jakýkoliv příkaz, hlídejte si, aby nezačínal zbytečnou mezerou. Když se vám do kódu taková nepatřičná mezera vloudí, jak tomu je ve výpisu 2.1 na řádku 3, systém vás na to ihned upozorní a neočekávanou mezeru označí.

Výpis 2.1: *Počáteční mezery a komentáře*

```
1 >>> 123 + 456 #Před zadáním příkazu nesmí být zbytečná mezera
2 579
3 >>>  432 - 10 #Tady jedna byla
4
5 SyntaxError: unexpected indent
6 >>>
```

Komentáře

I při maximální snaze o srozumitelnost zapsaného kódu se občas stane, že si u některé části nemůžete vzpomenout, proč jste ji do kódu zahrnuli, k čemu slouží a jak přesně se má používat. Jednou z cest, jak program učinit pochopitelnější i po letech, je doplnit jej vysvětlujícími komentáři, což jsou texty sloužící jako informace uživateli. Překladač i interpret je zcela ignorují – pro ně je komentář ekvivalentem mezery: kam se smí vložit mezera, tam se smí vložit komentář.

V *Pythonu* jsou komentáře uvozeny znakem # (mříž) a ukončeny koncem řádku. Chceme-li zadat komentář zabírající několik řádků, musíme na začátek každého řádku vložit znak #.

V interaktivním režimu, v němž nyní pracujeme, se komentáře většinou nepoužívají. Já je však budu občas vkládat, abych upozornil na to, čeho si máte na daném kódu všimnout, jak tomu bylo např. ve výpisu 2.1.

Výpis 2.1 současně ukazuje, jak budou formátovány výpisy naší komunikace v interaktivním režimu. Výpisy budou podbarveny. Naše zadání bude spolu s výzvou vysazeno tučně, odpovědi počítače obyčejně. Ohlásí-li však počítač chyby, bude toto hlášení vysazeno červeně a bez podbarvení. Bude-li v textu komentář, bude vysazen kurzivou a podbarven zeleně.

2.2 Celá čísla

Ve výpisu 2.1 jsme použili celá čísla. Ta nám spolu s texty, které probereme za chvíli, pro první etapu naší výuky postačí. Celé číslo můžeme zadat jako libovolnou posloupnost desítkových číslic nezačínající nulou. Celé číslo začínající nulou je v *Pythonu* syntaktická chyba – viz reakce na příkaz na řádku 1 ve výpisu 2.2.

Celá čísla můžete zadávat libovolně veliká. Potřebujeme-li napsat nějaké delší číslo, můžeme náš zápis zpřehlednit tím, že na vhodná místa vložíme znak podtržení (_), jak je tomu na řádku 3. Velikost jednotlivých skupin není definována (řádek 5). Podtržení ale nesmíme vložit na začátek ani na konec čísla, smí být pouze někde uprostřed.

Výpis 2.2: *Možnosti zápisu celých čísel*

```
1 >>> 012
2 SyntaxError: leading zeros in decimal integer literals are not permitted; use an
  0o prefix for octal integers
3 >>> 123_456_789
4 123456789
5 >>> 1_23_456_7890_12345
6 123456789012345
7 >>>
```

Je-li číslo záporné, vložíte před něj znak - (minus). Chcete-li výrazně odlišit kladná a záporná čísla, můžete před kladná čísla vložit znaménko +, ale to slouží opravdu jen k zpřehlednění zápisu. Teoreticky může být mezi znaménkem a příslušným číslem libovolný počet mezer (nebo komentářů), ale doporučoval bych vám této možnosti nevyužívat.

2.3 Reálná čísla

Po zvládnutí celých čísel nás v matematice učili používat čísla desetinná, o nichž nám posléze prozradili, že jsou podmnožinou čísel reálných. Termín *reálná čísla* se pro tato čísla používá i v programování.

V matematice nás učili oddělovat celou a desetinnou část čísla desetinnou čárkou. V drtivé většině programovacích jazyků (mezi nimi i v *Pythonu*) se pro tento účel používá desetinná tečka.

Reálné číslo můžeme zadat buď v tzv. *klasickém tvaru*, kdy v něm použijeme desetinnou tečku (např. `12.34`), anebo v exponentovém tvaru, při němž v zápisu čísla použijeme písmeno `e` nebo `E`, za něž zapíšeme číslo oznamující, kolikátou mocninou deseti se má vynásobit číslo zadané vlevo od znaku `e`, resp. `E`.

Jakmile je v zápisu čísla použita desetinná tečka nebo exponentový tvar, považuje je *Python* za desetinné, i kdyby se podle jeho hodnoty jednalo o číslo celé, tj. i kdyby mělo prázdnou desetinnou část. *Python* je vypíše s neprázdnou desetinnou částí, jak ve výpisu 2.3 ukazuje zobrazení čísel na řádcích 6 a 8.

Výpis 2.3: *Zadávání a zobrazování reálných čísel*

```
1 >>> 123_456_789_0_123_456_789_0_123_456_789.
2 1.2345678901234568e+28
3 >>> 123e-3
4 0.123
5 >>> 123.
6 123.0
7 >>> 123e3
8 123000.0
9 >>>
```

2.4 Textové řetězce – stringy

V programu často potřebujeme zadávat nejrůznější texty. Krátké texty, které zadáváme přímo v kódu, se oficiálně nazývají *textové řetězce*, ale programátoři je slangově označují jako *stringy*. Protože tento termín je mezi programátory hluboce zakořeněný, budu jej v dalším textu používat.

Pro text, který můžeme zadat na jednom řádku, poskytuje *Python* dva možné způsoby zápisu: zadávaný text musí být vždy uzavřen mezi apostrofy, nebo mezi uvozovky (viz výpis 2.4). Jejich význam se nijak neliší, jenom musíte dbát na to, abyste hraniční znak použitý na počátku použili i na konci.

Volba hraničního znaku může být ovlivněna tím, zda potřebujete některý z nich použít uprostřed textu – viz příkazy na řádcích 5 a 7 ve výpisu 2.4.

Výpis 2.4: *Zápis jednořádkového textu*

```
1 >>> "Text v uvozovkách"
2 'Text v uvozovkách'
3 >>> 'Text v apostrofech'
4 'Text v apostrofech'
5 >>> "Potřebuji 'apostrof'"
6 "Potřebuji 'apostrof'"
7 >>> 'Potřebuji "uvozovky"'
8 'Potřebuji "uvozovky"'
9 >>>
```

Python dává při zobrazení stringů přednost apostrofům. Uvozovky použije k ohraničení textu pouze tehdy, je-li ve vypisovaném textu použit apostrof, jak učinil na řádku 6.

Znak # ve stringu

V pasáži *Komentáře* na straně 39 jsem vám říkal, že část řádku za znakem # představuje komentář, což je text, který slouží uživateli a překladač jej ignoruje. To ale neplatí pro situaci, kdy se znak # objeví ve stringu. Uvnitř stringu tento znak žádný komentář neuvozuje a představuje sám sebe – viz ukázka ve výpisu 2.5. Nezařaďte se mezi ty, které znak # ve stringu zmátl.

Výpis 2.5: *Znak # ve stringu představuje sám sebe*

```
1 >>> 'Ve stringu představuje znak # sám za sebe.'
2 'Ve stringu představuje znak # sám za sebe.'
3 >>>
```

Víceřádkové stringy

Jak už bylo řečeno, takto označený text se musí vejít na jeden řádek. Chcete-li zadat několikařádkový text a rozhodnete se pokračovat na následujícím řádku před ukončením zadávaného textu uvozujícím znakem, systém ohlásí chybu (viz reakce na řádku 2 ve výpisu 2.6).

Jak ale výpis 2.6 naznačuje, ohraničíte-li zadávaný text trojicemi apostrofů či uvozovek, *Python* bude konec řádku považovat za součást zadávaného textu, dokud nezadáte uzavírající trojici. Jak se můžete přesvědčit na řádcích 5 a 10, při výpisu zadaných textů pak interpret na místech, kde končí jednotlivé řádky zadaného textu, vypíše dvojici znaků \n označující právě konec řádku. Tento znak můžete použít k označení konce řádku i vy, ale to si předvedeme v příští kapitole, až budete umět zobrazit, že jste opravdu zadali několikařádkový text.

String zadaný na řádcích 6-9 ukazuje ještě několik dalších zásad, které byste mohli využít:

Výpis 2.6: *Zápis víceřádkového textu*

```
>>> "Neukončený text
SyntaxError: EOL while scanning string literal
>>> """několikařádkový
text"""
'několikařádkový\ntext'
>>> '''
    Jiný způsob\
    'zápisu'
    '''
"\n    Jiný způsob    'zápisu'\n    "
>>> '''Musí být 'odděleny''''
SyntaxError: EOL while scanning string literal
>>> ''''Apostrof'.'''
"'Apostrof'."
>>> ""     #Dvojice uvozovek či apostrofů označuje prázdný string
''
>>> "Bezprostředně 'sousedící' stringy"  'Python "automaticky" sloučí'
'Bezprostředně \'sousedící\' stringyPython "automaticky" sloučí'
>>>
```

- Do pokračovacích řádků se započítávají i úvodní mezery.

  V následující kapitole si vysvětlíme, že správná velikost odsazení počátku řádku je v *Pythonu* důležitá. Počáteční mezery v pokračovacích řádcích víceřádkových stringů se však do tohoto pravidla nezapočítávají (viz např. string na řádcích 26–29 ve výpisu 5.4 na straně 85).

- Ukončíte-li řádek zpětným lomítkem jako na řádku 7, tak se následující konec řádku do výsledného stringu nezařadí.

- Uvnitř zadávaného stringu můžete použít stejný znak, jaký se vyskytuje v ohraničujících trojicích (na řádku 8 jsou např. použity apostrofy). Tyto znaky se jenom nesmějí nacházet v bezprostředním sousedství uzavírací trojice.

  Apostrof na konci řádku 8 je od uzavírající trojice oddělen koncem řádku a několika mezerami. Text zadávaný na řádku 11 však vyvolá chybu, protože je závěrečný apostrof nalepen na uzavírající trojici apostrofů. Jak ale ukazuje text na řádku 13, přilepení na otevírající trojici nevadí. Důležité je, aby mezi případným apostrofem a uzavírající trojicí apostrofů byl nějaký jiný znak. Jak si jistě domyslíte, totéž platí i pro uvozovky.

Reakce na příkaz na řádku 15 ukazuje, že *Python* je schopen pracovat i s *prázdným stringem*, tj. se stringem, který neobsahuje žádný znak.

Zadání na řádku 17 má ukázat, že stringy, které spolu bezprostředně sousedí, takže se mezi nimi nacházejí jen bílé znaky, *Python* automaticky sloučí do jediného. V našem případě to znamenalo také to, že při zobrazování bylo nutné zadat apostrofy pomocí escape sekvence.

Escape sekvence

Na řádku 16 jsou ve výpisu 2.6 zadány za sebou dva stringy, které odděluje jen pár mezer. Sousedící stringy, mezi nimiž se nacházejí pouze bílé znaky, *Python* automaticky sloučí do jediného stringu, a jak vidíte na řádku 17, spojí je, aniž by mezi ně cokoliv vložil (např. mezeru).

V našem případě se však objevil problém. V prvním stringu se vyskytovaly apostrofy, což nevadilo, protože string byl ohraničen uvozovkami. V druhém se zase vyskytovaly uvozovky, což opět nevadilo, protože string byl ohraničen apostrofy. Po sloučení však bylo třeba zvolit ohraničující znak (jak víme, *Python* dává přednost apostrofům), a v tu chvíli se tento znak stal uvnitř stringu zakázaným.

To byla ukázka situace, kdy zadávaný text obsahuje znak, který neumíte zadat z klávesnice, nebo se jeho přímé zadání z nějakého důvodu nehodí. Ve výpisu 2.6 byl ve strinzích zadaných na řádcích 3–4 a 6–8 takovým znakem znak konce řádku, ve stringu zadaném na řádku 16 byl tímto znakem apostrof.

Situace jako tato je poměrně častá, a proto jsou pro takovéto znaky v programovacích jazycích definovány tzv. *únikové posloupnosti*, které však programátoři nenazvou jinak než slangovým výrazem *escape sekvence* (čtěte iskejp sekvence). Jsou to skupiny znaků začínající znakem \ (zpětné lomítko, anglicky back slash). *Python* definuje řadu escape sekvencí, z nichž si v této knize vystačíme s následujícími čtyřmi:

`\\` Zpětné lomítko – vzhledem k tomu, že je použito jako metaznak uvozující tyto posloupnosti, tak chcete-li je vložit do stringu, musíte je zadávat zdvojené.

`\'` Apostrof pro případ, kdy jej do daného stringu není možné vložit přímo.

`\"` Uvozovky pro případ, kdy je do daného stringu není možné vložit přímo.

`\n` Konec řádku, nový řádek (New Line).

Bílé znaky

Na počátku předchozí pasáže jsem hovořil o strinzích, mezi nimiž se nacházejí pouze bílé znaky. Termínem *bílý znak* označujeme některý ze znaků, které se viditelně nezobrazují. Patří mezi ně mezera, vodorovný a svislý tabulátor, konec řádku a konec stránky.

2.5 Proměnné

Termínem *proměnná* (anglicky *variable*) označujeme oblast v paměti vyhrazenou pro uložení hodnoty, kterou si chceme zapamatovat. V programech používáme značné množství nejrůznějších proměnných. Proměnnou proto musíme pojmenovat, abychom později, až budeme chtít uloženou hodnotu použít, mohli interpretu jednoznačně sdělit, kde je hodnota uložena. Kdykoliv budete chtít uloženou hodnotu použít, zadáte v kódu název proměnné a interpret zařídí, aby byla použita hodnota uložená do této proměnné.

Identifikátor

Pro názvy proměnných se používá termín *identifikátory*, protože slouží k jejich identifikaci. V *Pythonu* platí pro identifikátory následující pravidla:

- Smějí obsahovat písmena, číslice a podtržítka.
- Nesmějí začínat číslicí.
- Rozlišuje se v nich velikost písmen. Identifikátory `ahoj`, `Ahoj` a `AHOJ` tak obecně představují různé objekty.
- Nesmějí být shodné s klíčovým slovem – viz tabulku 2.1.

Tabulka 2.1: *Klíčová slova jazyka Python*

`False`	`None`	`True`	`and`	`as`	`assert`	`async`
`await`	`break`	`class`	`continue`	`def`	`del`	`elif`
`else`	`except`	`finally`	`for`	`from`	`global`	`if`
`import`	`in`	`is`	`lambda`	`nonlocal`	`not`	`or`
`pass`	`raise`	`return`	`try`	`while`	`with`	`yield`

Název má co nejlépe naznačovat účel proměnné. Podle konvence se proměnné pojmenovávají malými písmeny, přičemž obsahuje-li název více slov, tak se mezi jednotlivá slova vkládá znak podtržení – např. `proměnná_s_poněkud_delším_názvem`.

Definujeme-li proměnnou, která je ve skutečnosti konstantou, zapisuje se její název velkými písmeny – např. `KONSTANTA_S_DLOUHÝM_NÁZVEM`.

Jazyk identifikátorů

Jak už jsem naznačil v úvodu, podle konvence by identifikátory měly vycházet z angličtiny, abyste se o svém programu mohli bez problému bavit i s cizinci. Přeložit komentáře a stringy může Google, ale identifikátory obsahují často tolik skrytých vzájemných vazeb, že je lepší se o jejich překlad raději nepokoušet.

Jak už jsem naznačil v úvodu, ve svých knihách jsem nakonec přistoupil ke kompromisu: identifikátory v AHA-příkladech budou české, protože hlavním účelem těchto příkladů je co nejlépe vysvětlit probíranou konstrukci. Identifikátory ve vytvářené aplikaci budou anglické.

Definice a použití proměnné, přiřazovací příkaz

Chceme-li pro potřeby našeho programu zavést nějakou proměnnou, do níž bychom mohli ukládat hodnoty, které si program potřebuje zapamatovat, vymyslíme pro ni vhodný název, zapíšeme jej, za něj napíšeme rovnítko a za něj pak přiřazovanou počáteční hodnotu, jak je tomu např. na řádku 1 ve výpisu 2.7. Zápis

```
identifikátor = hodnota
```

označujeme jako **přiřazovací příkaz**. Tedy přesněji je to speciální podoba přiřazovacího příkazu, jehož obecnou podobu bychom mohli popsat

```
cíl = výraz
```

kde označení `cíl` označuje předpis, podle nějž poznáme, kam se bude ukládat hodnota výrazu zadaného vpravo od rovnítka.

Zadáme-li v interaktivním režimu jako příkaz samotný název proměnné (řádek 2), *Python* vypíše její hodnotu (řádek 3). Použijeme-li však název proměnné, do které jsme ještě žádnou hodnotu neuložili (řádek 4), *Python* ohlásí chybu, protože z jeho pohledu tato proměnná neexistuje (na řádku 8 nám oznamuje, že proměnná doposud není definovaná). **Pro *Python* totiž začne proměnná existovat až po své inicializaci**, tj. až poté, co do ní uložíme počáteční hodnotu. Jakmile do proměnné s daným názvem uložíme nějakou hodnotu, tak ji tím současně vytvoříme (řádek 9), a *Python* začne být ochoten s touto proměnnou pracovat.

V příkazu na řádku 10 si všimněte, že proměnná `a` vystupuje na obou stranách přiřazovacího příkazu. Nejprve se ve výrazu vpravo použije její hodnota, aby se sečetla s hodnotou proměnné `x`, a výsledek tohoto součtu se uloží jako nová hodnota proměnné `a`. Systém nám v reakci na příkaz na řádku 11 tuto novou hodnotu zobrazí na řádku 12.

Výpis 2.7: *Definice a použití proměnných*

```
>>> a = 123      #Vytvářím a současně inicializuji novou proměnnou
>>> a            #Nechávám zobrazit její hodnotu
123
>>> a = a + x    #Proměnná x ještě není vytvořena
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    a = a + x   #Proměnná x ještě není vytvořena
NameError: name 'x' is not defined
>>> x = 456      #Vytvářím a současně inicializuji proměnnou x
>>> a = a + x    #Proměnná x je již použitelná
>>> a            #Obsah proměnné a se změnil
579
>>>
```

Nebezpečné změny hodnot

Jednou z vlastností *Pythonu*, pro kterou jej jedni chválí a druzí zatracují, je možnost ukládat do jedné proměnné různé typy hodnot. To sice umožňuje zkušeným programátorům provádět některé zajímavé obraty, ale na druhou stranu to nepozorným programátorům umožňuje zanášet do programu velmi záludné chyby, které se pak občas špatně hledají.

Ve výpisu 2.7 jsme měli v proměnných `a` a `x` uložená čísla. Výpis 2.8 ukazuje, co se stane, když programátor uloží do proměnné `a` string, zapomene na to a bude ji chtít opět sečíst s proměnnou `x`.

Když do proměnné b uložíme také string, jak je ukázáno na řádku 7, tak součet obou proměnných (řádek 8) proběhne bez problémů a vznikne nový string, který je spojením oněch sčítanců. Je třeba jen mít na paměti, že *Python* oba stringy pouze spojí a žádnou mezeru ani nic jiného mezi ně nevkládá (viz řádek 9). Budete-li proto chtít spojované stringy nějak oddělit, musíte mezi ně potřebný oddělovač explicitně vložit, jak je např. naznačeno na řádku 10.

Doposud jsem z lenosti používal pouze jednoznakové názvy proměnných. Na řádku 12 jsem to napravil a definoval proměnnou mezera, do níž jsem uložil jednoznakový string obsahující mezeru. Tuto proměnnou jsem pak použil na řádku 13. Příkaz je hned přehlednější.

Výpis 2.8: *Nebezpečné změny hodnot*

```
>>> a = "Sto dvacet tři"  #Opět měním hodnotu proměnné a
>>> x = a + x             #String není možné sčítat s číslem
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    x = a + x             #String není možné sčítat s číslem
TypeError: can only concatenate str (not "int") to str
>>> b = 'miliónů'     #Vytvářím a inicializuji proměnnou b
>>> a + b             #Dva stringy již sčítat mohu
'Sto dvacet třimiliónů'
>>> a + ' ' + b
'Sto dvacet tři miliónů'
>>> mezera = ' '
>>> a + mezera + b
'Sto dvacet tři miliónů'
>>>
```

2.6 Literály

Přímá zadání hodnoty, která jsme probírali v této kapitole, se označují jako *literály*. Obecně bychom mohli říci, že *literál je konstanta nazvaná svojí hodnotou*. Napíšete-li v programu např. 7 nebo "Dobrý den!", zapsali jste literál.

Každý jazyk definuje, které druhy dat je možné zapsat prostřednictvím literálů. Všechny jazyky, které znám, umožňují zapsat prostřednictvím literálů čísla a krátké texty, jež lze zadat na jeden řádek kódu. Některé jazyky podporují zadávání víceřádkových textů a dalších druhů dat.

Z pohledu na to, co vše lze zadat jako literál, patří *Python* mezi ty bohatší. Postupně se budeme seznamovat s tím, jak lze prostřednictvím literálů zadávat další druhy dat.

2.7 Volání funkcí

Termín *funkce* jistě znáte z matematiky. Napíšete-li ve výrazu například *sin(x)*, předpokládáte, že při výpočtu se tento text nahradí hodnotou funkce *sin* v bodě `x`. V programování je to podobné. Jediným rozdílem oproti tomu, co znáte z matematiky, je to, že funkce definované v programu mohou vyžadovat různý počet argumentů a některé trvají na tom, že žádný argument nechtějí.

Funkce se volá tak, že zadáme jméno proměnné, v níž je uložen odkaz na příslušný kód, za toto jméno napíšeme kulaté závorky a do nich příslušný argument. Nezadáváme-li žádný argument, budou závorky prázdné. Zadáváme-li více argumentů, oddělují se čárkami.

Zavoláme-li v programu funkci, funkce převezme zadané argumenty, spočte požadovaný výsledek a ten vrátí jako svoji funkční hodnotu. Tímto výsledkem (touto hodnotou) se v daném místě programu nahradí volání dané funkce. Pojďme si to vyzkoušet.

Příklady funkcí

Ve výpisu 2.7 jsme na řádku 2 ověřili, že číslo není možné sčítat se stringem. V knihovně ale existuje funkce `str()`, která svůj argument převede na string, a ten vrátí. Výsledek si můžete prohlédnout ve výpisu 2.9. Na řádku 1 je hodnota `x` převedena na string, který pak lze sčítat s dalšími stringy, takže příkaz bez problémů projde.

Druhou možností vytištění požadovaného textu je využití funkce `print()`. Té můžete zadat libovolný počet argumentů. Ona každý převede na string a vytiskne. Mezi každé dva tištěné stringy navíc vloží mezeru. Příkaz tisknoucí náš text je na řádku 3.

Další zajímavou funkcí je funkce `len()`. Té zadáte v argumentu objekt, který může mít několik prvků, a ona vrátí počet jeho prvků. Prvky stringu jsou jednotlivé znaky.

Výpis 2.9: *Použití funkcí*

```
>>> str(x) + mezera + b
'456 miliónů'
>>> print(x, b)
456 miliónů
>>> print("Proměnná b má", len(b), "znaků:", b)
Proměnná b má 7 znaků: miliónů
>>> print('Prázdný string má', len(''), 'znaků')
Prázdný string má 0 znaků
>>> print('První řádek\n Druhý řádek\n  Třetí řádek')
První řádek
 Druhý řádek
  Třetí řádek
>>> print('Při tisku se jak \'apostrofy\', tak \"uvozovky\" zobrazují normálně')
Při tisku se jak 'apostrofy', tak "uvozovky" zobrazují normálně
>>>
```

Na řádku 5 je volání funkce print(), které jsou předány čtyři argumenty: text, číslo vrácené funkcí len(), další text a hodnota uložená v proměnné b. Výsledek si můžete prohlédnout na řádku 6. Na řádku 7 je pak obdobný příkaz zobrazující počet znaků v prázdném stringu.

Ve výpisu 2.6 jsme se seznamovali se stringy, které mohou mít několik řádků. Tehdy jsem vám říkal, že znak pro ukončení řádku se zadává jako dvojice znaků \n. Narazí-li na tento znak v tištěném stringu funkce print(), tak opravdu odřádkuje. Vše předvádí příkaz na řádku 9.

Na řádku 13 je pak předvedeno, že escape sekvence se uplatní pouze při zadávání stringů, ale při tisku jsou již jimi reprezentované znaky zobrazeny normálně.

Parametr versus argument

V matematice se hovoří o parametrech funkcí a možná jste tento termín slyšeli i v souvislosti s programováním. Já jsem ale v předchozích odstavcích používal termín *argument*. Důvod je ten, že v programování je při práci s funkcemi potřeba rozlišovat dva druhy objektů: termínem **parametr** označujeme **proměnnou**, tj. schránku na hodnotu, s níž má funkce pracovat. Termínem **argument** pak označujeme **výraz**, jehož vyhodnocením získáme hodnotu, kterou do této proměnné uložíme.

2.8 Hodnota None

Občas je třeba definovat proměnnou, aby se s ní dalo pracovat, ale v okamžiku její definice ještě není známa její počáteční hodnota – ta se musí teprve spočítat. V některých jazycích se do takové proměnné vloží nula nebo nějaký její ekvivalent. *Python* definuje pro tyto účely konstantu None. Potřebujete-li zdůraznit, že daná proměnná nemá ještě platnou hodnotu, můžete ji inicializovat touto konstantou, jak je demonstrováno na řádku 1 ve výpisu 2.10.

Reakce na příkaz na řádku 2 dokazuje, že zadáte-li název proměnné obsahující hodnotu None, systém nic nevytiskne. Chcete-li se ubezpečit o obsahu proměnné, vytiskněte ji prostřednictvím funkce print(). Jak ukazují řádky 3–4, ta tuto hodnotu vytiskne.

Funkce print() je příkladem funkce, která má něco provést, ale nikdo po ní nechce, aby vracela nějakou funkční hodnotu. Protože však všechny funkce musejí něco vracet, tak ty, po nichž nikdo funkční hodnotu nechce, vracejí hodnotu None. To ukazuje i příkaz na řádku 5, kde prvním a třetím argumentem funkce print() je jiný tisk. Výpis současně ukazuje, že nejprve vytisknou své hodnoty funkce zadané jako argumenty, a teprve pak se dostane ke slovu funkce, jíž jsou tyto argumenty předány. Ta na řádku 8 vytiskne nejprve None představující první argument (návratovou hodnotu prvního volání funkce print()), pak zadaný string a za ním další None představující návratovou hodnotu druhého volání funkce print().

Výpis 2.10: *Hodnota* `None` *a její zobrazení*

```
1 >>> z = None      #Vytvořím proměnnou z, ale současně říkám, že v ní nic není
2 >>> z             #Je-li hodnotou proměnné None, interpret ji netiskne
3 >>> print(z)      #Funkce print ale tiskne i hodnotu None
4 None
5 >>> print(print('Tisknu 1'), "Funkce print() vrací:", print('Tisknu 2'))
6 Tisknu 1
7 Tisknu 2
8 None Funkce print() vrací: None
9 >>>
```

Podrobnosti o volání funkcí

Při volání funkcí se často volají jiné funkce při přípravě jejich argumentů. Studenti pak občas tápou v tom, kdy se která akce provede. Základní pravidlo jsem se snažil demonstrovat ve výpisu 2.10 příkazem na řádku 5. Zde je volána funkce `print()` se třemi argumenty. Výsledek na řádcích 6–8 dokazuje, že argumenty se vyhodnocují postupně zleva doprava a dokud nejsou hodnoty všech argumentů připraveny, tak se funkce nezavolá.

1. Prvním argumentem je hodnota vracená funkcí `print()` tisknoucí string `'Tisknu 1'`. Abychom ji získali, musíme funkci zavolat a funkce vytiskne text na řádku 6.
2. Druhým argumentem je string `"Funkce print() vrací:"`. Tato hodnota je známá, stačí ji pouze předat.
3. Třetím argumentem je hodnota vracená funkcí `print()` tisknoucí string `'Tisknu 2'`. Opět je třeba nejprve vytisknout příslušný text (řádek 7) a obdrženou návratovou hodnotu předat volané funkci jako třetí argument.
4. Teprve nyní jsou argumenty připraveny a můžeme funkci zavolat. Funkce vytiskne požadovaný text, z nějž se dozvíme, že obě předchozí volání funkce `print()` vrátila hodnotu `None`.

2.9 Zadání údajů z klávesnice

Prozatím jsme hodnoty konstant a proměnných pouze zobrazovali. Občas je ale také potřeba tyto hodnoty zjišťovat. Jednou z možností je využít funkce `input()`, která vypíše svůj argument (nezadáte-li jej, nevypíše nic) a očekává, co uživatel zadá. Uživatel zapíše své zadání, stiskne ENTER a funkce vrátí jako svoji návratovou hodnotu zadaný text.

Stručnou ukázku použití si můžete prohlédnout ve výpisu 2.11. V něm je na řádku 1 zavolána funkce `input()` s výzvou uživateli. Na řádku 2 funkce tuto výzvu zobrazila a počkala na odpověď uživatele. Text zadaný uživatelem pak vrátila jako svoji funkční hodnotu, kterou systém na řádku 3 vypsal.

U této funkce je třeba si pamatovat, že uživatelovo zadání se bude psát hned za výzvu. Budete-li proto chtít mít z estetických důvodů mezi výzvou a odpovědí mezeru, musíte ji zadat jako součást argumentu, jak je to ostatně ve výpisu 2.11.

Výpis 2.11: *Ukázka použití funkce `input()` a implicitní proměnné*

```
1 >>> input('Jak vás mám oslovovat? ')
2 Jak vás mám oslovovat? Vaše blahorodí
3 'Vaše blahorodí'
4 >>> print('Dobrý den,', _)
5 Dobrý den, Vaše blahorodí
6 >>>
```

2.10 Implicitní proměnná _

Zajímavý je ale i příkaz tisku na řádku 4. V něm je zavolána funkce `print()` se dvěma argumenty: prvním je string a druhým je proměnná s názvem `_`, o které jsme ještě nehovořili.

Do této proměnné systém uloží výsledek příkazu pokaždé, když příkaz zadaný v interaktivním režimu vrátí jinou hodnotu než `None`, a my ji neuložíme do nějaké proměnné. (Pokud proměnná `_` ještě neexistuje, tak ji vytvoří.)

V řadě případů bychom totiž chtěli se spočtenou hodnotou dále pracovat, načež si uvědomíme, že jsme si ji zapomněli uložit. Nevadí, nemusíme zadávat předchozí příkaz znovu, ale můžeme použít zmíněnou implicitní proměnnou.

2.11 Základní aritmetické operace

Než ukončíme kapitolu o superzákladech programování v jazyce *Python*, seznámím vás ještě se základními aritmetickými operacemi. V *Pythonu* máme k dispozici následující operace:

- `+` **Sčítání**. Jak již víte, vedle čísel můžeme sčítat i stringy.
- `-` **Odčítání**. Stringy se odčítat nedají.
- `*` **Násobení**. Násobit můžeme i číslo a string. Výsledkem je zadaný počet krát zopakovaný string (`3 * '21' == '212121'; '12' * 3 == '121212'`).
- `/` **Dělení**, jehož výsledkem je reálné číslo (`6 / 4 == 1.5`).
- `//` **Celočíselné dělení.** Výsledkem je celá část podílu (`6 // 4 == 1`).
- `%` **Zbytek po dělení** (`6 % 4 == 2`).
- `**` **Mocnění** (`6 ** 4 == 36 ** 2 == 1296`).

2.12 Formátovací stringy – f-stringy

V podkapitole 2.4 *Textové řetězce – stringy* na straně 41 jsme se naučili zadávat stringy prostřednictvím literálů s jednoduchými či trojitými apostrofy nebo uvozovkami. Vedle těchto základních literálů zavádí *Python* ještě několik speciálních literálů, v nichž se před uvozovky či apostrofy zadává prefix indikující zvláštní zacházení s daným stringem.

Jednou z možností je zadat před úvodní ohraničení (apostrof či uvozovky) znak `f` nebo `F`. Tento prefix zadává tzv. *formátovaný stringový literál* (*formatted string literal*) označovaný často jako *f-string*. Může obsahovat tzv. *nahrazovací pole* (*replacement fields*) ohraničená složenými závorkami `{}` a obsahující výraz doplněný o formátovací instrukce. Každé nahrazovací pole bude za chodu nahrazeno hodnotou příslušného výrazu naformátovanou podle zadaných instrukcí.

My zde nebudeme probírat, jak přesně zadat požadovaný formát – zájemce odkážu na dokumentaci *Pythonu* nebo na příručku [11]. Využijeme pouze toho, že takto si lze jednoduše objednat zobrazení požadovaných hodnot uvnitř stringu. Prozatím se spokojíme s tím, že zadáme-li ve formátovaném stringu výraz ve složených závorkách, bude jeho hodnota převedena na string a vložena do okolního stringu včetně případných úvodních či závěrečných mezer v daných závorkách.

Zadáme-li ve složených závorkách jako poslední nemezerový znak rovnítko, *Python* opíše do výsledného stringu obsah pole včetně onoho rovnítka, pak vyhodnotí výraz před rovnítkem a jeho hodnotu zapíše do stringu za ono rovnítko.

Ve výpisu 2.12 je zobrazeno využití f-stringů při zobrazování výsledků operací.

Výpis 2.12: *Použití formátovaných stringů*

```
>>> print(f'''Sčítání:{  6 + 4 = }, {'AB'+"CD"=};   Odčítání: {4-6=};
Násobení: {6 * 4 = }, {3*'21' = },{ '21' * 3 = };
Dělení:   {6/4=};  Celočíselné: {6//4=};  Zbytek: {6%4=};
Mocnění:  {6**4 = },  {36**2 = }''')
Sčítání:  6 + 4 = 10, 'AB'+"CD"='ABCD';   Odčítání: 4-6=-2;
Násobení: 6 * 4 = 24, 3*'21' = '212121', '21' * 3 = '212121';
Dělení:   6/4=1.5;  Celočíselné: 6//4=1;  Zbytek: 6%4=2;
Mocnění:  6**4 = 1296,  36**2 = 1296
>>>
```

2.13 Více příkazů na řádku

Někdy se nám hodí zapsat více příkazů na jeden řádek. V takovém případě se jednotlivé příkazy na řádku oddělují středníkem. Ve výpisu 2.13 najdete malou ukázku.

Výpis 2.13: *Více příkazů na jednom řádku*

```
>>> x = 'Přiřazovaná hodnota';   x
'Přiřazovaná hodnota'
>>> a = 1; b = 10; c = 100; print(a, b, c)
1 10 100
>>>
```

2.14 Shrnutí

Záznam komunikace s interpretem probíhající v této kapitole najdete v trojici souborů nazvaných `m02__fundamentals`.

Kapitola 3
Začínáme programovat

Co se v kapitole naučíte
Tato kapitola vás naučí definovat vlastní funkce. Nejprve vysvětlí, co to funkce je, a pak postupně probere definicí funkcí s návratovou hodnotou a funkcí s parametry včetně parametrů s implicitní hodnotou. Na závěr probere význam doposud nepoužívaných parametrů funkce `print()`.

3.1 Definice funkce

Až do příchodu objektově orientovaného programování byly základním programátorským nástrojem *podprogramy*, což jsou části kódu, které se jednou naprogramují a pak se mohou na mnoha místech programu použít – *zavolat*, aby provedly to, pro co byly naprogramovány. Umožňují tak dodržovat jednu z nejdůležitějších programátorských zásad, která říká, že se v programu nemá opakovat stejný (nebo skoro stejný) kód. V *Pythonu* se tyto podprogramy označují jako *funkce*.

V předchozí kapitole jsme se naučili funkce volat, nyní se naučíme definovat funkce vlastní. Definice funkce je tzv. ***složený příkaz***, který **vždy sestává z hlavičky ukončené dvojtečkou a z těla**.

- **Hlavičku funkce** tvoří klíčové slovo `def` následované názvem funkce (ten je současně názvem proměnné, do níž bude odkaz na objekt funkce uložen), závorkami se seznamem čárkami oddělených názvů parametrů (nemá-li funkce parametry, budou závorky prázdné) a závěrečnou dvojtečkou.

  Pro názvy funkcí platí stejné konvence jako pro názvy běžných proměnných: zapisují se malými písmeny a slova v několikaslovných názvech se oddělují podtržítky, např. `funkce_s_dlouhým_názvem`.

- Za hlavičkou následuje **tělo** tvořené posloupností příkazů, které budou po zavolání funkce postupně prováděny. Před prvním výkonným příkazem by měl být dokumentační komentář.

  Typicky se příkazy těla zapisují na dalších řádcích, přičemž syntaxe jazyka vyžaduje, aby všechny příkazy těla byly oproti hlavičce odsazené, a to všechny stejně. Jakákoliv odchylka je považována za syntaktickou chybu.

Obsahuje-li tělo složeného příkazu pouze jednoduché příkazy, je teoreticky možné je napsat na stejný řádek jako hlavičku. Tato možnost se však používá spíše výjimečně a navíc většinou jen tehdy, jedná-li se o tělo tvořené jediným příkazem. V této učebnici se s takovýmto zápisem setkáte v pasáži *Alternativní postup* na straně 229.

Nejdůležitější odchylkou definice funkce od příkazů, které jsme prováděli doposud, je to, že definice funkce zadané příkazy hned nespustí, ale pouze uloží. Provádět se budou až v okamžiku, kdy danou funkci někdo zavolá.

Funkce je objekt, na nějž odkazuje proměnná

Definici funkce můžeme považovat za zvláštní příkaz přiřazení, při jehož vykonávání se provede několik věcí:

1. Není-li ještě definována proměnná s názvem funkce, vytvoří se.
2. V paměti se alokuje (vyhradí) místo pro objekt reprezentující danou funkci. Součástí tohoto objektu bude i kód definovaný v těle funkce.
3. Do proměnné se uloží odkaz na vytvořený objekt.

V pasáži *Nebezpečné změny hodnot* na straně 46 jsem upozorňoval na nebezpečí uložení hodnoty do nesprávné proměnné. Uvědomte si, že toto platí i pro proměnné odkazující na funkci.

Proměnná odkazující na funkci je obyčejná proměnná obdobné té, která odkazuje na string. Zadáte-li pouhý název této proměnné, systém vypíše jeho hodnotu, např.:

```
>>> pozdrav
<function pozdrav at 0x000001D65E579820>
>>>
```

Funkce se zavolá pouze v případě, kdy za název proměnné zapíšete kulaté závorky se seznamem předávaných argumentů. Musíte se proto hlídat, abyste do proměnné odkazující na funkci nevložili jinou hodnotu, protože pak byste o odkaz na danou funkci přišli a už byste ji nemohli zavolat, leda byste ji znovu definovali.

Dokumentační komentář

Na počátek těla funkce je vhodné vložit string, v němž popíšete účel dané funkce a způsob jejího používání. Tento string označujeme jako dokumentační komentář, případně jako dokumentační string (v angličtině se ujala zkratka *docstring*). Aby se tento string stal dokumentačním komentářem, musí být v těle funkce uveden jako první hned za hlavičkou a před prvním příkazem těla.

Jako docstring lze použít libovolný stringový literál, nicméně konvence doporučují použít string ohraničený trojicemi uvozovek, a to i v případě, že je daný komentář jednořádkový – např. komentář na řádku 2 ve výpisu 3.1.

Získání nápovědy – dokumentace

Funkce si dokumentační komentář zapamatuje a můžete se jí na něj kdykoliv zeptat. V interaktivním režimu se na něj většinou ptáme tak, že zavoláme systémovou funkci `help()`, které zadáme název funkce (případně jiného dokumentovaného objektu) jako argument – viz příkaz na řádku 9 ve výpisu 3.1. Druhou možností je napsat název následovaný tečkou a identifikátorem `__doc__` (viz příkaz na řádku 15).

Definice funkce je obyčejný složený příkaz

Na závěr tohoto teoretického úvodu bych chtěl připomenout, že definice funkce je obyčejný složený příkaz (tj. příkaz s hlavičkou a tělem), který můžeme použít na místě, kde se smí použít příkaz, tedy i v definici jiné funkce.

Nyní vás nebudu rozptylovat samoúčelným AHA-příkladem, ale netrpěliví zájemci si mohou nalistovat výpis 11.4 na straně 157, kde je takováto funkce definována. Podrobněji se o interních funkcích dozvíte v publikaci [11].

3.2 Definice vlastní funkce

Pojďme si vše vyzkoušet a definujme funkci `pozdrav()` realizující činnost příkazů z posledního výpisu minulé kapitoly, tj. z výpisu 2.11 na straně 51. Výslednou definici si můžete prohlédnout ve výpisu 3.1.

Definice se od výpisu 2.11 syntakticky liší především hlavičkou následovanou dokumentačním komentářem. Další odchylkou je to, že v definici nemůžeme spoléhat na implicitní proměnnou, kterou jsme tehdy použili, ale musíme potřebnou proměnnou definovat sami. Proto je na řádku 3 definována **lokální proměnná** `oslovení`.

Výpis 3.1: *Definice funkce* `pozdrav()`

```
>>> def pozdrav():
        """Zjistí požadované oslovení uživatele a popřeje mu dobrý den."""
        oslovení = input('Jak vás mám oslovovat? ')
        print('Dobrý den,', oslovení)

>>> pozdrav()
Jak vás mám oslovovat? šéfe
Dobrý den, šéfe
>>> help(pozdrav)
Help on function pozdrav in module __main__:

pozdrav()
    Zjistí požadované oslovení uživatele a popřeje mu dobrý den.

>>> pozdrav.__doc__
'Zjistí požadované oslovení uživatele a popřeje mu dobrý den.'
>>>
```

Lokální proměnné

Lokální proměnná je proměnná, která je definována uvnitř funkce. Tím se stává soukromým majetkem funkce určeným pouze pro její vnitřní potřebu a nikdo z okolí nemá šanci zjistit, jakou hodnotu tato proměnná uchovává. Občas bychom ale potřebovali tuto hodnotu znát.

Teoreticky bychom mohli tuto potřebu vyřešit tak, že bychom příslušnou proměnnou definovali vně funkce, kde by na ní ostatní viděli a mohli s ní pracovat. Takové řešení je ale považováno přinejmenším za nešťastné, protože je potenciálním zdrojem řady chyb. Lepším řešením je definice funkce s návratovou hodnotou, kterou si za chvíli vysvětlíme.

3.3 Problémy s odsazováním v IDLE

Python vyžaduje, aby všechny příkazy těla byly stejně odsazené oproti hlavičce. Aby to bylo možné lépe dodržovat, tak ve většině IDE začínají v příkazových panelech případné pokračovací řádky automaticky třemi definovanými nebílými znaky (např. `•••`) následovanými mezerou, aby byly začátky všech řádků pod sebou. Stejně to řeší i *Python* spuštěný v konzolovém okně z příkazového řádku – viz výpis 3.2.

Příkazové okno prostředí IDLE bohužel tento zvyk nedodržuje, takže řádek za výzvou začíná v pátém sloupci (předchází mu výzva se třemi většítky a mezerou), kdežto následující řádky začínají ve sloupci prvním.

U jednoduchých definic, jakou je např. definice funkce `pozdrav()`, se s tím vypořádáme snadno – prostě následující řádky o trochu více odsadíme, aby jejich začátky byly vizuálně zarovnány pod pátým znakem prvního řádku. Pokud se však uvnitř těla funkce vyskytne nějaký složitější složený příkaz, může to být problém.

Výpis 3.2: *Odsazování pokračovacích řádků složených příkazů v konzolovém okně*

```
Microsoft Windows [Version 10.0.18362.900]
(c) 2019 Microsoft Corporation. Všechna práva vyhrazena.

Q:\65_PGM\65_PYT>python
Python 3.9.0b3 (tags/v3.9.0b3:b484871, Jun  9 2020, 20:36:59) [MSC v.1924 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> def funkce():
...     print('Takto to Python řeší v konzolovém okně.')
...
>>> funkce()
Takto to Python řeší v konzolovém okně.
>>>
```

Sloučení více řádků do jednoho

Považujeme-li za rozumné definovat takový složený příkaz v interaktivním režimu, můžeme si pomoci rozdělením hlavičky do více řádků. Pokud je totiž posledním znakem na řádku zpětné lomítko (\), tak *Python* připojí následující řádek na konec předchozího a tento sloučený útvar považuje za jeden řádek.

Budu-li proto chtít zadat v interaktivním režimu složitější složený příkaz, napíšu na první řádek samotné zpětné lomítko a vlastní zadání začnu od začátku druhého řádku, jak je tomu např. ve výpisu 3.3. Vzhled takto zadané definice se pak přiblíží vzhledu definic ve zdrojových souborech.

3.4 Funkce s návratovou hodnotou

Funkce `pozdrav()` definovaná ve výpisu 3.1 po svém zavolání pouze provede zadanou akci. Jak už jsem ale naznačil, často potřebujeme, aby funkce nejenom něco provedly, ale aby také předaly nějakou hodnotu. Například u funkce `pozdrav()` by se nám hodilo, kdybychom si zadané oslovení mohli někde zapamatovat, abychom ho mohli ještě někde použít.

Nejlepším řešením je v takovém případě definovat funkci s návratovou hodnotou. Dosáhneme toho tak, že před ukončením funkce zadáme příkaz začínající klíčovým slovem `return`, za nějž napíšeme výraz, jehož hodnota bude návratovou hodnotou dané funkce. Upravenou definici funkce `pozdrav()` si můžete prohlédnout ve výpisu 3.3.

Shoda názvu proměnných

Ve výpisu bych vás chtěl upozornit na definici proměnné `oslovení` na řádku **10**. Toto je zcela jiná proměnná než stejnojmenná lokální proměnná definovaná na řádku **6**. Jak už jsem řekl, lokální proměnné jsou viditelné pouze uvnitř funkce, v níž jsou definovány. V prostoru mimo tělo funkce o nich nikdo neví a nejsou pro nikoho přístupné.

Výpis 3.3: *Upravená definice funkce* `pozdrav()`

```
>>> \
def pozdrav():
    """Zjistí požadované oslovení uživatele, popřeje mu dobrý den
    a požadované oslovení vrátí jako svoji návratovou hodnotu.
    """
    oslovení = input('Jak vás mám oslovovat? ')
    print('Dobrý den,', oslovení)
    return oslovení

>>> oslovení = pozdrav()
Jak vás mám oslovovat? kolego
Dobrý den, kolego
>>> oslovení
'kolego'
>>>
```

3.5 Funkce s parametry

Většina funkcí modifikuje svoji činnost na základě dat, které jim volající program předá při jejich zavolání. Označujeme je jako **funkce s parametry**.

Jak jsme si vysvětlili již v pasáži *Parametr versus argument* na straně 49, *parametry* funkce jsou proměnné, které volající program inicializuje tím, že do nich při volání funkce vloží hodnoty *argumentů*, což jsou výrazy, které v kódu zadáváme do závorek za odkazem na volanou funkci.

Parametry jsou lokálními proměnnými dané funkce. Od běžných lokálních proměnných se liší v jediné věci: parametry inicializuje volající program zadáním odpovídajících argumentů, kdežto běžné lokální proměnné musí být inicializovány explicitně v těle funkce.

Definice jednoduché funkce s parametry je spolu s demonstrací jejího možného volání ve výpisu 3.4.

Výpis 3.4: *Definice a použití funkce* `bmi`

```
>>> def bmi(výška, hmotnost):
        """Spočte BMI index osoby se zadanou váhou a výškou."""
        return hmotnost / (výška * výška)

>>> bmi(1.85, 90)
26.296566837107374
>>>
```

Zadávání argumentů

Argumenty volané funkce, tj. výrazy definující počáteční hodnoty jejích parametrů, je možné zadávat dvěma způsoby:

- Zadáme jednotlivé argumenty ve stejném pořadí, v jakém jsou v hlavičce funkce uvedeny odpovídající parametry (viz řádek 5 ve výpisu 3.4). Takto zadané argumenty označujeme jako **poziční** (positional arguments), protože to, který parametr je daným argumentem inicializován, se odvozuje od pozice daného argumentu v seznamu argumentů.
- Zapíšeme název inicializovaného parametru následovaný rovnítkem a zadávaným argumentem. Takto zadané argumenty označujme jako **pojmenované** (anglicky se označují jako keyword arguments). U tohoto způsobu nezáleží na pořadí, v jakém budou argumenty zadány.

Oba způsoby zápisu můžeme kombinovat, jenom musíme **dbát na to, abychom pojmenované argumenty zadávali až za pozičními**, přičemž v sekci pozičně zadávaných argumentů nesmíme některý přeskočit s tím, že jej posléze zadáme jako pojmenovaný. Každý argument lze zadat jenom jednou.

Implicitní hodnoty argumentů

Domníváte-li se, že některý z argumentů bude často zadáván s nějakou hodnotou, můžete v seznamu parametrů tuto hodnotu příslušnému parametru hned zadat jako jeho *implicitní počáteční hodnotu* (*default initial value*), která se použije v případě, kdy uživatel daný argument nezadá. Myslete jen na to, že inicializované parametry musejí být v seznamu uvedeny jako poslední.

Argumenty s implicitní hodnotou budeme v dalším textu označovat jako **implicitní argumenty**. AHA-příklad s definicí funkce s implicitními argumenty demonstrující různé možnosti zadávání argumentů najdete ve výpisu 3.5.

Výpis 3.5: *Používání pozičních a pojmenovaných argumentů*

```
>>> def pos_key_demo(p1, p2, p3, p4='Nezadán', p5='Nezadán'):
        """Pomocná funkce pro demonstraci možností zadávání argumentů."""
        print(f'{p1=}, {p2=}, {p3=}, {p4=}, {p5=}')

>>> pos_key_demo('a1', 'a2', 'a3', 'a4')
p1='a1', p2='a2', p3='a3', p4='a4', p5='Nezadán'
>>> pos_key_demo(10, 20, p5=55, p3=33)
p1=10, p2=20, p3=33, p4='Nezadán', p5=55
>>>
```

Povinně poziční a povinně pojmenované argumenty

Pro úplnost bych vám měl prozradit, že *Python* umožňuje zadat požadavek, aby jistá skupina počátečních argumentů byla vždy zadávána pozičně a/nebo aby jistá skupina závěrečných argumentů byla vždy zadávaná pojmenovaně. Objeví-li se v seznamu parametrů nějaké funkce fiktivní parametr nazvaný `/` (lomítko), musejí být všechny předchozí parametry zadávány jako poziční. Objeví-li se fiktivní parametr nazvaný `*` (hvězdička), musejí být všechny následující argumenty zadávány jako pojmenované.

Jako fiktivní označuji tyto parametry proto, že to nejsou skutečné parametry, ale pouze značky označující v seznamu parametrů pozici, odkud začne daný požadavek platit.

3.6 Funkce `print()` a její parametry

Typickým příkladem funkce s implicitními argumenty je funkce `print`, kterou jsme již mnohokrát použili. Její hlavička má tvar

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

Počáteční hvězdička není tou hvězdičkou, o níž jsem hovořil před chvílí, protože nereprezentuje fiktivní parametr – není za ní čárka. Tentokrát hvězdička naznačuje, že na počátku může být libovolný počet argumentů (s touto možností vás podrobněji seznámím v pasáži *Hvězdičkový parametr* na straně 124).

Význam parametrů s implicitními hodnotami je následující:

`sep` Definuje oddělovač (separátor) jednotlivých tištěných položek. Nezadáte-li nic, budou položky oddělovány mezerou.

`end` Definuje ukončovací text vkládaný za poslední tištěnou položku. Implicitně je nastaveno ukončení řádku.

`file` Definuje název souboru, do nějž se bude tisknutý text ukládat. Implicitně je nastaven standardní výstup.

`flush` Definuje, zda se bude po daném tisku „splachovat“, tj. zda program zabezpečí, aby se informace přesunula z vyrovnávací paměti do výstupního souboru. Podrobněji o něm hovořím v doprovodné učebnici [11].

Samostatný výpis s ukázkami možného použití sem vkládat nebudu, protože tuto funkci v dalším textu ještě mnohokrát použijeme.

3.7 Výrazy versus příkazy

Jakoukoliv akci, jejímž výsledkem je získání nějaké hodnoty, označujeme jako **výraz**. Výrazem jsou proto matematické operace nebo volání funkce, a to přesto, že účelem volání řady funkcí je provedení nějaké akce a obdrženým výsledkem je hodnota `None`.

Operace, jejímž provedením žádnou hodnotu nezískáme (ani `None`), označujeme jako **příkaz**. Doposud jsme se setkali se dvěma druhy příkazů: s příkazem přiřazení a s definicí funkce (i když to je vlastně také jistý druh přiřazení).

Výrazy používáme vždy jako součást příkazů – v příkazu přiřazení je např. na pravé straně rovnítka výraz, který se musí spočíst, a jeho hodnotu pak uložíme do proměnné definované výsledkem výrazu na levé straně rovnítka.

Zvláštním hybridem jsou tzv. **výrazové příkazy**, což jsou příkazy, jejichž součástí není nic jiného než výraz.

Výrazové příkazy má smysl zadávat pouze při komunikaci se systémem v interaktivním režimu, kdy chceme, aby nám interpret zobrazil jejich hodnotu. Použijí-li se kdekoliv jinde než v přímé komunikaci s interpretem (např. uvnitř definice funkce), tak se sice v požadovanou chvíli provedou, ale nic se nezobrazí.

3.8 Definice prázdné funkce

Jednou za čas se hodí definovat prázdnou funkci, která pak slouží pouze jako připomínka toho, že danou funkci chceme definovat. Funkci bez těla nám *Python* definovat nedovolí. Musí obsahovat alespoň jeden příkaz.

Podle mne je optimálním řešením zadat v těle dokumentační komentář, v němž naznačíte, proč danou funkci definujete. V učebnicích ale většinou v takových případech autoři používají příkaz `pass`, jenž zastupuje prázdný příkaz z jiných programovacích jazyků. Třetí možností je použít výpustku. *Python* totiž definuje objekt nazvaný `...` (tři tečky), jenž je jedinou instancí třídy `Ellipsis`. Zkuste jej zadat jako příkaz.

Ve výpisu 3.6 najdete všechny tři podoby definice prázdné funkce. Jak jsem ale již řekl, osobně dávám přednost té první.

Výpis 3.6: *Možné podoby definice prázdné funkce*

```
>>> def prázdná_funkce():
        """Dokumentační komentář."""

>>> def funkce_s_prázdným_příkazem():
        pass

>>> ...
Ellipsis
>>> def funkce_s_výpustkou():
        ...

>>> prázdná_funkce()
>>> funkce_s_prázdným_příkazem()
>>> funkce_s_výpustkou()
>>>
```

3.9 Datový typ

Datový typ (nebo zkráceně jen typ) je označení pro trojici charakteristik specifikujících vlastnosti hodnot, které označujeme jako data daného typu. Svůj typ mají veškerá data, s nimiž program pracuje. Datový typ specifikuje:

- ☞ množinu přípustných hodnot, resp. stavů,
- ☞ způsob uložení těchto hodnot (stavů) v paměti (o ten se zatím nebudeme zajímat a zpracování této informace přenecháme virtuálnímu stroji),
- ☞ množinu operací, které lze s instancemi daného typu provádět, spolu se specifikací zpráv, které je možné danému objektu posílat.

Jinými slovy: datový typ prozrazuje, co můžeme od hodnot daného typu očekávat a co s nimi můžeme dělat.

Řada programovacích jazyků požaduje, abyste u každé proměnné deklarovali typ hodnot, které se do ní mohou uložit. To sice programátora poněkud omezuje a zdržuje, ale na druhou stranu tento požadavek snižuje počet potenciálních chyb. Současně poskytuje překladači řadu dodatečných informací, takže může přeložit program efektivněji a tím zvýšit efektivitu práce výsledného programu.

Python takovýmito požadavky programátora neomezuje. Vede to sice k pomalejšímu běhu programu, protože program musí za běhu dělat řadu rozhodnutí, která u jazyků z druhé skupiny mohl dělat překladač. Na druhou stranu je to však bohatě vykoupeno výrazně větší produktivitou programátora, která nás v prvním přiblížení zajímá nejvíce, protože efektivitu programu lze snadno zvýšit dobře napsanými knihovnami. Nevýhodou této svobody je vyšší množství potenciálních chyb.

3.10 Anotace

Jak jsme si právě vysvětlili, *Python* umožňuje, aby se typ objektů v proměnných průběžně měnil, což přispívá k rychlejšímu vývoji, ale na druhou stranu to také zvyšuje pravděpodobnost nejrůznějších chyb z přehlédnutí. *Python* proto umožňuje deklarovat typy proměnných a typ návratové hodnoty funkcí.

Prozatím jsme pracovali s hodnotami čtyř datových typů: celá čísla jsou typu `int`, reálná čísla jsou typu `float`, stringy jsou typu `str` a hodnota `None` je jedinou hodnotou typu `NoneType`. V příští kapitole se naučíte vytvářet datové typy vlastní.

Deklaraci typů realizujeme prostřednictvím **anotací**, což jsou výrazy, k nimž *Python* primárně přistupuje obdobně jako ke komentářům – zapamatuje si je, aby programy mohly v případě potřeby jejich dodržení zkontrolovat. Na rozdíl od komentářů však při překladu zkontroluje, jde-li daný výraz vyhodnotit. Úplné nesmysly tam proto psát nemůžeme.

Chcete-li anotovat proměnnou (většinou parametr funkce), vložte za identifikátor (a před případné rovnítko) dvojtečku následovanou anotačním výrazem. Chcete-li

anotovat návratovou hodnotu funkce, vložte za zavírací závorku seznamu parametrů šipku (znaky `->`) a vlastní anotaci pak vložte mezi tuto šipku a dvojtečku uzavírající hlavičku. Vše se snaží demonstrovat výpis 3.7.

Na řádcích 1-4 je v něm definována funkce `anotace()`, která má anotované parametry i návratovou hodnotu. Pokud byste se chtěli na její anotace zeptat, požádejte ji o atribut `__annotations__`, jak je předvedeno na řádku 6. Jeho obsah se vypisuje jako seznam dvojic typu *`název-parametru : anotace`*. Pro návratovou hodnotu se místo názvu parametru uvádí `return`.

Výpis 3.7: *Definice funkce demonstrující použití anotací*

```
>>> def anotace(celé: int, reálné: float = 3.14) -> None:
        """Funkce demonstrující použití anotací."""
        proměnná: str = f'{celé * reálné = } = {celé} * {reálné}'
        print(proměnná)

>>> anotace.__annotations__
{'celé': <class 'int'>, 'reálné': <class 'float'>, 'return': None}
>>> anotace(3, 'Chyba ')
celé * reálné = 'Chyba Chyba Chyba ' = 3 * Chyba
>>>
```

Jak ukazuje příkaz na řádku 3, anotovat lze i proměnné. Ale protože se v tomto případě jedná o lokální proměnnou, je to soukromá věc dané funkce, protože na lokální proměnné zvenku nikdo nevidí.

Příkaz na řádku 8 demonstruje, že *Python* nic nekontroluje, takže nehlásí chybu, že jsem si objednal `float` a dodal string.

Anotace ve svých programech používat nemusíte. Je to jen informace navíc, která může posloužit k porozumění vašemu programu těm, kteří jej budou po vás číst. Jejich používání v situacích, kdy je požadovaný typ argumentu a/nebo typ návratové hodnoty zřejmý, vám však vřele doporučuji.

3.11 Shrnutí

Záznam komunikace s interpretem probíhající v této kapitole najdete v trojici souborů nazvaných `m03__start_def`.

Kapitola 4
Základy OOP

Co se v kapitole naučíte
Kapitola seznamuje se základy objektově orientovaného paradigmatu, které tvoří teoretický základ pro objektově orientované programování. Vysvětlí vám základní principy, představí podstatu objektů a seznámí s významem jejich atributů a používání kvalifikace. Poté probere význam tříd, naučí vás definovat vlastní třídy a předvede, jak se s nimi pracuje.

4.1 Proč se učit objektové paradigma

Na počátku úvodu jsem říkal, že: „*Python je moderní programovací jazyk, který umožňuje velmi jednoduše navrhovat jednoduché programy, ale na druhou stranu nabízí dostatečně 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/á/é*“ používat zkratku OO a termín *objektově orientované programování* budu nahrazovat zkratkou OOP.

OOP se na konci 20. století stalo hlavním programovacím paradigmatem a lze očekávat, že tento stav ještě pěkných pár let vydrží. Bohužel většina kurzů (a to jak na školách, tak v soukromé sféře) doposud OOP neučí. Ty, které o sobě tvrdí, že učí OO programování, učí často pouze OO kódování, tj. učí, jak v daném programovacím jazyku navržený program zapsat. O tom, jak při návrhu OO programů přemýšlet a jak takový program správně navrhnout, se toho účastníci těchto kurzů většinou moc nedozvědí.

Tato učebnice se pokouší z těchto stereotypů vymanit a soustředit se spíše na to, jak program navrhovat. Popisu možností jazyka a konstrukcí, které je možné k zakódování navrženého programu použít, se věnuje doprovodná učebnice [11].

Znalost principů OOP vám u jednodušších programů umožní pochopit jejich chování v některých situacích a u složitějších projektů vám pomůže tyto projekty správně navrhnout.

Řada programátorů používajících *Python* tvrdí, že OOP ve svých programech nepoužívají, takže jeho znalost nepotřebují. Přiznejme si, že zejména v *Pythonu*, kde je všechno objekt, je prakticky nemožné napsat program nepoužívající objekty. To, že se někdo tváří, že objekty nepoužívá, ještě neznamená, že tomu tak doopravdy je.

Neznalost objektové podstaty jazyka a její ignorování vede často k tomu, že programátor nechápe, proč se jeho program chová tak divně, a neumí program správně opravit. Náhradní záplaty neřešící vlastní podstatu problému pak vedou často k tomu, že se chyba projeví jinde.

Mohu-li vám něco doporučit, **neignorujte objektovou podstatu jazyka**. Mnohé konstrukce se pak zprůzrační, budou pochopitelnější, a budete v nich proto dělat méně chyb.

4.2 Základní princip OOP

OOP vychází z myšlenky, že **všechny programy, které vytváříme, jsou simulací buď 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 skutečností a budou i použitelnější, spolehlivější a snáze udržovatelné.

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.

Zprávy × metody

Řekli jsme si, že objekty si navzájem posílají zprávy. Zprávy bychom mohli chápat jako žádost o reakci. Posílám-li objektu zprávu, chci, aby nějak reagoval.

Část programu, která definuje, jak bude objekt reagovat na obdrženou zprávu, označujeme termínem **metoda**. Každá zpráva, na kterou objekt umí zareagovat, má v programu přiřazenu svoji vlastní metodu (případně několik metod).

Zde se objevuje drobný terminologický guláš. Analytici totiž dávají přednost termínu „poslat zprávu“, programátoři hovoří spíše o volání metod. Svým způsobem je jedno, jestli řeknete, že instanci pošlete zprávu nebo že zavoláte její metodu.

Termín *zpráva* budu proto používat spíš v souvislosti s popisem chování programu, termínu *metoda* pak budu dávat přednost ve chvíli, kdy budu hovořit o konkrétní části programu realizující odpověď na zaslání příslušné zprávy.

Jak jsem ale řekl, oba termíny jsou do jisté míry ekvivalentní. Rozdíl je spíš pocitový, protože oznámením *„posílám zprávu“* říkám, co po oslovovaném objektu chci, kdežto termínem *„volám metodu“* jakoby oznamuji, jaký kód bude pro vyřešení mého požadavku spuštěn. Jenomže to programátoři často vůbec netuší. Který kód se v reakci na můj požadavek opravdu spustí, nejde totiž při psaní programu v řadě případů ani odhadnout – to se určuje až za běhu.

Metody × funkce

Aby byl terminologický guláš dokonalý, používá se v *Pythonu* ještě termín funkce. Funkce je v *Pythonu* metoda, která zdánlivě není atributem žádného objektu, takže ji můžeme volat bez kvalifikace (co to je, si řekneme za chvíli v pasáži *Práce s objekty – kvalifikace*).

Berte to tak, že tento termín je určen pro programátory, kteří nechtějí programovat objektově a chtějí mít proto pocit, že žádné objekty nepoužívají. Nepoužívat objekty v jazyku, v němž je všechno objekt, sice nejde, ale syntaxe jazyka jim umožňuje se tak alespoň tvářit.

Já se této zvyklosti pokusím přizpůsobit a funkční objekty, které o sobě nekřičí, že jsou atributy nějakého objektu, budu označovat jako funkce, kdežto funkční objekty, které to přiznávají, budu označovat jako metody. Je to ale opravdu jen úlitba zavedeným zvyklostem, protože ve skutečnosti jsou prakticky všechny funkční objekty něčím atributem. Jedinou výjimkou jsou funkce definované jako lokální proměnné, a ty se používají jen minimálně.

4.3 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 (připojení, 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. 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***.

Atributy jsou definovány jako proměnné, které odkazují 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.

To, jestli je atribut datový, typový nebo funkční, závisí pouze na tom, na co daná proměnná právě odkazuje. Když do proměnné odkazující na metodu vložíte omylem (nebo možná i záměrně) nějaký údaj, změní se daný atribut z funkčního na datový.

Berte to tak, že v *Pythonu* je aktuální druh daného atributu pouze vlastností daného okamžiku. Na druhou stranu se druh atributů mění spíše výjimečně a v řadě situací pomáhá v orientaci v kódu, tak budu toto označení nadále používat.

Práce s objekty – kvalifikace

Kdykoliv chceme pracovat s některým z atributů objektu, musíme objekt nejprve oslovit, a poté mu sdělit, se kterým jeho atributem chceme pracovat. Děláme to následovně:

- Získáme odkaz na objekt (zadáme název proměnné, v níž je uložen, nebo jej získáme jako výsledek výrazu – nejčastěji volání funkce).
- Napíšeme tečku.
- Za ní pak napíšeme název atributu, s nímž chceme pracovat. Obsahuje-li atribut odkaz na funkci, kterou chceme zavolat, přidáme závorky se seznamem argumentů.

Oslovení objektu, tj. část před tečkou, označujeme termínem **kvalifikace**. Kvalifikace tak označuje, komu daný atribut patří. Stejně nazvaný atribut totiž může vlastnit více objektů.

Správně bychom měli kvalifikovat každou svoji žádost o atribut objektu, nicméně programátoři si chtějí vždy ušetřit co nejvíce psaní, takže autoři jazyků definují vždy několik situací, kdy si můžeme kvalifikaci odpustit a nechat na překladači, aby ji doplnil za nás. Postupně vás s nimi seznámím.

4.4 Třídy a jejich instance

Ve větších programech se vyskytují tisíce a desetitisíce 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.

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

Třídy mají mezi objekty zvláštní postavení – 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 je to buď třída, anebo pouze zprostředkoval získání instance vyrobené nějakou třídou.

Třída (přesněji objekt reprezentující danou třídu) definuje společné vlastnosti svých instancí a definuje i nástroje pro vytváření těchto instancí. Přirovnáme-li programování k hraní si na pískovišti, pak třída je něco jako formička a její instance jsou bábovičky, které za pomoci této formičky vytváříme. Při jiném přirovnání bychom mohli prohlásit, že třída je vlastně továrna na objekty – její instance.

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

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

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 dovolí 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.

Vytváření instancí – konstruktor, alokátor, initor

Třídu můžeme v *Pythonu* vnímat i jako funkci, jejímž zavoláním vytvoříme instanci dané třídy, kterou nám tato funkce vrátí jako svoji funkční hodnotu. S tím jsme se setkali např. ve výpisu 2.9 na straně 48, kde jsme volali funkci `str()`.

Tuto funkci můžeme chápat jako tzv. **konstruktor**, což je funkce, která má na starosti vytvoření (zkonstruování) objektu – zde instance dané třídy. Vytvoření instance přitom provádí ve dvou krocích:

- Nejprve zavolá metodu `__new__()`, která pro daný objekt alokuje paměť, tj. vyhradí místo v paměti a provede několik nízkoúrovňových inicializačních operací. Budu ji proto označovat **alokátor**.

  Návratovou hodnotou alokátoru je odkaz na alokovanou paměť, kterou bychom mohli označit jako polotovar instance.

- Pak zavolá metodu `__init__()`, které předá odkaz vrácený alokátorem a případné další obdržené argumenty. Tato metoda pak polotovar instance inicializuje a případně provede některé pomocné operace.

  Některé třídy považují polotovar vytvořený metodou `__new__()` za plnohodnotný objekt a metodu `__init__()` vůbec nedefinují. Většina tříd ji ale definuje. Uvidíte, že v naší aplikaci se setkáme s oběma případy.

  Metoda `__init__()` bývá v některých učebnicích *Pythonu* označována jako *konstruktor*. To je ale nešťastné označení, protože, jak již víte, metoda instanci nevytváří, ale pouze inicializuje. Výstižnější by bylo označovat ji jako *inicializátor*, ale ani to není zcela přesné. Budu ji proto označovat nepřeložitelným slovem **initor**.

4.5 Definice třídy a jejích atributů

Definice třídy je v *Pythonu* složený příkaz, jehož hlavička začíná klíčovým slovem `class`, za nímž následuje název dané třídy. Tělo třídy pak může obsahovat inicializace atributů třídy, mezi něž zahrnujeme i definice jejích metod a případných interních tříd.

Název třídy by měl podle konvencí začínat velkými písmeny a v případě několikaslovných názvů by slova měla být na sebe nalepena a každé by mělo začínat velkým písmenem – např. `TypickýNázevTřídy`.

Definice třídy se překládá obdobně jako definice funkce. Překladač zjistí, jestli již existuje proměnná s názvem třídy, a pokud ne, tak ji vytvoří. Pak začne provádět tělo třídy příkaz za příkazem, a když je hotov, tak do vytvořené proměnné uloží odkaz na objekt dané třídy.

Základní rozdíl mezi definicí třídy a definicí funkce je v tom, že při definici funkce se příkazy z jejího těla překládají, kdežto při definici třídy se tyto příkazy ihned provádějí.

Ve výpisu 4.1 je připravena definice třídy `DemoClass`, na níž si předvedeme některé důležité prvky. Tato definice je sice ještě zadána v interaktivním režimu, ale díky rozdělení počátečního řádku podle doporučení v podkapitole *3.3 Problémy s odsazováním v IDLE* na straně 57 vypadá obdobně, jako by byla zapsána v nějakém editoru.

Dokumentační komentář

Na řádku 2 je hlavička třídy následovaná na řádcích 3–5 dokumentačním komentářem. Stejně jako u funkcí dalších dokumentovaných objektů je dokumentačním komentářem string, který je uveden na začátku těla. A stejně jako funkce a další dokumentované objekty si třída svůj dokumentační komentář pamatuje a můžete se jí na něj kdykoliv zeptat způsobem popisovaným v pasáži *Získání nápovědy – dokumentace* na straně 56.

Při dotazu na atribut `__doc__` se stejně jako u funkce zobrazí jen dokumentační komentář, ale při zavolání funkce `help()` obdržíte vedle tohoto komentáře i seznam všech deklarovaných metod s jejich komentáři a seznam všech atributů třídy s jejich aktuálními hodnotami. Vyzkoušejte si to.

Příkazy těla třídy se provádějí

Za dokumentačním komentářem následují příkazy, které se všechny provedou. Na řádku 6 je příkaz tisku, který zobrazí informaci o startu definice dané třídy, obdobně jako závěrečný příkaz těla třídy na řádku 30, jenž zobrazí zprávu o ukončení definice. Zprávy vytištěné oněmi příkazy tisku najdete ve výpisu 4.1 na řádcích 32–33.

Mezi nimi jsou příkazy definující atributy – datové (číselné hodnoty a odkazy na objekty), funkční (odkazy na metody) a typové (odkazy na objekty tříd). Jak jsme si však říkali v podkapitole *3.7 Výrazy versus příkazy* na straně 62, kdyby se v těle třídy objevily výrazové příkazy, tak se sice provedou, ale nic se nezobrazí.

Atributy třídy

Na řádcích 8 a 9 jsou definovány dva datové atributy. Jsou to atributy objektu reprezentujícího danou třídu, takže budu-li s nimi chtít pracovat, musím oslovit jejich majitele, tj. třídu `DemoClass`.

Na řádcích 12–14 je definována metoda třídy (tj. funkční atribut). Problém je, že systém předpokládá, že všechny metody, které ve třídě definujeme, jsou metodami instancí. Třídních metod se totiž v OOP definuje výrazně méně. *Python* proto zavedl pravidlo, že když nějakou takovou definujeme, měli bychom ji jednoznačně označit jako metodu třídy, aby se s danou metodou pracovalo maličko nestandardně.

Výpis 4.1: *Definice demonstrační třídy* `DemoClass`

```
>>> \
class DemoClass:
    """Demonstrační třída definující svůj datový a funkční atribut,
    tj. datovou proměnnou a metodu a vedle toho i důležité instanční metody.
    """
    print('Spouští se definice třídy DemoClass')

    c_atribut = 'Třídní datový atribut'
    instancí  = 0     # Počet doposud vytvořených instancí

    @staticmethod
    def c_mtd(id=None):
        """Metoda třídy - statická metoda."""
        print(f'Spuštěna třídní metoda DemoClass.c_mtd({id=})')

    def i_mtd(self, id=None):
        """Instanční metoda."""
        print(f'Spuštěna metoda i_mtd({id=}) instance {self}')

    def __init__(self):
        """ Initor (přesněji inicializátor) instancí třídy DemoClass."""
        DemoClass.instancí = DemoClass.instancí + 1
        self.ID = DemoClass.instancí
        print(f'Instance vytvořena: {self}')

    def __repr__(self):
        """Vrátí textový podpis dané instance."""
        return f'DemoClass(ID={self.ID})'

    print('Konec definice třídy DemoClass')

Spouští se definice třídy DemoClass
Konec definice třídy DemoClass
>>>
```

Dekorátory

K označení nestandardních součástí programu slouží tzv. *dekorátory*, což jsou speciální objekty, které zadáme před onu nestandardní definici – v našem případě před definici metody `c_mtd`. Před název dekorátoru pak vložíme ještě znak `@` (zavináč, šnek) indikující, že následující identifikátor představuje dekorátor.

Teoreticky definice projde i bez označení dekorátorem (tj. překladač ji přeloží a při korektním použití bude i korektně pracovat), ale konvence použití dekorátoru vřele doporučují, protože se pak snáze odhalují některé chyby.

Konkrétně dekorátor `staticmethod()` zabrání tomu, aby se při kvalifikaci dané metody instancí překladač pokoušel předat tuto instanci jako nultý argument. Třídní atributy bychom měli vždy kvalifikovat třídou a jejich kvalifikace instancí je považována za nemravnost, které byste se měli určitě vyvarovat. Na druhou stranu vám tento dekorátor kryje záda v případě, kdy takovou nemravnost z nejrůznějších důvodů provedete.

Metody instancí

Všechny tři další metody jsou již metodami instancí, což poznáte podle toho, že se jejich první parametr jmenuje `self`. Teoreticky by se sice mohl jmenovat jakkoliv, ale podle konvence byste jej měli pojmenovat `self`. Tento parametr odkazuje na instanci, pro níž daná metoda pracuje.

Abychom byli přesní, tyto metody jsou vlastně také definovány jako metody třídy a můžeme k nim proto přistupovat i prostřednictvím třídy. Jejich definice však počítají s tím, že prvním parametrem je výše zmíněný odkaz na instanci dané třídy, se kterou mají pracovat. Jejich volání proto můžeme kvalifikovat instancí, kterou pak překladač dosadí do počátečního parametru (přesněji uloží do něj odkaz na tuto instanci).

Tak pracuje např. metoda `i_mtd()` definovaná na řádcích `16-18`. Ta pouze oznamuje své spuštění, přičemž součástí této zprávy je zobrazení tzv. textového podpisu dané instance výrazem `{self}` v tištěném f-stringu.

Initor

Metodu `__init__()` definovanou na řádcích `20-24` jsem v pasáži *Vytváření instancí – konstruktor, alokátor, initor* na straně 70 označil jako initor. Pojďme se blíže podívat na to, co tato metoda dělá. Na řádku `22` zvětší atribut `instancí` o jedničku. Zavoláním metody `__init__()` končí proces vytváření další instance, takže touto inkrementací zabezpečí, že třída bude vědět, že má o instanci více.

Na řádku `23` pak metoda vytváří instanční datový atribut `ID` a ukládá do něj aktuální počet instancí dané třídy. Ten bychom mohli považovat za rodné číslo dané instance. V *Pythonu* se instanční atributy nevytvářejí běžnou deklarací, jako tomu bylo u atributů třídy na řádcích `8` a `9`, ale vytvářejí se v instančních metodách tak, jak vidíme na řádku `23`.

Kdyby proměnná `ID` nebyla kvalifikována parametrem `self`, tak by daný příkaz vytvořil lokální proměnnou dané funkce – v našem případě initoru. Její kvalifikací však zařídíme, že se z této proměnné stane instanční atribut.

Na závěr pak na řádku `24` initor oznamuje vytvoření dané instance.

Speciální metody

Na initoru i na následující metodě `__repr__()` je důležité to, že jejich název začíná a končí dvojicí podtržítek, což je příznak, že se jedná o speciální metody, které jsou automaticky volány při definovaných příležitostech.

O metodě `__init__()` jsme si řekli, že je volána na závěr konstrukce objektu a má za úkol jej inicializovat. Metoda `__repr__()` je zase volána v situacích, kdy potřebujeme získat textovou reprezentaci daného objektu – tzv. podpis. Když má proto být na řádku `18` nebo na řádku `24` tištěna hodnota instance `self`, zavolá se její metoda `__repr__()`,[6] která textový podpis dané instance dodá.

[6] Není to sice zcela přesné, ale pro naše účely to prozatím postačí. Přesnější výklad najdete v dokumentaci anebo v [11].

Zvláštností speciálních metod uvozených a ukončených dvojicí podtržení je, že se systém v definovanou chvíli podívá, jestli má vaše třída či instance definovanou příslušnou metodu, a pokud ano, zavolá ji. Nemá-li ji definovanou, provede se nějaká implicitní akce.

Nepotřebujeme-li vytvářenému objektu definovat atributy ani provádět nějaké doprovodné akce, nemusíme mu definovat initor. Na závěr inicializace se pak nic neprovede. Nepotřebujeme-li, aby se instance podepisovaly nějakým speciálním způsobem, nemusíme definovat metodu `__repr__` a instance převezme podpis od systému.

Definice třídy je obyčejný příkaz

Na závěr tohoto teoretického úvodu bych chtěl upozornit, že definice třídy je (obdobně jako definice funkce) obyčejný složený příkaz, který můžeme kdykoliv použít na místě, kde se smí použít příkaz, tedy i v definici funkce nebo jiné třídy. Nebudeme si to zde předvádět, jenom bych chtěl, abyste si to uvědomovali a aby vás nepřekvapilo, když na takový obrat někde narazíte.

Takovéto definice se jednou za čas opravdu objeví. Zájemce, kteří by chtěli vidět nějaké příklady, odkážu (jako již ostatně několikrát) na publikaci [11].

4.6 Práce s vytvořenou třídou a jejími instancemi

Kdykoliv něco naprogramujeme, měli bychom to ihned otestovat. Ve výpisu 4.2 je záznam zadání několika příkazů a reakce na ně.

Na řádku 1 je volána metoda `c_mtd()`, která je metodou třídy. Její volání je proto logicky kvalifikováno její třídou. Metoda pak na řádku 2 vypíše zprávu o svém zavolání a spuštění.

Na řádku 2 je třída dotazována na hodnotu svého atributu `instancí`. Protože třída prozatím ještě žádnou instanci nevytvořila, je jeho hodnotou logicky nula.

Na řádku 5 jsou dva příkazy vytvářející instance třídy `DemoClass` a ukládající vytvořené instance do proměnných `i1` a `i2`. Jak vidíte, instance se vytvoří tak, že se zavolá daná třída, jako by to byla funkce. Jak už jsme si řekli, na konci konstrukce se automaticky zavolá initor, který v našem případě ukončí svojí činnost vypsáním zprávy o vytvoření dané instance. Tyto zprávy jsou na řádcích 6 a 7.

Když je třída na řádku 8 znovu dotázána na hodnotu atributu `instancí`, vrátí nám dvojku, protože již byly vytvořeny dvě instance.

Na řádku 10 je volána instanční `i_mtd()`, která je kvalifikovaná třídou. V takovém případě je třeba zadat všechny parametry včetně počátečního. Metoda pak vypíše zprávu o svém zavolání doplněnou podpisem instance.

Přiznejme si však, že toto není příliš častý případ použití instančních metod a ukazuji vám jej spíše proto, abyste nebyli překvapeni, pokud se s ním někdy setkáte. Mnohem

častější je kvalifikace volání instančních metod jejich instancí, jak je předvedeno na řádku 12. V takovém případě za nás počáteční argument zadá překladač, a my máme v kódu na starosti zadání zbývajících argumentů.

Výpis 4.2: *Vytvoření instancí třídy* `DemoClass` *a práce s nimi*

```
>>> DemoClass.c_mtd(111)
Spuštěna třídní metoda DemoClass.c_mtd(id=111)
>>> DemoClass.instancí
0
>>> i1 = DemoClass();   i2 = DemoClass()
Instance vytvořena: DemoClass(ID=1)
Instance vytvořena: DemoClass(ID=2)
>>> DemoClass.instancí
2
>>> DemoClass.i_mtd(i1, 'Kvalifikováno třídou')
Spuštěna metoda i_mtd(id='Kvalifikováno třídou') instance DemoClass(ID=1)
>>> i2.i_mtd('Kvalifikováno instancí')
Spuštěna metoda i_mtd(id='Kvalifikováno instancí') instance DemoClass(ID=2)
>>>
```

4.7 Použití initoru s parametry

Initor třídy `DemoClass` bychom mohli považovat za bezparametrický, protože kromě povinného parametru `self` žádný jiný parametr nedefinuje. V řadě případů je ale potřeba initoru nějaké informace předat.

Výpis 4.3: *Definice třídy s initorem s parametry*

```
>>> \
class Kvádr:
    """Instance reprezentují kvádry se zadanými rozměry.
    """
    def __init__(self, délka: float, šířka: float, výška: float) -> None:
        """Vytvoří a inicializuje požadovaný kvádr."""
        self.délka = délka
        self.šířka = šířka
        self.výška = výška

    def __repr__(self) -> str:
        """Vrátí textový podpis dané instance."""
        return f'Kvádr(délka={self.délka}, šířka={self.šířka}, ' \
               f'výška={self.výška})'

>>> k1=Kvádr(1, 2, 3);   k2=Kvádr(10, 20, 30);   print(f'{k1 = }\n{k2 = }')
k1 = Kvádr(délka=1, šířka=2, výška=3)
k2 = Kvádr(délka=10, šířka=20, výška=30)
>>>
```

Ve výpisu 4.3 najdete definici třídy `Kvádr`, jejíž initor má kromě parametru `self` ještě tří další parametry, kterými inicializuje instanční proměnné, jejichž hodnoty se pak objeví v podpisu dané instance.

Na řádku 16 je pak ukázáno, že při vytváření instance se parametry předají třídě, která v tu chvíli vystupuje jako funkce, a třída je pak ve vhodnou chvíli předá initoru.

Kromě toho je na řádcích 11–14 definována metoda `__repr__()`, jejíž funkci a použití jsem vám vysvětloval v pasáži *Speciální metody* na straně 73.

Výraz na více řádcích

Na řádcích 13–14 je definován dlouhý string tak, že je rozdělen na dva, každý je zapsán na samostatném řádku a první řádek končí zpětným lomítkem. O této možnosti jsem hovořil v pasáži *Sloučení více řádků do jednoho* na straně 58.

Druhou možností řešení problému, kdy je výraz příliš dlouhý na to, abychom jej mohli zapsat na jeden řádek (konvence totiž doporučují nepoužívat řádky delší než 79 znaků a u dokumentačních komentářů dokonce 72 znaků), je uzavřít celý výraz do závorek. Jakmile *Python* narazí na otevírací závorku, tak řádky automaticky slučuje, dokud nedojde až k příslušné zavírací závorce. Takovýto způsob sloučení řádků je použit např. v definici příkazu `return` na řádcích 34–41 ve výpisu 9.1 na straně 134.

4.8 Definice prázdné třídy

Jednou za čas se hodí definovat prázdnou třídu, která pak slouží pouze jako připomínka toho, že danou třídu chceme definovat. S definicí třídy je to obdobné jako s definicí prázdné funkce, kterou jsme probírali v podkapitole *3.8 Definice prázdné funkce* na straně 62.

Opět se přimlouvám za to, abyste v takovém případě použili komentář, jak je předvedeno ve výpisu 4.4. V něm můžete naznačit, proč jste se rozhodli danou prázdnou třídu definovat.

Výpis 4.4: *Definice prázdné třídy*

```
>>> class PrázdnáOkomentovaná:
        """Dokumentační komentář s účelem třídy"""

>>> PrázdnáOkomentovaná
<class '__main__.PrázdnáOkomentovaná'>
>>>
```

4.9 Typy hodnot

V podkapitole *3.9 Datový typ* na straně 63 jsme si vysvětlili význam termínu *datový typ*. V objektově orientovaných jazycích je pojem *datový typ* zhruba ekvivalentní s pojmem

třída. Každý objekt, s nímž v programu pracujeme, můžeme chápat jako instanci nějaké třídy. Tuto třídu budeme označovat jako datový typ daného objektu.[7]

Při ladění, ale často i během běžného chodu programu je občas potřeba zjistit typ objektu, s nímž pracujeme. K tomu slouží funkce `type()`, které předáme daný objekt jako argument, a ona nám vrátí typ daného objektu.

Aby se nám tyto informace jednodušeji zjišťovaly, definujme funkci `pvt()`, které předáme jako argument string definující daný výraz, u nějž nás zajímá typ výsledku, a ona nám vypíše daný výraz, jeho hodnotu a jeho typ. String jí přitom předáváme proto, aby nám mohla vypsat, co přesně jsme jí zadali. A aby jej pak předala funkci `eval()`, která zadaný string vyhodnotí.

Definici funkce najdete na řádcích 2–5 ve výpisu 4.5. Nenechte se zde znervóznit znakem `#`, který normálně uvozuje komentář.

Na řádku 7 je pak připraveno několik výrazů, které jsou na řádku 9 předány postupně dané funkci jako argumenty. O tom, proč má na řádku 13 třída `DemoClass` předponu `__main__`, se dočtete v příští kapitole.

Výpis 4.5: *Definice a použití funkce `pvt()`*

```
1  >>> \
2  def pvt(expression: str) -> None:
3      """Vytiskne zadaný výraz, jeho hodnotu a typ výsledku."""
4      value = eval(expression)
5      print(f'{expression} = {value} # type = {type(value)}')
6
7  >>> celé = 1;   reálné = 1.5;   string = "String";   objekt = DemoClass()
8  Instance vytvořena: DemoClass(ID=3)
9  >>> pvt('celé');   pvt('reálné');   pvt('string');   pvt('objekt')
10 celé = 1 # type = <class 'int'>
11 reálné = 1.5 # type = <class 'float'>
12 string = String # type = <class 'str'>
13 objekt = DemoClass(ID=3) # type = <class '__main__.DemoClass'>
14 >>>
```

4.10 Shrnutí

Záznam komunikace s interpretem probíhající v této kapitole najdete v trojici souborů nazvaných `m04__classes`. Definice třídy `DemoClass` je uložena ještě jednou samostatně v souboru `m04a_DemoClass.py`.

[7] Termín *datový typ* je v jazyku *Python* ve skutečnosti poněkud obecnější než termín třída, ale podrobnější výklad rozdílu mezi nimi přesahuje zaměření této učebnice.

Kapitola 5
Moduly a práce s nimi

Co se v kapitole naučíte
Tato kapitola vám představí moduly a jejich význam. Naučí vás používat příkaz `import` v jeho různých podobách. Pak vám vysvětlí, jak definovat vlastní modul, jak jej použít a v případě potřeby znovu načíst.

5.1 Moduly – základní informace

Python nabízí velké množství nejrůznějších definic. Abychom se v nich vyznali a aby se omezila pravděpodobnost, že pojmenujete proměnnou stejně jako nějakou existující (a bude proto docházet k záměnám), jsou všechny klíčové definice rozděleny do tematických skupin označovaných jako **moduly**.

Jako modul označujeme část kódu uloženou v samostatném souboru. Vedle modulů vytvořených v *Pythonu* však existují celé knihovny modulů vytvářené v zájmu maximální efektivity v jiných programovacích jazycích, především pak v jazyku C. Tam to může být s rozdělením modulů do souborů maličko jiné.

Vše je součástí nějakého modulu

Jak už jsme si řekli, v OOP je vše objekt, takže i modul je objekt. Vše, co je v daném modulu definováno, mu patří. Každý objekt je vytvořen v nějakém modulu. Platí to i pro objekty, které vytváříme v interaktivním režimu. Modul, jehož součástí je vše, co vytvoříme v interaktivním režimu, se jmenuje `__main__`.

Chceme-li pracovat s atributem nějakého modulu, musíme název atributu kvalifikovat názvem daného modulu. Z tohoto pravidla existují dvě výjimky – kvalifikovat nemusíme:

- atributy modulu `builtins`, v němž jsou definovány všechny zabudované třídy a funkce,
- atributy modulu, v němž se nachází kód, který tyto atributy používá.

Prozatím jsme mohli kvalifikaci vynechávat, protože jsme využívali výše uvedené výjimky: pracovali jsme pouze s názvy, které jsme definovali v interaktivním režimu (v modulu `__main__`), anebo s názvy z modulu `builtins`.

Dva názvy objektů

Na problematiku oslovování objektů, o níž jsem před chvílí hovořil, se můžeme dívat i jinak – můžeme prohlásit, že každý objekt má vlastně dva názvy:

- Jednoduchý název uvedený v definici objektu, tj. v příkazu, jímž daný objekt vytváříme. U proměnné je to název zadaný v příkazu, kde ji inicializujeme, u funkce je to název použitý v její hlavičce.
- Úplný název doplněný o všechny předpony – kvalifikace. Prozatím jsme kvalifikovali názvem modulu, ale časem uvidíte, že spektrum je mnohem širší, takže úplný název může obsahovat celou sérii kvalifikací.

Zdrojový soubor

Text zadávaných příkazů označujeme jako **zdrojový kód** (slangově *„zdroják“*). S výjimkou modulu `__main__`, jehož zdrojový kód je tvořen naší konverzací se systémem, se zdrojový kód všech ostatních modulů ukládá do zdrojových souborů. Zdrojový soubor s programem v *Pythonu* je obyčejný textový soubor s příponou `py`. Název zdrojového souboru současně definuje název daného modulu, a proto musí vyhovovat podmínkám pro identifikátory.

Přeložený soubor

Kdyby měl systém pracovat se zdrojovými soubory, ztrácel by jejich interpretací příliš mnoho času. Proto se zdrojový text modulu před jeho prvním použitím přeloží do interního tvaru, který je možné interpretovat mnohem rychleji. Systém vytvoří v podsložce `__pycache__` složky se zdrojovým souborem stejnojmenný soubor s příponou `pyc` (zkratka z python compiled), do nějž uloží přeložený kód.

Když pak budete chtít později s daným kódem znovu pracovat, *Python* se nejprve podívá do podsložky `__pycache__`. Najde-li tam přeložený soubor, který nebude starší než příslušný zdrojový soubor, použije jej. Bude-li přeložený soubor starší nebo tam nebude vůbec, *Python* zdrojový kód přeloží, přeložený kód do této složky uloží a následně použije.

5.2 Příkaz import

Před chvílí jsme si řekli, že názvy definované ve stejném modulu jako příkaz, který je používá, nemusíme kvalifikovat. Pro použití názvů definovaných v jiných modulech však samotná kvalifikace nestačí. Chceme-li v modulu používat atribut nějakého jiného modulu, musíme o něj nejprve daný modul (tj. jeho vlastníka) požádat.

Abychom mohli nějaký modul o něco požádat, musí být nejprve v paměti objekt, který bude daný modul reprezentovat. Tento objekt se vytvoří (a daný modul zavede) příkazem `import`, jenž má několik variant.

Čistý import jiného modulu

Nejjednodušší verze začíná klíčovým slovem `import`, za nímž následuje identifikátor importovaného modulu, který musí být shodný s názvem souboru, v němž se nachází přeložený či zdrojový kód daného modulu:

```
import  název_souboru_s_importovaným_modulem
```

Tento příkaz importuje daný modul. Vytvoří objekt, který bude daný modul reprezentovat, a odkaz na tento objekt uloží do proměnné nazvané jako daný modul. Od této chvíle můžeme jeho atributy používat. Musíme je však pokaždé kvalifikovat názvem proměnné s odkazem na objekt jejich modulu.

Ukažme si to na příkladu. *Python* obsahuje modul `turtle` definující třídu `Turtle`, která nám zprostředkuje práci se světem počítačové želvy. Zavoláme-li tuto třídu jako bezparametrickou funkci, otevře okno se světem želvy schopné kreslit želví grafiku, vytvoří objekt želvy zobrazovaný jako hlavička šipky, umístí ho do středu otevřeného okna a vrátí jako svoji funkční hodnotu.

Ve výpisu 5.1 je na řádku 1 pokus o vytvoření nové želvy – instance třídy `Turtle` nacházející se v modulu `turtle`. Jak vidíte z reakce na řádcích 2 až 5, příkaz vyvolal chybu a systém tvrdí, že název `turtle` nezná. Proměnná s odkazem na modul totiž ještě neexistuje. Jinými slovy: systém se nezná k žádnému jinému modulu, dokud daný modul neimportujeme.

Na řádku 6 byl tedy modul importován, tj. byla vytvořena proměnná pojmenovaná názvem modulu a do ní byl uložen odkaz na modul. Příkaz na řádku 7 tentokrát prošel. Při jeho vykonání se otevřelo okno, v jehož středu se objevila malá šipka symbolizující želvu.

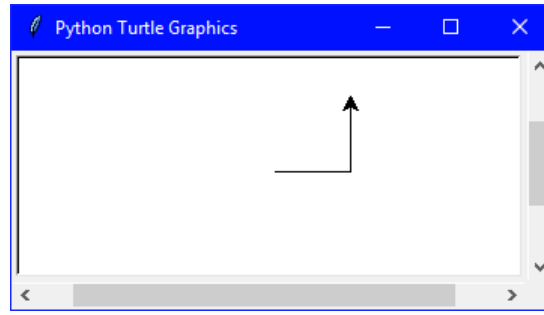
Abyste si ověřili práci s objekty z jiného modulu, je na řádku 8 příkaz k přesunu želvy o `50` bodů vpřed následovaný příkazy k jejímu otočení vlevo a dalšímu posunu. Výslednou podobu okna želví grafiky najdete na obrázku 5.1.

Připomínám, že příkazem na řádku 7 jsme definovali proměnnou `t`, která patří aktuálnímu modulu (modulu `__main__`). Když s ní proto chceme pracovat, nemusíme ji kvalifikovat. To, že proměnná odkazuje na objekt v nějakém modulu, je už její interní věc. My se na řádku 8 obracíme na proměnnou, o níž víme, že odkazuje na objekt definující metody `forward()` a `left()`. Volání této funkce proto stačí kvalifikovat názvem proměnné odkazující na oslovovaný objekt (želvu).

Výpis 5.1: *Importování modulů*

```
>>> turtle.Turtle()   # Před importem modulu jsou jeho atributy nepoužitelné
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    turtle.Turtle()   # Před importem modulu jsou jeho atributy nepoužitelné
NameError: name 'turtle' is not defined
>>> import turtle
>>> t = turtle.Turtle()      # Otevře se okno, v jehož středu se objeví želva
>>> t.forward(50); t.left(90); t.forward(50)   # Nakreslí čáru doprava a vzhůru
>>>
```

S želví grafikou přišel v roce 1969 jazyk Logo určený pro vstupní kurzy programování. Ten se stal v 80. letech po nástupu osobních počítačů velmi populárním a s ním i jeho želví grafika. Moduly se želví grafikou již dnes existují pro většinu jazyků používaných ve vstupních kurzech programování. Mimo jiné je tento modul součástí standardní instalace *Pythonu*.



Obrázek 5.1:
Okno světa želvy po provedení zadaných příkazů

5.3 Import modulu pod jiným názvem

Někdy nám originální název modulu z nejrůznějších důvodů nevyhovuje, např. je příliš dlouhý nebo hrozí kolize s nějakým používaným názvem. Mohli bychom samozřejmě uložit odkaz na modul do proměnné s jiným názvem, ale mnohem jednodušší je vytvořit tuto proměnnou hned v rámci importu. Dosáhneme toho např. tak, že v příkazu `import` přidáme za název souboru s kódem importovaného modulu klíčové slovo `as` následované názvem proměnné, do níž se bude odkaz na modul ukládat:

```
import  název_souboru_s_kódem_modulu  as  název_proměnné
```

Pojďme si vše vyzkoušet. Ve výpisu 5.2 je na řádku 1 uložen odkaz na importovaný modul do proměnné `želva`.

Když pak na řádku 2 kvalifikuji tímto názvem třídu Turtle, systém zjistí, na který modul proměnná odkazuje, v něm najde třídu Turtle, jejíž instanci vytvoří a odkaz na vytvořený objekt uloží do proměnné ž.

Na řádku 3 pak požádáme objekt odkazovaný touto proměnnou, aby o 50 bodů couvl, otočil se vlevo a popojel o 50 bodů vpřed.

Příkaz na řádku 4 využívá toho, že proměnná s odkazem na modul je zcela obyčejná proměnná. Vytvoří novou proměnnou korytnačka, což je slovenský název pro želvu, a uloží do ní odkaz na modul s želví grafikou.

Příkaz na řádku 6 pak vyrobí třetí želvu, kterou příkaz na řádku 6 otočí vlevo a popojede s ní o 50 bodů vpřed.

Výpis 5.2: *Uložení odkazu na modul pod jiným názvem*

```
1 >>> import turtle as želva    # Vytvoří proměnnou želva, do níž uloží odkaz na
  modul
2 >>> ž = želva.Turtle()        # Ve středu okna se objeví nová želva
3 >>> ž.forward(-50); ž.left(90); ž.forward(50)    # Nakreslí čáru doleva a vzhůru
4 >>> korytnačka = želva        # Korytnačka = želva slovensky
5 >>> k = korytnačka.Turtle()   # Ve středu okna se objeví třetí želva
6 >>> k.left(90); k.forward(50) # Nakreslí čáru vzhůru
7 >>>
```

To, že ukazuji, že se na modul dá odkazovat z několika proměnných, neberte jako doporučení, ale pouze jako ukázku toho, co se při troše nepozornosti může stát. V zájmu přehlednosti programu se proto snažte přiřadit importovanému modulu jen jeden název a důsledně používat pouze ten.

Přímý import vyjmenovaných objektů

Nevýhodou příkazů import probraných v předchozí pasáži je pro řadu programátorů to, že při oslovování objektů v daném modulu musejí být názvy oslovovaných objektů kvalifikovány názvy jejich modulu (přesněji názvem proměnné s odkazem na daný modul). Programátoři jsou totiž známí tím, že se snaží minimalizovat počet znaků, které jsou nuceni napsat.

Příkaz import má proto ještě druhou podobu, s jejíž pomocí importují zadané objekty přímo a při následném použití je pak již nemusejí kvalifikovat, protože vytvoří v aktuálním modulu proměnnou, do níž se uloží odkaz na importovaný objekt. Tento příkaz má podobu:

```
from název_souboru_s_kódem_modulu import název_objektu as název_proměnné
```

Nebudeme-li chtít objekt přejmenovávat, můžeme část začínající as vynechat a odkaz na importovaný objekt se uloží do stejnojmenné proměnné.

Počítejte s tím, že tímto importem vytvoříte proměnné s odkazy na atributy modulu, ale nevytvoříte proměnnou odkazující na objekt modulu. Po daném modulu proto nemůžete nic chtít, dokud jej neimportujete příkazem import, který vytváří proměnnou s odkazem na objekt modulu.

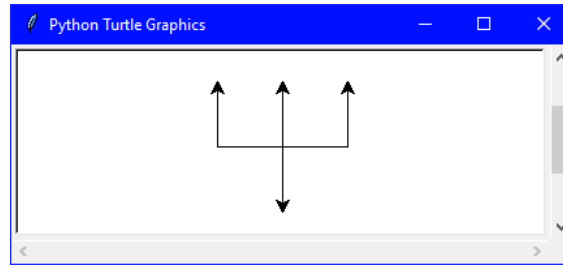
Příkaz umožňuje zadat více importovaných objektů současně, přičemž některé (nebo všechny) mohou mít svoji část `as` a jiné ne. Informace o jednotlivých importovaných objektech se pak oddělují čárkami.

Pojďme si opět vše ukázat. Kdybychom chtěli s želvou vytvářet něco většího, určitě by nás nutnost neustálé kvalifikace obtěžovala. Bylo by proto vhodné využít možnosti příkazu `from … import`, který nám navíc umožní třídy želvy pojmenovat tak, aby si její název studenti lépe pamatovali.

Ve výpisu 5.3 je na řádku 1 z modulu `turtle` importován odkaz na třídu `Turtle`, jenž je uložen do proměnné nazvané `Želva`. Na řádku 2 pak je importovaný objekt použit a vytvoří se nová, tentokrát již čtvrtá želva. Ta pak na řádku 5 zobrazí čáru dolů. Výsledek těchto experimentů si můžete prohlédnout na obrázku 5.2.

Výpis 5.3: *Import zadaných objektů ze zadaného modulu*

```
1 >>> from turtle import Turtle as Želva  # Proměnná Želva odkazuje na třídu
  turtle.Turtle
2 >>> žž = Želva()                        # Vytvoří se čtvrtá želva
3 >>> žž.right(90); žž.forward(50)       # Nakreslí čáru dolů
4 >>>
```



Obrázek 5.2:
Okno světa želvy po provedení zadaných příkazů

5.4 Vytvoření vlastního modulu

Ukončíte-li interpret *Pythonu*, budou zadané definice ztraceny. Budete-li je příště potřebovat, musíte je zapsat znovu. Chcete-li proto napsat poněkud delší program, je lepší použít pro přípravu vstupu textový editor (např. IDLE) a vytvořený soubor pak spustit, a to buď v interaktivním režimu, anebo přímo. Je-li program delší, můžete jej

rozdělit do několika souborů a usnadnit tak jeho následnou údržbu. Výhodou uložení programu do souboru je, že definované (a pokud možno i vyzkoušené) definice můžete použít v dalších programech.

Jak už jsme si řekli, jako modul označujeme v *Pythonu* část kódu definovanou v samostatném souboru. Modul může obsahovat libovolné příkazy včetně přiřazení, definic funkcí a definic tříd. Všechny tyto příkazy se provedou při zavádění modulu do paměti. Po zavedení je tak modul vlastníkem vytvořených objektů a můžeme jej o ně žádat, jak jsme to dělali v předchozí podkapitole s atributy modulu `turtle`.

Ve výpisu 5.4 je zobrazen modul `m05a_module_demo`, který najdete v doprovodných programech a který můžete otevřít v libovolném textovém editoru včetně IDLE. Pojďme si jej projít.

Název modulu

Název modulu má být co nejkratší a měl by obsahovat pouze malá písmena. Pokud to zvýší čitelnost, může obsahovat i znaky podtržení, ale ve standardní knihovně jsou i moduly s názvy z několika slov bez podtržení – např. `simpledialog`.

Kódová stránka

První dva řádky jsou komentářové a nejsou povinné. Řádek `1` má dva úkoly vyvolané zmateným používáním kódování v operačním systému *Windows*. *Python* sice od verze 3 nastavil jako standard kódování UTF-8, nicméně *Windows* z důvodů zpětné kompatibility s něčím, co zavedly někdy před třiceti lety, přepínají v průběhu práce mezi několika kódovými stránkami a občas tím způsobují chaos. Řádek `1` proto obsahuje text, který:

- Nabízí potřebné znaky editorům, které umějí nastavit používanou kódovou stránku automaticky podle několika prvních znaků.
- Ukazuje uživateli sady ne-ASCII znaků, aby si mohl zkontrolovat, že zdrojový kód byl otevřen se správně nastavenou kódovou stránkou a nemusí proto editoru vysvětlovat, že má text načíst v kódování UTF-8, ani se obávat překvapení vyplývajících z dezinterpretace zadaného znaku.

Řádek `2` je informační a obsahuje název daného zdrojového souboru a cestu k němu. Pokud vás tyto řádky obtěžují, můžete je oba vyhodit. V příštích výpisech je již nebudu zobrazovat, i když v doprovodných programech budou.

Dokumentační komentář

String zadaný na řádcích `3–7` představuje tzv. *dokumentační komentář* (*docstring*) daného modulu (proto je také zobrazen jako komentář), který by měl obsahovat důležité informace, jež by měl vývojář o daném modulu vědět.

Jak jistě odhadnete, i modul si svůj dokumentační komentář pamatuje. Když se budete na dokumentaci ptát, získáte obdobné výsledky jako v případě třídy: při dotazu na atribut `__doc__` získáte vlastní dokumentační komentář a při zavolání funkce `help()` obdržíte vedle tohoto komentáře ještě seznam dokumentací všech dokumentovaných atributů plus seznam zbylých atributů s jejich aktuálními hodnotami.

Výpis 5.4: *Modul `m05a_module_demo` sloužící k demonstraci chování Pythonu při zavádění modulu*

```
1  #Příliš žluťoučký kůň úpěl ďábelské ó - PŘÍLIŠ ŽLUŤOUČKÝ KŮŇ ÚPĚL ĎÁBELSKÉ Ó
2  #Q:/65_PGM/65_PYT/m05a_module_demo.py
3  """
4  Modul demonstrující základní pravidla pro tvorbu modulů,
5  jejich zavádění a následnou práci s nimi.
6  """
7
8  print('===== START načítání modulu', __name__, '=====')
9
10 text = 'Datový atribut modulu'
11 print("Při zavádění modulu se hodnoty výrazových příkazů nezobrazují")
12 text  # Výrazový příkaz => při zavádění modulu se nevytiskne
13 print('Chcete-li objekt zobrazit, musíte jej explicitně vytisknout\n',
14       f"{text=}")
15
16 def m_fce(arg='Nezadán') -> None:
17     """Samostatně definovaná funkce = funkční atribut modulu."""
18     print(f'Spuštěna funkce m_fce({arg=}) definovaná v modulu {__name__}')
19
20 class MCls:
21     """Třída definovaná v modulu."""
22
23     @staticmethod
24     def s_mtd() -> None:
25         """Statická metoda třídy MCls."""
26         print(f'''
27     Spuštěna metoda {MCls.s_mtd.__name__}()
28         definované ve třídě {MCls.__name__}
29            definované v modulu {__name__}''')
30
31 print('===== KONEC načítání modulu', __name__, '=====')
```

Zadané příkazy

Zbytek zdrojového kódu začíná a končí příkazy tisku (řádky 8 a 31), které při jeho importu oznamují začátek a konec jeho zavádění. Až získáte jisté zkušenosti, můžete tyto řádky smazat nebo je upravit do podoby, o níž budu hovořit v podkapitole *19.7 Kontrolní tisky* na straně 228. Na začátku vaší programátorské dráhy však doporučuji u prvních několika modulů takovéto řádky použít, i když budou třeba zakomentované. Jejich přítomnost usnadní hledání případných chyb.

Další příkazy definující datový atribut (řádek 10), funkci/metodu (řádky 16-18) a třídu (řádky 20-29) již pouze demonstrují, jaké definice lze v modulu očekávat. Příkazy na řádcích 11-14 pouze ukazují, že výsledky výrazových příkazů se nezobrazují a budete-li je chtít (nejspíš pro kontrolu) zobrazit, musíte je explicitně vytisknout.

Tady bych vám připomněl pasáž *Editační okno* na straně 37, kde jsem se zmiňoval o příkazu **Option → Show Code Context**. Zkuste tento příkaz zadat a sledujte, jak se bude měnit obsah podšeděné oblasti u horního okraje okna, když budete obsah souboru postupně posouvat nahoru a hlavičky začnou opouštět zobrazovanou oblast.

5.5 Práce s vytvořeným modulem

Modul je vytvořen, můžeme jej použít. Ve výpisu 5.5 je záznam seance, při níž je modul importován a následně použit.

Příkaz importu na řádku 1 zadává současně uložení odkazu na importem vytvořený objekt modulu do proměnné `m05a`. Po spuštění importu se začnou provádět jednotlivé příkazy modulu, mezi nimiž jsou v našem případě i kontrolní tisky, které se vypsaly na řádcích 2-6.

Výpis 5.5: *Použití modulu `m05a_module_demo`*

```
 1 >>> import m05a_module_demo as m05a
 2 ===== START načítání modulu m05a_module_demo =====
 3 Při zavádění modulu se hodnoty výrazových příkazů nezobrazují
 4 Chcete-li objekt zobrazit, musíte jej explicitně vytisknout
 5  text='Datový atribut modulu m05a_module_demo'
 6 ===== KONEC načítání modulu m05a_module_demo =====
 7 >>> m05a
 8 <module 'm05a_module_demo' from 'Q:\\65_PGM\\65_PYT\\m05a_module_demo.py'>
 9 >>> m05a.text
10 'Datový atribut modulu m05a_module_demo'
11 >>> m05a.MCls
12 <class 'm05a_module_demo.MCls'>
13 >>> m05a.MCls.s_mtd()
14
15     Spuštěna metoda s_mtd()
16         definované ve třídě MCls
17            definované v modulu m05a_module_demo
18 >>> from m05a_module_demo import *
19 >>> MCls.s_mtd()
20
21     Spuštěna metoda s_mtd()
22         definované ve třídě MCls
23            definované v modulu m05a_module_demo
24 >>>
```

Proměnná s odkazem na objekt modulu

Po doběhnutí importu můžeme proměnnou, do níž jsme uložili odkaz na objekt modulu, používat ke kvalifikaci atributů modulu. Když je na řádku 7 zadán samotný název této proměnné, vypíše se podpis modulu. Všimněte si, že modul zná svoje jméno a umístění svého zdrojového souboru.

Modul sice své jméno zná, ale program (přesněji překladač či interpret) je nezná, takže se jím kvalifikovat nedá. Ledaže byste použili čistý import, který už nic neimportuje (objekt modulu je již vytvořen), ale vytvoří proměnnou pojmenovanou stejně jako modul a uloží do ní odkaz na objekt modulu.

Na řádku 11 je výrazový příkaz žádající o vypsání podpisu třídy. Všimněte si, že v podpisu je její úplný název obsahující i název modulu, jehož je atributem.

Oprava načteného modulu

Když se na řádku 13 spouští její statická metoda `s_mtd()`, mohlo by nám vadit, že tisk začíná prázdným řádkem. Pro opravu lze využít toho, že jsme si v pasáži *Víceřádkové stringy* na straně 42 řekli, že končí-li řádek zpětným lomítkem, nevkládá se mezi části na tomto a následujícím řádku přechod na nový řádek.

Vložíme proto na konec řádku 26 ve výpisu 5.4 zpětné lomítko a při té příležitosti také zakomentujeme (nebo smažeme) řádky 11–14, které už jen zbytečně zaclánějí.

Zadáme nový import. S tím to ale nebude tak jednoduché. Reakce na příkaz na řádku 18 dokazuje, že pokus o nový import již importovaného modulu neudělá nic, protože systém ví, že objekt daného modulu je již vytvořen a může se používat.

Příkaz `import … from` pouze zařídí, že se vytvoří a inicializují proměnné odkazující na importované atributy, abychom je mohli používat bez kvalifikace. O nezměněné činnosti opravované funkce nás přesvědčí i reakce na příkaz na řádku 19.

5.6 Opětovné načtení opraveného modulu

Nechce-li se nám restartovat systém, protože máme některé věci rozdělané, musíme postupovat tak, jak ukazuje výpis 5.6. V něm se na řádku 1 importuje funkce `reload()` z modulu `importlib`. Té zadáme v argumentu odkaz na existující objekt modulu. Funkce příslušný modul znovu načte a pro případ potřeby vrátí odkaz na načtený modul jako svoji funkční hodnotu.

Když funkci na řádku 2 zavoláme, tak vidíme, že úvodní a závěrečný tisk modulu se zobrazily (ty zbylé jsme zakomentovali) a na řádku 5 se pak zobrazil podpis návratové hodnoty = načteného modulu.

Pak vám ale asi bude divné, proč volání opravené metody na řádku 8 pořád zobrazuje prázdný řádek. A ještě divější vám asi bude, že kvalifikované volání na řádku 11 jej již nezobrazuje.

Vysvětlení je jednoduché. Při opětovném načtení se znovu přeloží zdrojový kód modulu a znovu se provedou všechny příkazy. Vzniknou tak nové objekty atributů. Proměnné vytvořené příkazem `import … from` však stále odkazují na staré verze těchto atributů.

Musíme proto zadat znovu příkaz `import … from`, aby se do daných proměnných načetly nové odkazy (řádek 15). Pak už volání metody na řádku 16 spustí kód nově definované verze a vše bude v pořádku.

Výpis 5.6: *Načtení opravené verze modulu* `m05a_module_demo`

```
>>> from importlib import reload
>>> reload(m05a)
===== START načítání modulu m05a_module_demo =====
===== KONEC načítání modulu m05a_module_demo =====
<module 'm05a_module_demo' from 'Q:\\65_PGM\\65_PYT\\m05a_module_demo.py'>
>>> MCls.s_mtd()

    Spuštěna metoda s_mtd()
        definované ve třídě MCls
           definované v modulu m05a_module_demo
>>> m05a.MCls.s_mtd()
    Spuštěna metoda s_mtd()
        definované ve třídě MCls
           definované v modulu m05a_module_demo
>>> from m05a_module_demo import *
>>> MCls.s_mtd()
    Spuštěna metoda s_mtd()
        definované ve třídě MCls
           definované v modulu m05a_module_demo
>>>
```

5.7 Shrnutí

Záznam komunikace s interpretem probíhající v této kapitole najdete v trojici souborů nazvaných `m05__modules`.

Část B
Připravujeme aplikaci

V druhé části připravíme nástroje pro následné vytvoření naší aplikace – textové konverzační hry. Nejprve se dozvíte něco o návrhu objektové architektury, pak začneme architekturu aplikace skutečně navrhovat a připravíme si její zárodek. Poté vám představím metodiku TDD a podle jejího doporučení připravíme test naší budoucí aplikace, který by nám měl v následujících etapách vývoje pomoci výrazně zefektivnit její vybudování. Při jeho přípravě se seznámíte s kontejnery a naučíte se navrhovat kód, který bude umět na základě vyhodnocení aktuálního stavu zvolit správné pokračování. Současně se dozvíte, jak reagovat na situace vzniklé jako důsledek chyby v programu nebo jiné mimořádné události.

Kapitola 6
Základy objektové architektury

Co se v kapitole naučíte

Tato kapitola bude opět spíše teoretická. Má vás seznámit s termíny, které budu v delším textu používat, a především pak s některými zásadami, jež se vám budu snažit ve zbytku knihy vštípit.

Kdo preferuje postup, při němž se nejprve řeší praktický příklad, a teprve poté se vysvětluje, proč byl řešen právě takto, může tuto kapitolu přeskočit a dodatečně se k ní vrátit.

6.1 Předmluva

Schopnosti současných programovacích jazyků a vývojových nástrojů umožňují i relativním začátečníkům poměrně snadno vyvinout poměrně složitý program. Základní společnou nectností řady programů vyvíjených začínajícími programátory jsou „velké oči" jejich autorů. Chtějí hned od počátku vyvinout dokonalý program s řadou nabízených možností a schopností, avšak jejich nezkušenost vede záhy k tomu, že se jejich monstrózní návrh začne hroutit vlastní vahou.

Společnou nevýhodou takovýchto programů je, že vznikají obdobně chaoticky a živelně jako slum. Kdykoliv se v nich objeví nový obyvatel (požadavek na novou funkcionalitu programu), najde si nějaké volné místo a postaví si přístřešek z toho, co je zrovna po ruce nebo co cestou posbírá (co programátor najde na Googlu).

Obdobně zpočátku vznikaly i profesionální programy. S růstem vyvíjených programů softwaroví vývojáři pochopili, že před vlastním návrhem a zakódováním vyvíjeného programu je třeba provést důkladnou analýzu řešeného problému. S rostoucí složitostí vyvíjených programů se však doba potřebná pro analýzu, návrh a vývoj programu prodlužovala, takže se v důsledku různých vnějších změn častokrát stalo, že se během prací na programu změnily požadavky a celý cyklus bylo třeba zopakovat.

Postupně proto vznikaly nejrůznější metodiky, které měly vývoj zefektivnit. V průběhu devadesátých let se objevila řada metodik, které byly schopny průběžně akceptovat změny v zadání a přizpůsobovat tomu vývoj. Tyto metodiky jsou souhrnně označovány jako *agilní metodiky*. Značná část v současnosti vyvíjených programů je vyvíjena některou z těchto metodik.

6.2 Architektura

Základem úspěchu je takový návrh programu, který předjímá možné změny zadání a minimalizuje tím úsilí potřebné k zapracování těchto změn. S rostoucí velikostí a složitostí softwarových systémů se návrh a specifikace architektury systému stává významnějším problémem než volba algoritmů a datových struktur.

Architektura má řadu definic, které nejsou vzájemně zcela konzistentní. V této knize přijmeme následující: *Architektura programu definuje strukturu součástí programu, jejich vzájemné vazby, principy a předpisy určující jejich návrh a vývoj v průběhu času.*

Chcete-li volnější „definici“, tak bychom mohli říci, že *architektura programu je to, co se na něm v průběhu času nemění.*

Jinými slovy: není důležité, jakou definici architektury přijmeme. Důležité je to, abychom se naučili navrhovat své programy tak, abychom toho v nich při měnícím se zadání museli měnit co nejméně.

6.3 Hlavní zásady návrhu

Při návrhu programu bychom se měli řídit několika důležitými zásadami, jejichž dodržování nám umožní efektivně vyvíjet programy, které budou spolehlivé a v budoucnu snadno rozšiřitelné.

Tyto zásady se v začátečnických kurzech většinou neprobírají, což vede v konečném důsledku k tomu, že si je později osvojí jen skupinka těch neschopnějších. Ostatní „bastlí“ většinu svého života své stále obludnější začátečnické programy, které se po čase začnou hroutit vlastní vahou.

Nebudu vám je teď představovat všechny, protože pro jejich pochopení a docenění je třeba jistá zkušenost. Budu vás s nimi proto seznamovat postupně. Další zásadu představím vždy poté, co probereme teorii, která je k jejímu pochopení potřeba.

Připravenost na změny

Svým studentům s oblibou říkám: *„Jedinou konstantou současného programování je jistota, že zadání se brzy změní.“* Naučte se proto programovat tak, aby vás tyto změny zadání nezaskočily, ale dokázali jste je do vyvíjeného programu rychle zapracovat.

Tato zásada (i když trochu jinak formulovaná) je klíčovou zásadou *agilního programování*, což je v současné době převažující přístup. Je však třeba přiznat, že mnozí programátoři o sobě sice tvrdí, že programují agilně, ale ve skutečnosti bychom jejich způsob práce označili spíše jako *hurá přístup*.

Zásada připravenosti na změnu je velmi obecná, takže si programátoři vytvořili řadu úžeji zaměřených zásad. Pojďme si představit alespoň některé z nich.

CRIDP – maximální přehlednost

O této zásadě jsem hovořil již v pasáži *Změny přístupu k tvorbě programů* na straně 27. Říkal jsem, že jedním z důvodů je to, že programátoři věnují mnohem více času čtení programu než jeho psaní. Přitom jsem uváděl i známý citát Martina Fowlera.

Svým studentům vždy říkám: vybíráte-li ze dvou řešení, která se svojí efektivitou dramaticky neliší, vyberte si vždy to přehlednější a srozumitelnější, protože vám to v budoucnu ušetří hodně času.

KISS – maximální jednoduchost

KISS je zkratka z anglického *Keep it simple, stupid* (udržuj to jednoduché a prosté), případně *Keep it stupid simple*. Tato zásada nabádá vývojáře, aby udržoval program maximálně jednoduchý.

S porušováním této zásady se typicky setkáváme u začátečníků, kteří ještě neumějí odhadnout své schopnosti a možnosti. Svůj program se snaží navrhnout maximálně dokonalý, aniž by přemýšleli nad tím, zda umějí veškerou naplánovanou funkcionalitu naprogramovat a už vůbec nemají představu o tom, jak dlouho jim to bude trvat.

Známou skutečností totiž je, že dokud si zákazník program nevyzkouší, tak neví přesně, co bude opravdu potřebovat a jaké budou jeho koncové požadavky. To platí i v případě, že autor programu je svým vlastním zákazníkem.

Moderním trendem proto je navrhnout nejprve jenom tu nejzákladnější podmnožinu požadované funkcionality a po jejím rozchození pak postupně přidávat další. Přitom se často ukáže, že mnohé z původně požadovaných funkcí již nejsou potřeba a naopak se vynoří potřeba jiných, s nimiž se původně vůbec nepočítalo.

YAGNI – žádné zbytečnosti

YAGNI je zkratka z anglického *You aren't gonna need it* (vždyť to nebudeš potřebovat). Je příbuzná s předchozí zásadou. Nabádá vývojáře, aby do programu začleňovali pouze takové funkce, které jsou doopravdy potřeba, a ignorovali funkce charakterizované heslem „*to by se mohlo hodit*".

Zabudování veškerých takových funkcionalit je třeba odložit na dobu, kdy o ně zákazník doopravdy požádá, protože začne mít pocit, že je potřebuje (a je ochoten jejich vývoj zaplatit).

DRY – bez kopií

DRY je zkratka z anglického *Don't repeat yourself* (neopakuj se). Podle této zásady by se v programu neměly vyskytovat dva shodné, nebo velmi podobné úseky kódu (nebo dokonce více). Pokud něco takového hrozí, měly by se použít takové konstrukce, aby bylo možné daný kód naprogramovat na jednom místě, na které by se pak místa, kde má být použit, odvolávala.

Není-li tato zásada dodržována, tak je při výše zmíněných úpravách vždy třeba najít všechny výskyty podobného kódu a ve všech zanést vždy správnou opravu. To často vede k nepříjemným zavlečeným chybám.

Jednou z aplikací této zásady je pravidlo, že v programu by se neměly vyskytovat tzv. magické hodnoty zadávané jako literály, přičemž za magické hodnoty se považují jakákoliv čísla s výjimkou nuly a jedničky, různé texty s výjimkou prázdného stringu apod.

Pro každou takovouto hodnotu bychom měli definovat pojmenovanou konstantu a tu pak používat místo literálu. Literál použijeme pouze při definici dané konstanty.

Pokud např. posuneme začátek pracovní doby ze 7 hodin na 8, nemusíme obíhat celý program, hledat sedmičky a zjišťovat, která z nich označuje původní začátek pracovní doby a která např. počet dní v týdnu.

SoC – jediný zodpovědný

SoC je zkratka z anglického *Separation of Concerns* (oddělení zodpovědností). Tato zásada doporučuje definovat program tak, aby se funkcionalita jednotlivých částí co nejméně překrývala.

Zásada může být interpretována také tak, že každá funkcionalita by měla být naprogramována na jediném místě, tj. měl by za ni být zodpovědný jediný objekt. Když pak bude špatně fungovat, víme přesně, co máme opravit.

Tato zásada je poměrně úzce svázána s předchozí. Rozdíl spočívá v tom, že zásada DRY se zabývá tím, **jak** je něco zakódováno, kdežto zásada SoC řeší otázku, **proč** je třeba něco naprogramovat.

SRP – jediná zodpovědnost

SRP je zkratka z anglického *Single-responsibility principle* (princip jediné zodpovědnosti). Podle této zásady by objekt (modul, třída, funkce, …) měl být zodpovědný za jedinou věc. Jinak bychom mohli říci, že k požadavku na změnu jeho kódu smí vést pouze jeden důvod.

Tato zásada je doplňkem zásady SoC. Ta říkala, že za každou funkcionalitu má být zodpovědný jediný objekt, naproti tomu zásada SRP říká, že tento objekt má být zodpovědný za jedinou věc.

6.4 Návrhové vzory

Jednou z důležitých zásad produktivních programátorů je „nevynalézat již vynalezené“. Celé objektově orientované programování je „vyladěno“ k tomu, aby při řešení nových úloh programátor nemusel znovu vynalézat nové třídy, ale aby místo toho mohl znovu použít třídy, které již naprogramoval někdy dříve a/nebo které naprogramoval někdo jiný.

Jednou z cest zefektivnění vývoje programů a „nevymýšlení“ dříve vymyšleného je používání tzv. **návrhových vzorů** (anglicky design patterns), které bychom mohli charakterizovat jako doporučení, jak řešit některé typické, často se vyskytující úlohy.

Návrhové vzory jsou programátorskou obdobou matematických vzorců. Radí, jak řešit nějaké typy úloh. Netvrdí, že navrhované řešení je jediné možné, ale všichni vědí, že je přiměřeně univerzální a především ověřené časem.

Vezměme si třeba známý vzorec na řešení kvadratické rovnice. Ve škole jste se učili, že řešení rovnice:

$$ax^2 + bx + c = 0$$

získáte dosazením do vzorce:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Nikdo vás nenutí tento vzorec používat (pokud se z něj zrovna ve škole nezkouší). Existuje celá řada kvadratických rovnic, které lze řešit rychleji bez vzorce. Nicméně ve většině případů je použití tohoto vzorce zdaleka nejvýhodnější cestou k řešení. Tím, že se ho naučíte používat, získáte několik výhod:

- Získáte řešení rychleji, protože nebudete muset přemýšlet, jak úlohu vyřešit.
- Výrazně zmenšíte pravděpodobnost chyby, kterou byste mohli udělat, kdybyste řešení teprve vymýšleli.
- Se vzorcem často přichází terminologie, která usnadňuje komunikaci. Když spolupracovníkovi řeknete, že diskriminant je záporný, hned ví, co to pro řešení vašeho problému znamená, a nemusíte mu sáhodlouze vysvětlovat, jak je to s řešitelností kvadratické rovnice.

Obdobné výhody získáte i vhodným použitím návrhových vzorů. To je jeden z důvodů, proč jsou tak populární.

Oblibu návrhových vzorů odstartovalo v roce 1995 vydání knihy *Design Patterns* ([2]), která se stala velice rychle biblí všech objektově orientovaných programátorů.

Autoři této knihy bývají v literatuře často označováni jako „banda čtyř“ (anglicky Gang of Four – ve zkratce GoF). Narazíte-li proto někde na odkaz GoF nebo gang čtyř, jedná se určitě o odkaz na výše uvedenou knihu.

Kniha obsahuje 23 základních návrhových vzorů všeobecného použití. V počítačové literatuře autoři na tyto vzory často odkazují, takže se zkušený programátor bez jejich znalosti neobejde. Hlavním přínosem knihy však nebyl jen katalog základních návrhových vzorů, ale také demonstrace některých zásad, jimiž by se měl moderní programátor řídit.

Řada programátorů po přečtení této knihy (nebo některé z jejích následovnic) výrazně změnila styl svého programování. Bohužel většiny učebnic programování se tato změna nedotkla. Převážná většina učebnic programování je ve skutečnosti učebnicemi syntaxe a základního použití vybraného programovacího jazyka, a proto tyto učebnice žádné zmínky o návrhových vzorech neobsahují (případně jen okrajově) a počítají s tím, že se s nimi čtenář ve své další praxi seznámí.

Protože se domnívám, že znalost alespoň těch nejdůležitějších návrhových vzorů patří k základním znalostem objektového programátora, tak vás v dalším textu s některými seznámím, abyste se je naučili vnímat.

Vážnější zájemce bych odkázal na svoji knihu [9], která sice uvádí demonstrační příklady v *Javě*, ale to na podstatě vzorů nic nemění. Rád bych ji ale v nejbližší době aktualizoval i pro *Python*, který přece jenom přistupuje k řadě problémů poněkud odlišně.

6.5 Antivzory

Návrhové vzory ukazují, jak se daný problém řešit má. Vedle nich ale existuje i řada oblíbených postupů, kterou jsou vzorovými ukázkami toho, jak se něco řešit určitě nemá.

Za všechny uvedu ten nejoblíbenější, označovaný jako *Velká koule bláta* (anglicky *Big ball of mud*). Při něm programátor bez přemýšlení plácá kód na hromadu, která nemá žádnou strukturu. V tom zmatku se pak vyzná pouze on, což zdůvodňuje nedostatečnou inteligencí ostatních. Ve chvíli, kdy velikost programu přesáhne jistou mez, a ten se začne hroutit vlastní vahou, tak programátor změní zaměstnání a jde jinam plácat další hroudu.

Líbilo se mi, když jeden autor označil tento vzor jako *slum*. Jeho obyvatelé také skládají své příbytky z toho, co kde najdou. Když takový „slumový“ programátor potřebuje svůj kód vylepšit, dojde si na skládku *Google*, kde něco najde a ke svému programu to přilepí, aniž by chápal, jak to funguje. Když posléze zjistí, že daný přílepek funguje z nějakých důvodů jenom v úterý dopoledne, naplánuje předvádění svého díla na tuto dobu. Jakékoliv pozdější stížnosti pak svádí na to, že si dotyčný něco špatně nainstaloval či udělal jinou chybu.

Antivzory mívají často lidovější názvy jako např. *Švýcarský nožík*, *Zlaté kladivo* nebo *Proud lávy*. Na podrobnější výklad zde bohužel není prostor. Snad se „potkáme“ ve výše zmíněné chystané knize o návrhových vzorech v *Pythonu*.

6.6 Rozhraní versus implementace

Každý objekt má dvě tváře: rozhraní a implementaci.

- *Rozhraní* (anglicky *interface*) představuje to, co je o objektu veřejně známé a co se např. můžete dozvědět z jeho nápovědy, tj. zavoláte-li funkci `help()`, které předáte daný objekt jako argument.
- *Implementace* (anglicky *implementation*) pak představuje to, jak je zařízeno, že objekt umí to, co umí (pamatuje si údaje, provádí akce pod.).

PINI

Jednou z nejdůležitějších zásad moderního programování je pravidlo PINI, jehož název je zkratkou z anglického *Program to an Interface, not an Implementation*.

Podle tohoto pravidla musíme program navrhovat tak, abychom využívali jenom to, co je definováno v rozhraní daného objektu, a abychom naopak nevyužívali své znalosti jeho implementace.

Když se později upraví definice objektu, aniž by se změnilo jeho rozhraní, všechny spolupracující programy, které se drží tohoto pravidla, pracují bez problémů dál. U programů, které se jej nedrží, hrozí nebezpečí, že se začnou hroutit.

Nezveřejňované atributy

Pokud se o nějakém atributu domníváme, že slouží pouze pro interní potřebu daného objektu, a není proto vhodné o jeho existenci informovat okolí, je dobré označit daný atribut jako nezveřejňovaný. V *Pythonu* platí pravidlo, že atributy, jejichž názvy začínají jedním podtržítkem, jsou považovány za interní atributy daného modulu, resp. třídy. Platí pro ně:

- „Hvězdičková verze“ příkazu `from … import *` je neimportuje. Pokud je opravdu potřebujete, musíte je importovat explicitně.
- V dokumentaci daného objektu (modulu, třídy) vyvolané např. voláním funkce `help()` se o nich nedozvíte. Chcete-li se o nich něco dozvědět, musíte se podívat do zdrojového kódu daného objektu.

Jak vidíte, nejsou zcela nedosažitelné. Objekt se jimi nechlubí, ale pokud o nich víte a opravdu je potřebujete, můžete je použít.

V některých ostatních jazycích jdou s chráněním atributů pro interní potřebu ještě dále. Označují je jako soukromé (anglicky *private*) a překladač kontroluje, že je nepoužije nikdo, kdo není jejich majitelem.

Python ví, že každé pravidlo má své výjimky a že v některých výjimečných situacích (např. při testování) je třeba i s těmito atributy pracovat. Takovéto situace se sice v učebnicích neobjevují, ale existují. *Python* považuje programátora za svéprávného člověka, který si nebude pod sebou podřezávat větev nadužíváním těchto atributů, a tak mu řešení oněch nestandardních situací zbytečně nekomplikuje.

6.7 Návrh programu

Když klasicky vychovaný programátor obdrží od zákazníka zadání, začne hned přemýšlet nad tím, jak by je zakódoval. Tím ale zcela zmizí za mentálním obzorem zákazníka, který mu nemůže říci, že se jeho úvahy ubírají špatným směrem a že po něm chtěl naprogramovat něco jiného. Zákazník si objedná auto a má na mysli dodávku, kterou by přepravoval materiál. Programátor navrhne dokonalé auto, které může závodit ve formuli 1, jenom toho moc neuveze.

Doporučovaný postup je následující:

Účastníci

Při návrhu programu začneme tím, že navrhneme účastníky, tj. objekty, které budou v programu vystupovat. Při vhodné volbě termínů nás při tom může zákazník sledovat a usměrňovat.

Budeme-li programovat účtárnu, dozvíme se, že by v programu měly vystupovat faktury, zákazníci, termíny splatnosti, upomínky, penále a další objekty. Budeme-li navrhovat auto, budou tam vystupovat účastníci jako motor, podvozek, odpružení, ložná plocha, nosnost, výkon apod.

Schopnosti

V druhém kole bychom si měli ujasnit požadované schopnosti jednotlivých účastníků. Co by měli jednotliví účastníci umět, a co proto budeme muset následně definovat jako jejich metody.

I v této druhé etapě nás zákazník dokáže často sledovat. Objeví se sice jistě schopnosti, o nichž víme, že je objekt z nějakých systémových důvodů musí mít, i když zákazník jejich účel ani nutnost jejich existence nechápe, ale značná část z nich bude spadat do jeho oblasti znalostí. U nich nám může nastavovat mantinely, pokud budou plánované schopnosti za hranicí toho, co potřebuje a tím i za hranicí toho, co je ochoten zaplatit.

Vlastnosti

V další etapě bychom si měli ujasnit vlastnosti jednotlivých účastníků, což by mohlo následně vést k definici jejich atributů. Potřebuji-li např. kladivo, je rozdíl, zda potřebuji hodinářské, zámečnické či tesařské. I zde je nenulová pravděpodobnost, že nám bude zákazník moci s lecčíms poradit.

Kódování

Když jsme si ujasnili účastníky a jejich charakteristiky, můžeme přistoupit k jejich zapsání v kódu použitého programovacího jazyka. I zde bychom si však měli včas ujasnit, je-li daná část dostatečně jednoduchá k přímému zakódování, anebo je poněkud složitější a bylo by výhodné na ni znovu aplikovat postup vyhledání účastníků atd.

6.8 Druhy vytvářených objektů

Při rozhodování o tom, jaký druh objektu pro daný účel vytvořit, je zřejmé, že pro objekty, jichž bude více, bychom měli definovat třídu. U objektů, které se budou v celé aplikaci vyskytovat jenom v jednom exempláři, je rozhodování těžší. Obecně máme v *Pythonu* tři možnosti:

- Definovat objekt jako modul. Tato možnost bude asi nejjednodušší, ale na druhou stranu není vhodné mít celou aplikaci rozbitou do záplavy drobných modulů. Má proto smysl popřemýšlet, jestli by v zájmu maximální přehlednosti aplikace nebylo výhodnější sloučit několik definic „sólistů“ do jednoho modulu.
- Definovat třídu, která nebude mít žádnou instanci a bude sama oním objektem. Tato možnost by se mohla hodit ve chvíli, kdy bychom měli takovýchto „sólistů“ více a definice každého z nich by byla relativně jednoduchá, takže by se mohlo ukázat výhodné sloučit všechny do jednoho modulu.
- Definovat pro daný objekt třídu, která bude mít jedinou instanci – daný objekt. Po této možnosti sáhneme ve chvíli, kdy bychom tuto třídu mohli začlenit do dědické hierarchie a díky tomu zdědit implementaci některých funkcionalit. Tuto možnost zde nebudu rozebírat, ale v [11] je uvedeno několik možných způsobů realizace takovéto třídy.

6.9 Dva způsoby návrhu

Pro řešení složitějších úloh existují obecně dva principiální přístupy: návrh shora dolů a naopak návrh zdola nahoru. V praxi se většinou používá nějaká jejich kombinace, ale zkusím vám je nejprve představit v jejich čisté podobě.

Návrh shora dolů

Návrh metodou shora dolů (anglicky *top-down design*) vychází z přirozeného lidského postupu při řešení složitých problémů, který je označován jako *dekompozice problému* (*problem decomposition*) a je vlastně programátorskou aplikací známé římské zásady *divide et impera* neboli česky *rozděl a panuj*.

I pro nás je velice výhodné rozdělit si složitý problém nejprve na řadu dílčích podproblémů, ty vyřešit a z jejich řešení pak sestavit řešení problému původního. Pokud se nám bude zdát některý z podproblémů stále příliš složitý, není nic přirozenějšího než jej opět rozdělit. Proto bývá tento přístup někdy označován jako *postupný návrh* (*stepwise design*).

Po několika úrovních postupného zjemňování problému se nakonec musíme dostat do situace, kdy jednotlivé podproblémy budou již natolik jednoduché, že naprogramovat je bude pro nás hračkou, protože stačí vhodně poskládat hotové součásti.

O tomto způsobu návrhu můžeme říci, že v každém okamžiku víme, co má prvek dělat, ale zpočátku netušíme, jak to bude dělat, jaká je jeho struktura. To se ujasní až v průběhu další analýzy a následného návrhu a kódování.

Výhodou návrhu shora dolů je, že se můžeme od počátku tvářit, že je aplikace hotová, i když se budeme muset některým částem vyhýbat, anebo budou místo nich prozatím k dispozici pouze nějaké záslepky.

Nevýhodou je naopak to, že máme delší dobu k dispozici pouze nějaký polotovar, v němž nic pořádně nefunguje, takže si při testech musíme dávat velký pozor, abychom nezabloudili do nějaké části, která je závislá na něčem, co ještě není hotové.

Ve zbytku učebnice budeme vyvíjet jednoduchou textovou konverzační hru. V takovém případě bychom měli při návrhu shora dolů začít od modulu `game` a spouštět hned celou hru. Při tom bychom postupně objevovali, kterou část by bylo v danou chvíli optimální navrhnout. Tu část bychom se pokusili navrhnout, přičemž při jejím návrhu bychom postupovali obdobně.

Návrh zdola nahoru

Při návrhu metodou zdola nahoru (anglicky *bottom-up design*) vycházíme z úplně jiné filozofie řešení. Můžeme ji interpretovat tak, že při ní nám naopak připadá, že instrukční soubor použitého procesoru neodpovídá řešené úloze a my se pokoušíme vytvořit nad tímto instrukčním souborem soubor nový: soubor instrukcí, které nám umožní řešit naši úlohu snáze.

Při objektovém přístupu bychom začali návrhem objektů, které při své práci vystačí s objekty z dostupných knihoven a frameworků. Když je rozchodíme, začneme navrhovat objekty, které při své práci vystačí s objekty z knihoven a těmi již navrženými. Tak budeme pokračovat, dokud z hotových částí nesestavíme celou aplikaci.

Při návrhu textové konverzační hry zmíněné v předchozí pasáži (zájemci najdou stručné zadání v podkapitole *7.2 Zadání* na straně 103) bychom nejprve definovali třídu předmětů, s nimiž se ve hře manipuluje. Pak bychom definovali třídy prostorů, v nichž se dané předměty mohou vyskytovat, a třídu batohu, v němž hráč předměty přenáší. Poté bychom asi navrhovali svět tvořený těmito prostory a skončili bychom naprogramováním reakcí na zadávané příkazy.

Nevím, jestli je z předchozího textu dostatečně zřejmé, že návrh metodou zdola nahoru vyžaduje dobrou znalost dostupných knihoven a současně předvídavost, jaké požadavky na právě vytvářené objekty (třídy, moduly, …) se v horních patrech návrhu mohou objevit.

Osobně se domnívám, že pro začátečníky je vhodný pouze ve chvíli, kdy si pouze hrají a staví různé součásti bez přesného zadání, co má být cílem daného návrhu. Tak trochu podle hesla: *„Ono se ukáže, k čemu to může být dobré.“*

Porovnání

Závěrečné srovnání obou přístupů uvádím v tabulce 6.1, kterou jsem převzal z [7]. Termínem *základna* je v ní míněn souhrn nástrojů, které má programátor k dispozici, tj. prostředky vlastního jazyka rozšířené o nástroje v dostupných knihovnách.

Z tabulky můžete odvodit, kdy kterou z metod zvolit, případně jak je zkombinovat. Výše uvedený příklad se zákazníkem, jemuž nechceme se svým návrhem příliš brzy utéci za jeho mentální obzor, vede zcela zřejmě k přístupu shora dolů.

Naopak při návrhu programů spolupracujících s nějakými zařízeními (např. s chytrou domácností), při němž si musíte nejprve ujasnit, co vše můžete po takovém zařízení chtít a jak toho dosáhnout, je evidentní, že výhodnější bude naopak postup zdola nahoru.

My budeme při budování aplikace v následujících kapitolách používat metodu shora dolů a ukážeme si, jak při její aplikaci postupovat.

Tabulka 6.1: *Porovnání metody shora dolů a zdola nahoru*

	Shora dolů	Zdola nahoru
Podstatou postupu je	Analýza	Syntéza
Nový prvek je nám	Prostředkem	Cílem
Že nový prvek půjde nad danou základnou vytvořit	Nevíme jistě, spíše předpokládáme	Víme jistě
Jaké bude mít nový prvek vlastnosti	Víme přesně	Přizpůsobíme to možnostem dostupných knihoven
Jak bude nový prvek zkonstruován	Vyřešíme později	Musíme se tím zabývat hned
Konkrétní použití nového prvku	Známe	Nezajímá nás (může být libovolné)
Předpokládaná aplikace	V jednom projektu	V mnoha projektech
Návratnost vynaložené námahy	Brzká, jednorázová	Opožděná, postupná
Hlavní riziko	Požadujeme něco, co nepůjde naprogramovat nebo nebude efektivní	Vytvoříme něco, co se nebude dostatečně využívat

6.10 UML diagramy

Ve chvíli, kdy máme ujasněné účastníky s jejich schopnostmi a vlastnostmi včetně druhu objektu, který je bude reprezentovat, je vhodné dosavadní znalosti zobrazit ve tvaru, který je pro běžného člověka mnohem přehlednější a snáze zapamatovatelnější než text. Programátoři pro tyto účely používají UML diagramy, z nich pak především diagram tříd.

Účastníci zmiňovaní v minulé podkapitole jsou většinou instancemi nějakých tříd, v *Pythonu* může být takovým účastníkem i modul. V diagramu je každá třída či modul zobrazen(a) obdélníkem, vztahy mezi nimi se zobrazují různými druhy čar a šipek.

Stručný popis použitých součástí UML diagramů se objeví v podkapitole *14.1 Pravidla pro kreslení UML diagramů* na straně 181 před prvním složitějším UML diagramem. Na podrobnější výklad UML diagramů zde není prostor. Zájemce o hlubší vhled do UML diagramů bych odkázal na vynikající knihu [3], resp. anglický originál [6].

6.11 Shrnutí

V této kapitole se nevyskytovaly žádné doprovodné programy.

Kapitola 7
Návrh základní architektury

Co se v kapitole naučíte

V této kapitole se pokusíme převést verbální (slovní) zadání hry z první kapitoly do programové podoby tak, aby se požadavky tohoto zadání pokud možno přeměnily do požadavků na definici modulů, tříd a jejich atributů. V závěru kapitoly vás pak seznámím s koncepcí balíčků, jejíž znalost je pro tvorbu větších aplikací klíčová. Současně definujeme zárodek naší budoucí aplikace.

7.1 Koncepce vyvíjené aplikace

Myslím, že teorie již bylo dost a že je nejvyšší čas vše pomalu směřovat k praxi. V následujících kapitolách se postupně pokusíme realizovat jednoduchou aplikaci. V této kapitole vám pouze přiblížím ideu budované aplikace, abych mohl vysvětlit i některé zvláštní možnosti. V následující kapitole pak tuto koncepci shrneme do stručného a výstižného zadání, na jehož základě navrhneme architekturu aplikace.

Vytvářenou aplikací bude jednoduchá konverzační hra (adventura), při níž se hráč snaží v konečném počtu kroků dostat k předem zadanému cíli. V každém kroku zadá programu textový příkaz naznačující, co má v daném okamžiku udělat, a program v reakci na tento příkaz změní svůj stav a hráči odpoví. V odpovědi hráči poví, co provedl, a případně naznačí, v jakém stavu se po provedení tohoto příkazu nachází a jaké jsou hráčovy další možnosti. Bude to tedy takový rozhovor anebo – chcete-li – textový ping-pong: hráč pošle text příkazu, program vrátí text odpovědi, a tak stále kolem dokola, dokud hru nedohraje do konce (nezávisle na tom, zda dosáhne či nedosáhne požadovaného cíle hry), nebo dokud jej hra nepřestane bavit.

Instance hry bude přitom definována jako jedináček, který po ukončení jednoho běhu bude schopen opětovného spuštění nového běhu hry, aniž by bylo třeba ukončovat program.[8]

Hra bude probíhat ve virtuálním světě, v němž existuje několik prostorů, které spolu zadaným způsobem sousedí. Těmito prostory mohou být místnosti v budově, části krajiny, planety, etapy života apod. Hra musí umět provést akci realizující přechod z aktuálního prostoru do sousedního prostoru.

Za sousední prostor přitom považujeme takový, do nějž lze jednoduše přejít. Je-li třeba nejprve splnit nějakou podmínku, stane se onen potenciálně sousední prostor skutečně sousedním až po splnění dané podmínky. Tyto podmínky mohou být různé:

- Jsou-li sousedními prostory místnosti v budově, může být potřeba nejprve odemknout spojovací dveře.
- Jsou-li sousedními prostory části krajiny, můžeme se na sousední louku za potokem dostat např. až poté, co postavíme most přes potok.
- Jsou-li sousedními prostory semestry vysoké školy, kterou máte za cíl absolvovat, musíte odemknout následující semestr třeba tak, že dojdete na studijní oddělení a ukážete index s absolvovanými předměty, aby vás mohli zapsat do dalšího semestru.

V každém prostoru se mohou nacházet různé *h-objekty* (viz podšeděný blok). Některé z nich může hráč vzít, uložit je do pomyslného batohu, aby mu v budoucnu pomohly ke splnění nějakého pomocného úkolu nebo dokonce cíle celé hry. Hra musí proto umět provést akci, která přesune zadaný h-objekt z aktuálního prostoru (tj. prostoru, v němž se právě nachází hráč) do batohu, a současně akci, která naopak přesune h-objekt z batohu do aktuálního prostoru.

Akce × Příkaz

V předchozím textu vás možná zarazilo, že jsem střídavě hovořil o příkazech a o akcích. Dohodněme se, že termínem **akce** budu označovat druh činnosti, např. přesun z prostoru do prostoru nebo zvednutí h-objektu. Naproti termínem **příkaz** budu označovat konkrétní zadání, např. „`jdi koupelna`“ nebo „`zvedni meč`“.

Množství h-objektů, které se do batohu vejdou, je však omezené (to je jedna z omezujících podmínek hry). Akce pro přesun h-objektu z prostoru do batohu proto nesmí povolit překročení kapacity batohu.

Navíc musí ve světě hry existovat h-objekty, které není možné zvednout a přemístit do batohu (okno, potok, studijní referentka, …). Akce pro zvednutí h-objektu proto nesmí umožnit uložit do batohu nepřenositelný h-objekt.

[8] Návrhový vzor *Jedináček* probírá možnosti, jak zabezpečit, aby třída měla jedinou instanci.

Co to je *h-objekt*

Termín *h-objekt* používám z nedostatku nápadů na lepší pojmenování. Termín *objekt*, který by byl v dané situaci nejvýstižnější (ze základů OOP víme, že v simulovaném světě je všechno objekt), vyvolává problémy při potřebě odlišení obecného objektu v programu a objektu (*h-objektu*) ve hře.

Říkat mu *položka*, což by byl další možný překlad anglického *item*, zvoleného jako identifikátor pro tento druh objektů ve frameworku a ukázkovém programu, mi také nepřipadalo zcela vhodné. Pak by totiž mohla mít vaše hra v jeskyni tři položky: ohniště, meč a trpaslíka.

Stejně nevhodná mi připadala i synonyma *předmět* nebo *věc*, protože (jak jsem před chvílí naznačil) bychom pak mezi předměty/věci museli zařadit i pohádkové bytosti, osoby, zvířata a další tvory, kteří by se v našich hrách vyskytli a které bychom neradi označovali jako věci nebo předměty.

Nakonec jsem proto zakotvil u termínu *h-objekt* (= objekt hry), který mi připadá nejakceptovatelnější, na čemž jsme se shodli i se spolupracovníky. Budu-li citovat „klasika“: *„Můžete s tím nesouhlasit, můžete proti tomu protestovat, ale to je také vše, co s tím můžete dělat.“*

Mezi h-objekty mohou (ale nemusejí) být i takové, s nimiž může hráč komunikovat a případně od nich získat informaci potřebnou ke zdárnému pokračování. Tyto h-objekty mohou být jak živé (trpaslík, kouzelný dědeček, …), tak neživé (robot, knížka, magnetofon, …).

Mezi h-objekty v prostoru mohou (ale opět nemusejí) být i takové, které jsou za jistých okolností novým prostorem s jeho vlastními h-objekty. Takovýmto h-objekt-em-prostorem může být např. truhla, trezor, nalezený batoh apod. Tyto nestandardní prostory nemají žádné sousedy, a proto se do nich musíme „přesunout“ nějakým nestandardním příkazem – např. `otevři truhla` a zpět příkazem `zavři truhla`.

Takovýmto prostorem může být např. i deska stolu nebo okno. Do takového prostoru se přesunete např. příkazem `koukni_na_stůl`. Pak uvidíte jenom h-objekty, které se na stole nacházejí, a některý z nich se vám může hodit.

7.2 Zadání

Pojďme se tedy zamyslet nad tím, jaké objekty budou v naší hře vystupovat, a co proto budeme muset postupně navrhnout. Jeden ze způsobů návrhu doporučovaný začátečníkům je formulovat co nejstručněji (ale současně co nepřesněji) zadání a zvýraznit v něm všechna podstatná jména. To budou kandidáti na budoucí objekty a třídy objektů.

Volný popis hry v předchozí podkapitole není pro výchozí zdání programu optimální, protože je příliš „ukecaný“. Musíme jej proto zestručnit, aniž bychom přitom nějaké požadavky vypustili. Kromě toho bychom měli do výchozího zadání přidat i požadavky, které jsou v úvodním popisu „skryty mezi řádky“, anebo v něm vůbec nejsou, ale ze zkušenosti víme, že by se záhy objevily jako požadavky na rozšíření.

Takovým typickým požadavkem je např. rozšíření aplikace o základní nápovědu nebo možnost jejího předčasného ukončení a opětovného spuštění (restartu).

Výchozí zadání naší aplikace by pak mohlo vypadat např. následovně:

Vytvořte **aplikaci** realizující textovou konverzační **hru** splňující následující požadavky:

- **Hra** probíhá ve virtuálním **světě**, v němž existuje několik **prostorů**, které spolu zadaným způsobem sousedí.
- Ve hře **hráč** postupně zadává **příkazy**, na které **hra** odpovídá.
- **Hra** musí akceptovat **příkaz** realizující **přechod** z aktuálního **prostoru** do zadaného sousedního **prostoru**.
- Každý **prostor** má svůj **název**. Abychom se v **prostorech hry** vyznali, nesmějí v ní být dva **prostory** se stejným **názvem**.
- V každém **prostoru** se mohou nacházet různé **h-objekty**. Některé z nich je možné přenášet, jiné ne.
- Každý **h-objekt** má svůj **název**. V **prostoru** se může nacházet několik **h-objektů** se stejným **názvem**.
- Některé z přenositelných **h-objektů** v **prostoru** může **hráč** vzít a uložit je do pomyslného **batohu**, aby mu v budoucnu pomohly ke splnění nějakého pomocného **úkolu** nebo dokonce **cíle** celé **hry**.
- **Hra** musí proto akceptovat **příkaz**, který přesune zadaný **h-objekt** z aktuálního **prostoru** (tj. **prostoru**, v němž se právě nachází **hráč**) do **batohu**, a **příkaz**, který naopak přesune zadaný **h-objekt** z **batohu** do aktuálního **prostoru**.
- **Množství h-objektů**, které se do **batohu** vejdou, je omezené. **Příkaz** pro přesun **h-objektu** z **prostoru** do **batohu** proto nesmí vyvolat překročení **kapacity batohu**. Zrovna tak nesmí umožnit uložit do **batohu** nepřenositelný **h-objekt**.
- **Hra** musí umožňovat předčasné **ukončení** a opakované **spuštění** bez **ukončení aplikace**.
- Současně musí být schopna poskytnout **nápovědu** s **přehledem příkazů** a jejich **významu**. Tato **nápověda** nemusí být kontextově citlivá.
- Při zadávání příkazů nesmí záležet na **velikosti písmen**.
- Aby se vše snáze zpracovávalo, musejí být **názvy prostorů**, **h-objektů** i **příkazů** jednoslovné.
- **Hra** se spustí zadáním prázdného **příkazu**, tj. **příkazu** tvořeného prázdným **stringem**. Prázdný **příkaz** se však smí zadat pouze tehdy, když **hra** neběží. Zadání prázdného **příkazu** za běhu **hry** je považováno za **chybu**.
- **Hra** musí umět korektně reagovat na chybně zadané **příkazy uživatele**.

7.3 Účastníci – objekty vystupující ve hře

Zadání máme, podstatná jména jsme v něm zvýraznili, můžeme přistoupit k definici účastníků. Pojďme projít zadání, provést analýzu nalezených podstatných jmen a rozhodnout, která z nich si zaslouží, aby je ve vytvářeném rámci reprezentovala nějaká třída či jiný druh objektu.

Názvy, které pro ně budeme navrhovat, zatím vysadíme malými písmeny. V průběhu další analýzy se ukáže, který z nich bude názvem třídy a který by měl proto začínat velkým písmenem.

Aplikace, Hra – `game`

Prvním podstatným jménem je slovo **aplikace**, za nímž následuje slovo **hra**. Tato dvě slova můžeme v tomto případě považovat za synonyma, protože obě označují vyvíjenou aplikaci. Přikloníme se k druhému, protože je výstižnější.

V celé aplikaci bude jediný takový objekt, takže bude asi nejvýhodnější definovat hru jako modul, který nazveme `game`. Při dalších úvahách se pak rozhodneme, co vše umístíme do tohoto modulu a co umístíme do samostatných modulů.

Svět – `world`

Dalším podstatným jménem je **svět**, v němž se bude celá hra odehrávat. Svět pak už dále v zadání nevystupuje, ale působí dojmem objektu, který by mohl mít v programu důležitou roli. Pro tento objekt se nabízí jediný vhodný název: `world`.

Prostor – `place`

Následujícím podstatným jménem v zadání je **prostor**. Prostory, v nichž se hra odehrává, jsou její klíčovou součástí. Je tedy více než zřejmé, že bude nanejvýš vhodné definovat třídy, jejichž instance budou představovat prostory dané hry.

Při hledání názvu této třídy naskočí nejprve název `Space`, ale když se zamyslíme nad použitím ve hře, přicházejí v úvahu i názvy `Area` nebo `Place`. Jak jste jistě z nadpisu odhadli, já jsem se nakonec přiklonil k názvu `Place`.

Název – `name`

Prostory musejí mít různá jména. Jméno tedy bude atribut prostoru. Budou-li nám stačit obyčejné stringy, nemusíme zavádět žádnou novou třídu. Initor třídy `Place` však musí umět vytvořit prostor se zadaným názvem.

Příkaz – Akce – `action`

Při dalším čtení narazíme na podstatné jméno **příkaz**. Příkazy jsou dalšími klíčovými prvky hry, na které jsou kladeny dodatečné podmínky. Jak si ale možná vzpomenete, tak jsem vám v poznámce na stránce 102 říkal, že budu rozlišovat termíny **příkaz** a **akce**. Objekty našeho programu by pak měly představovat akce, které budou schopny zareagovat na různé příkazy.

Možná vám bude připadat divné, proč by měla být akce objektem. Uvědomte si ale, že objekty představující jednotlivé akce (přesun mezi prostory, položení, resp. zvednutí h-objektu atd.) budou uchovávat informace o tom, jak má hra zareagovat v případě, kdy hráč zadá příkaz vedoucí k realizaci dané akce. Přitom musejí umět rozumně zareagovat i v případě, kdy hráč zadá příkaz špatně. Špatně zadaný příkaz nesmí program zhroutit, ale pouze vyvolá reakci, která hráče upozorní na jeho chybu.

Přechod

Následuje podstatné jméno **přechod**. Tady už to není tak jasné. Mohli bychom sice požadovat vytvoření objektu definujícího přechod z prostoru do prostoru, ale na druhou stranu bychom mohli pojmout otázku těchto přechodů tak, že to, jestli je možné z jednoho prostoru přejít do druhého, je interní věci daného prostoru, proto by tuto informaci měl spravovat daný prostor a neměla by být uložená v nějakém veřejném objektu. Stačilo by, aby si každý prostor pamatoval prostory, do nichž je možné v daném okamžiku přejít.

Při takovémto uspořádání ale bude třeba, aby si někdo pamatoval, ve kterém prostoru se hráč zrovna nachází a kde bude proto program zjišťovat, kam se odtud může přesunout.

To, že jsem zde nedoporučil zavádět pro přechody jejich vlastní třídu, ještě neznamená, že se to při trochu jiné koncepci hry nemůže hodit. V literatuře se setkáte s aplikacemi realizujícími hru, v níž se prochází bludištěm a v níž jsou přechody (nebo spíše průchody) docela užitečné datové objekty.

H-objekt – `item`

Při dalším čtení přeskočíme několik prostorů (přesněji řečeno podstatných jmen prostor), o nichž už jsme rozhodli, a narazíme na podstatné jméno **h-objekt**. H-objekty jsou opět klíčovým prvkem hry s dalšími omezujícími podmínkami (některé musí jít zvednout a jiné ne), takže není pochyb o tom, že by pro ně měla být v aplikaci definována třída. Rozhodl jsem se ji nazvat `Item`.

Nesmíme zapomenout na to, že **h-objekty** jsou pojmenované, takže instance třídy `Item` musejí mít atribut `name`.

Hráč

Následuje podstatné jméno **hráč**. Tady asi opět trochu zaváháme. Musíme si rozmyslet, jestli pro nás hráče představuje uživatel (případně testovací program), anebo jestli se jedná opravdu o objekt hry. Musíme si proto ujasnit, jaká by mohla být role hráče v programu.

Ze zadání vyplývá, že hráč se někde nachází a má něco v batohu. Je tedy otázkou, jestli to držet v programu jako samostatné informace (definovat např. samostatný batoh s tím, že si pozici hráče bude pamatovat třeba svět, v němž se hráč nachází), anebo tyto informace uchovávat v objektu hráče.

Obě cesty jsou možné. Když jsem se o těchto možnostech rozhodoval já, dospěl jsem k závěru, že informace týkající se hráče jsou poněkud heterogenní (nesourodé), takže by hráč buď dělal několik relativně různých věcí, anebo by fungoval jen jako přepravka pro informace získané specializovanými objekty. Rozhodl jsem se proto objekt hráče do hry nezavádět a využít pro získání informací o hráči specializované objekty: pro informaci o aktuálním prostoru svět a pro informaci o obsahu batohu speciální objekt reprezentující pouze batoh.

Na druhou stranu, kdyby byla hra koncipována maličko jinak a mohlo ji např. hrát několik hráčů, bylo by asi výhodnější objekty hráčů zavést.

Batoh – Bag – BAG

Při dalším čtení zadání narazíme vzápětí na podstatné jméno **batoh**. O něm jsem před chvílí hovořil, takže je vám jistě jasné, že je o něm rozhodnuto: bude jej reprezentovat samostatný objekt.

Protože je batoh v celé hře jediný, je třeba vybrat, pro který druh objektu (viz podkapitolu *6.8 Druhy vytvářených objektů* na straně 98) se rozhodneme. Dopředu jistě odhadnete, že schopnosti tohoto objektu budou minimální, takže se asi nevyplatí je definovat jako modul. Definujeme pro něj proto třídu, nazveme ji `Bag` a batoh bude reprezentován její jedinou instancí, kterou nazveme `BAG` (podle konvencí velkými písmeny, protože se bude jednat o konstantu).

Úkol, cíl

V následujícím textu se hovoří o pomocném **úkolu** a o **cíli** celé hry. Tyto pojmy jsou však evidentně pouze doplňkové a není třeba pro ně v rámci definovat nějakou speciální reprezentaci. Opět ale platí, že by při maličko změněné koncepci hry mohlo být výhodně tyto objekty definovat.

Množství, kapacita

Následuje řada podstatných jmen, které se v textu již vyskytly a na jejichž výskyt jsme již zareagovali. Poslední doposud neprodiskutovaná podstatná jména jsou **množství** h-objektů v batohu a **kapacita** batohu.

Obě označují totéž, přičemž se asi shodneme, že se jedná o interní informaci batohu, kterou by si měl batoh spravovat sám a není třeba pro ni definovat nějaký zvláštní datový typ – batoh si může pamatovat svoji kapacitu jako číslo. Jenom je třeba, aby měl atribut nazvaný `CAPACITY` (velká písmena naznačují, že to bude konstanta).

Ukončení, spuštění

Spuštění a ukončení nebudeme v tomto příkladu definovat jako samostatné objekty, protože se o ně postarají příslušné příkazy. Jenom nesmíme zapomenout takovéto příkazy do portfolia příkazů zařadit.

Nápověda, přehled

Nápověda bude text vypsaný při zadání konkrétního příkazu. Bude to proto předem definovaný string a není ji proto třeba definovat jako nový objekt.

7.4 Správci skupin objektů

Při návrhu architektury je třeba mít na paměti, že jakmile se někde vyskytne větší množství objektů (přičemž větší množství objektů bývá většinou více než dva), bývá nanejvýš vhodné, ne-li přímo nutné definovat objekt, který bude vystupovat jako jejich správce. Správce můžeme definovat několika způsoby:

- Ten nejméně vhodný je pověřit správcovstvím objekt, který má již na starosti jiné věci. Jedna z důležitých programátorských zásad říká, že **každý objekt má být zodpovědný pouze za jednu věc** (viz pasáž *SRP – jediná zodpovědnost* na straně 93).
- Můžete definovat speciální objekt, který bude mít správu daných objektů na starosti. Ve většině OO jazyků k tomu musíte definovat třídu, jejíž bude daný objekt instancí, a poté zabezpečit, aby byl jedinou instancí. S tímto řešením se sice setkáte, ale pro mne je zbytečně komplikované, i když je v některých případech jediné možné.
- Jsou-li všechny spravované objekty instancí jediné třídy, může se jejich správcem stát objekt dané třídy. S tímto řešením se ale příliš často nesetkáte, protože řada programátorů se bojí považovat třídu za objekt.
- V *Pythonu* se může správcem objektů stát modul, a to zejména tehdy, jsou-li mateřské třídy všech spravovaných objektů definovány v daném modulu. Jak ale uvidíte v poslední kapitole, není to podmínka nutná.

Správci v naší aplikaci

Podíváme-li se na naše účastníky z pohledu vyhledávače potenciálních správců, můžeme dojít k následujícím závěrům:

- Svět naší hry je tvořen skupinou **prostorů**. Proto by objekt reprezentující svět mohl být správcem těchto prostorů. Uvážíme-li výše uvedené možnosti, pak se nabízí definovat svět jako modul `world`, v němž bude definována třída `Place`, jejímiž instancemi budou prostory, které bude modul `world` spravovat.

  Tento objekt pak bude zákonitě tím, jenž si bude pamatovat, ve kterém prostoru se hráč právě nachází, jak už jsem se zmínil v pasáži *Přechod* na straně 106. Definujeme v něm proto atribut `current_place`, v němž si svět bude aktuální prostor pamatovat.
- Další skupinou objektů jsou **akce**. Prozatím se ukazuje, že bude nejvýhodnější definovat pro ně speciální modul `actions`. V něm pak budou definovány objekty všech akcí. Při následné podrobnější analýze se rozhodneme, zda pro každou akci definujeme samostatnou třídu, anebo zvolíme nějaké jiné řešení.
- Následujícími účastníky by měly být h-objekty. Má jich být více, takže pro ně bude potřeba definovat třídu – nazveme ji `Item`. Otázkou je, kdo bude správcem jejích instancí.

Vzhledem k tomu, že tyto instance jsou „rozcourány“ po jednotlivých prostorech a batohu, tak by asi bylo nejrozumnější, aby každá z nich byla spravována svým majitelem, tj. příslušným prostorem či batohem. Pokud by některá z instancí měla změnit majitele, musejí se na tom její majitelé dohodnout.

- Z předchozího vyplývá, že třídu `Bag` bude nejlepší definovat také v modulu `world` a s její jedinou instancí `BAG`, která tak bude atributem modulu.

7.5 Vytvoření zárodku budoucí aplikace

Uzavřeme tuto kapitolu tím, že vytvoříme zárodek naší budoucí aplikace. Vytvořte soubory odpovídající třem naplánovaným modulům: `game`, `actions` a `world`. První dva prozatím vybavíme pouze dokumentačním komentářem a případným zaváděcím a závěrečným tiskem. V modulu `world` definujeme prázdné třídy `Item`, `Place` a `Bag` (jak se to dělá, jsme si vysvětlovali v podkapitole *4.8 Definice prázdné třídy* na straně 76).

Z předchozích úvah víme, že bychom v definovaných modulech a jejich třídách měli navíc definovat následující objekty:

- H-objekty a prostory, tj. instance tříd `Item` a `Place` mají mít svůj vlastní název. V těchto třídách proto definujeme initor s parametrem `name` inicializujícím stejnojmenný instanční atribut.
- Svůj název mají mít i jednotlivé akce, nicméně celá problematika definice akcí je trochu složitější, takže se zatím omezíme na definici prázdného modulu `actions`, jehož detailní podobou se začneme zabývat, až jej bude potřeba opravdu navrhnout. Prozatím jej ponecháme prázdný.
- Batoh má mít omezenou kapacitu. Ve třídě `Bag` proto definujeme atribut `CAPACITY`, který bude tuto kapacitu uchovávat.
- V předchozí pasáži jsme odvodili, že by v modulu `world` měly být definovány atributy `current_place` a `BAG`.

  Aktuální prostor zatím neznáme, tak atribut `current_place` inicializujeme konstantou `None`. Atribut `BAG` můžeme inicializovat vytvořením instance třídy `Bag`.
- Reakce na prázdný příkaz má záviset na tom, je-li hra spuštěná. V modulu `game` proto definujeme atribut `is_active` a inicializujeme jej hodnotou `False`, protože na začátku hra spuštěná není.

Definice prázdného modulu `actions` a skoro prázdného modulu `game` vám předvádět nebudu, ale zájemci si mohou ve výpisu 7.1 prohlédnout zárodečnou definici modulu `world` bez úvodních komentářových řádků.

Výpis 7.1: *Zárodečná definice modulu* `world`

```
"""
Modul world reprezentuje svět a obsahuje definice tříd prostorů,
předmětů (h-objektů) a batohu.
"""
print(f'===== Modul {__name__} ===== START')

################################################################################
class Item():
    """Instance reprezentují h-objekty v prostorech hry a v batohu."""

    def __init__(self, name: str):
        """Vytvoří h-objekt se zadaným názvem."""
        self.name = name


################################################################################
class Place:
    """Instance reprezentují prostory hry."""

    def __init__(self, name: str):
        """Vytvoří prostor se zadaným názvem."""
        self.name = name


################################################################################
class Bag:
    """Instance třídy reprezentuje batoh."""

    CAPACITY = 0    # Maximální kapacita batohu


################################################################################

# Aktuální prostor = prostor, v němž se hráč právě nachází
current_place = None

# Jediná instance batohu
BAG = Bag()


################################################################################
print(f'===== Modul {__name__} ===== STOP')
```

7.6 Balíčky

Zatím to tedy vypadá, že naše aplikace bude tvořena třemi moduly: `game`, `world` a `actions`. Bylo by ale vhodné, aby moduly dané hry nebyly promíchány s moduly sloužícími zcela jinému účelu – v tomto případě s moduly s doprovodnými programy kapitol. *Python* zavádí pro takový účel tzv. *balíčky* (anglicky *packages*).

Trocha teorie

Chceme-li vytvořit nějakou jenom trochu větší aplikaci, není vhodné naskládat všechny moduly na hromadu. Soubory na disku také nemáte všechny v jedné složce (adresáři), ale vytvoříte na disku řadu podsložek a soubory mezi ně vhodně rozmístíte, abyste se v nich lépe vyznali.

Stejně jako modul odpovídá souboru, tak balíček odpovídá složce. A protože složka (adresář) je pouze speciální typ souboru, tak i balíček je pouze speciální typ modulu.

Balíček je modul, který může obsahovat jiné moduly. Tak jako složky (adresáře) v souborovém systému, i balíčky tvoří hierarchickou stromovou strukturu, přičemž balíček může obsahovat podbalíčky (podsložky, podadresáře) i koncové moduly (soubory). Koncept balíčků slouží k lepší organizaci modulů a zavedení hierarchie jmen.

Obdobně jako u souborů a složek, i zde zavádíme **podbalíčky**, které označujeme jako **dceřiné balíčky**, a „**nadbalíčky**“ jako **rodičovské balíčky**. Balíček, který již nemá žádného rodiče, pak označujeme jako **kořenový balíček**.

Protože balíček je speciálním případem modulu, tak toto označení rozšiřujeme i mezi moduly. Jako **rodičovský modul** tak označujeme balíček, v němž se daný modul nachází (při souborové interpretaci: složku, v níž se nachází daný soubor).

Název modulu

Název modulu sestává z názvu rodičovského modulu následovaného tečkou a vlastním názvem daného modulu. Protože jsou balíčky pouze speciálním případem modulů, platí to i pro ně.

Pokud bychom v kořenovém balíčku měli balíček nazvaný `pkg1`, v něm podbalíček nazvaný `pkg2` a v něm podbalíček `pkg3`, byl by název tohoto balíčku `pkg1.pkg2.pkg3`. Pokud bychom v tomto balíčku definovali modul `mod`, pak by úplný název tohoto modulu byl `pkg1.pkg2.pkg3.mod`. Tento název také udává atribut `__name__`.

Initor balíčku

Jako každý objekt i balíček je třeba zkonstruovat. Initorem balíčku je kód uložený v souboru `__init__.py`. Na disku se složka představující balíček pozná podle toho, že obsahuje tento soubor. Soubor může být prázdný, ale musí tam být.

Stejně jako u ostatních objektů, *Python* spustí zadaný kód v okamžiku, kdy vytváří objekt reprezentující daný balíček a potřebuje jej inicializovat.

Šablona initoru balíčku

V doprovodných programech najdete soubor `__init__.py` i v kořenovém balíčku. Ten se však v interaktivním režimu implicitně nenačítá a jeho definice je potřeba až při vytváření samostatně spustitelné aplikace, což budeme probírat v podkapitole *19.8 Přímé spuštění zadaného* skriptu na straně 229.

V kořenovém balíčku doprovodných programů najdete (mimo jiné) soubor `__init__Template.py`, jenž bude sloužit jako šablona initorů nově vytvářených balíčků. Jeho zdrojový kód si můžete prohlédnout ve výpisu 7.2. Když vytvářím nový balíček, tak do něj tento soubor zkopíruji, smažu v něm řádky 4-5 a upravím text v řádcích 6-7 a 11-12.

Než získáte větší zkušenosti, doporučuji vám, abyste také použili příkazy tisku podobné příkazu na řádcích 10-13. Až získáte větší zkušenosti, zavedete si vlastní konvence.

Výpis 7.2: *Šablona initorů balíčků v souboru `__init__Template.py`*

```
1  """
2  Šablona pro initory podbalíčků - ty budou
3  v dokumentaci i v kontrolním tisku obsahovat text:
4  Balíček s verzí hry na konci X. kapitoly
5  po «stručná charakteristika posledního rozšíření»\
6  """
7
8  print(f'''##### {__name__} - \
9  Balíček s verzí hry na konci X. kapitoly
10      po «stručná charakteristika posledního rozšíření»\
11 ''')
```

Když se objeví příslušný tisk, víte, že se daný balíček načítal. Můžete si tak ověřit, že když pomocí funkce `reload()` znovu načtete nějaký modul (viz podkapitolu *5.6 Opětovné načtení opraveného modulu* na straně 87), tak se kvůli němu jeho balíček znovu nezavádí.

Rozdělení doprovodných programů do balíčků

Pro hru, kterou se právě chystáme vybudovat, je ve složce s doprovodnými programy vytvořena podsložka/balíček `game`. V ní pak najdete řady podsložek s názvy typu `game_v1x`, kde `x` zastupuje postupně znaky `a, b, c` atd. Jednička označuje první verzi hry a písmena postupně označují stav vývoje našeho projektu.

Zdrojové kódy modulů, k nimž jsme se dopracovali v této kapitole, jsou ve složce/balíčku `game_v1a`. Pokud ji otevřete, najdete v ní ještě modul `scenarios`, který budeme vytvářet v kapitole *9 Připravujeme test* začínající na straně 128. Použil jsem v ní stejný balíček proto, že v něm pouze přibyl jeden modul, ale nic stávajícího se nezměnilo.

Obecně vždy zůstanu ve stejném balíčku do chvíle, než začneme některý ze souborů měnit. Pak vždy vytvořím nový balíček, v jehož názvu změním poslední písmeno (po `game_v1a` přijde `game_v1b`, po něm `game_v1c` atd.), zkopíruji do něj moduly z předchozího balíčku a začnu zanášet úpravy.

Až celou hru rozchodíme a začneme přemýšlet nad tím, jak ji vylepšit, začnu vytvářet balíčky řady `game_v2`.

Celou tuto operaci provádím pouze proto, aby se vám porovnáním souborů v různých balíčcích dařilo snáze odhalovat provedené změny a nemuseli jste kvůli tomu bloudit textem a hledat, ve kterou chvíli se něco změnilo.

Budete-li si chtít při čtení následujících kapitol vytvářet vlastní verzi projektu, připravte si vedle vlastní složku, vložením souboru `__init__.py` z ní udělejte balíček a můžete paralelně experimentovat se svojí verzí programu a dívat se, co se stane, když naprogramujete něco jinak. Ve svojí verzi samozřejmě jednotlivé „krokové balíčky" vytvářet nemusíte.

Porovnání dvou souborů nabízí všechna lepší vývojová prostředí, chytřejší editory a dokonce i mnozí správci souborů (ve *Windows* např. populární *Total Commander*). Kromě toho existuje řada programů specializovaných právě na porovnání souborů. V anglické *Wikipedii* jich najdete bohatou sbírku na adrese https://en.wikipedia.org/wiki/Comparison_of_file_comparison_tools.

Relativní import

Python umožňuje zadávat v příkazu `from … import` cestu k importovanému modulu i relativně. Rodičovský balíček importujícího modulu můžete reprezentovat tečkou, prarodičovský dvěma tečkami atd. Nesmíte jen vycestovat až do kořenového balíčku.

Pro demonstraci relativního importu jsem do balíčku `game_v1a` přidal k zárodku naší hry ještě demonstrační modul `m07a_relative_import`. Do rodičovského balíčku `game` jsem pak přidal modul `m07b_imported` a do balíčku `game_v1b` modul `m07c_imported`.

Definici modulu `m07a_relative_import` si můžete prohlédnout ve výpisu 7.3, definice dalších dvou demonstračních modulů jsou podobné, jenom neobsahují žádný import.

Výpis 7.3: *Definice modulu `m07a_relative_import` v balíčku `game.game_v1a`*

```
"""
Pomocný importující modul pro demonstraci funkce relativního importu.
"""

print(f'===== Modul {__name__} ===== START')

from .          import game
from ..         import m07b_imported
from ..game_v1b import m07c_imported

print(f'===== Modul {__name__} ===== STOP')
```

Modul `m07a_relative_import` nedělá nic jiného, než tiskne zprávy o svém zavedení a importuje tři moduly: ¨

- modul `game` ze svého rodičovského balíčku,
- modul `m07b_imported` ze svého prarodičovského balíčku `game`,
- modul `m07c_imported` z balíčku `game.game_v1b`.

Reakci na import tohoto modulu si můžete prohlédnout ve výpisu 7.4.

Výpis 7.4: *Import modulu `game.game_v1a.m07a_relative_import`*

```
>>> import game.game_v1a.m07a_relative_import
##### game - Společný rodičovský balíček balíčků jednotlivých verzí her
##### game.game_v1a - Balíček s verzí hry na konci 7., resp. 9. kapitoly
      po prvotní analýze (7. kapitola) a
      po návrhu šťastného scénáře a simulačních metod (9. kapitola)
===== Modul game.game_v1a.m07a_relative_import ===== START
===== Modul game.game_v1a.game ===== START
===== Modul game.game_v1a.game ===== STOP
===== Modul game.m07b_imported ===== START
===== Modul game.m07b_imported ===== STOP
##### game.game_v1b - Balíček s verzí hry na konci 11. kapitoly
      po definici testu hry
===== Modul game.game_v1b.m07c_imported ===== START
===== Modul game.game_v1b.m07c_imported ===== STOP
===== Modul game.game_v1a.m07a_relative_import ===== STOP
>>>
```

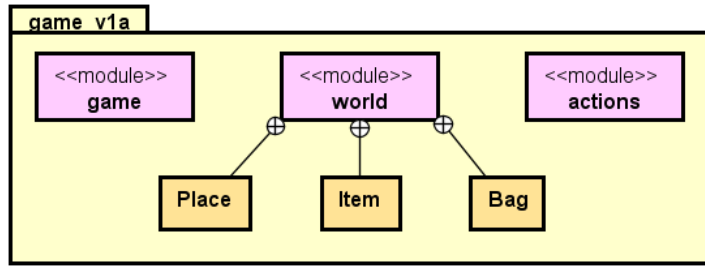
7.7 UML diagram

UML diagram dosavadní verze našeho projektu si můžete prohlédnout na obrázku 7.1. Jak vidíte, třídy i moduly jsou v něm zobrazeny jako obdélníky, které jsou všechny umístěny v balíčku `game_v1a`. Ten je pak zobrazen jako obdélník s „ouškem“.

Abychom poznali, že objekty `game`, `actions` a `world` jsou moduly, jsou jejich obdélníky doplněny tzv. *stereotypem*, což je text uzavřený do «francouzských uvozovek».[9]

Koncové plus v kroužku u spojnic mezi třídami `Place`, `Item` a `Bag` a modulem `world` označuje, že třídy jsou definovány uvnitř tohoto modulu.

[9] Podle pravidel českého pravopisu (viz např. [1]) by se sice tyto uvozovky měly psát »špičkami k sobě«, ale v UML diagramech se píší podle francouzských a amerických pravidel, tj. «špičkami od sebe».



Obrázek 7.1:
Diagram tříd modulu `game_v1a`

Objekty se v UML oficiálně zobrazují tak, že se v hlavičce za názvem objektu uvede dvojtečka a mateřská třída daného objektu (moduly jsou v *Pythonu* instancí třídy `module`) a celý název se podtrhne. Použití stereotypu mi ale připadá přehlednější. Kromě toho, nástroje pro kreslení UML diagramů, které se mi dostaly do ruky, se zobrazováním jednotlivých objektů nepočítají. Norma UML sice od počátku zavedla i diagram objektů, ale tato skutečnost se do některých učebnic UML stále neprobojovala.

7.8 Shrnutí

Záznam komunikace s interpretem probíhající v této kapitole najdete v trojici souborů nazvaných `m07__basic_architecture`. Pomocné demonstrační moduly jsou spolu se svým umístěním popsány v podkapitole *7.6 Balíčky*. Zdrojové kódy modulů a tříd odpovídající stavu vývoje aplikace na konci této kapitoly (a výše uvedenému UML diagramu) najdete v balíčku `game.game_v1a`.

Kapitola 8
Kontejnery a práce s nimi

Co se v kapitole naučíte

Tato kapitola vám představí kontejnery, s nimiž budete pracovat v následujících kapitolách. Postupně vás seznámí se základními údaji o posloupnostech, seznamech, n-ticích, množinách a slovnících. Vysvětlí vám, že zvláštním druhem posloupností jsou i stringy, a naučí vás pracovat s posloupnostmi prostřednictvím indexace. Na závěr vysvětlí problematiku jmenných prostorů.

8.1 Kontejnery

Naše programy dosud většinou pracovaly pouze s jednotlivými objekty. V praxi je však velice často potřeba pracovat s celými skupinami objektů. K tomu používáme speciální objekty – kontejnery, což jsou objekty určené k ukládání jiných objektů. Objekty tvořící skupinu uložíme do kontejneru a pracujeme-li pak s kontejnerem, pracujeme s celou skupinou v něm uložených objektů.

Programové kontejnery mají trochu jiné vlastnosti než ty, které znáte z běžného života. Protože tyto odchylky studenty občas zmatou, vypíchl jsem je do podšeděného rámečku.

8.2 Proměnné a neměnné objekty

Všechny objekty, s nimiž jsme doposud pracovali (tj. čísla, logické hodnoty i stringy), patří mezi neměnné objekty (tj. mezi objekty, které za svého života nemohou změnit svoji hodnotu). Chceme-li jakkoliv upravit hodnotu neměnného objektu, musíme vytvořit nový objekt s onou upravenou hodnotou.

Má to důležité důsledky. Vystupují-li v jakémkoliv vzorci neměnné objekty, mohu se spolehnout na to, že se jejich hodnota do doby výpočtu nezmění. Vystupuje-li ve vzorci pětka, vím, že bude mít hodnotu 5 na věky věků. Obdobně tomu bude, vystupuje-li ve vzorci nějaký string.

Naproti tomu u proměnných objektů se na jejich obsah spolehnout nemohu. Kdyby byly např. stringy proměnné, mohlo by se stát, že bych požádal o vytištění daného stringu a než by k němu v důsledku řady jiných příkazů došlo, někdo by mi jej za mými zády změnil a vytisklo by se něco jiného.

Proto je znalost toho, zda je daný objekt proměnný či neměnný, v programování velice důležitá.

Zvláštnosti programových kontejnerů

Z běžného života jste zvyklí na to, že když uložíte objekt do kontejneru, tak už jej nemáte – je v kontejneru. Při práci ve většině objektově orientovaných programovacích jazyků ale nepracujeme přímo s objekty, ale pouze s odkazy na objekty. Do kontejneru tak neukládáme objekt, ale pouze odkaz na objekt.

Aby toho nebylo málo, tak tam velmi často ukládáme dokonce pouhou kopii tohoto odkazu. Máte-li odkaz v proměnné a vložíte-li obsah proměnné do kontejneru, původní obsah v proměnné zůstane a do kontejneru se vloží pouze jeho kopie. Nezapomínejte na tento detail, studenti s ním mají občas problémy.

Důsledkem této vlastnosti je např. i to, že oproti kontejnerům, jaké známe z běžného života, můžeme v programu jeden objekt uložit do několika kontejnerů současně.

8.3 Druhy kontejnerů

V *Pythonu* rozeznáváme několik základních druhů kontejnerů:

- **Posloupnosti** (anglicky *sequence*) jsou kontejnery, u nichž je definováno uspořádání uložených prvků. Víme, kdo je první, kdo poslední a kdo je umístěn za kým. Na prvek posloupnosti se můžeme odvolávat indexem definujícím jeho pořadí, přičemž počáteční prvek má index `0`.
  - Mezi posloupnosti patří i **stringy**, které občas chápeme jako neměnné posloupnosti znaků. Stringy jsou instancemi třídy `str`.
  - **Seznamy** jsou instancemi třídy `list`. Jsou to proměnné posloupnosti obecných objektů. Vhledem k jejich proměnnosti do nich můžeme další prvky přidávat (nemusí to být nutně na kraj, může to být i na zadanou pozici), anebo je z nich odebírat.
  - **N-tice** jsou instancemi třídy `tuple`. Jsou to neměnné posloupnosti obecných objektů. Od seznamů se liší pouze tím, že jsou neměnné, takže do vytvořené n-tice již není možno nic předat ani z ní odebrat, ani v ní prvek vyměnit. Mohli bychom je proto chápat jako zmrazené seznamy.

- **Množiny** jsou instancemi třídy `set`. Jsou to obecně proměnné kontejnery, v nichž může být každá hodnota jenom jednou.
- Speciálním druhem množin jsou **zmrazené množiny**, které jsou instancemi třídy `frozenset`. Od běžných množin se liší tím, že jsou neměnné. Jakmile takovou množinu vytvoříte, už její obsah nemůžete změnit.
- **Slovníky** jsou instancemi třídy `dir`. Mohli bychom je chápat jako množiny uspořádaných dvojic (klíč, hodnota). Nabízejí možnost rychle vrátit hodnotu sdruženou se zadaným klíčem.

8.4 Vytváření kontejnerů

Všechny druhy kontejnerů můžeme zadávat prostřednictvím literálů (tj. každý druh má svůj literál), prostřednictvím volání jejich tříd a s výjimkou stringů je můžeme všechny zadávat také pomocí generátorové notace.

Jednotlivé typy literálů a generátorových notací si jsou velmi podobné. Abychom měli jistotu, že jsme vždy vytvořili to, co jsme vytvořit chtěli, definujeme na počátku seance v této kapitole funkci `pvt()`, kterou jsme definovali ve výpisu 4.5 na straně 77.

Možná, že si někteří z vás budou myslet, že by bylo jednodušší definovat tuto funkci v nějakém modulu a ten pak importovat. Bohužel funkce `eval()`, která je ve funkci `pvt()` použita, vidí jen globální proměnné svého modulu a doplnění jejích vědomostí je zbytečně komplikované. Jednodušší je definovat funkci `pvt()` znovu.

Pojďme se tedy podívat, jak je možno vytvořit jednotlivé druhy kontejnerů.

Seznam – `list`

Pomocí literálu zadáme seznam tak, že posloupnost čárkami oddělených hodnot uzavřeme do hranatých závorek, jak je uvedeno na řádcích 1 a 3 ve výpisu 8.1. Jak se můžete přesvědčit na řádcích 2 a 4, výsledný seznam se zobrazuje naprosto stejně, jak bychom jej zadávali jako literál.

Druhou možností zadání seznamu je využít volání třídy `list`, které zadáme jako argument buď budoucí prvky vytvářeného seznamu, nebo nějaký kontejner, anebo nějaký generátor hodnot. Ukázku najdete na řádku 5, kde je jako argument zadán string `'Python'` a třída vytvoří seznam jeho písmen. Na řádku 6 pak opět vidíte, jak se výsledný seznam zobrazuje.

Na řádku 7 je použita třetí možnost, kterou je generátorová notace. Ta se vytváří podle pravidla

```
[ výraz for proměnná in zdroj ]
```

Výsledkem je pak seznam hodnot výrazu aplikovaného na jednotlivé hodnoty poskytované zdrojem. Je-li zdrojem nějaký kontejner, aplikuje se výraz postupně na hodnoty uložené v daném kontejneru.

Na řádku 9 je na závěr ukázáno, že v případě potřeby je možné vytvořit i prázdný seznam.

Výpis 8.1: *Vytváření seznamů*

```
>>> prvočísla_L = [3, 5, 7, 11];   pvt('prvočísla_L')
prvočísla_L = [3, 5, 7, 11] # type = <class 'list'>
>>> pořadí_L = ['Nultý', 'První', 'Druhý', 'Třetí'];   pvt('pořadí_L')
pořadí_L = ['Nultý', 'První', 'Druhý', 'Třetí'] # type = <class 'list'>
>>> písmena_L = list('Python');   pvt('písmena_L')
písmena_L = ['P', 'y', 't', 'h', 'o', 'n'] # type = <class 'list'>
>>> mocniny_L = [n*n for n in prvočísla_L];   pvt('mocniny_L')
mocniny_L = [9, 25, 49, 121] # type = <class 'list'>
>>> prázdný_L = [];   pvt('prázdný_L')
prázdný_L = [] # type = <class 'list'>
>>>
```

N-tice – `tuple`

N-tice se vytváří téměř stejně jako seznamy. Literál však nepoužívá hranaté závorky, ale kulaté. Nicméně jakmile napíšete mezi dvě hodnoty čárku, bude je systém považovat za n-tici. To ukazuje i příkaz na řádku 1 ve výpisu 8.2. Použití závorek by sice zápis jistě zpřehlednilo, ale jak vidíte, není nezbytně nutné.

Příkaz na řádku 2 ukazuje, jak je možné použít volání třídy `tuple`, přičemž jako argument je opět použit string `"Python"`.

Volání třídy je potřeba i v případě použití generátorové notace. Když bychom totiž tento zápis pouze uzavřeli do kulatých závorek, jak je předvedeno na řádku 5, bude výsledek chápán jako uzávorkovaný výraz reprezentující generátor (viz zpráva funkce `pvt()` vytištěná na řádku 6).

Abychom pro tvorbu n-tic využili generátorovou notaci, musíme příslušný generátor zadat jako argument volání třídy, jak je předvedeno na řádku 6. Současně se zde můžete přesvědčit, že funkce tohoto generátoru nevyžaduje nutně, aby zdrojem byla n-tice.

Jak ukazuje řádek 8, při vytváření n-tic není problém vytvořit prázdnou n-tici. Problémy by mohly nastat až ve chvíli, kdy bychom potřebovali vytvořit n-tici s jediným prvkem. Jak ukazuje reakce na příkazy na řádku 10, uzavřeme-li jednu hodnotu do kulatých závorek, překladač ji bere jako uzávorkovaný výraz. Abychom jej zadali jako n-tici, musíme za zadávanou hodnotu přidat ještě čárku.

Výpis 8.2: *Vytváření n-tic*

```
>>> prvočísla_T = 3, 5, 7, 11;   pvt('prvočísla_T')
prvočísla_T = (3, 5, 7, 11) # type = <class 'tuple'>
>>> písmena_T = tuple('Python');   pvt('písmena_T')
písmena_T = ('P', 'y', 't', 'h', 'o', 'n') # type = <class 'tuple'>
>>> mocniny_T = (n*n for n in prvočísla_L);   pvt('mocniny_T')
mocniny_T = <generator object <genexpr> at 0x0000024D5B887900> # type = <class
'generator'>
>>> mocniny_T = tuple(n*n for n in prvočísla_L);   pvt('mocniny_T')
mocniny_T = (9, 25, 49, 121) # type = <class 'tuple'>
>>> prázdná_T = ();   pvt('prázdná_T')
prázdná_T = () # type = <class 'tuple'>
>>> číslo = (135);   ntice = (135,);   pvt('číslo');   pvt('ntice')
číslo = 135 # type = <class 'int'>
ntice = (135,) # type = <class 'tuple'>
>>>
```

Množiny – set, frozenset

Množiny můžeme vytvářet stejnými třemi způsoby jako seznamy. Pomocí literálu zadáme množinu tak, že posloupnost čárkami oddělených hodnot uzavřeme do složených (lidově chlupatých) závorek, jak je uvedeno na řádcích 1 a 3 ve výpisu 8.3. Na výsledcích na řádcích 2 a 4 si však všimněte, že množiny vám nezaručí pořadí vypisování jednotlivých hodnot.

Obdobné se seznamem je i použití volání třídy a generátorové notace. Po vytvoření běžné množiny voláme třídy `set` (řádek 5), zápis pomocí generátoru se vkládá do složených závorek (řádek 7).

Výpis 8.3: *Vytváření množin*

```
>>> prvočísla_S = {3, 5, 7, 11};   pvt('prvočísla_S')
prvočísla_S = {11, 3, 5, 7} # type = <class 'set'>
>>> pořadí_S = {'Nultý', 'První', 'Druhý', 'Třetí'};   pvt('pořadí_S')
pořadí_S = {'Třetí', 'První', 'Druhý', 'Nultý'} # type = <class 'set'>
>>> písmena_S = set('Python');   pvt('písmena_S')
písmena_S = {'y', 'n', 'P', 'o', 't', 'h'} # type = <class 'set'>
>>> mocniny_S = {n * n for n in prvočísla_S};   pvt('mocniny_S')
mocniny_S = {121, 49, 9, 25} # type = <class 'set'>
>>> prázdná_S = {};   pvt('prázdná_S')
prázdná_S = {} # type = <class 'dict'>
>>> prázdná_S = set();   pvt('prázdná_S')
prázdná_S = set() # type = <class 'set'>
>>> písmena_F = frozenset('Python');   pvt('písmena_F')
písmena_F = frozenset({'y', 'n', 'P', 'o', 't', 'h'}) # type = <class 'frozenset'>
>>> mocniny_F = frozenset(n * n for n in prvočísla_S);   pvt('mocniny_F')
mocniny_F = frozenset({121, 49, 9, 25}) # type = <class 'frozenset'>
>>>
```

Problém může nastat při vytváření prázdné množiny. Necháte-li se zlákat podobností se seznamem a n-ticemi a zadáte prázdné složené závorky (řádek 9), budete asi překvapeni, že nevznikne prázdná množina, ale prázdný seznam (o těch se budeme bavit vzápětí), jak nám na řádku 10 ukázala funkce `pvt()`. Potřebujete-li prázdnou množinu, zavolejte její třídu bez parametrů, jak je ukázáno na řádku 11. Na řádku 12 si pak můžete všimnout, že i podpis prázdné množiny vypadá jako volání její třídy.

Zmrazenou množinu můžete vytvořit pouze zavoláním třídy `frozenset`. Na řádcích 13a 15 jsou předvedena dvě volání s různými druhy argumentů.

Slovník – `dict`

Posledním uvedeným, nicméně velmi důležitým a často používaným druhem kontejneru je slovník. Slovník bychom mohli vnímat jako množinu uspořádaných dvojic typu (*klíč* : *hodnota*), kde klíčem musí být neměnný objekt. Primárním použitím slovníku je operace, při níž zadáme klíč, a slovník vrátí sdruženou hodnotou.

Při vytváření slovníků musíme vytvářet výše zmíněné dvojice. Slovník si pamatuje, v jakém pořadí jsme dvojice vkládali, a v tomtéž pořadí je pak vypisuje. Ukázky příkazů vytvářejících slovníky najdete ve výpisu 8.4.

Na řádku 1 je slovník zadáván prostřednictvím literálu. Jednotlivé prvky se zadávají jako dvojice hodnot oddělených dvojtečkou, přičemž klíč je zadáván vždy jako první.

Při vytváření slovníků voláním třídy musíte předávat argument jako zdroj (např. kontejner nebo generátor) dvojic. Na řádku 3 je proto vytvořen seznam dvojic, a tento seznam je pak na řádku 5 předán třídě `dict` jako zdroj.

Všimněte si, že v generátoru seznamu na řádku 3 jsme jako zdroj použili množinu `prvočísla_S`. Pořadí prvků v seznamu proto odpovídá pořadí, v jakém své prvky dodávala množina (viz řádek 2 ve výpisu 8.3). V tomtéž pořadí je pak seznam dodával slovníku a v tomtéž pořadí je pak slovník vypisuje (viz řádek 6).

Na řádku 7 je pak slovník zadán prostřednictvím generátorové notace. Při ní záleží na tom, co dodává zdroj.

Výpis 8.4: *Vytváření slovníků*

```
 1 >>> pořadí_D = {'Druhý':2, 'První':1, 'Nultý':0};   pvt('pořadí_D')
 2 pořadí_D = {'Druhý': 2, 'První': 1, 'Nultý': 0} # type = <class 'dict'>
 3 >>> dvojice_L = [(n, n*n) for n in prvočísla_S];   pvt('dvojice_L')
 4 dvojice_L = [(11, 121), (3, 9), (5, 25), (7, 49)] # type = <class 'list'>
 5 >>> dvojice_D = dict(dvojice_L);   pvt('dvojice_D')
 6 dvojice_D = {11: 121, 3: 9, 5: 25, 7: 49} # type = <class 'dict'>
 7 >>> mocniny_D = {n:n*n for n in prvočísla_T};   pvt('mocniny_D')
 8 mocniny_D = {3: 9, 5: 25, 7: 49, 11: 121} # type = <class 'dict'>
 9 >>> prázdný_D = {};   pvt('prázdný_D')
10 prázdný_D = {} # type = <class 'dict'>
11 >>>
```

8.5 Získání prvku z kontejneru

S výjimkou množin můžete po všech ostatních kontejnerech chtít, aby nám poskytly prvek zadaný svým indexem. U seznamů a n-tic představuje index pořadí daného prvku v kontejneru, přičemž počáteční prvek má index `0`, a mohli bychom jej tedy označovat jako nultý. U slovníků pak je index reprezentován hodnotou klíče.

Zadáme-li za odkaz na kontejner (např. za název proměnné, která na něj odkazuje) index uzavřený v hranatých závorkách, získáme prvek se zadaným indexem.

Příkaz na řádku 8 pak už má jenom dokázat, že odkaz na kontejner nemusíme získat pouze z proměnné, ale může to být (jako v tomto případě) návratová hodnota funkce nebo výsledek nějakého výrazu.

V našem příkladu je na řádku 7 definována n-tice s názvy několika dříve definovaných kontejnerů. Na řádku 8 je pak první prvek této n-tice předán jako argument funkci `kontejner()`, která vrátí odkaz na daný kontejner. Za voláním funkce jsou hranaté závorky s indexem `0`, takže výsledkem je nultý prvek kontejneru vráceného funkcí `kontejner()`. Protože jsme funkci zadali název `písmena_T` (první prvek n-tice `názvy`), získáme tak ve výsledku znak `P`, jako počáteční písmeno textu `Python`.

Výpis 8.5: *Získání prvku z kontejneru pomocí indexu*

```
>>> f'{pořadí_L[3]=}, {mocniny_T[2]=}, {pořadí_D["První"]=}'
'pořadí_L[3]=\'Třetí\', mocniny_T[2]=49, pořadí_D["První"]=1'
>>> def kontejner(název: str):
        """Vrátí odkaz na kontejner se zadaným názvem."""
        return eval(název)

>>> názvy = ('prvočísla_L', 'písmena_T', 'mocniny_L')
>>> print(f'{kontejner(názvy[1])[0] = }')
kontejner(názvy[1]")[0] = 'P'
>>>
```

8.6 Projití celého kontejneru – cyklus for

V řadě případů potřebujeme postupně zpracovat všechny prvky daného kontejneru. Jinými slovy: potřebujeme opakovaně provádět nějakou činnost, pouze pokaždé s nějakým jiným prvkem. Konstrukce, které zadání takovéto opakované akce umožňují, označujeme jako cykly. Příkaz `for` proto často označujeme jako **cyklus for** a ještě častěji jako **cyklus s parametrem**.

Syntakticky je cyklus `for` podobný konstrukci, kterou známe z generátorové notace kontejnerů. Příkaz `for` je složený příkaz, který má tvar:

```
for parametr_cyklu in zdroj :
    příkazy_těla_cyklu
```

kde *zdroj* je v našem případě procházený kontejner. Cyklus přebírá ze zdroje jeden prvek za druhým, odkaz na něj vloží do proměnné označované jako *parametr_cyklu* a pak provede příkazy těla cyklu.

Ukázky použití tohoto cyklu najdete ve výpisu 8.6. Na řádcích 2–7 je v něm definována funkce `prvky_pod_sebou()`, které předáme kontejner, a ona vytiskne jeho jednotlivé prvky v pořadí, v jakém je kontejner dodává, přičemž každý řádek (prvek) očísluje počínaje nulou.

Na dalších řádcích jsou vytištěny prvky různých druhů kontejnerů. Na řádcích 10–13 jsou vytištěny prvky seznamu, na řádcích 15–17 hodnoty uložené ve slovníku a na řádcích 19–22 prvky množiny. Na závěr je na řádku 23 metoda pověřená vytištěním prázdného seznamu, ale jak vidíte, protože seznam byl prázdný, nevytiskla správně nic, takže funkce tělo cyklu ani jednou neprovedla.

Výpis 8.6: *Procházení kontejnerem pomocí příkazu* `for`

```
>>> \
def prvky_pod_sebou(kontejner) -> None:
    """Vypíše číslovaně prvky kontejneru, každý na samostatný řádek."""
    index = 0
    for prvek in kontejner:
        print(f'{index}. {prvek}')
        index = index + 1

>>> prvky_pod_sebou(pořadí_L)
0. Nultý
1. První
2. Druhý
3. Třetí
>>> prvky_pod_sebou(pořadí_D)
0. Druhý
1. První
2. Nultý
>>> prvky_pod_sebou(prvočísla_S)
0. 11
1. 3
2. 5
3. 7
>>> prvky_pod_sebou(prázdný_L)
>>>
```

8.7 Funkce s proměnným počtem argumentů

Všechny naše doposud definované funkce měly vždy pevný počet argumentů. Občas nebylo třeba všechny zadávat a za nezadané argumenty se dosadily jejich implicitní hodnoty, ale to nic neměnilo na tom, že jich byl předem známý pevný počet.

V průběhu dosavadního výkladu jsme se ale již setkali i s funkcí `print()`, která je schopna akceptovat proměnný počet argumentů. Pojďme si ukázat, jak takovou funkci definovat.

Hvězdičkový parametr

Základ úspěchu tkví v tom, že i když je počet argumentů proměnný, počet parametrů zůstává pevný. Dosáhneme toho tím, že před název posledního pozičního parametru napíšeme hvězdičku – v dalším textu jej proto budu v případě potřeby označovat jako hvězdičkový parametr.

Tento parametr musí být zadán jako poslední poziční parametr a bude definován jako n-tice, do níž se poskládají všechny zbylé pozičně zadané argumenty. AHA-příklad s ukázkou jeho použití najdete ve výpisu 8.7. V praxi jej v našem programu použijeme např. v kapitole *15 Příkazy Vezmi a Polož* na straně 189.

Výpis 8.7: *Použití hvězdičkového parametru*

```
>>> def hp(*par: int) -> None:
        print(f'Obdržené argumenty: {par=}')

>>> hp(3, 2, 1, 0)
Obdržené argumenty: par=(3, 2, 1, 0)
>>>
```

Hvězdičkový argument

Vedle hvězdičkových parametrů sbírajících do n-tice zbylé argumenty zavádí *Python* i hvězdičkové argumenty, které naopak generují jednotlivé hodnoty. Zadáme-li mezi argumenty zdroj, který označíme hvězdičkou, interpret jej požádá o jeho hodnoty a naplní jimi příslušné parametry. Zdrojem přitom může být libovolný generátor, ale nejčastěji to bývá nějaký kontejner.

V AHA-příkladu ve výpisu 8.8 je ve volání funkce `ha()` na řádku `4` použita jako argument funkce `range()`, jež ve své jednoparametrické verzi vrací generátor posloupnosti hodnot od nuly až po zadanou hraniční hodnotu, kterou však již nevrátí. Jak napovídá vytištěný text na řádku `5`, prvními třemi hodnotami generátoru byly inicializovány parametry `a`, `b` a `c`, zbytek vygenerovaných hodnot byl předán hvězdičkovému parametru `zbytek`.

Výpis 8.8: *Použití hvězdičkového argumentu*

```
>>> def ha(a, b, c, *zbytek) -> None:
        print(f'Obdržené argumenty: {a=}, {b=}, {c=}, {zbytek=}')

>>> ha(*range(6))
Obdržené argumenty: a=0, b=1, c=2, zbytek=(3, 4, 5)
>>>
```

Dvouhvězdičkový parametr

Má-li funkce zpracovávat proměnný počet pojmenovaných argumentů, definuje na konci seznamu parametrů parametr, jehož identifikátor bude označen úvodními dvěma hvězdičkami. V dalším textu jej proto budu v případě potřeby označovat jako **dvouhvězdičkový parametr**.

Tento parametr pak bude definován jako slovník, do nějž se poskládají všechny zbylé pojmenované argumenty, přičemž klíčem bude vždy string se zadaným názvem argumentu.

V AHA-příkladu ve výpisu 8.9 jsou ve volání funkce `dhp()` na řádku 4 zadány tři pojmenované argumenty. Jak je vidět v textu na řádku 5, zadané argumenty byly umístěny do slovníku, který byl volané funkci předán jako hodnota parametru `args`.

Výpis 8.9: *Použití dvouhvězdičkového parametru*

```
>>> def dhp(**args) -> None:
        print(f'Zadané argumenty: {args=}')

>>> dhp(a=1, b=2, c=3)
Zadané argumenty: args={'a': 1, 'b': 2, 'c': 3}
>>>
```

Dvouhvězdičkový argument

Uvedete-li před nějakým argumentem dvojici hvězdiček, překladač jej při volání funkce interpretuje jako slovník, který rozbalí a zadá jeho jednotlivé prvky jako pojmenované argumenty.

Občas může být výhodné mít v nějakém modulu předdefinované oblíbené kombinace pojmenovaných argumentů a poté místo vypisování konkrétních hodnot prostě zadat příslušný slovník.

V AHA-příkladu ve výpisu 8.10 je ve volání funkce `dha()` na řádku 5 předán jako dvouhvězdičkový argument slovník s inicializacemi parametrů `x`, `y` a `z`. Jejich počáteční hodnoty jsou pak vytištěny na řádku 6.

S praktickým použitím dvouhvězdičkových parametrů i argumentů se setkáte v podkapitole *13.7 Definice rodičovské a dceřiné třídy* na straně 176.

Výpis 8.10: *Použití dvouhvězdičkového argumentu*

```
>>> def dha(x: int, y: int, z: int) -> None:
        print(f'Souřadnice: {x=}, {y=}, {z=}')

>>> pozice = {'z': 30, 'x': 100, 'y': 2}
>>> dha(**pozice)
Souřadnice: x=100, y=2, z=30
>>>
```

8.8 Specifika slovníků

Slovníky slouží především k získávání uložených hodnot na základě znalosti jejich sdružených klíčů. Zavedl jsem si takovou konvenci, že proměnné odkazující na slovníky a jim podobné mapovací objekty pojmenovávám složeninou sestávající z označení klíče, po němž následuje znak 2 reprezentující anglické *to* a označení získané hodnoty. Ve výpisu 8.11 jsem tak na řádcích 1 a 3 pojmenoval proměnné, do nichž jsem uložil odkazy na dříve vytvořené slovníky.

Často ale potřebujeme pracovat s pouhými hodnotami, pouhými klíči, případně s celými uloženými dvojicemi. Pro takovýto účel jsou zavedeny tzv. *pohledy* (anglicky *views*), které mohou sloužit jako generátor prvků pro kontejner prvků, s nimiž chceme pracovat. Slovníky poskytují tři pohledy:

`items()`
Vrátí nový pohled použitelný jako generátor dvojic, v nichž je první položkou klíč a druhou položkou jemu přiřazená hodnota. Ukázka použití je na řádku 5.

`keys()`
Vrátí pohled použitelný jako generátor vracející klíče zadaného slovníku. Ukázka použití je na řádku 7.

`values()`
Vrátí pohled použitelný jako generátor vracející hodnoty zadaného slovníku. Ukázka použití je na řádku 9.

Výpis 8.11: *Specifika slovníků*

```
 1 >>> název_2_číslo = pořadí_D;   název_2_číslo
 2 {'Druhý': 2, 'První': 1, 'Nultý': 0}
 3 >>> číslo_2_mocnina = dvojice_D;   číslo_2_mocnina
 4 {11: 121, 3: 9, 5: 25, 7: 49}
 5 >>> název_číslo = {dvojice for dvojice in název_2_číslo.items()};   název_číslo
 6 {('Nultý', 0), ('Druhý', 2), ('První', 1)}
 7 >>> názvy = tuple(název for název in název_2_číslo.keys());   názvy
 8 ('Druhý', 'První', 'Nultý')
 9 >>> mocniny = [mocnina for mocnina in číslo_2_mocnina.values()];   mocniny
10 [121, 9, 25, 49]
11 >>>
```

8.9 Jmenné prostory

Předem upozorňuji, že v *Pythonu* se termínem *jmenný prostor* označuje něco zcela jiného než v některých jiných programovacích jazycích, tak se při rozhovoru s programátory pracujícími v těchto jazycích nenechte zmást.

Termínem *jmenný prostor* (anglicky *namespace*) se v *Pythonu* označuje slovník pamatující si názvy a hodnoty všech proměnných vlastněných daným vlastníkem, tj. názvy všech atributů daného vlastníka, v případě funkcí také názvy všech jejich lokálních proměnných. Jmenný prostor je přitom sám atributem `__dict__` daného vlastníka.

Ve jmenném prostoru daného objektu jsou všechny jeho atributy včetně těch, jejichž názvy začínají znakem podtržení (viz pasáž *Nezveřejňované atributy* na straně 96). Jenom se některé z nich v některých případech nezveřejňují. Jsou však stále dostupné.

V *Pythonu* existují dva druhy vlastníků jmenných prostorů: běžné objekty a těla funkcí. Jmenné prostory objektů obsahují názvy atributů daného objektu, jmenné prostory těl funkcí obsahují lokální proměnné dané funkce včetně jejích parametrů. Vzhledem k tomu, že funkce je současně objekt, jsou s ní spojeny dva jmenné prostory. Který se použije, záleží na tom, zda k dané funkci přistupujeme jako k objektu, nebo zda voláme jí reprezentovaný kód.

Z předchozího tvrzení vyplývá, že mezi názvy uloženými v jednotlivých jmenných prostorech není žádný vztah. Jsou-li ve dvou jmenných prostorech stejné názvy, každý z nich může (ale také nemusí) odkazovat na jiný objekt.

8.10 Shrnutí

Záznam komunikace s interpretem probíhající v této kapitole najdete v trojici souborů nazvaných `m08__containers`.

Kapitola 9
Připravujeme test

Co se v kapitole naučíte
V této kapitole se soustředíme na přípravu testu, který bude prověřovat správnost vytvořeného programu. Představíme si metodiku *Test Driven Development* a budeme podle ní postupovat i při návrhu naší aplikace. Přitom probereme i základy práce s kontejnery a ošetřování chyb.

9.1 Metody `__repr__()` a `__str__()`

V průběhu kapitoly definujeme metodu `__str__()`, o níž jsem se doposud nezmiňoval. Protože pak nechci trhat výklad, povím vám o ní nyní. Tato metoda souvisí s tím, že *Python* rozeznává dva druhy podpisů:

- Metoda `__repr__()` je určena spíše pro vývojáře a měla by optimálně převést daný objekt na takový string, z nějž by interpret dokázal daný objekt vytvořit. Její funkci a použití jsem vám vysvětloval v pasáži *Speciální metody* na straně 73.

  Volá ji např. interpret, když v systémovém okně IDLE nebo v příkazovém řádku zadáme výrazový příkaz, jehož hodnotu systém následně zobrazí. Když budu proto chtít zdůraznit, že výsledek byl získán touto metodou, budu jej označovat jako **systémový podpis**.

  Tuto metodu mají definovánu všichni, protože třídy, v nichž není explicitně definována, použijí verzi zděděnou od třídy `object`.

- Metoda `__str__()` je určena pro převod do tvaru, který bude číst uživatel. Když budu proto chtít zdůraznit, že výsledek byl získán touto metodou, označím jej jako **uživatelský podpis**.

  Tuto metodu použije funkce `print()`, když daný objekt převádí na string. Není-li tato metoda pro daný objekt definována, zavolá se metoda `__repr__()`.

U formátovacích stringů (probírali jsme je v podkapitole *2.12 Formátovací stringy – f-stringy* na straně 52) je to trochu složitější. Objekt, jenž je výsledkem výrazu ve složených závorkách, bude převeden na string prostřednictvím metody `__str__()`, ale přidáte-li do složených závorek rovnítko, bude objekt převeden prostřednictvím své metody `__repr__()`.

Formát výstupu sice můžete ovlivnit, ale popis formátovacích pravidel leží mimo záběr této učebnice. Zájemci se mohou dozvědět podrobnosti v dokumentaci, anebo v již několikrát citované knize [11].

9.2 Jak testovat

Program nestačí jenom napsat, ale každý program je potřeba také otestovat, aby autor včas odhalil případné chyby. V dřívějších dobách zaujímali pracovníci pověření testováním programů nejspodnější místa hodnotového žebříčku. Situace se od té doby výrazně změnila. Většina vývojářů pochopila, že správný návrh a realizace testů může na jednu stranu výrazně zefektivnit proces vývoje a na druhou stranu dobře otestovaný, bezchybný program zvyšuje renomé týmu u zákazníků, což zvyšuje pravděpodobnost lukrativních zakázek v budoucnu. Dnes proto patří testeři k uznávaným, vyhledávaným a velmi dobře placeným členům vývojových týmů.

Programování řízené testy

V moderním programování se často používá metodika vývoje označovaná jako **vývoj řízený testy**, pro niž se používá zkratka **TDD** z anglického *Test Driven Development*. Tato metodika doporučuje, abychom vždy nejprve napsali testy, a teprve pak začali vyvíjet program. Při psaní programu se pak stačí soustředit pouze na to, aby výsledný program prošel napsanými testy.

Výhodou takto koncipovaného vývoje je, že včasně psané testy opravdu testují, co má program dělat. Testy psané na konci vývoje většinou testují, že program správně pracuje, ale někdy jim unikne, že správně dělá něco jiného, než co bylo původním zadáním. Kdysi jsem psal učebnici programování ve verších a tam jsem tuto skutečnost popisoval básničkou:

Když se k cíli dostat chceme,
bývá jedno, kudy jdeme.
Dej však pozor, abys hnedle
nezjistil, že cíl je vedle.

Problém je podobný, jako když si na fotbalovém hřišti stoupnete do brány, zavřete oči a vyrazíte k druhé bráně. Když po padesáti krocích oči otevřete, zjistíte, že vás to poněkud stočilo a míříte někam na tribunu. Tak to zkusíte znovu – a po padesáti krocích zjistíte, že vás to stočilo na druhou stranu a míříte kamsi do lesa.

S programátory je to obdobné. Po obdržení zadání vyrazí požadovaným směrem, ale postupně se více a více soustředí na detaily, jak úkol naprogramovat, a ztrácejí ze zřetele původní cíl. Na konci pak často odevzdají fungující program, který ale dělá něco trochu jiného, než zákazník požadoval.

Metodika TDD nabízí testy jako mantinely, které vám nedovolí sejít z cesty k cíli. Proto se jí budu řídit i v probírané ukázce návrhu aplikace. Před tím, než se pustíme do programování zadaného úkolu, proto nejprve navrhneme, jak bychom průběžně testovali, nakolik se nám zadaný úkol daří řešit. Vyvstává však otázka, jak vyvíjenou hru testovat.

Jednotkové, integrační a regresní testy

Programátoři často používají sadu testovacích metod prověřujících třídy testovaného programu a jejich instance. Takovéto testy označujeme jako **jednotkové testy** (anglicky *unit tests*), protože testují jednotlivé jednotky budoucího programu. U větších programů však s jednotkovými testy nevystačíme. U nich totiž potřebujeme vědět nejen to, že jednotlivé jednotky pracují správně, ale také to, že tyto jednotky správně vzájemně spolupracují.

U rozsáhlejších programů proto po odladění jednotlivých jednotek přicházejí na řadu **integrační testy** (anglicky *integration tests*), které mají za úkol otestovat integraci jednotlivých jednotek do programu a jejich vzájemnou spolupráci. Naše hra už bude natolik složitý program, že by si integrační testy zasloužil. Na druhou stranu ale zase nebude natolik složitá, abychom nemohli oba dva druhy testů sloučit do jednoho kompletu, který otestuje celkové chování programu a přitom současně prověří i funkci jeho jednotlivých jednotek.

Třetím druhem testů, s nímž se při debatách o testování programů často setkáte, jsou regresní testy (anglicky *regression tests*), které mají ověřit, že dříve odladěná část programu je funkční i po provedených modifikacích. Napíšeme-li náš test rozumně, může sloužit i k tomu, abychom průběžně otestovali, že jsme při dalším vývoji nezkazili něco s toho, co jsme vyvinuli dříve.

Možnosti testování naší hry

Znovu se tedy vracíme k otázce, jak naši hru testovat. Jak je v programování obvyklé, nabízí se několik možností:

- Připravit testovací metody, které budou testované hře zadávat jednotlivé příkazy a kontrolovat, jak na ně hra reaguje, tj. co po jejich zadání hráči odpoví a do jakého se dostane stavu. Nevýhodou takovéhoto testu je jeho značná pracnost – před testováním každého příkazu bychom museli nejprve připravit výchozí situaci, pak zadat příkaz a poté zkontrolovat výsledný stav. Vznikne tak sada relativně nezávislých testů, při jejichž vytváření můžeme přehlédnout některé vzájemné vazby.

- Druhou možnosti je definovat jakýsi scénář, jenž by obsahoval zadání posloupnosti příkazů specifikující, jak by se hra mohla hrát. Ke scénáři bychom připravili nějakou univerzální testovací metodu, které bychom tento scénář požadovaného průchodu hrou zadali. Naše univerzální testovací metoda by pak hře postupně zadávala příkazy z tohoto scénáře a po každém příkazu zkontrolovala, zda na něj hra správně zareagovala.

  Pro tuto kontrolu však potřebujeme, aby krok takového scénáře obsahoval nejenom příkaz, který se má hře zadat, ale i jakýsi popis stavu, do kterého by měla hra po provedení zadaného příkazu přejít.

Druhé řešení přiblíží test realitě. Takovýto scénář totiž převede původní verbální (slovní) zadání úlohy, jemuž rozumí jenom člověk, na zadání, kterému je schopen porozumět i počítač. Počítač je mu navíc schopen nejenom porozumět, ale také s jeho pomocí otestovat, zda se výsledný program opravdu chová podle předem zadaných pravidel. Vytvořené scénáře budeme navíc moci použít nejenom k testování správného chodu hry, ale i k demonstraci toho, jak je možné hru hrát, nebo naopak toho, jak se hra hrát nemá. Vydáme se proto touto cestou.

Ve zbytku této kapitoly navrhneme testovací modul, v němž bude definován scénář jednoduché hry, a vzápětí si ukážeme, jak lze takovýto scénář převést do podoby programu.

9.3 Scénáře

Používání termínu *scénář* se neomezuje pouze na filmy či divadelní hry, ale používá se i v softwarovém inženýrství, kde označuje popisy možných průchodů programem, resp. popisy chování programu v různých situacích. Sestává z posloupnosti kroků, které vedou ke splnění zadané úlohy. Rozsáhlejší aplikace mají definovány i stovky scénářů pro jednotlivé dílčí úlohy.

Mezi scénáři popisujícími postup řešení konkrétní úlohy je vždy jeden, který označujeme jako **základní úspěšný scénář**. Popisuje situaci, kdy se vše daří. Uživatel nedělá chyby a ani v systému se nic nepokazilo. Kromě něj ještě existují **alternativní scénáře** popisující postup, jak je třeba reagovat na nějakou chybu, a to jak chybu uživatele, tak chybu, která se objeví v systému (např. nelze se připojit k síti, nejde přečíst nějaký soubor apod.).

V knize [8] označili tento scénář jako *Happy Scenario – šťastný scénář*, protože se v něm všechno daří. Pojďme navrhnout šťastný scénář naší hry, v němž popíšeme, jak by mohla hra v ideálním případě probíhat. V dalších etapách pak můžeme zkusit navrhnout i scénáře pro situace, kdy se něco nedaří.

Náš scénář se bude od těch klasických odlišovat v jedné věci: nebude popisovat průchody programem (tj. naší hrou) slovy, ale bude to objekt, který bude tento průchod specifikovat způsobem využitelným v testovacích programech, pomocí nichž budeme tento průchod simulovat a testovat.

Modul scenarios

Abychom naše scénáře a případný další kód související s testováním vyvíjené aplikace nemíchali s kódem vlastní hry, definujeme pro ně samostatný modul, který nazveme `scenarios`. V něm definujeme veškerý kód související s testováním.

9.4 Kroky definující stav hry

Scénář má reprezentovat možný průběh hry. Průběh hry můžeme rozložit na sadu kroků. V každém kroku zadáme hře nějaký příkaz, jímž ji přivedeme do nějakého dalšího stavu, který nám hra ve své odpovědi naznačí. Jak už jsme si řekli, v objektovém programování je vše objekt. Objekty budou i kroky hry – budou instancemi třídy `Step`.

Krok scénáře by měl obsahovat informace o stavu, do nějž má hra přejít po provedení (případně odmítnutí) zadaného příkazu. Měli bychom si tedy ujasnit, co charakterizuje stav hry.

- Nejdůležitější informací je **zadávaný příkaz**. Reakci na něj budeme testovat a následující informace tak budou popisovat stav hry po vykonání tohoto příkazu, resp. po pokusu o jeho vykonání.
- První, co nás bude při testu reakce hry na zadaný příkaz nejspíše zajímat, je **odpověď hry**, tj. jakou zprávu hráči po provedení příkazu vypíše.
- Ze zadání víme, že hráč má v průběhu hry procházet různými prostory. Jednou z charakteristik by tedy měl být **aktuální prostor**, tj. prostor, v němž se bude hráč nacházet po provedení příslušného příkazu.
- Z aktuálního prostoru můžeme přejít do některých jiných prostorů a do některých odtud naopak přejít nemůžeme (chceme-li např. ze sklepa na dvůr, musíme nejprve do haly). **Množina přímo dostupných sousedů aktuálního prostoru** se může v průběhu hry měnit – např. odemčením dveří se zpřístupní další místnost. Další charakteristikou by proto mohly být prostory, které s aktuálním prostorem bezprostředně sousedí, a můžeme se do nich proto bez problémů přesunout.
- V prostorech se mohou nacházet nejrůznější **h-objekty**. Mezi charakteristiky stavu hry by tak měla patřit i **množina h-objektů vyskytující se v aktuálním prostoru** po provedení příkazu.

  Do charakteristiky stavu by měly patřit i h-objekty v ostatních prostorech, ale protože do nich hráč nevidí, stačí je zkontrolovat v okamžiku, kdy do nich hráč, resp. jím řízená postava vstoupí.
- Víme, že hráč si může některé h-objekty vzít s sebou do pomyslného batohu na další cestu. **Množina h-objektů, které má hráč aktuálně v batohu**, může výrazně ovlivnit další průběh hry (např. k tomu, aby hráč mohl zabít draka, musí mít v batohu meč). Další charakteristikou výsledného stavu by tedy měl být obsah batohu.

- Když budeme procházet scénářem krok za krokem, bylo by asi vhodné jednotlivé kroky číslovat, abychom případně věděli, kde zadání kroku hledat anebo naopak kolik kroků nám ještě zbývá do konce hry. Přidáme tedy ještě **pořadí kroku v daném scénáři**.

9.5 Definice třídy Step

Instance třídy `Step` mají sloužit pouze jako přepravky na hrst objektů uchovávajících informace popsané v předchozí podkapitole. Definuje pouze initor a metodu `__str__()` definující uživatelský podpis daného kroku.

Aby byly v uživatelském podpisu viditelně odlišeny příkazy a odpovědi a aby byly odděleny jednotlivé kroky, je před vlastní definicí funkce definována konstanta `_single_line` představující krátkou jednoduchou oddělující čáru, a konstanta `_double_line` představující delší dvojitou čáru, jimiž metoda jednotlivé části výstupu odděluje.

Definice initoru je jednoduchá, protože pouze vytvoří instanční atributy, do nichž uloží hodnoty argumentů. Jedinou chytrost, kterou bychom do ní mohli vložit, je automatická inkrementace indexu. Tu můžeme zařídit tak, že si třída bude pamatovat index naposledy vytvořené instance a initor vždy toto číslo o jedničku zvětší a uloží do instančního atributu jako index.

Definici třídy `Step` odpovídající předchozím úvahám si můžete prohlédnout ve výpisu 9.1. V definici initoru vás nesmí zaskočit jeho hlavička, která má řadu parametrů, takže samotná zabírá řádky `11-19`. Tělo je pak už jednoduché, protože se v něm pouze inicializují instanční atributy hodnotami stejnojmenných parametrů.

Jedinou výjimkou je atribut `index`, jehož hodnotu metoda spočte sama. Aby tak mohla učinit, je ve třídě definován třídní atribut `last_index`, v němž si třída pamatuje index příští vytvářené instance.

Poměrně jednoduchá je i definice metody `__str__()` na řádcích `32-41`. Teoreticky by v ní bylo možné použít víceřádkový string, ale mně se na něm nelíbí, že pak jeho jednotlivé řádky musejí začínat na začátku řádku kódu, čímž narušují celkové zarovnání. Dávám proto přednost posloupnosti stringů, které *Python* automaticky sloučí. Když ji navíc uzavřu do závorek, tak nemusím na konec každého řádku vkládat zpětné lomítko oznamující, že jsem ještě neskončil a že daný logický řádek pokračuje na následujícím fyzickém řádku.

Anotace deklarující prvky kontejnerů

Některé z vás možná zarazí, proč je v anotacích uveden výraz `tuple[str]`. To je způsob, jakým se naznačuje, že dané argumenty mají být n-tice stringů. Pamatujte na to, až budete chtít uživatelům svých programů předat informace podobného druhu.

Výpis 9.1: *Definice třídy `Step` v modulu `scenarios` v balíčku `game_v1a`*

```
_single_line = 30*'-'
_double_line = 60*'='

class Step:
    """Krok scénáře definuje zadávaný příkaz a stav hry po jeho zpracování.
    Pomocí sekvence kroků tvořících scénář lze otestovat funkcionalitu hry.
    """
    # Atribut třídy pamatující si index naposledy vytvářeného kroku
    last_index:int = -1

    def __init__(self,
            command: str,   # Zadaný příkaz
            message: str,   # Zpráva hry vypsaná v reakci na příkaz
            # Stav hry po provedení příkazu
            place:     str,         # Aktuální prostor
            neighbors: tuple[str],  # Sousedé aktuálního prostoru
            items:     tuple[str],  # H-objekty v aktuálním prostoru
            bag:       tuple[str],  # H-objekty v batohu
        ):
        """Vytvoří krok scénáře, přičemž použije index o jedničku větší,
        než byl index předchozího vytvářeného kroku.
        """
        self.command    = command
        self.message    = message
        self.place      = place
        self.neighbors  = neighbors
        self.items      = items
        self.bag        = bag
        Step.last_index = Step.last_index + 1
        self.index      = Step.last_index

    def __str__(self) -> str:
        """Vrátí uživatelský podpis instance."""
        return (f'{self.index}.\n'
                f'{self.command}\n{_single_line}\n'
                f'{self.message}\n{_single_line}\n'
                f'Aktuální prostor: {self.place}\n'
                f'Sousedé prostoru: {self.neighbors}\n'
                f'Předměty v prostoru: {self.items}\n'
                f'Předměty v batohu:   {self.bag}\n'
                f'{_double_line}')
```

9.6 Definice šťastného scénáře

Scénář můžeme definovat jako n-tici, jejíž jednotlivé prvky budou kroky, tj. instance třídy `Step` definované ve výpisu 9.1. Při tvorbě této instance musíme zadat argumenty definující stav hry po vyplnění příkazu, jenž je prvním argumentem.

Pojďme zadat nějakou hru, která bude dostatečně jednoduchá, ale na druhou stranu umožní otestovat všechny požadavky našeho zadání. Vybral jsem hru inspirovanou pohádkou o Červené Karkulce. Počátek scénáře této hry si můžete prohlédnout ve výpisu 9.2. Celý scénář najdete v doprovodných programech v modulu `scenarios`.

Výpis 9.2: *Definice počátku n-tice reprezentující „šťastný scénář“ vybrané hry v modulu `scenarios` v balíčku `game_v1a`*

```
# Základní úspěšný scénář demonstrující průběh hry,
# v něm hráč nezadává žádné chybné příkazy
# a směřuje co nejkratší cestou k zadanému cíli.
_HAPPY_SCENARIO = (
    _Step('',
          'Vítejte!\n'
          'Toto je příběh o Červené Karkulce, babičce a vlkovi.\n'
          'Svými příkazy řídíte Karkulku, aby donesla věci babičce.\n'
          'Nebudete-li si vědět rady, zadejte znak ?.',
          'Domeček',
          ('Les',),
          ('Bábovka', 'Víno', 'Stůl', 'Panenka', ),
          (),
          ),
    _Step('Vezmi víno',
          'Karkulka dala do košíku objekt: Víno',
          'Domeček',
          ('Les', ),
          ('Bábovka', 'Stůl', 'Panenka', ),
          ('Víno', ),
          ),
    _Step('Vezmi bábovka',
          'Karkulka dala do košíku objekt: Bábovka',
          'Domeček',
          ('Les', ),
          ('Stůl', 'Panenka', ),
          ('Bábovka', 'Víno', ),
          ),
    _Step('Jdi LES',
          'Karkulka se přesunula do prostoru:\n'
          'Les s jahodami, malinami a pramenem vody',
          'Les',
          ('Domeček', 'Temný_les', ),
          ('Maliny', 'Jahody', 'Studánka', ),
          ('Bábovka', 'Víno', ),
          ),
```

9.7 Simulace běhu hry

Než se pustíme do vlastního návrhu testů, můžeme definovat metody, které budou průběh hry simulovat, abychom získali jistou představu, jak by hra mohla probíhat.

Jednoduchá simulace

Ve výpisu 9.3 je zobrazena definice metody `simulate_simple()`, která realizuje velice jednoduchou simulaci průběhu dané hry. Prochází scénářem `_HAPPY_SCENARIO` krok za krokem a zobrazuje, jaký příkaz v daném kroku zadává a co by na to měla hra odpovědět.

Výpis 9.3: *Definice metody `simulate_simple()` v modulu `scenarios` v balíčku `game_v1a`*

```
def simulate_simple() -> None:
    """Vytiskne jednoduchou simulaci běhu hry podle šťastného scénáře,
    přičemž v každém kroku zobrazí pouze příkaz a odpověď hry.
    """
    for step in _HAPPY_SCENARIO:
        print(f'{step.index}. {step.command}\n{_single_line}\n'
              f'{step.message}\n{_double_line}')
```

Importujete-li daný modul a zadáte-li příkaz `simulate_simple()`, rozběhne se simulace, jejíž počátek vidíte ve výpisu 9.4 spolu s příkazy vedoucími k jeho spuštění.

Výpis 9.4: *Import modulu `scenarios` v balíčku `game_v1a` a spuštění metody `simulate_simple()`*

```
============================== RESTART: Shell ==============================
>>> from game.game_v1a.scenarios import simulate_simple
##### game - Společný rodičovský balíček balíčků jednotlivých verzí her
##### game.game_v1a - Balíček s verzí hry na konci 7., resp. 9. kapitoly
      po prvotní analýze (7. kapitola) a
      po návrhu šťastného scénáře a simulačních metod (9. kapitola)
===== Modul game.game_v1a.scenarios ===== START
===== Modul game.game_v1a.scenarios ===== STOP
>>> simulate_simple()
0.
------------------------------
Vítejte!
Toto je příběh o Červené Karkulce, babičce a vlkovi.
Svými příkazy řídíte Karkulku, aby donesla věci babičce.
Nebudete-li si vědět rady, zadejte znak ?.
============================================================
1. Vezmi víno
------------------------------
Karkulka dala do košíku objekt: Víno
============================================================
2. Vezmi bábovka
------------------------------
Karkulka dala do košíku objekt: Bábovka
============================================================
3. Jdi LES
------------------------------
Karkulka se přesunula do prostoru:
Les s jahodami, malinami a pramenem vody
============================================================
...
```

Podrobnější simulace

Předchozí simulace nám zobrazovala jenom průběh hry pro případ, že se chceme rychle dozvědět, jak hra probíhá. Nic jsme se z ní však nedozvěděli o plánovaném stavu hry na konci daného kroku. Zájemci o tuto informaci proto mohou definovat podrobnější metodu, která za odpovědí hry zobrazí ještě očekávaný stav. Díky tomu, že třída `Step` definuje uživatelský podpis svých instancí, je tato metoda jednodušší, přestože toho vypisuje více.

Výpis 9.5: *Definice metody `simulate_with_state()` v modulu `scenarios` v balíčku `game_v1a`*

```
def simulate_with_state() -> None:
    """Vytiskne simulaci běhu hry podle šťastného scénáře,
    přičemž v každém kroku vytiskne za příkazem a odpovědí
    informace o požadovaném stavu hry po provedeném kroku.
    """
    for step in _HAPPY_SCENARIO:
        print(step)
    print("KONEC HRY")
```

Výpis 9.6: *Začátek simulace vypsané metodou `simulate_with_state()`*

```
>>> from game.game_v1a.scenarios import simulate_with_state as sws
>>> sws()
>>> from game.game_v1a.scenarios import simulate_with_state as sws
>>> sws()
0.

------------------------------
Vítejte!
Toto je příběh o Červené Karkulce, babičce a vlkovi.
Svými příkazy řídíte Karkulku, aby donesla věci babičce.
Nebudete-li si vědět rady, zadejte znak ?.
------------------------------
Aktuální prostor: Domeček
Sousedé prostoru: ('Les',)
Předměty v prostoru: ('Bábovka', 'Víno', 'Stůl', 'Panenka')
Předměty v batohu:   ()
============================================================
1.
Vezmi víno
------------------------------
Karkulka dala do košíku objekt: Víno
------------------------------
Aktuální prostor: Domeček
Sousedé prostoru: ('Les',)
Předměty v prostoru: ('Bábovka', 'Stůl', 'Panenka')
Předměty v batohu:   ('Víno',)
============================================================
...
```

9.8 Nezveřejňované atributy

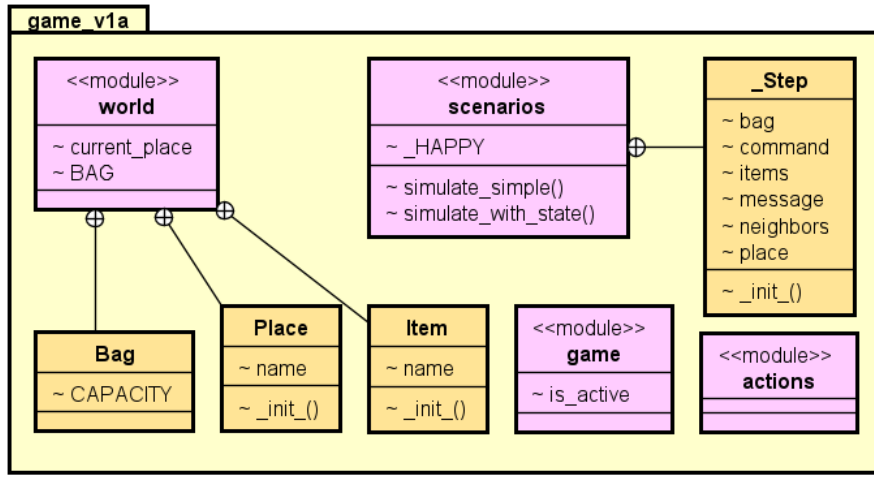
V předchozích programech vás možná udivilo, proč názvy konstant `_single_line` a `_double_line` začínají podtržítkem. Je to proto, že podle mne slouží jenom pro interní potřebu daného modulu a není třeba jejich existenci zveřejňovat do celého světa.

V *Pythonu* totiž platí pravidlo, že atributy, jejichž názvy začínají jedním podtržítkem, jsou považovány za interní atributy daného modulu, resp. třídy. Jsou sice součástí jmenného prostoru (viz podkapitolu *8.9 Jmenné prostory* na straně 127), ale:

- „Hvězdičková verze“ příkazu `from … import *` je neimportuje. Pokud je opravdu potřebujete, musíte je importovat explicitně.
- V dokumentaci daného objektu (modulu, třídy) vyvolané např. voláním funkce `help()` se o nich nedozvíte. Chcete-li se o nich něco dozvědět, musíte se podívat do zdrojového kódu daného objektu.

9.9 Shrnutí

Aktuální diagram tříd balíčku `game_v1a` najdete na obrázku 9.1. Záznam komunikace s interpretem probíhající v této kapitole najdete v trojici souborů nazvaných `m09__test_prepare`. Zdrojové kódy modulů a tříd odpovídající stavu vývoje aplikace na konci této kapitoly (a výše uvedenému UML diagramu) najdete v balíčku `game.game_v1a`, v němž oproti minulé kapitole pouze přibyl modul `game.game_v1a.scenarios`.



Obrázek 9.1:
Diagram tříd balíčku `game_v1a`

Kapitola 10
Rozhodování

Co se v kapitole naučíte
Tato kapitola vás seznámí se směsicí konstrukcí, které budou použity v závěrečném testu. Začne konstrukcemi používanými v případech, kdy se program potřebuje rozhodnout, jak pokračovat. Poté vám představí konstrukci definující jak pokračovat v případě, kdy se ukáže, že v prováděném kódu je chyba. Na závěr ještě vysvětlí, jak se zachovat, chceme-li ve funkci pracovat s jinými proměnnými, než jsou její lokální proměnné.

V předchozí kapitole jsme začali psát testy naší plánované hry. Skončili jsme prozatím simulacemi průběhu hry podle zadaného scénáře. Nyní bychom měli připravit skutečný test, který bude schopen prověřit funkci programu.

Při testování se vždy zadají nějaké příkazy, a pak se prověřuje, zda obdržený výsledek odpovídá požadovanému. Než proto začneme psát skutečný test, musíme se nejprve naučit, jak zakódovat požadavek na rozhodování. Než si ale začneme vysvětlovat, jak se do programu zapisuje rozhodování, musíme si nejprve povědět něco o logických hodnotách, podle nichž se rozhoduje.

10.1 Logické hodnoty

Pro ukládání výsledků logických operací zavedl *Python* datový typ `bool`, který má dvě hodnoty:

- `True` reprezentuje informaci, že tvrzení je pravdivé, podmínka je splněná apod.
- `False` reprezentuje informace, že tvrzení není pravdivé, podmínka není splněná apod.

Na hodnotu typu `bool` lze převést hodnotu jakéhokoliv datového typu. Přitom se postupuje podle pravidla:

- Nula, `None` a prázdný kontejner (např. prázdný string či seznam) se převedou na hodnotu `False`.
- Jakákoliv jiná hodnota se převede na `True`.

Toto pravidlo by měl demonstrovat výpis 10.1. Ve výpisu jsou sice všechny hodnoty převáděny explicitně tak, že jsem je zadal jako argumenty třídě `bool`, ale tento převod probíhá automaticky. Kdykoliv budete mít za úkol zadat logickou hodnotu, stačí zadat libovolnou hodnotu. Musíte ovšem vědět, že se převede na tu, kterou chcete zadat. V dalším textu si to mnohokrát předvedeme.

Výpis 10.1: *Převod libovolné hodnoty na logickou*

```
>>> bool(0), bool(None), bool(''), bool([]), bool({})
(False, False, False, False, False)
>>> bool(1), bool('A'), bool((1,)), bool([2,3]), bool({1, "Raz"})
(True, True, True, True, True)
>>>
```

10.2 Terminologie výrazů

V následujících pasážích si budeme povídat o logických výrazech, tj. o výrazech, jejichž výsledkem je logická hodnota. Nejprve vás ale musím seznámit s několika termíny, které budu používat:

Operace

Operace je to, co se provede. Operace sčítání sečte dva objekty, operace porovnání tyto objekty porovná.

Operátor

Operátor je znak, skupina znaků nebo skupina slov, které oznamují, jaká operace se má v daném místě provést. Chceme-li dvě čísla sečíst, vložíme mezi ně operátor `+` (plus).

Operand

Operandy jsou objekty, s nimiž se provádí operace. Je-li operátor vložen mezi ně, pak hovoříme o levém a pravém operandu.

Arita operátorů

Toto slovo asi mnozí z vás ještě neslyšeli. Arita operátoru říká, kolik operandů se při provádění dané operace zpracovává (kolik operandů vstupuje do dané operace). V *Pythonu* potkáme operátory:

- nulární (bez operandu – např. konstanta `True`),
- unární (s jedním operandem – např. negace),
- binární (se dvěma operandy – např. sčítání),
- ternární (se třemi operandy – podmíněný výraz, který probereme za chvíli),
- kvadrální (čtyři operandy může mít operátor vykrajování).

Priorita operátorů

Použijeme-li v programu nějaký operátor, interpret najde jeho operandy, provede požadovanou operaci a použije obdržený výsledek. Občas ale nastane situace, že ve výrazu je operátorů více a interpret se musí rozhodnout, v jakém pořadí bude jednotlivé operace provádět.

V matematice nás učili, že násobení má vždy přednost před sčítáním, takže napíšeme-li výraz `3 + 5 * 7` (v programování se jako znak násobení používá hvězdička), vynásobí se nejprve `5 * 7` a výsledek se přičte k číslu `3`. Operace násobení má totiž větší prioritu než operace sčítání.

Nepamatujeme-li si, který operátor má před kterým operátorem přednost, ovlivníme pořadí vyhodnocování jednotlivých operátorů vhodným umístěním závorek.

10.3 Porovnávání hodnot

Python definuje standardní šestici operátorů porovnání spolu s dvojicí operátorů pro test totožnosti. Všimněte si, že operátor rovnosti je tvořen dvěma rovnítky – to aby byl viditelně odlišen od znaku označujícího přiřazení.

`<`	Ostře menší.	`>`	Ostře větší.
`<=`	Menší nebo rovno.	`>=`	Větší nebo rovno.
`==`	Rovno.	`!=`	Nerovno.
`is`	Totožno.	`is not`	Není totožno.

Předpokládám, že většina z vás se domnívá, že jim je význam operátorů porovnání jasný, protože s nimi pracuje již od základní školy. Bohužel mezi tím, jak porovnává objekty počítač a jak člověk, je jistý rozdíl, který je třeba mít průběžně na paměti. Pro jistotu proto nejdůležitější odchylky připomenu.

Porovnání reálných čísel

S porovnáváním celých čísel nebývají problémy. Ty ale občas nastanou při porovnávání čísel reálných. Tam se může stát, že čísla, která jsou teoreticky shodná, se budou na konci výpočtu v důsledku zaokrouhlovacích chyb poněkud lišit.

Reálná čísla proto nikdy netestujte na rovnost, ale na to, zda se *dostatečně málo* liší. Co to je „dostatečně málo“, závisí na tom, s jakou přesností jste schopni porovnávaná čísla spočítat. V matematice mají výrazy `(100*0.1)/3` a `(100/3)*0.1` stejnou hodnotu, v programu však zjistíte, že:

```
((100 * 0.1) / 3)  -  ((100 / 3) * 0.1)  ==  -4.440892098500626e-16
```

Zřetězené porovnávání

Python podporuje *zřetězené porovnávání*, při němž pravý operand jednoho porovnání může současně vystupovat jako levý operand následujícího porovnání. Jednotlivá porovnání se vyhodnocují postupně zleva doprava tak dlouho, dokud jsou pravdivá nebo dokud se nedosáhne konce celého řetězu. Takže platí:

```
1 < 5 < 17 < 99;          25 < 125 > 100;
```

Porovnávání textů

Texty jsou porovnávány znak po znaku podle kódu stejnolehlých znaků. Jakmile znaky nejsou shodné, je za větší prohlášen ten text, v němž je znak s větším kódem. Pořadí kódu znaků obecně neodpovídá abecedě, zejména použijete-li znaky s diakritikou. Např. platí

```
'a' > 'B' < 'C' < 'č' > 'd'
```

Stringy můžete porovnávat na rovnost, ale nesnažte se je řadit přesně podle abecedy. K tomu slouží speciální funkce.

Porovnávání totožnosti objektů

Při práci s objekty zjišťuje operátor `==`, zda oba porovnávané objekty reprezentují stejnou hodnotu. Občas ale potřebujeme zjistit, zda se v daném případě jedná o totožný objekt, tj. zda v obou případech pracujeme s informacemi uloženými ve stejném místě v paměti. To zjišťujeme pomocí operátoru `is`.

Rozdíl mezi informacemi obdrženými aplikací těchto operátorů jsem se pokusil demonstrovat ve výpisu 10.2. Na řádku 1 jsou v něm definovány proměnné `a`, `b` a `ab`. Příkazem na řádcích 2-6 je pak nejprve vytištěna hodnota proměnné `ab` a hodnota součtu `(a + b)`. Na dalších řádcích jsou pak obě hodnoty porovnány nejprve na rovnost hodnot a poté na totožnost objektů. Jak sami vidíte, hodnoty jsou chápány jako shodné, ale každý ze stringů je reprezentován jiným objektem, takže objekty nejsou totožné.

Zvědavcům ještě prozradím, že zavírací závorka na řádku 6 je osamocená proto, abych vizuálně oddělil příkaz od následujícího vyhodnocení. Jinak by mohla být klidně na řádku za trojicí apostrofů uzavírajících víceřádkový string. Kdyby řádek 2 nekončil zpětným lomítkem, zabezpečil by odřádkování tištěný string.

Výpis 10.2: *Porovnání hodnot versus porovnání objektů*

```
1  >>> a = 'Dobrý ';   b = 'den!';   ab = a + b
2  >>> print(f'''\
3  Objekty:   {ab = },   {a+b = }
4  Rovnost:   {ab == (a+b) = }
5  Totožnost: {ab is (a+b) = }'''
6      )
7  Objekty:   ab = 'Dobrý den!',   a+b = 'Dobrý den!'
8  Rovnost:   ab == (a+b) = True
9  Totožnost: ab is (a+b) = False
10 >>>
```

10.4 Podmíněný výraz

V programu potřebujeme někdy spočíst nějakou hodnotu, ale to, jak se má daná hodnota spočítat, závisí na nějakých vnějších podmínkách. Představte si např. úlohu, při níž máme znaménko číselné hodnoty říci slovy: je-li číslo kladné, máme před ně zapsat slovo `plus`, je-li záporné, máme před ně napsat `minus`.

Toho můžeme dosáhnout např. pomocí podmíněného výrazu, což je ternární operace s následující syntaxí:

výraz1 `if` *podmínka* `else` *výraz2*

Vyhodnocení podmíněného výrazu začíná vyhodnocením podmínky. Je-li splněná, je výsledkem hodnota výrazu `1`, není-li splněná, je výsledkem hodnota výrazu `2`. Definici takové funkce a její použití najdete ve výpisu 10.3.

Podmíněný výraz je v závorkách na řádku 3. Jeho výsledkem je podle hodnoty argumentu `value` string `'plus '` nebo `'minus '`. K němu pak ještě funkce přidá absolutní hodnotu argumentu `value` převedenou na string.

Všimněte si posledního výrazu na řádku 3: hodnota parametru `value` je předána jako argument funkci `abs()`, která vrátí jeho absolutní hodnotu. Vrácená hodnota je okamžitě předána třídě `str` volané jako funkce, a ta vyrobí string představující obdržený argument převedený na řetězec.

Výpis 10.3: *Podmíněný výraz a jeho použití*

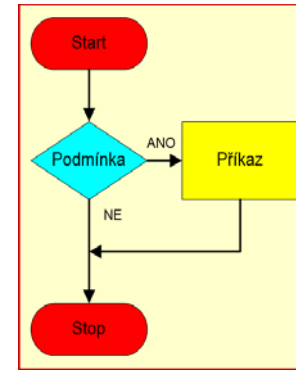
```
1 >>> def ifExpr(value) -> str:
2         """Nahradí znaménko jeho slovním vyjádřením."""
3         return ('plus ' if value > 0 else 'minus ')  +  str(abs(value))
4
5 >>> f'{ifExpr(+5) = },     {ifExpr(-5) = }'
6 "ifExpr(+5) = 'plus 5',     ifExpr(-5) = 'minus 5'"
7 >>>
```

10.5 Podmíněný příkaz

Potřebujeme-li se v programu rozhodnout, jak dále řešit zadanou úlohu, používáme podmíněný příkaz, který vyhodnotí zadanou podmínku a na základě obdrženého výsledku se rozhodne, co bude dělat dál.

Hlavička podmíněného příkazu začíná klíčovým slovem `if`, za nímž následuje podmínka a dvojtečka. Tělo je pak tvořeno blokem příkazů, které se mají provést v případě, je-li podmínka splněna.

Podmíněných příkazů je několik druhů. Pojďme si je postupně probrat.



Obrázek 10.1: *Postup zpracování jednoduchého podmíněného příkazu*

Jednoduchý podmíněný příkaz

Vystačíme-li s tím, že se pouze rozhodujeme, jestli nějaké příkazy provést, či neprovést, jedná se o *jednoduchý podmíněný příkaz*. Postup zpracování jednoduchého podmíněného příkazu je znázorněn na obrázku 10.1. Jeho ukázku si můžete prohlédnout ve výpisu 10.4 na řádcích 5-7 v definici funkce `ifThen`, která řeší stejnou úlohu, jakou jsme řešili před chvílí pomocí podmíněného výrazu.

- Je-li argumentem kladné číslo, podmínka není splněna. Tělo podmíněného příkazu se přeskočí a pokračuje se vykonáváním následujícího příkazu, kterým je tisk textového vyjádření znaménka následovaného hodnotou čísla (řádek 8).
- Je-li argumentem záporné číslo, je podmínka splněna a tělo se provede. V něm se zamění hodnota proměnné `znam` a zadané číslo se nahradí svým kladným protějškem. Poté se pokračuje příkazem tisku (řádek 8), který hodnoty obou proměnných vytiskne.

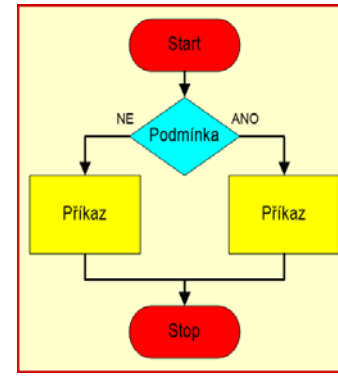
Výpis 10.4: *Použití jednoduchého podmíněného příkazu*

```
1  >>> \
2  def ifThen(číslo) -> None:
3      """Doplní číslo slovním vyjádřením znaménka."""
4      znam = 'plus'       # Předpokládám, že číslo je kladné
5      if číslo < 0:       # Oprava pro záporné číslo
6          znam  = 'minus'
7          číslo = -číslo
8      print (znam, číslo);
9
10 >>> ifThen(5); ifThen(-7)
11 plus 5
12 minus 7
13 >>>
```

Větev `else`

Občas se v programu potřebujeme rozhodnout, zda se má provést jedna či druhá akce. V takovém případě doplníme jednoduchý podmíněný příkaz větví `else`, která vypadá syntakticky jako složený příkaz, jehož hlavička sestává z klíčového slova `else` následovaného dvojtečkou. Jeho tělo pak obsahuje blok příkazů, které se mají provést v případě, že podmínka není splněna.

Jednoduchý podmíněný příkaz doplněný o větev `else` tvoří komplet označovaný jako *úplný podmíněný příkaz*. Vývojový diagram úplného podmíněného příkazu zobrazující postup jeho zpracování si můžete prohlédnout na obrázku 10.2.



Obrázek 10.2: *Vývojový diagram úplného podmíněného příkazu*

Příklad použití úplného podmíněného příkazu najdete ve výpisu 10.5 na řádcích 4–7 v definici funkce `ifElse`. Dělá totéž co předchozí funkce `ifThen`, jenom to dělá trochu jinak. Za jednoduchý podmíněný příkaz přidá větev `else` s blokem příkazů, které se mají provést v případě, že je podmínka nepravdivá.

Nezávisle na tom, který blok příkazů se provedl, se po jeho vykonání pokračuje vykonáváním následujícího příkazu, kterým je v našem případě tisk na řádku 8.

Výpis 10.5: *Použití úplného podmíněného příkazu*

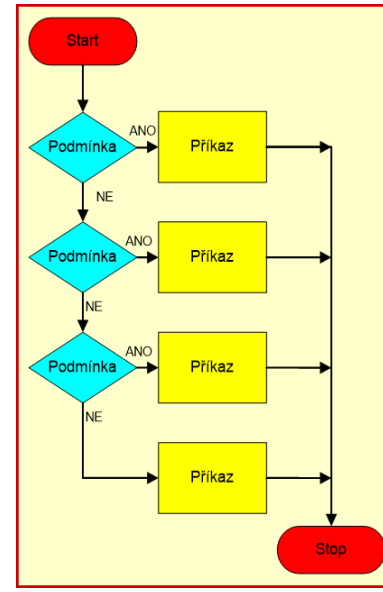
```
>>> \
def ifElse(číslo) -> None:
        """Doplní číslo slovním vyjádřením znaménka."""
        if (číslo > 0):      # Verze pro kladná čísla
            text = "plus " + str(číslo)
        else:                # Verze pro záporná čísla
            text = "minus " + str(-číslo)
        print(text)

>>> ifElse(5);   ifElse(-7)
plus 5
minus 7
>>>
```

Rozhodování s více větvemi: rozšířený podmíněný příkaz

Občas musíme vybírat z více možností než ze dvou. K takovému rozhodnutí ale jedna podmínka nestačí. V takovém případě vložíme za jednoduchý podmíněný příkaz potřebný počet *rozšiřujících podmíněných větví*. Za poslední rozšiřující větví může následovat větev `else`, jejíž tělo definuje příkazy, které se mají provést v případě, kdy žádná z uvedených podmínek není splněna.

Rozšiřující podmíněné větve mají syntaxi složených příkazů, jejichž hlavičky začínají klíčovým slovem `elif` (zkratka z `else if`) následovaným podmínkou pro danou větev a dvojtečkou. Je-li daná podmínka splněna, provedou se příkazy příslušného těla a poté se pokračuje prvním příkazem za celým rozšířeným podmíněným příkazem.

Není-li podmínka splněna, pokračuje se vyhodnocením podmínky následující rozšiřující větve. Pokud již žádná nenásleduje, vykoná se tělo případné větve `else`. Není-li definována ani ta, pokračuje se prvním příkazem za celým rozšířeným podmíněným příkazem.



Obrázek 10.3: *Vývojový diagram zobrazující postup vykonávání rozšířeného podmíněného příkazu*

Použití příkazu si můžete prohlédnout ve výpisu 10.6, kde je definice funkce, které zadáte název ročního období (malými písmeny) a ona vrátí text oznamující, co v zadaném období dělá jabloň.

Výpis 10.6: *Použití rozšířeného podmíněného příkazu*

```
>>> \
def dotaz(sezóna: str) -> str:
    """Vytiskne, co dělá jabloň v zadané sezóně."""
    if   sezóna == 'zima':
        aktivita = 'spí'
    elif sezóna == 'jaro':
        aktivita = 'kvete'
    elif sezóna == 'léto':
        aktivita = 'zraje'
    elif sezóna == 'podzim':
        aktivita = 'plodí'
    else:
        return "Špatně zadaná sezóna: " + str(sezóna)
    return 'Je-li ' + str(sezóna) + ', jabloň ' + str(aktivita)

>>> print(f'{dotaz("jaro")}\n{dotaz("podzim")}\n{dotaz("nevím")}')
Je-li jaro, jabloň kvete
Je-li podzim, jabloň plodí
Špatně zadaná sezóna: nevím
>>>
```

10.6 Tři druhy chyb

Prozatím jsme se rozhodovali pouze na základě nějaké hodnoty. Občas je třeba vybrat následující pokračování podle toho, jestli se v prováděném kódu objevila chyba. Než si ale tuto možnost probereme, povíme si nejprve něco o chybách.

Je známou pravdou, že se málokdy podaří napsat program na první pokus bez chyby. Pravdou je spíše opačné tvrzení – téměř každý program obsahuje nějakou chybu. Chyby v programu bychom mohli rozdělit do tří skupin.

Syntaktické chyby

Syntaktické chyby (syntax errors) jsou prohřešky proti pravidlům zápisu jazyka. Tyto chyby jsou svým způsobem „nejpříjemnější“, protože je odhalí již překladač a poměrně jasně nás na ně upozorní, i když občas přesnou příčinu chyby neodhalí. Řada programátorů je proto ani nepovažuje za chyby. Je to něco, co zůstane mezi nimi a překladačem, a prakticky nikdo jiný se o takovýchto chybách nedozví.

Podle rozsáhlosti a přísnosti syntaktických pravidel pak můžeme programovací jazyky rozdělit na:

- *přísné*, které zadávají řadu pravidel, aby se maximum chyb jevilo jako syntaktické,
- a *benevolentní*, které poskytují programátorům větší svobodu s tím, že pak program důkladně otestují. *Python* patří mezi ty benevolentní.

Běhové chyby

Běhové chyby[10] (runtime errors) jsou chyby, na které se nepodaří přijít během překladu, ale které se projeví až za běhu programu a vedou většinou ke zhroucení systému. Mezi takovéto chyby patří např. dělení nulou, pokus o použití objektu, který ještě nebyl vytvořen, apod.

Logické chyby

Logické chyby (často se setkáte s označením *sémantické*[11] *chyby*) jsou chyby v logice programu, které nevedou k žádnému chybovému hlášení, nicméně zapříčiní, že program nedělá přesně to, co má. Ty se umějí nejlépe maskovat a dá většinou nejvíce práce je odhalit a opravit.

10.7 Reakce na vznik běhových chyb

Někoho možná překvapí, že i s chybami se v programu pracuje jako s objekty. A protože programátoři jsou přesvědčeni, že se v jejich programech objevují chyby jen výjimečně, označují je hromadně jako **výjimky** (anglicky **exceptions**).

Výjimka je objekt, který systém vytvoří v okamžiku, kdy nastane nějaká výjimečná situace: buďto se při vykonávání programu narazí přímo na *chybu* (anglicky *error*), anebo je daná část programu natolik podezřelá, že systém vytvoří výjimku reprezentující *varování* (anglicky *warning*). Do tohoto objektu uloží důležité informace:

- o jaký druh chyby či varování se jednalo (to se pozná podle názvu mateřské třídy dané výjimky),
- na kterém místě programu k němu došlo,
- jak se program do daného místa dostal.

Podle terminologie zavedené v daném jazyce se o vytvořené výjimce říká, že se buď **vyvolá** (anglicky **raise**), nebo **vyhodí** (anglicky **throw**), což znamená, že se další vykonávání programu v daném místě přeruší v aktuálně prováděné funkci, resp. v posloupnosti volání hledá, jestli cestou narazí na někoho, kdo bude připraven výjimku **zachytit** (anglicky **catch**) a **ošetřit** (anglicky **handle**). Pokud nikoho takového nenajde, tak systém ukončí provádění daného programu a vypíše chybovou zprávu.

---

[10] Ve slangu jsou občas označovány jako „*rantajmové*“.

[11] Sémantický = významový.

10.8 Zachycení a ošetření výjimky

Máme-li v programu příkaz (většinou se jedná o volání funkce), jenž může v nějaké situaci vyhodit výjimku, kterou se rozhodneme zachytit a ošetřit, musíme daný příkaz nejprve umístit do bloku (složeného příkazu) `try` – viz nástin syntaxe ve výpisu 10.7.

Za tento blok vložíme blok `except`, v jehož hlavičce uvedeme typ zachytávané výjimky a v jeho těle kód ošetřující vzniklou situaci. Chceme-li v daném bloku s výjimkou pracovat, můžeme v hlavičce předat klíčové slovo `as` a za něj napsat název proměnné, do níž bude výjimka uložena.

Takovýchto bloků může za sebou následovat více. Každý by měl mít v hlavičce uvedenu výjimku (případně n-tici výjimek), jejichž ošetření následující blok obsahuje. Poslední v hlavičce žádnou výjimku uvedenu mít nemusí. Použijeme-li jej, tak se bude vztahovat na všechny výjimky nepokryté předchozími bloky.

Za bloky `except` může ještě následovat blok `finally`, v němž jsou zadány příkazy, které se musí provést nezávisle na tom, jestli v bloku `try` byla vyhozena nějaká výjimka.

Jak už jsem řekl, nástin podoby části kódu, v níž by mohla být vyhozena výjimka, kterou bychom chtěli ošetřit, naznačuje výpis 10.7. Připomínám, že bloků `except` může, ale nemusí být více, a že nepovinný je i blok `finally`.

Výpis 10.7: *Nástin syntaxe příkazů pro zachycení a ošetření výjimky*

```
try:
    Příkazy, které mohou vyhodit výjimku
except Očekávaná výjimka as Název proměnné :
    Příkazy ošetřující danou výjimku
except:
    Příkazy ošetřující zbylé výjimky
finally:
    Příkazy, které se provedou vždy
```

Průchod programu bloky `try … except … finally`

Program postupně provádí příkazy v bloku `try` a pokud vše projde a nebude vyhozena žádná výjimka, přeskočí blok(y) `except`, je-li uveden blok `finally`, pokračuje jím a poté příkazem následujícím za komplexem příkazu `try`.

Vyhodí-li některý z příkazů v bloku `try` výjimku a existuje-li větev `except` s ošetřením pro tuto výjimku, skočí se do příslušné větve `except`. Po vykonání příslušné činnosti se pokračuje do případného bloku `finally`.

Většinu možností jsem se pokusil zachytit v definici funkce `dělení()` ve výpisu 10.8. Abyste vyzkoušeli rozdíl, můžete blok `finally` na chvíli zakomentovat (nebo smazat) a podívat se, jak program pracuje, když tento blok definován není.

Řekl bych, že si reakce na příkazy na řádcích `20` a `23` dokážete odvodit, takže se zmíním pouze o reakci na příkaz na řádku `28`. Při jeho vykonávání byla vyhozena výjimka typu `TypeError`, která spustila ošetření v bloku `except` na řádcích `13–15`. Zde se připraví zpráva, která se předá v argumentu výjimce typu `Exception` vytvářené a vyhazované na řádku `15`.

Výpis 10.8: *Zachycení a ošetření výjimky*

```
>>> \
def dělení(a, b):
    """Zjistí, zda je možné zadané objekty dělit, a pokud ano,
    vrátí výsledek. Pokus o dělení nulou oznámí a vrátí None.
    Nelze-li zadané objekty dělit, vyhodí výjimku.
    """
    try:
        c = a / b
    except ZeroDivisionError as zde:
        print(f'Nepovolená operace: dělení nulou\n'
              f'Vyhozena výjimka {zde}')
        return None
    except:
        zpráva = f'Nelze dělit objekt «{a}» objektem «{b}»'
        raise Exception(zpráva)
    finally:
        print('--- Toto se provede vždy ---')
    return c

>>> f'{dělení(5, 4) = }'
--- Toto se provede vždy ---
'dělení(5, 4) = 1.25'
>>> f'{dělení(5, 0) = }'
Nepovolená operace: dělení nulou
Vyhozena výjimka division by zero
--- Toto se provede vždy ---
'dělení(5, 0) = None'
>>> f'{dělení(5, "4") = }'
--- Toto se provede vždy ---
Traceback (most recent call last):
  File "<pyshell#22>", line 8, in dělení
    c = a / b
TypeError: unsupported operand type(s) for /: 'str' and 'str'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    f'{dělení(a, b) = }'
  File "<pyshell#22>", line 15, in dělení
    raise Exception(zpráva)
Exception: Nelze dělit objekt «Dobrý » objektem «den!»
>>>
```

Všimněte si ale, že vzhledem k tomu, že je definován blok `finally`, tak se před vlastním vyhozením výjimky nejprve provede jeho tělo. Až poté se vyhodí zadaná výjimka.

Chybová zpráva na řádcích 30-42 však hovoří o dvou výjimkách. Na řádcích 30-33 je nejprve popsána výjimka vyhozená v důsledku toho, že jsme chtěli dělit číslo stringem, načež nám řádek 35 oznámí, že při ošetřování této výjimky byla vyhozena jiná výjimka, a o ní pak hovoří zbytek zprávy na řádcích 37-42.

10.9 Použití nelokálních proměnných

V testovací funkci, kterou vám chci v příští podkapitole představit, jsem použil ještě jeden doposud neprobraný obrat, kterým je použití proměnných, jež nejsou v dané funkci lokální. V publikaci [11] jsem na výklad tohoto obratu se všemi souvislostmi spotřeboval deset stránek. Protože vás tu již nechci zdržovat další teorií, zkusím to stručněji.

V pasáži *Definice funkce je obyčejný složený příkaz* na straně 56 jsem vám vysvětlil, že funkci mohu definovat kdekoliv, kde mohu použít příkaz, tedy i uvnitř jiné funkce. To se hodí, potřebujete-li definovat funkci, kterou nepoužijete nikde jinde než uvnitř dané funkce.

Kromě toho jsem vám předtím v pasáži *Definice a použití proměnné, přiřazovací příkaz* na straně 45 vysvětlil, že když do proměnné s daným názvem poprvé uložíme nějakou hodnotu, tak ji tím současně vytvoříme. Zde bych ještě dodal, že se vytvoří v místě použití daného přiřazovacího příkazu. Pak platí:

- Vytvořím-li ji uvnitř funkce či metody, stane se lokální proměnnou dané funkce či metody.
- Vytvořím-li ji uvnitř nějaké třídy, ale mimo definice jejích metod, stane se atributem dané třídy.
- Vytvořím-li ji mimo definice tříd, stane se atributem aktuálního modulu, který bývá označován jako *globální proměnná modulu*.

Potřebuji-li kdekoliv použít atribut třídy, musím jej kvalifikovat danou třídou, a to i uvnitř dané třídy a jejích metod. Potřebuji-li použít globální proměnnou aktuálního modulu, mohu se na ni obrátit přímo. To jsme ostatně udělali i ve výpisu 9.3 na straně 136, kde jsme v metodě `simulate_simple()` použili globální (tj. definované vně metody) proměnné `_single_line` a `_double_line`.

Tehdy jsme ale jejich hodnotu pouze použili a nechtěli ji měnit. V takovém případě postupuje překladač tak, že když danou proměnnou nenajde uvnitř funkce, pokusí se ji najít v rámci modulu.

Příkaz `global`

Zcela jiná situace nastane v okamžiku, kdy bychom chtěli dané proměnné přiřadit novou hodnotu. Bude-li to uvnitř funkce, tak překladač vytvoří novou lokální proměnnou a danou hodnotu jí přiřadí.

Když bychom chtěli přiřadit novou hodnotu globální proměnné, musíme to překladači nejprve oznámit. K tomu slouží příkaz `global`, za nějž napíšeme název globální proměnné (nebo čárkami oddělený seznam názvů), které chceme přiřazovat nějakou hodnotu, např.

```
global proměnná1, proměnná2
```

Překladač je ochoten příkaz akceptovat i v případě, kdy zatím žádná proměnná daného jména neexistuje a vytvoří se až v dané funkcí prvním přiřazením.

S praktickým použitím příkazu `global` se setkáte například ve výpisu 12.3 na straně 168, kde najdete i podrobný rozbor, proč je ho zde nutné použít.

Příkaz `nonlocal`

Obdobná situace nastane, když máme funkci definovanou uvnitř jiné funkce a chceme ve vnitřní funkci pracovat s lokálními proměnnými její vnější funkce. V takovém případě musíme překladači tento svůj záměr oznámit i v případě, kdy se chystáme danou proměnnou pouze číst. K takovému oznámení slouží příkaz `nonlocal` – např.

```
nonlocal proměnná1, proměnná2
```

Tentokrát ale musí daná proměnná již existovat, tj. ve vnější funkci musí být inicializována ještě před definicí dané vnitřní funkce.

10.10 Shrnutí

Záznam komunikace s interpretem probíhající v této kapitole najdete v trojici souborů nazvaných `m10__decisions`.

Kapitola 11
Definice testu hry

Co se v kapitole naučíte

V této kapitole vás nejprve seznámím s přiřazovacím výrazem, který zanedlouho použijeme. Pak doprovodíme hru do stavu, abychom ji mohli začít testovat, a poté definujeme test hry. Ten sice skončí neúspěchem, protož hra ještě není hotová, ale od této chvíle nás test povede dalším vývojem.

11.1 Přiřazovací výraz

V řadě programovacích jazyků není přiřazení příkaz, ale výraz, jehož hodnotou je přiřazovaná hodnota. *Python* jej zpočátku nezavedl, ale řadě programátorů, kteří přišli z jiných jazyků, se po této možnosti stýskalo.

Python proto nakonec zavedl *přiřazovací výraz* (*assignment expression*) a s ním i přiřazovací operátor `:=` (vzhledem k jeho podobě je občas označován jako *mroží operátor* – *walrus operator*), který přiřadí hodnotu pravého operandu do proměnné zadané jako levý operand. Výsledkem této operace je přiřazovaná hodnota.

Když budeme např. chtít získat seřazený seznam hodnot z nějakého generátoru, máme dvě možnosti zobrazené ve výpisu:

- První možnost je zobrazena na řádcích `1` a `3.` Vyžaduje použití dvou příkazů. V prvním příkazu se vytvoří seznam a odkaz na něj se uloží do proměnné. Ve druhém příkazu se pak tento seznam seřadí.
- Druhá možnost je zobrazena na řádku `5`. Při ní se odkaz na vytvořený seznam uloží do proměnné prostřednictvím přiřazovacího výrazu, jehož hodnotou je přiřazovaná hodnota, v našem případě odkaz na vytvořený seznam. Ten je pak vzápětí požádán, aby seznam seřadil.

Nebudu vás přemlouvat, abyste přiřazovací výraz používali. Představují vám jej proto, abyste se při čtení cizích programů při setkání s ním nelekli. Své programy vytvářejte vždy tak, aby pro vás byly co nejpřehlednější. Pamatujte na to, že mnohem více času strávíte čtením vyvíjeného programu, než jeho psaním.

Výpis 11.1: *Demonstrace použití přiřazovacího výrazu*

```
>>> lst_a = list('Python');   print(f'{lst_a = }')
lst_a = ['P', 'y', 't', 'h', 'o', 'n']
>>> lst_a.sort();   print(f'{lst_a = }')
lst_a = ['P', 'h', 'n', 'o', 't', 'y']
>>> (lst_b := list('Python')).sort();   print(f'{lst_b = }')
lst_b = ['P', 'h', 'n', 'o', 't', 'y']
>>>
```

11.2 Balíček game_v1b

V této kapitole začneme doposud definované moduly měnit. Proto jsem pro doprovodné programy zavedl nový balíček `game.game_v1b`, do nějž jsem zkopíroval moduly z balíčku `game.game_v1a`.

Pokud se snažíte paralelně vyvíjet svoji verzi aplikace, nemusíte nic měnit. Já jsem zavedl nový balíček pouze proto, abyste mohli snáze dohledat změny oproti původnímu stavu, který najdete v balíčku `game.game_v1a`.

11.3 Jak budeme testovat

Veškerou teorii potřebnou k napsání testu již znáte, takže se do něj můžeme pustit. Pojďme si ale nejprve rozmyslet, co a jak budeme prověřovat. Připomeňte si současný stav naší aplikace popsaný v podkapitole *7.5 Vytvoření zárodku budoucí aplikace* na straně 109 a vedle návrhu testu si současně ujasněme, jak budeme muset aplikaci upravit, abychom vůbec mohli začít testovat.

Zadání příkazu hře

V testu budeme procházet scénář krok za krokem. Pokaždé zadáme hře příkaz, počkáme na odpověď hry a poté prověříme, zda se hra nachází v očekávaném stavu. Bude-li vše v pořádku, přejdeme na další krok. Objeví-li se nesoulad, oznámíme, čím se skutečný stav liší od očekávaného, a ukončíme hru, aby vývojář mohl zanést opravy.

K tomu musíme navrhnout způsob, jak zadávat hře příkazy. Nejjednodušší možností bude definovat v modulu `game` funkci `execute_command()`, které v argumentu předáme příkaz zadaný hráčem (nebo uložený v kroku scénáře), a ona nám vrátí odpověď hry. Možná prozatímní podoba této funkce je naznačena ve výpisu 11.2.

Tato provizorní funkce nic nedělá, ale záhy začne vznikat požadovaná definice celé hry, kterou tato metoda spustí. Musíme být proto připraveni na to, že někde ve hře může být chyba, která povede k vyhození výjimky, a proto musíme v testu volat tuto metodu v bloku `try` a být připraveni na ošetření případné výjimečné situace.

Výpis 11.2: *Definice modulu `game` v balíčku `game_v1b`*

```
"""
Modul reprezentuje hru.
"""
print(f'===== Modul {__name__} ===== START')
################################################################################

# Příznak toho, zda hra právě běží (True), anebo jen čeká na další spuštění
is_active = False   # Na počátku hra čeká, až ji někdo spustí


def execute_command(command: str) -> str:
    """Zpracuje zadaný příkaz a vrátí string se zprávou pro uživatele."""
    return 'Prozatímní text'


################################################################################
print(f'===== Modul {__name__} ===== STOP')
```

Odpověď a pozice

Po získání odpovědi víme, že hra příkaz zpracovala, a můžeme začít testovat její stav. Především bychom měli prověřit **odpověď hry**, jestli hra opravdu říká to, co jí scénář naplánoval.

Poté bychom měli prověřit, jestli **se hráč nachází ve správném prostoru**. Z podkapitoly *7.5 Vytvoření zárodku budoucí aplikace* na straně 109 ale víme, že odkaz na aktuální prostor je uchováván v atributu current_place modulu world. Protože jej v dalších testech ještě použiju několikrát, bylo by možná vhodné jej uložit do lokální proměnné, abych si o něj nemusel pořád říkat modulu `world`.

Sousedé

Když ověříme, že se nacházíme ve správném prostoru, měli bychom ověřit jeho stav. Především bychom měli zjistit, jestli **má požadované sousedy**.

Ze zadání víme, že různé prostory musejí mít různá jména. Z předchozí pasáže také víme, že při požadavku na přemístění zadá uživatel název prostoru, program musí tento název převést na objekt reprezentující daný prostor a ten uložit do proměnné `current_place`.

Optimálním prostředkem pro převod něčeho na něco jiného bývá většinou slovník. Mohli bychom proto definovat v prostoru jako instanční atribut slovník `name_2_neighbor`, který by převáděl názvy sousedů na příslušné objekty. Rozšíříme tedy definici initoru o definici tohoto atributu, jemuž prozatím přiřadíme prázdný slovník.

H-objekty v prostoru

Obdobně bychom měli prověřit, že **v prostoru jsou ty správné h-objekty**. Tady ale nemůžeme použít mapu, jak jsme to dělali u sousedů, protože v prostoru může být několik objektů se stejným názvem. Protože h-objekty v prostoru mohou průběžně

přibývat a ubývat, uchováme jejich názvy v seznamu pojmenovaném `item_names`. Ten bude dalším instančním atributem prostorů. Rozšíříme proto definici initoru o definici tohoto atributu, jemuž prozatím přiřadíme prázdný seznam.

Výpis 11.3: *Prozatímní definice modulu `world` v balíčku `game_v1b`*

```
"""
Modul world reprezentuje svět a obsahuje definice tříd prostorů,
předmětů (h-objektů) a batohu.
"""
print(f'===== Modul {__name__} ===== START')

################################################################################
class Item():
    """Instance reprezentují h-objekty v prostorech hry a v batohu."""

    def __init__(self, name: str):
        """Vytvoří h-objekt se zadaným názvem."""
        self.name = name


################################################################################
class Place:
    """Instance reprezentují prostory hry."""

    def __init__(self, name: str, description: str):
        """Vytvoří prostor se zadaným názvem a stručným popiskem."""
        self.name = name
        self.description = description
        self.name_2_neighbor = {}
        self.item_names = []


################################################################################
class Bag:
    """Instance třídy reprezentuje batoh."""

    CAPACITY = 0    # Maximální kapacita batohu

    def __init__(self):
        """Vytvoří batoh."""
        self.item_names = []


################################################################################

# Aktuální prostor = prostor, v němž se hráč právě nachází
current_place = None

# Jediná instance batohu
BAG = Bag()

################################################################################
print(f'===== Modul {__name__} ===== STOP')
```

H-objekty v batohu

Nakonec bychom ještě měli prověřit **správný obsah batohu**. Protože zde ale budeme řešit stejný problém jako v případě prostorů, definujeme i pro batohy instanční atribut `item_names`, v němž budou uchovávány názvy h-objektů umístěných v batohu. Rozšíříme proto definici initoru o definici tohoto atributu, jemuž prozatím přiřadíme prázdný seznam.

Protože bude mít batoh pouze jedinou instanci, definujeme v modulu `world` atribut `BAG`, do nějž odkaz na tuto instanci uložíme.

Oznámení o navštíveném prostoru

Při oznamování zvednutého a položeného předmětu se ve scénáři v odpovědi hry vyskytl název daného předmětu (h-objektu), ale při přesunu do sousedního prostoru se vypisovala jeho stručná charakteristika. Měli bychom na to myslet a uložit v každém prostoru jeho stručný popisek.

Souhrn

Definici prozatímní metody v modulu `game` jsem již uvedl ve výpisu 11.2. Nyní ještě přidám výpis 11.3 s upravenou definicí modulu `world`. Sice se od té minulé moc neliší, ale budete ji tu mít vedle testu, jímž ji začneme prověřovat.

11.4 Vlastní test hry

Definici funkce `test_scenario()` si můžete prohlédnout ve výpisu 11.4. Pojďme ji projít a analyzovat.

Přeskočíme prozatím úvodní příkazy a podíváme se rovnou na test na řádcích `21-43`. Zde najdeme cyklus, který postupně prochází scénář krok za krokem a v každém kroku otestuje chování hry v reakci na zadaný příkaz a zapíše průběh.

Na řádcích `22-23` nejprve vytiskne pořadové číslo daného kroku následované zadávaným příkazem. Tento příkaz pak na řádku `25` předá hře a obdrženou odpověď na řádku `26` vytiskne. Poslední dva příkazy jsou v bloku `try`, protože se může stát, že v testované hře bude chyba a zadání daného příkazu skončí vyhozením výjimky.

Byla-li vyhozena výjimka, tak blok `except` tuto skutečnost oznámí, vytiskne očekávaný krok, který se nepovedlo realizovat, a sám pak vyhodí výjimku. Možná vám toto řešení bude připadat zbytečně překombinované, ale berte to tak, že je zde vše připraveno pro to, abychom mohli v budoucnu test vylepšit a např. přidat podrobnější informace o aktuálním stavu hry a možné příčině dané výjimky.

Výpis 11.4: *Definice funkce `test_scenario()` v modulu `scenario` v balíčku `game_v1b`*

```
1  from . import world, game
2
3  def test_scenario() -> None:
4      """Prověří, zda hra pracuje podle zadaného scénáře;
5      zadané příkazy a odpovědi průběžně tiskne.
6      """
7      print(f'Spuštěna metoda {__package__}.{__name__}.test_scenario')
8      from_scenario = None
9      from_game     = None
10
11     def compare_containers(cont1, cont2) -> bool:
12         """Porovná obsah dvou kontejnerů a jako vedlejší efekt uloží do
13         proměnných from_scenario a from_game seznamy s názvy v zadaných
14         kontejnerech převedenými na malá písmena a seřazenými dle abecedy.
15         """
16         nonlocal from_scenario, from_game
17         (from_scenario := [item.lower() for item in cont1]).sort()
18         (from_game     := [item.lower() for item in cont2]).sort()
19         return from_scenario != from_game
20
21     for step in _HAPPY_SCENARIO:
22         print(f'{step.index}.\n{_single_line}\n'
23               f'{step.command}\n{_single_line}')
24         try:
25             answer = game.execute_command(step.command)
26             print(f'{answer}\n{_double_line}\n')
27         except Exception as Ex:
28             print('Při vykonávání příkazu byla vyhozena výjimka:\n', Ex)
29             print('Očekávaný stav po kroku č.', step)
30             raise Exception(Ex)
31
32         if step.message != answer:
33             _error(step, step.message, answer, 'odpověď hry')
34         current_place = world.current_place
35         if  step.place != current_place.name:
36             _error(step, step.place, current_place, 'aktuální prostor')
37         if compare_containers(step.neighbors,
38                               current_place.name_2_neighbor.keys()):
39             _error(step, from_scenario, from_game, 'aktuální sousedé')
40         if compare_containers(step.items, current_place.item_names):
41             _error(step, from_scenario, from_game, 'objekty v prostoru')
42         if compare_containers(step.bag, world.BAG.item_names):
43             _error(step, from_scenario, from_game, 'objekty v batohu')
44
45     print('Hra úspěšně otestována podle šťastného scénáře')
```

Úvodní testy

Pojďme ale dále. Na řádku 32 začíná prověřování stavu hry a toho, nakolik tento stav odpovídá stavu naplánovanému ve scénáři. Nejprve je otestována shoda odpovědi hry s odpovědí naplánovanou. Budou-li se lišit, je zavolána funkce `_error()`, která tuto informaci vytiskne.

Protože se několik dalších testů bude týkat aktuálního prostoru, je na řádku 34 vytvořena proměnná current_place, do níž je uložen aktuální prostor získaný od světa hry. Na řádku 35 se pak otestuje, zda název aktuálního prostoru odpovídá naplánovanému. Neshoduje-li se, opět se zavolá funkce _error(), která uživateli poskytne informace o odhalené chybě.

Funkce _error()

Pojďme se podívat na výpis 11.5, kde je definice funkce _error(). Ta má čtyři parametry. V prvním obdrží krok scénáře, aby mohla zjistit jeho index. V druhém je hodnota očekávaná podle scénáře a ve třetím tatáž hodnota obdržená od hry. Posledním parametrem je string naznačující, co se neshodovalo.

Funkce si na základě obdržených hodnot připraví zprávu, kterou pak vytiskne spolu s oznámením, že následný stisk klávesy ENTER ukončí aplikaci. Poté počká, až uživatel zadá řádek textu (stačí ale pouhý stisk klávesy ENTER) a zavoláním zabudované funkce exit() aplikaci ukončí.

Výpis 11.5: *Definice metody _error() v modulu scenario v balíčku game_v1b*

```
1 def _error(step: _Step, scenario: object, game: object, reason: str) -> None:
2     """Ohlásí zprávu o chybě a ukončí aplikaci."""
3     msg = f'V {step.index}. kroku neodpovídá: {reason}\n' \
4           f'   Očekáváno: {scenario}\n' \
5           f'   Obdrženo:  {game}'
6     print( msg )
7     input('Stisk klávesy Enter aplikaci ukončí')
8     exit(1)
```

Funkce compare_containers()

Vraťme se ale k našim testům. Další příkazy, tj. příkazy na řádcích 37–43, prověřují shodu obsahu kontejnerů. Na řádcích 37–38 se porovnávají naplánované a obdržené názvy sousedů aktuálního prostoru, na řádku 40 názvy h-objektů v aktuálním prostoru a na řádku 42 názvy h-objektů v batohu.

Protože se jedná o netriviální činnost, která bude prováděna opakovaně, je pro ni definována funkce. A protože tuto činnost nebudeme potřebovat provádět nikde jinde, je tato funkce definována jako interní. Je nazvána compare_containers a je definována na řádcích 11–19.

Definice funkce začíná na řádku 16 oznámením, že bude používat proměnné from_scenario a from_game, které jsou definované v její vnější funkci. Ty jsou inicializované na None, protože musejí být definované před tím, než se k nim začne vnitřní funkce hlásit (viz pasáž *Příkaz nonlocal* na straně 151).

Funkce očekává v parametrech kontejnery s názvy objektů. U každého z nich nejprve vytvoří seznam jeho prvků převedených na malá písmena. Tento seznam seřadí a uloží do proměnné. Tam budou k dispozici pro případ, že by se seznamy neshodovaly a bylo třeba o jejich obsahu informovat uživatele.

Na závěr funkce vrátí informaci o tom, zda se seznamy shodují.

Proč na malá písmena

Převod na malá písmena je velmi důležitý, protože v zadání je napsáno, že chod hry nesmí záviset na tom, jakou velikost písmen použije uživatel při zadání příkazu. Ve scénáři je proto střídavě používána různá velikost písmen, aby se ověřilo, že na ni program skutečně nereaguje.

Když pak ale potřebujeme porovnávat naplánovaný obsah kontejneru s obsahem vráceným hrou, také by toto porovnání nemělo záviset na velikost písmen. Bylo by proto vhodné převést všechny používané názvy vždy buď na malá, nebo na velká písmena – vyberte si, co je vám sympatičtější. Já budu v ukázkových programech používat převod na malá písmena.

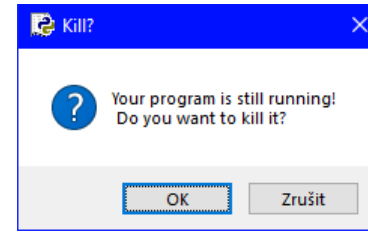
11.5 Spouštíme test

Aby se testy spouštěly co nejsnadněji, přidal jsem za závěrečný tisk o načtení modulu `scenarios` příkaz

```
test_scenario()
```

který spustí test hry podle scénáře v n-tici `_HAPPY_SCENARIO`. Vše je nyní připraveno, můžeme importovat a tím i spustit test naší hry. Příkaz k importu testovacího modulu a následnému spuštění testu a reakce na něj si můžete prohlédnout ve výpisu 11.6.

Když test nalezne chybu, oznámí nám, co našel (viz zpráva na řádcích 20–25) a vyzve nás, abychom stiskli klávesu ENTER a aplikaci ukončili. Pokud jste nedali přednost jinému vývojovému prostředí a pracujete pořád v IDLE, tak se poté objeví dialogové okno z obrázku 11.1, v němž vám systém oznamuje, že váš program stále běží a ptá se, zda jej chcete opravdu „zabít“.



Obrázek 11.1:
Dotaz systému, má-li ukončit běžící program

Odpovíte-li **OK**, ukončíte tím práci IDLE. Odpovíte-li **Zrušit** nebo stisknete klávesu ESC, program se vrátí do IDLE a vypíše standardní výzvu tvořenou třemi většítky, po níž můžete pokračovat v práci.

Výpis 11.6: *Import modulu* `scenarios` *z balíčku* `game_v1b`

```
============================== RESTART: Shell ==============================
>>> import game.game_v1b.scenarios
##### game - Společný rodičovský balíček balíčků jednotlivých verzí her
##### game.game_v1b - Balíček s verzí hry na konci 11. kapitoly
      po definici testu hry
===== Modul game.game_v1b.scenarios ===== START
===== Modul game.game_v1b.world ===== START
===== Modul game.game_v1b.world ===== STOP
===== Modul game.game_v1b.game ===== START
===== Modul game.game_v1b.game ===== STOP
===== Modul game.game_v1b.scenarios ===== STOP
Spuštěna metoda game.game_v1b.game.game_v1b.scenarios.test_scenario
0.
------------------------------

------------------------------
Prozatímní text
============================================================

V 0. kroku neodpovídá: odpověď hry
   Očekáváno: Vítejte!
Toto je příběh o Červené Karkulce, babičce a vlkovi.
Svými příkazy řídíte Karkulku, aby donesla věci babičce.
Nebudete-li si vědět rady, zadejte znak ?.
   Obdrženo:  Prozatímní text
Stisk klávesy Enter aplikaci ukončí
>>>
```

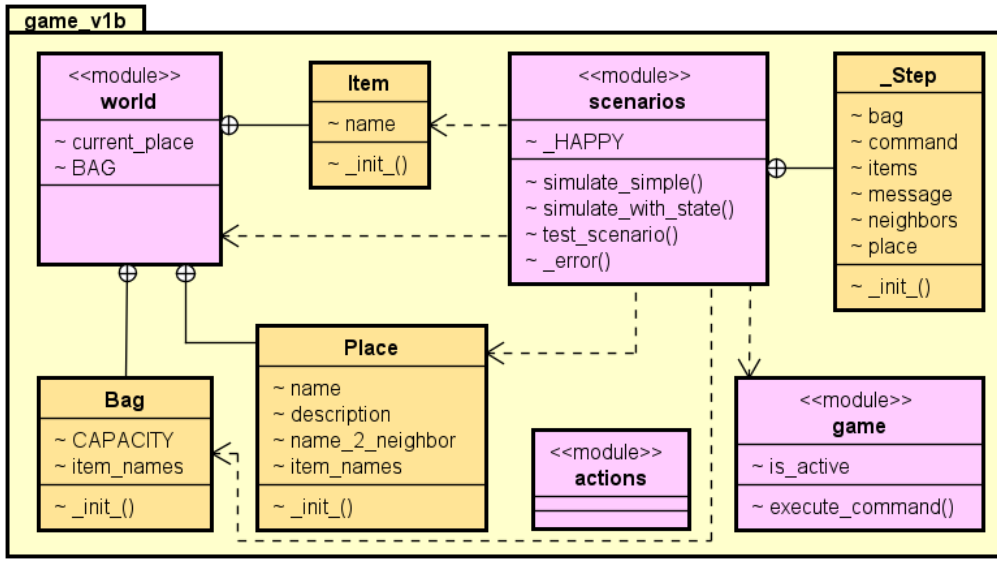
11.6 Další postup

Test jsme spustili a podle očekávání našel hned v úvodním (nultém) příkazu chybu. Nyní už můžeme pokračovat podle doporučení metodiky TDD: opravit nalezenou chybu a znovu spustit test. To vše stále dokola tak dlouho, dokud test neprojde.

Poté se budeme muset podívat, jestli jsme otestovali opravdu vše. Pokud ne, vytvoříme další test a znovu budeme upravovat aplikaci, dokud všechny testy neprojdou.

11.7 Shrnutí

Aktuální diagram tříd balíčku `game_v1b` najdete na obrázku 11.2. Záznam komunikace s interpretem probíhající v této kapitole najdete v trojici souborů nazvaných `m11_test_define`. Zdrojové kódy modulů a tříd odpovídající stavu vývoje aplikace na konci této kapitoly najdete v balíčku `game.game_v1b`, především pak v modulu `game.game_v1b.scenarios`.



Obrázek 11.2:
Diagram tříd balíčku `game_v1b`

Část C
Budujeme aplikaci

Ve třetí části začneme opravdu budovat plánovanou aplikaci. Pod vedením a nápovědou dříve vytvořeného testovacího programu nejprve vytvoříme spustitelný zárodek aplikace, pak si povíme něco o dědění a následně vytvoříme kód reprezentující svět hry. Poté definujeme kód realizující jednotlivé akce, abychom aplikaci přivedli do stavu, v němž kompletně projde testy. Na závěr se naučíte definovat aplikaci jednoduché uživatelské rozhraní a vytvořit její distribuovatelnou verzi spustitelnou jako řádnou aplikaci a reagující na případné argumenty příkazového řádku.

Kapitola 12
Definujeme start hry

Co se v kapitole naučíte
V této kapitole začneme opravdu budovat hru. Nejprve se dozvíte o dělení objektů na datové, výkonné a řídicí a připomeneme si pravidlo jediné odpovědnosti. Pak podle toho upravíme funkci v modulu `game` a definujeme nové funkce v modulu `actions`. Na konci bude úspěšně proveden startovní krok.

Ve výpisu 11.6 na straně 160 jsme se na řádcích `20-25` dozvěděli, že po startu hra vrátila špatnou úvodní zprávu. Podíváme-li se na definici funkce `execute_command()`, která odpověď hry připravila (viz řádky `11-13` ve výpisu 11.2 na straně 154), je vše jasné: jedná se o prozatímní definici, kterou je třeba nahradit. Pojďme se zamyslet nad tím, jak optimálně navrhnout definitivní verzi.

12.1 Balíček `game_v1c`

V této kapitole budeme opět měnit doposud definované moduly. Proto jsem pro doprovodné programy zavedl balíček `game.game_v1c`, do nějž jsem zkopíroval moduly z balíčku `game.game_v1b` a zde je budu měnit.

Jak už jsem dříve říkal, pokud se snažíte paralelně vyvíjet svoji verzi aplikace, nemusíte nic měnit. Nový balíček zavádím pouze proto, abyste mohli později snáze dohledat změny oproti původnímu stavu, který najdete v balíčku `game.game_v1b`.

12.2 Tři druhy objektů

Objekty používané v programech můžeme dělit podle různých kritérií. Podle jednoho z nich je můžeme rozdělit do tří skupin:

- **Datové objekty** – jejich hlavním úkolem je uchovávat nějaká data. Sem bychom mohli zařadit např. různé kontejnery.
- **Výkonné objekty** – jejich hlavním úkolem je něco spočítat nebo jinak zpracovat. V naší hře sem budou patřit např. objekty realizující jednotlivé akce.
- **Řídicí objekty** – jejich hlavním úkolem je organizovat spolupráci ostatních objektů. Jsou to takoví manažeři, kteří si udržují přehled o stavu plnění svěřených úkolů a pověřují své podřízené objekty různými dílčími úkoly.

V každé jenom trochu větší aplikaci by měl být nějaký organizátor. Většinou jich najdete hned několik. V naší aplikaci si o roli hlavního organizátora říká objekt hry.

12.3 Delegování zodpovědnosti

Již jsme si říkali v pasáži *SRP – jediná zodpovědnost* na straně 93, každý objekt má být zodpovědný pouze za jednu věc. Určíme-li, že nějaký objekt bude pracovat jako organizátor, měl by být definován tak, aby se soustředil na delegování různých činností na specializované objekty, které pak danou činnost provedou, a nerozptyloval se zbytečně prováděním pomocných akcí. Měli bychom se při jeho návrhu řídit známým pravidlem: špatný velitel, na kterého zbyde práce.

V naší aplikaci si o roli hlavního organizátora říká objekt hry. Ten zastupuje celou aplikaci vůči okolí. Jako správný organizátor by měl ostatním objektům přidělovat jejich úkoly.

V předchozích úvahách jsme hovořili o tom, že za zpracování příkazů bude zodpovědný modul `actions`. Bylo by tedy vhodné delegovat zpracování příkazu na něj. Můžeme v něm definovat stejnojmennou funkci, kterou funkce v modulu `game` zavolá a obdržený výsledek předá tomu, kdo volal ji, což bude ještě nějakou dobu testovací program. Přitom nesmíme zapomenout, že pro správnou funkci volání funkce v jiném modulu musíme oslovený modul importovat.

Definici nové podoby funkce `execute_command()` si můžete prohlédnout ve výpisu 12.1. Mohli bychom v ní sice zadat volání funkce v modulu `actions` přímo do příkazu `return`, ale takto se můžeme v případě potřeby přesvědčit, co bude funkce vracet.

Výpis 12.1: *Definice metody `execute_command()` v modulu `game` v balíčku `game_v1c`*

```
1 from . import actions
2
3 def execute_command(command: str) -> str:
4     """Zpracuje zadaný příkaz a vrátí string se zprávou pro uživatele."""
5     message = actions.execute_command(command)
6     return message
```

12.4 Funkce execute_command() v modulu actions

Přesuneme se nyní do modulu actions. Ten prozatím obsahuje pouze dokumentační komentář. Pojďme se zamyslet nad tím, jak definovat jeho funkci execute_command().

V podkapitole *7.2 Zadání* na straně 103 jsme definovali, že hru odstartuje prázdný příkaz, ale že tento příkaz se smí zadat pouze tehdy, když hra neběží. Informaci o tom, zda hra běží, ale uchovává atribut is_active v modulu game. Abychom se k němu dostali, museli bychom modul game (nebo alespoň jeho atribut) importovat.

Problém je, že jsme před chvílí v modulu game importovali modul actions, takže se chystáme na import kruhem. To je sice při jisté dávce opatrnosti možné, nicméně je to považováno za velmi křehké řešení náchylné k chybám.

Uděláme to proto raději tak, že tento atribut přestěhujeme z modulu game do modulu actions. Bude-li se někdy někdo hry ptát, jestli je aktivní, tak se hra zeptá modulu actions, který beztak importuje.

Vraťme se k definici metody. Ta musí vyřešit čtyři možné kombinace stavu hry a zadaného příkazu:

- Prázdný příkaz, hra neběží.
- Prázdný příkaz, hra běží.
- Neprázdný příkaz, hra neběží.
- Neprázdný příkaz, hra běží.

Záleží na nás, zda budeme nejprve testovat prázdnost příkazu, nebo spuštěnost hry. Já jsem se rozhodl pro prázdnost zadaného příkazu. Definici funkce najdete ve výpisu 12.2.

Jak vidíte, metoda nejprve odstraní ze zadaného příkazu úvodní a závěrečné mezery (přesněji bílé znaky) a obdrží-li prázdný string, zavolá interní funkci _execute_empty_command(), která tento příkaz zpracuje.

Je-li zadaný příkaz neprázdný, zjistí, zda hra již běží, a pokud ano, tak zavolá funkci _execute_standard_command(), aby zadaný příkaz zpracovala. Pokud však hra neběží, nesmí na takové zadání standardně reagovat a pouze vrátí zprávu s upozorněním uživatele na jeho chybu.

Výpis 12.2: *Definice funkce execute_command() v modulu actions v balíčku game_v1c*

```
def execute_command(command: str) -> str:
    """Zpracuje zadaný příkaz a vrátí text zprávy pro uživatele.
    """
    command = command.strip()   # Smaže úvodní a závěrečné bílé znaky
    if command == '':
        return _execute_empty_command()
    elif is_active:
        return _execute_standard_command(command)
    else:
        return ('Prvním příkazem není startovací příkaz.\n' +
                'Hru, která neběží, lze spustit pouze startovacím příkazem.')
```

Definice má být krátká

Na předchozí definici si všimněte, že řeší pouze základní myšlenku a zpracování detailů deleguje na jiné funkce. Ty však prozatím neexistují a musíme je teprve definovat.

Všeobecná tendence moderního programování je, že definované funkce mají být krátké. Tak dlouhou funkci, jako je např. funkce `test_scenario()` ve výpisu 11.4 na straně 157, by řada puristů považovala za nepřiměřenou. Omluvou pro ni může být jen to, že se v ní prováděla sekvence testů, která neměla žádnou složitou vnitřní strukturu.

12.5 Funkce `_execute_empty_command()`

První z funkcí, které jsme předběžně použili ve funkci `execute_command()`, byla funkce `_execute_empty_command()`, která má zpracovat prázdný příkaz. Její úkol je jednoduchý: pokud hra běží, oznámí nepovolené použití příkazu, pokud hra neběží, tak ji odstartuje (kód najdete ve výpisu 12.3).

Při odstartování musí změnit hodnotu proměnné `is_active` na `True`, protože od této chvíle již hra běží. Kromě toho by měla zabezpečit, že se vše uvede do správného počátečního stavu. Tím můžeme pověřit místní metodu `_initialize()`, kterou můžeme pro začátek definovat jako prázdnou (viz výpis 12.4), a postupně do ní doplňovat inicializační příkazy tak, jak to budou spouštěné testy naznačovat. Start ukončíme vrácením stringu se zprávou o odstartování hry.

Ve výpisu 12.4 je již připraven příkaz spouštějící inicializaci světa hry. Zatím je zakomentovaný, protože jsme danou funkci v modulu `world` ještě nedefinovali. Protože je však jasné, že v nejbližším kroku budeme muset svět hry oživit, je příkaz již připraven a stačí jej jen odkomentovat (a samozřejmě doplnit import modulu `world`).

Důvod použití příkazu `global`

Ve výpisu 12.3 s definicí funkce si všimněte v řádku `5` příkazu `global`. Zkuste jej zakomentovat a uvidíte, že překlad neprojde a překladač bude hlásit `Unresolved reference 'is_active'`.

Kdybychom do proměnné `is_active` nepřiřazovali v řádku `9` hodnotu `True` (opět zkuste řádek zakomentovat), mohli bychom globální proměnnou používat bez deklarace. Překladač by zjistil, že taková lokální proměnná ještě nebyla vytvořena, a použil by globální. Jenže když poprvé přiřazujeme do nedeklarované proměnné, tak se jí překladač snaží vytvořit. Jenomže před chvílí stejnojmennou proměnnou použil. Nebyla sice deklarovaná, ale v globálním jmenném prostoru ji našel. Je z toho proto zmaten a ohlásí chybu.

Kdybychom sice v řádku `9` hodnotu `True` přiřazovali, ale v řádku `6` by nebyl test této proměnné, tak by překlad také proběhl, protože by překladač vytvořil lokální proměnnou `is_active` a hodnotu `True` do ní přiřadil. To by sice od něj byl záludný podraz,

protože my žádnou lokální proměnnou `is_active` nepotřebujeme, ale na takovéto „podrazy“ si musíme v *Pythonu* zvyknout a vyvarovat se jejich přehlédnutí důkladným testováním.

Když použijeme na řádku 5 příkaz `global`, tak se na řádku 6 překladač nesnaží vytvářet lokální proměnnou, protože daná proměnná je již deklarovaná, a vše proběhne tak, jak to proběhnout má.

Výpis 12.3: *Definice metody `_execute_empty_command()` v modulu `actions`*

```
1  def _execute_empty_command() -> str:
2      """Zpracuje prázdný příkaz, tj. příkaz zadaný jako prázdný řetězec.
3      Tento příkaz odstartuje hru, ale v běžící hře se nesmí použít.
4      """
5      global is_active
6      if is_active:
7          return 'Prázdný příkaz lze použít pouze pro start hry'
8      else:
9          is_active = True
10         _initialize()
11         return ('Vítejte!\n'
12                 'Toto je příběh o Červené Karkulce, babičce a vlkovi.\n'
13                 'Svými příkazy řídíte Karkulku, aby donesla věci babičce.\n'
14                 'Nebudete-li si vědět rady, zadejte znak ?.')
```

Výpis 12.4: *Prozatímní definice metody `_initialize()` v modulu `actions`*

```
1  def _initialize() -> None:
2      """Inicializuje všechny součásti hry před jejím spuštěním"""
3      # world.initialize()
```

12.6 Funkce `_execute_standard_command()`

Zbývá nám poslední nedefinovaná funkce – `_execute_standard_command()`. Ta má zpracovávat standardní příkazy. Ty sestávají z názvu spouštěné akce následovaného případnými parametry. Aby se s příkazem lépe pracovalo, je vhodné jej nejprve zavoláním metody `split()` rozdělit na jednotlivá slova. Nulté slovo je pak zadaný název akce a případná další slova jsou argumenty.

Otázkou je, jak poznat, jakou akci je třeba spustit. Optimální se jeví definovat atribut, jímž bude slovník, jehož klíče budou názvy akcí a jehož hodnoty budou odkazy na instance těchto akcí. Aby nezáleželo na zadané velikosti písmen, budou všechny názvy uloženy v malých písmenech.

Protože slovník převádí názvy na akce, pojmenujeme jej podle konvence popsané v podkapitole *8.8 Specifika slovníků* na straně 126 `_NAME_2_ACTION`. Pro začátek může zůstat prázdný, ale jak budeme postupně definovat jednotlivé akce, budeme jej paralelně plnit.

Definice funkce je ve výpisu 12.5. Dotaz na slovník je v bloku `try`, protože není-li ve slovníku dvojice s klíčem zadaným jako index, tak slovník vyhodí výjimku `KeyError`. Pokud se tak stane, funkce vrátí zprávu o neexistující akci.

Najde-li program ve slovníku akci se zadaným názvem, spustí se její metoda `execute`, které se v argumentu předá seznam argumentů zadaného příkazu.

Výpis 12.5: *Definice funkce `_execute_standard_command()` v modulu `actions`*

```
1  def _execute_standard_command(command: str) -> str:
2      """Připraví parametry pro standardní akci hry,
3      tuto akci spustí a vrátí zprávu vrácenou metodou dané akce.
4      Byla-li zadána neexistující akce, vrátí oznámení.
5      """
6      words = command.lower().split()
7      action_name = words[0]
8      try:
9          action  = _NAME_2_ACTION[action_name]
10     except KeyError:
11         return 'Tento příkaz neznám: ' + action_name
12     return action.execute(words)
13
14
15 # Slovník, jehož klíče jsou názvy akcí převedené na malá písmena
16 # a hodnotami jsou příslušné akce
17 _NAME_2_ACTION = {}
```

12.7 Spuštění testu

Vše by mělo být připraveno. Nejvyšší čas spustit test, aby nás navedl, čím pokračovat. Záznam z importu jednotlivých modulů a následného spuštění testu si můžete prohlédnout ve výpisu 12.6.

Záznam je velmi podobný záznamu ve výpisu 11.6 na straně 160. Liší se na počátku tím, že v rámci natahování modulu `game` na řádcích `7-12` se na řádcích `8-11` nejprve natáhne importovaný modul `actions`.

Liší se i výsledek. Zpráva vrácená hrou sice prošla, ale test na řádcích `30-32` oznamuje, že v souboru `scenarios.py` došlo na řádku `113` ve funkci `test_scenario()` k chybě. Program se ptá objektu `current_place` na hodnotu atributu `name`, ale v proměnné je objekt typu `NoneType`, a ten žádný takový atribut nemá.

Nezbývá, než se pustit do vybudování základu světa hry. Před tím si však musíme povědět něco o dědění, což bude tématem příští kapitoly.

Protože jsme v této kapitole řešili primárně podobu balíčku `actions` a balíček `world` jsme nechali netknutý, zatímco ve 14. kapitole to bude právě naopak, rozhodl jsem se řešit problematiku obou kapitol ve společném balíčku.

Výpis 12.6: *Import modulu* `scenarios` *z balíčku* `game_v1c`

```
============================== RESTART: Shell ==============================
>>> import game.game_v1c.scenarios
##### game - Společný rodičovský balíček balíčků jednotlivých verzí her
##### game.game_v1c - Balíček s verzí hry na konci 12. kapitoly
      po definici startu hry
===== Modul game.game_v1c.scenarios ===== START
===== Modul game.game_v1c.world ===== START
===== Modul game.game_v1c.world ===== STOP
===== Modul game.game_v1c.game ===== START
===== Modul game.game_v1c.actions ===== START
===== Modul game.game_v1c.actions ===== STOP
===== Modul game.game_v1c.game ===== STOP
===== Modul game.game_v1c.scenarios ===== STOP
Spuštěna metoda game.game_v1c.game.game_v1c.scenarios.test_scenario
0.
------------------------------

------------------------------
Vítejte!
Toto je příběh o Červené Karkulce, babičce a vlkovi.
Svými příkazy řídíte Karkulku, aby donesla věci babičce.
Nebudete-li si vědět rady, zadejte znak ?.
============================================================

Traceback (most recent call last):
  File "<pyshell#94>", line 1, in <module>
    import game.game_v1c.scenarios
  File "Q:\65_PGM\65_PYT\game\game_v1c\scenarios.py", line 215, in <module>
    test_scenario()
  File "Q:\65_PGM\65_PYT\game\game_v1c\scenarios.py", line 112, in test_scenario
    if  step.place != current_place.name:
AttributeError: 'NoneType' object has no attribute 'name'
>>>
```

12.8 Shrnutí

Záznam komunikace s interpretem probíhající v této kapitole najdete v trojici souborů nazvaných `m12__start_game`. Zdrojové kódy modulů a tříd odpovídající stavu vývoje aplikace na konci této kapitoly najdete v balíčku `game.game_v1c`. Architekturu jsme nijak významně nerozšířili, a proto UML diagram neuvádím.

Kapitola 13
Dědění

Co se v kapitole naučíte
Tato kapitola je opět teoretická a probírá jednu z nejdůležitějších objektových konstrukcí, kterou je dědění. Vysvětlí vám jeho základní principy a pravidla a ukáže vám, jak se v *Pythonu* definuje dědění tříd, a to včetně násobného dědění. Na závěr vás pak seznámí s abstraktními třídami a metodami a naučí vás je definovat.

Zpráva po spuštění testu na konci minulé kapitoly nám ukázala, že nyní bychom se měli zaměřit na vybudování základů světa hry. K jeho kvalitnímu návrhu však potřebujeme něco vědět o jedné z klíčových konstrukcí objektově orientovaného programování, kterou je dědění. Jeho výkladu bude věnována tato kapitola.

13.1 Základní terminologie

Mezi instancemi nějaké třídy můžeme velmi často najít skupinu instancí, které mají určité 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 často označována jako **potomek** (anglicky *child*) původní třídy. Z toho logicky vyplývá, že původní třída bývá pro změnu označována jako **rodičovská třída** (anglicky *parent class*), případně **nadtřída** (anglicky *superclass*) dané podtřídy. A když už jsem u té terminologie, tak někdy bývá rodič označován jako **základní třída** (anglicky *base class*) a potomek jako **odvozená třída** (anglicky *derived class*). A aby vyučující studenty 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** (anglicky direct parents) a naopak třídy, které se k dané třídě hlásí ve svých hlavičkách, označíme jako její **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. Proto se také o dceřiných třídách někdy říká, že *rozšiřují* (anglicky *extends*) svoji rodičovskou třídu.

Hierarchie dědění

V *Pythonu* má každá třída svého předka (může jich mít i víc, ale k tomu se dostaneme za chvíli). Tohoto předka je třeba uvést v hlavičce třídy v kulatých závorkách za názvem třídy. Není-li v hlavičce třídy uveden žádný předek, tak se bezprostředním předkem dané třídy stává třída `object`, která je současně jedinou třídou, jež nemá žádného předka.

13.2 Tři druhy dědění

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.

Dědění rozhraní

Dědění rozhraní (viz podkapitolu *6.6 Rozhraní versus implementace* na straně 96) je v programování klíčové. Zdědím-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**.

Připomínám, 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.

Dědění implementace

Dědění implementace znamená, že potomek deklaruje veškerý rodičem definovaný kód také za svůj. Nemusí tedy definovat pro všechno vlastní metody, ale v řadě případů může použít metodu zděděnou od svého rodiče či od některého z prarodičů.

V dědění implementace jsou skryty mnohé ze záludností, o nichž jsem hovořil. Nezkušený programátor je často tak nadšen tím, že by mohl zdědit implementaci nějaké metody a nemusel tuto metodu programovat, že zcela zapomene na nutnost dodržet také rozhraní. Rodičovské rozhraní pak může kolidovat s naprogramovaným, takže potomek již není schopen plnohodnotně vystupovat v roli instance svého 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 žábu, která umí jezdit na kole. Se schopností skákat po kamenech cyklista automaticky zdědil i schopnost plodit pulce, pořádat večerní žabí koncerty a řadu „užitečných“ žabích schopností.

13.3 LSP – substituční princip Liskové

V květnu 1988 měla Barbara Liskov na konferenci SIGPLAN úvodní přednášku, 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.

Řada 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 nějž ke zděděným atributům se souřadnicemi středu a poloměrem přidáme úhel výseče. Kruhová výseč není speciálním případem kruhu, a proto třída výsečí nemůže být definována jako potomek třídy kruhů.

13.4 Virtuální metody a jejich přebíjení

Když rodičovská třída definuje metodu, jejíž definice se nám v dceřiné třídě nehodí, můžeme v dceřiné třídě definovat metodu se stejným názvem upravenou tak, aby vyhovovala požadavkům dceřiné třídy. (Nechceme-li porušovat LSP, nesmí současně narušovat požadavky rodičovské třídy.) Příslušný atribut dceřiné třídy pak bude 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 v daném kontextu přestane platit.

Není tedy pravda, že se metoda přepíše (overwrite) nebo předefinuje (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 původní verze této metody definovaná, a instance všech potomků této třídy, které metodu nepřebily, ji stále používají v její původní podobě. Původní verze metody může dokonce volat i instance potomka.

V některých jazycích se musí předem uvést, které metody se smějí (C++, C#), resp. nesmějí (*Java*) přebít. Metody, které se smějí přebít, označujeme jako *virtuální*. V *Pythonu* jsou všechny instanční metody automaticky virtuální.

Proces volání virtuální metody bychom mohli interpretovat tak, že se počítač nejprve podívá, zda je daná metoda definována v mateřské třídě daného objektu. Když je, tak ji zavolá, když není, tak se podívá do její rodičovské třídy. Takto postupuje v rodičovské hierarchii až do chvíle, kdy najde třídu definující danou metodu (anebo když ji nenajde, tak vyhodí příslušnou výjimku).

Skutečný mechanismus je sice trochu jiný, výrazně efektivnější, ale chová se stejně, jako kdyby pracoval podle výše uvedeného návodu.

Polymorfismus

Termín *polymorfismus* označuje skutečnost, že se oslovený objekt rozhodne, jak zareaguje na zaslanou 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 zaslanou zprávu spustí (tj. jaká konkrétní metoda se zavolá), záleží na osloveném objektu a na jeho mateřské třídě.

Když programátor definuje funkci, která obdrží v parametru objekt, jehož virtuální metodu bude volat, tak dokud nezná přesný typ argumentu, netuší, kterou metodu zavolá. Musí se spoléhat na to, že metody daného objektu dodržují LSP a mají tedy předvídatelné chování, i když jeho detaily nelze obecně odhadnout.

13.5 Rodičovský podobjekt

Aby instance dceřiné třídy zdědily opravdu všechny schopnosti instancí rodičovské třídy, jsou 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. Důsledkem je sloučení jmenných prostorů dané instance a oněch rodičovských podobjektů.

13.6 Initory v procesu dědění

Aby byly rodičovské podobjekty plnohodnotné, musejí se korektně inicializovat. Jinými slovy, součástí definice initoru objektu by mělo být volání initoru rodičovské třídy, který inicializuje příslušný rodičovský podobjekt.

V *Pythonu* jsou initory standardní metody, které můžeme použít kdekoliv (bohužel). Správně bychom je neměli používat vůbec a nechat jejich použití na třídě, ať si je zavolá při vytváření svých instancí, ale nikdo to nehlídá.

Rodičovský initor (a obecně libovolnou rodičovskou metodu) můžeme volat několika způsoby. Nejpoužívanější způsoby jsou:

- Volání initoru rodičovského podobjektu se kvalifikuje názvem příslušné rodičovské třídy.
- Volání initoru se kvalifikuje instancí třídy `super`, kterou vytvoříte zavoláním konstruktoru. Ten má sice dva parametry (mateřskou třídu volající instance a tuto instanci), ale ty nemusíte zadávat, protože je za vás dosadí překladač.

Názory na výhodnost jednotlivých způsobů se různí. Někteří doporučují jeden, jiní druhý. Kvalifikace rodičovskou třídou je nepatrně rychlejší a řada učebnic ji používá, ale zkušení programátoři před ní varují. Při složitější hierarchii dědění s více předky je totiž tato cesta náchylná k chybám (zájemci, kteří nechtějí zůstat jen na povrchu, najdou podrobný výklad i s příkladem v příručce [11]). Doporučují proto dát přednost volání funkce (konstruktoru) `super()`. Tímto doporučením se budeme řídit i my.

13.7 Definice rodičovské a dceřiné třídy

Řekl bych, že teorie už bylo dost, takže bychom si mohli takovou definici sami vyzkoušet. Ve výpisu 13.1 najdete definici tříd `Matka` a `Dcera` z modulu `m13a_MDPV`. Obě definice si jsou podobné. Obě definují třídní atribut, instanční metodu a initor, který vytváří instanční atribut. Initor tiskne zprávy o svém startu a konci, instanční metoda tiskne informace o své instanci.

Rozdílem, který stojí za povšimnutí, je to, že initor matky již žádný rodičovský initor nevolá, protože initor třídy `object` je prázdný a není třeba jej spouštět. Dcera ale již matčin initor volat musí a opravdu tak na řádku 28 činí.

Metody `mtd()` vypisují atributy svého parametru `self` a jeho mateřské třídy, přičemž metoda třídy `Dcera` na závěr ještě volá stejnojmennou metodu třídy `Matka`.

Výpis 13.1: *Definice tříd `Matka` a `Dcera` v modulu `m13a_MDPV`*

```
 1 from utils import prSK
 2
 3 class Matka():
 4     """Protože třída nedeklaruje explicitně předka,
 5     je bezprostředním potomkem třídy object.
 6     """
 7     cam = 'Třídní atribut matky'
 8
 9     def __init__(self, argm='M', argm2='', **kwds):
10         prSK(1, Matka, f'{argm=}, {argm2=}, {kwds=}')
11         super().__init__(**kwds)
12         self.iam = f'Instanční atribut matky ({argm})'
13         prSK(0, Matka)
14
15     def mtd(self):
16         print(f'Instanční metoda třídy Matka\n'
17               f'   pro instanci typu {type(self)}.\n'
18               f'   {self.cam = }\n   {self.iam = }')
19
20
21 class Dcera(Matka):
22     """Třída je bezprostředním potomkem třídy Matka.
23     """
24     cad = 'Třídní atribut dcery'
25
26     def __init__(self, argd='D', **kwds):
27         prSK(1, Dcera, f'{argd=}, {kwds=}')
28         super().__init__(argm2='dm', **kwds)
29         self.iad = f'Instanční atribut dcery ({argd})'
30         prSK(0, Dcera)
31
32     def mtd(self):
33         print(f'Instanční metoda třídy Dcera\n'
34               f'   pro instanci typu {type(self)}.\n'
35               f'   {self.cad = }\n   {self.iad = }\n'
36               f'   {self.cam = }\n   {self.iam = }')
37         super().mtd()
```

Některé možná překvapí dvouhvězdičkové parametry initorů. K jejich významu se dostaneme za chvíli v pasáži *Návrh třídy s více bezprostředními rodiči* na straně 178.

Ve výpisu 13.2 je pak záznam reakcí na vytvoření instancí a následně na zavolání jejich instanční metody. Domnívám se, že tento záznam nepotřebuje žádný další vysvětlující komentář.

Výpis 13.2: *Záznam vytvoření instancí tříd* `Matka` *a* `Dcera` *a volání jejich metod*

```
>>> import m13a_MDPV as mdpv
===== Modul m13a_MDPV ===== START
===== Modul m13a_MDPV ===== STOP
>>> m = mdpv.Matka()
Matka - START - argm='M', argm2='', kwds={}
Matka - KONEC
>>> d = mdpv.Dcera()
Dcera - START - argd='D', kwds={}
   Matka - START - argm='M', argm2='dm', kwds={}
   Matka - KONEC
Dcera - KONEC
>>> m.mtd()
Instanční metoda třídy Matka
   pro instanci typu <class 'm13a_MDPV.Matka'>.
   self.cam = 'Třídní atribut matky'
   self.iam = 'Instanční atribut matky (M)'
>>> d.mtd()
Instanční metoda třídy Dcera
   pro instanci typu <class 'm13a_MDPV.Dcera'>.
   self.cad = 'Třídní atribut dcery'
   self.iad = 'Instanční atribut dcery (D)'
   self.cam = 'Třídní atribut matky'
   self.iam = 'Instanční atribut matky (M)'
Instanční metoda třídy Matka
   pro instanci typu <class 'm13a_MDPV.Dcera'>.
   self.cam = 'Třídní atribut matky'
   self.iam = 'Instanční atribut matky (M)'
>>>
```

13.8 Násobné dědění a diamantový problém

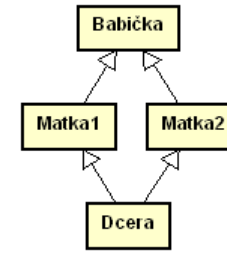
V jazycích *Simula 68* a *Smalltalk*, které stály u zrodu OOP, měla třída povoleného pouze jediného bezprostředního předka. Takovýto způsob dědění bývá označován jako *jednoduché dědění* (anglicky *simple inheritance*).

Problém jednoduchého dědění spočívá v tom, že reprezentovaný objekt může být speciálním případem několika obecnějších množin objektů. Autoři programovacích jazyků proto 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* (anglicky *multiple inheritance*).

Zkušenosti s používáním více bezprostředních rodičů v jazyce C++ ukázaly řadu možných problémů, vyvolaných možností společného prarodiče několika rodičů (viz obrázek 13.1), které byly hromadně označeny jako *diamantový problém* (*diamond problem*).

Touto problematikou se zde ale nebudu podrobněji zabývat, zájemce odkážu na obligátní zdroj [11]. Jen bych chtěl, abyste si zapamatovali, že dokud nezískáte dostatek zkušeností, měla by jediným společným prarodičem vašich tříd být třída `object`.



Obrázek 13.1: *Diamantový problém*

Návrh třídy s více bezprostředními rodiči

Diamantovým problémem se zabývat nebudeme, nicméně zcela na násobné dědění zanevřít nemůžeme, protože jednou za čas se ukazuje jako velice užitečné. Ve výpisu 13.3 jsem vám proto předvedl definici druhé rodičovské třídy nazvané `Přítelkyně` a pak definici třídy `VnučkaDP`, která je společnou dceřinou třídou tříd `Dcera` a `Přítelkyně` v uvedeném pořadí, které jsou také definované v modulu `m13a_MDPV`. Kromě nich je v tomto modulu definovaná i třída `VnučkaPD`, která je společnou dceřinou třídou tříd `Přítelkyně` a `Dcera` a jejíž definice se liší pouze pořadím jejích rodičů.

Věnujte pozornost hlavně initoru třídy `VnučkaDP` definovanému na řádcích `14-18`. Třída má dva bezprostřední předky, a musí se proto postarat o inicializaci dvou rodičovských podobjektů. Volá však pouze initor kvalifikovaný instancí `super()` (řádek `16`) a spoléhá na to, že ta se o vše postará, a to včetně předání správných argumentů jednotlivým initorům.

Aby byla definice jednodušší, tak třída `Přítelkyně` definuje pouze jeden instanční atribut a žádnou metodu a stejně tak třída `VnučkaDP` zděděnou metodu nepřebíjí, ale spokojí se s tou zděděnou.

Výpis 13.3: *Definice tříd `Přítelkyně` a `VnučkaDP` v modulu `m13a_MDPV`*

```
1  class Přítelkyně():
2      """Třída je bezprostředním potomkem třídy object.
3      """
4      def __init__(self, argp='P', **kwds):
5          prSK(1, Přítelkyně, f'{argp=}, {kwds=}')
6          super().__init__(**kwds)
7          self.ipr = 'Instanční atribut přítelkyně'
8          prSK(0, Přítelkyně)
9
10
11 class VnučkaDP(Dcera, Přítelkyně):
12     """Třída je bezprostředním potomkem tříd Dcera a Přítelkyně.
13     """
14     def __init__(self, argv='DP', **kwds):
15         prSK(1, VnučkaDP, f'{argv=}, {kwds=}')
16         super().__init__(**kwds)
17         self.iav = f'Instanční atribut první vnučky ({argv})'
18         prSK(0, VnučkaDP)
```

Ve výpisu 13.4 najdete záznam vytváření instance třídy `VnučkaDP` následovaný na řádku `10` záznamem vytváření instance třídy `VnučkaPD`. Všimněte si, jak se instance třídy `super` postupně postarala o volání jednotlivých initorů, které ze slovníku reprezentujícího dvojhvězdičkový parametr odebíraly hodnoty svých argumentů.

Současně si všimněte, jak změna pořadí deklarace rodičů ovlivní postup volání jednotlivých initorů. Sami si pak můžete vyzkoušet, že pro vnučky jsou k dispozici instanční atributy obou jejich rodičů.

Aby to vše fungovalo, je důležité, aby rodičovský initor volaly i initory tříd, které jsou bezprostředními potomky třídy `object`, a které by proto teoreticky rodičovský initor volat nemusely. Můžete zkusit zakomentovat volání initorů ve třídách `Matka` a `Přítelkyně` a zjistíte, že se inicializoval pouze první z rodičů. Jakmile však jeden z nich odkomentujete, vše opět poběží správně. Doporučuji však neriskovat a používat je u obou tříd.

Výpis 13.4: *Záznam vytvoření instancí tříd `VnučkaDP` a `VnučkaPD` z modulu `m13a_MDPV`*

```
>>> vDP = mdpv.VnučkaDP(argm='vM', argd='vD', argp='vP')
VnučkaDP - START - argv='DP', kwds={'argm': 'vM', 'argd': 'vD', 'argp': 'vP'}
   Dcera - START - argd='vD', kwds={'argm': 'vM', 'argp': 'vP'}
      Matka - START - argm='vM', argm2='dm', kwds={'argp': 'vP'}
         Přítelkyně - START - argp='vP', kwds={}
         Přítelkyně - KONEC
      Matka - KONEC
   Dcera - KONEC
VnučkaDP - KONEC
>>> vPD = mdpv.VnučkaPD(argm='vM', argd='vD', argp='vP')
VnučkaPD - START - argv='PD', kwds={'argm': 'vM', 'argd': 'vD', 'argp': 'vP'}
   Přítelkyně - START - argp='vP', kwds={'argm': 'vM', 'argd': 'vD'}
      Dcera - START - argd='vD', kwds={'argm': 'vM'}
         Matka - START - argm='vM', argm2='dm', kwds={}
         Matka - KONEC
      Dcera - KONEC
   Přítelkyně - KONEC
VnučkaPD - KONEC
>>>
```

13.9 Zobecňování

Řekli jsme si, že definicí potomka zavádíme specializaci, protože definujeme třídu, jejíž instance patří do speciální podmnožiny instancí předka. Při návrhu programu ale občas postupujeme i obráceně: když vidíme, že instance skupiny tříd mají společné vlastnosti 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, 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 (viz *DRY – bez kopií* na straně 93).

13.10 Abstraktní třídy

Jednou za čas je vhodné definovat společného rodiče skupiny tříd, který reprezentuje nějakou abstrakci, od níž není vhodné vytvářet instance. Rodičovské třídy, jimž je zakázáno vytvářet instance, protože vytváření instancí je povoleno pouze jejich potomkům, označujeme jako *abstraktní třídy* (anglicky *abstract classes*).

Abstraktní třídy mívají často definovány také *abstraktní metody*, což jsou metody s prázdným tělem, u nichž se od všech neabstraktních potomků dané třídy vyžaduje, aby je definovaly. Pokud některou ze zděděných abstraktních metod nedefinují, je to považováno za syntaktickou chybu, z níž se daná třída může vykroutit jedině tak, že se také prohlásí za abstraktní.

Abstraktní třídy se v *Pythonu* definují tak, že se v hlavičce třídy uvede pojmenovaný argument `metaclass` s hodnotou `ABCMeta`, což je třída, kterou je třeba nejprve importovat z modulu `abc`. Z téhož modulu se importuje i dekorátor `abstractmethod`, který slouží k označování abstraktních metod.

AHA-příklad s definicí abstraktních tříd tu uvádět nebudu, protože si jejich definici v následujících kapitolách několikrát vyzkoušíte.

13.11 Shrnutí

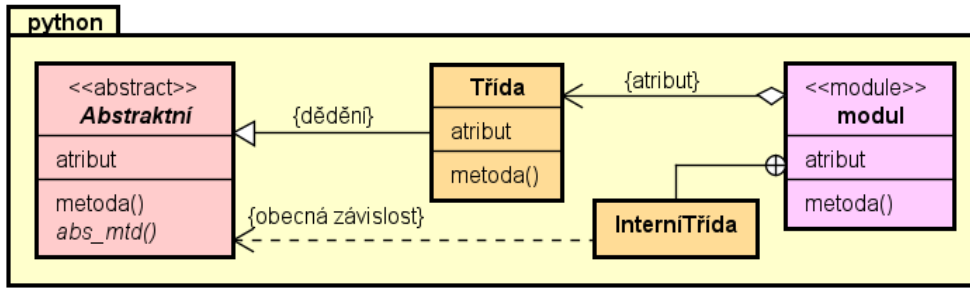
Záznam komunikace s interpretem probíhající v této kapitole najdete v trojici souborů nazvaných `m13__inheritance`.

Kapitola 14
Vytváříme svět hry

Co se v kapitole naučíte
Jak název napovídá, v této kapitole začneme vytvářet svět hry. Upravíme definice některých tříd a vytvoříme základní sadu prostorů, v nichž se bude hráč pohybovat.

14.1 Pravidla pro kreslení UML diagramů

Naše úvahy o dalším postupu začneme opět analýzou UML diagramu zobrazujícího aktuální architekturu hry. Než ale s touto analýzou začneme, připomenu pár pravidel, která budu při tvorbě diagramů tříd dodržovat. Jejich realizaci si můžete prohlédnout na obrázku 14.1.



Obrázek 14.1:
Prvky používané při tvorbě UML diagramů

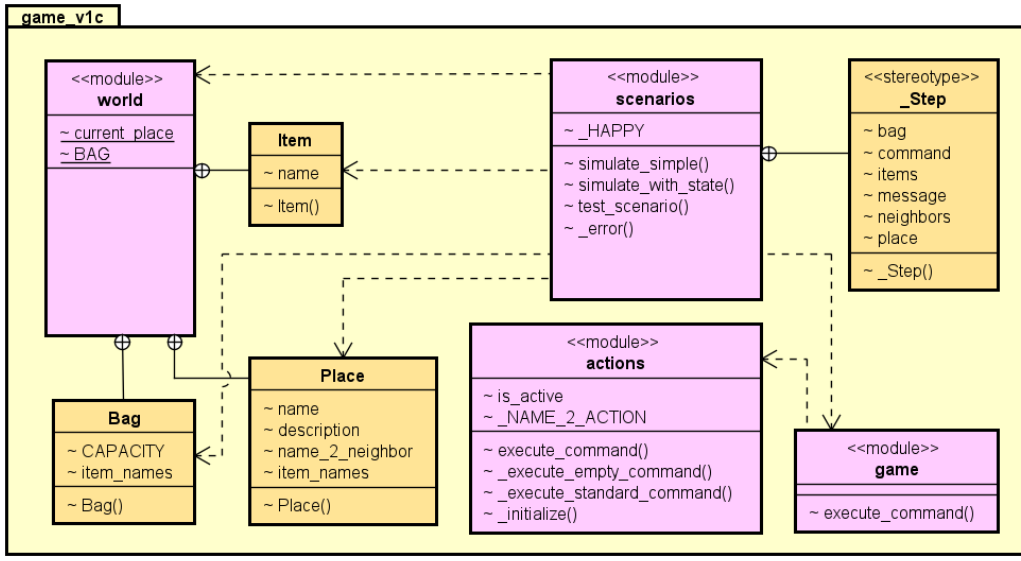
- Třídy a objekty se zobrazují jako obdélníky, které mohou být rozděleny na několik částí.
  - V horní části je tučně zobrazen název dané třídy či objektu.
  - Ve střední části se zobrazují definované atributy.
  - Ve spodní části se zobrazují definované metody.

U atributů a metod si můžete vybrat, zda se vůbec budou zobrazovat, a pokud je zobrazíte, tak zda je zobrazíte všechny, anebo jenom ty veřejné, které patří do oficiálního rozhraní daného objektu.

- Informace mohou být doplněny tzv. *stereotypy* zapisovanými jako texty uzavřené ve «francouzských uvozovkách».
- Dědění se zobrazuje šipkou tvořenou plnou čárou zakončenou trojúhelníkovou hlavičkou, která směřuje od dceřiné třídy k rodičovské třídě.
- Závislosti mezi jednotlivými objekty se zobrazují prostřednictvím čar či šipek. Já budu používat výhradně šipky.
  - To, že objekt používá jako atribut instanci nějaké třídy, budu znázorňovat plnou šipkou s čárovou hlavičkou směřující k třídě, jejíž instance je použita. U paty čáry bude navíc zobrazen prázdný kosočtverec (diamant).
  - Obecnější závislost (např. instance třídy je použita jako parametr či návratová hodnota, vystupuje ve výrazu apod.) budu znázorňovat čárkovanou šipkou s čárovou hlavičkou směřující od objektu, který něco používá, k objektu, který je používán.
- To, že je objekt definován uvnitř jiného objektu, se znázorňuje čarou vedoucí od interního objektu k jeho majiteli, přičemž na hraně majitele se zobrazí kroužek se znaménkem +, jak je to na obrázku předvedeno u interní třídy definované uvnitř modulu.
- To, že třída nebo metoda je abstraktní, se zdůrazňuje vysazením jejího názvu kurzivou. Protože mi to připadá příliš nenápadné, budu u abstraktních tříd doplňovat stereotyp `«abstract»`.
- Jak už jsem se zmínil v podkapitole *7.7 UML diagram* na straně 114, v UML je oficiální způsob zobrazování běžných objektů (tj. ne datových typů) poměrně nenápadný a navíc jej většina nástrojů na kreslení UML diagramů nepodporuje. Budu je proto označovat prostřednictvím stereotypu `«module»`.
- Barvy nejsou předepsané, ale já budu vybarvovat moduly zeleně, běžné třídy krémově a abstraktní třídy růžově, aby byly základní informace patrné již z dálky bez nutnosti je číst. Majitelé papírových knih, kteří mají obrázky pouze černobílé, si mohou barevné UML diagramy stáhnout na stránce knihy.

14.2 Aktuální UML diagram

Pojďme si připomenout současnou podobu naší aplikace prostřednictvím diagramu tříd, v němž jsou tentokrát již zobrazeny všechny doposud definované veřejné atributy a metody a s nimi i vzájemné závislosti. Diagram si můžete prohlédnout na obrázku 14.2.



Obrázek 14.2:
Diagram tříd balíčku `game_v1c`

14.3 Přípravné akce, balíček `game_v1d`

Vraťme se ke kódu naší hry. Začneme rozlučkou v podkapitole *12.7 Spuštění testu* na straně 169 a především pak výpisem 12.6. Zde jsme se ze závěrečné chybové zprávy dozvěděli, že v modulu `world` ještě není správně inicializován atribut `current_place` uchovávající odkaz na aktuální prostor, tj. na prostor, v němž se hráč právě nachází.

Abyste mohli co nejsnáze zpětně analyzovat realizované změny, vytvořil jsem balíček `game_v1d` a zkopíroval do něj obsah balíčku `game_v1c`. Prostřednictvím úprav, které budeme provádět v této kapitole, postupně převedeme jeho obsah do podoby, v níž jej najdete v doprovodných programech.

Pojďme na to. Začneme tím, že si vzpomeneme na výpis 12.4 na straně 168 se zakomentovaným příkazem k inicializaci modulu `world`. Příkaz odkomentujeme a abychom vše zprovoznili, přidáme na začátek modulu ještě příkaz

```
from . import world
```

14.4 Pojmenované objekty

Přejděme nyní k modulu `world`, jehož vylepšení bude věnován zbytek této kapitoly. Ze tří tříd, které jsou definovány v modulu `world`, mají dvě své instance pojmenované. Přidáme-li požadavek na pojmenování jednotlivých akcí, naznačuje to, že by bylo vhodné definovat pro ně jednoho společného abstraktního rodiče, který se bude starat o záležitosti související se jmény.

Třídu nazveme ANamed, přičemž počáteční A má symbolizovat, že se jedná o abstraktní třídu. Jako abstraktní ji definujeme proto, že nepotřebujeme nějaké obecné pojmenované instance, ale vždy pouze instance některého z potomků. Její prozatímní definici si můžete prohlédnout ve výpisu 14.1. Protože se jedná o první abstraktní třídu v daném modulu, je součástí výpisu i potřebný import.

Všimněte si, že třída definuje metodu __repr__(), která se zavolá při žádosti o podpis objektu. Metoda __str__() definována není, tak se bude systémový i uživatelský podpis shodovat a bude jím název daného objektu.

Výpis 14.1: *Definice třídy ANamed v modulu world v balíčku game_v1d*

```
from abc import ABCMeta

class ANamed(metaclass=ABCMeta):
    """Společný rodič všech tříd s pojmenovanými instancemi,
    v případě našich textových her je to rodič h-objektů, prostorů a akcí.
    """

    def __init__(self, name):
        """Zapamatuje si jméno dané instance."""
        self.name = name

    def __repr__(self) -> str:
        """Pojmenovaný objekt se bude podepisovat svým názvem."""
        return self.name
```

14.5 Úprava definice třídy Item

Instance tříd Place a Bag obsahují kontejnery položek typu Item, takže by určitě bylo vhodné definovat tuto třídu před tím, než se pustíme do těch zbylých.

Úprava definice třídy Item bude jednoduchá: definujeme třídu jako potomka třídy ANamed a v initoru zavoláme initor rodičovské třídy. Upravenou definici si můžete prohlédnout ve výpisu 14.2.

Výpis 14.2: *Upravená definice třídy Item v modulu world v balíčku game_v1d*

```
class Item(ANamed):
    """Instance reprezentují h-objekty v prostorech hry a v batohu."""

    def __init__(self, name: str):
        """Vytvoří h-objekt se zadaným názvem."""
        super().__init__(name)
```

14.6 Úprava definice třídy Place

Podívejme se nyní na třídu Place, jejíž instance reprezentují prostory hry. Měly by si pamatovat sousedy daného prostoru a h-objekty, které se v něm nacházejí. K tomu se ale musí nejprve dozvědět, které to zpočátku jsou. Upravíme proto definici initoru a přidáme mu parametry, v nichž obdrží názvy sousedů a názvy uchovávaných h-objektů na počátku hry.

Vhledem k tomu, že tyto údaje budou zadány jako n-tice (viz anotace v hlavičce), o nichž víme, že se nemohou změnit, bude nejjednodušší uložit odkazy na ně do stejnojmenných atributů. Při inicializaci na počátku hry se inicializují atributy, které se na základě těchto názvů naplní odpovídajícími objekty.

Toto jsou však pouze výchozí objekty, avšak v průběhu hry se mohou jak h-objekty v prostoru, tak jeho sousedé měnit. Výchozí objekty si však potřebujeme zapamatovat, abychom je mohli znovu nastavit při příštím startu hry. V zadání totiž je, že hra musí jít znovu spustit bez vypnutí celé aplikace.

Proto se také test neptá na výchozí stav, ale na atributy pojmenované item_names a name_2_neighbor. Definujeme proto i tyto atributy a inicializujeme je.

S uložením seznamu názvů h-objektů nebude problém. Horší to bude se sousedy, protože u nich jsme se rozhodli uchovávat informace ve slovníku. Přitom je evidentní, že o daných sousedech víme prozatím jen to, jak se jmenují. Některé z budoucích sousedů jsme již možná vytvořili, ale další se budou teprve vytvářet.

S tím nám ale může pomoci metoda fromkeys popsaná v podšeděném rámečku. Pomocí této metody vytvoříme slovník se zadanými klíči, ale bez hodnot, a až se bude hra inicializovat, tak ke klíčům příslušné prostory doplníme.

Nyní byste již uměli třídu upravit sami. Svoji verzí si můžete porovnat s verzí ve výpisu 14.3.

Výpis 14.3: *Upravená definice třídy Place v modulu world v balíčku game_v1d*

```
class Place(ANamed):
    """Instance reprezentují prostory hry."""

    def __init__(self, name: str, description: str,
                 initial_neighbor_names: Tuple[str],
                 initial_item_names:     Tuple[str]):
        """Vytvoří prostor se zadaným názvem a stručným popisem napojený
        zpočátku na zadané sousedy a obsahující zadané h-objekty."""
        super().__init__(name)
        self.description            = description
        self.initial_item_names     = initial_item_names
        self.initial_neighbor_names = initial_neighbor_names
        self.item_names             = list(initial_item_names)
        self.name_2_neighbor        = dict.fromkeys(initial_neighbor_names)
```

Vytváření slovníků pomocí metody `fromkeys()`

Jednou za čas potřebujeme vytvořit slovník jako schránku na budoucí hodnoty sdružené s předem známými klíči. Tyto hodnoty se mohou v průběhu další práce teprve zjišťovat, ale již od začátku může být jasné, jaké budou odpovídající klíče. Položky vytvářeného slovníku proto můžeme inicializovat nějakou hodnotou, jež nám umožní poznat, jestli už hodnota dané položky byla zadána, anebo která má smysl do chvíle, než ji někdo změní.

Pro tento účel je možné použít metodu `fromkeys()` třídy `dict`. Tato metoda má dva parametry, přičemž druhý je volitelný. Prvním parametrem je zdroj klíčů vytvářeného slovníku, druhým je implicitní hodnota jeho položek. Není-li druhý parametr zadaný, použije se hodnota `None`.

Připomínám, že vzhledem k tomu, že se jedná o metodu třídy, musím její volání kvalifikovat její třídou, tj. musím ji volat `dict.fromkeys(…)`.

Výpis 14.4: *Definice atributu `_NAME_2_PLACE` v modulu `world` v balíčku `game_v1d`*

```
_NAME_2_PLACE = { p.name.lower(): p for p in (
    Place('Domeček',
          'Domeček, kde bydlí Karkulka',
          ('Les',),
          ('Bábovka', 'Víno', 'Stůl', 'Panenka', ),
         ),
    Place('Les',
          'Les s jahodami, malinami a pramenem vody',
          ('Domeček', 'Temný_les',),
          ('Maliny', 'Jahody', 'Studánka', ),
         ),
    Place('Temný_les',
          'Temný_les s jeskyní a číhajícím vlkem',
          ('Les', 'Jeskyně', 'Chaloupka', ),
          ('Vlk', ),
          ),
    Place('Chaloupka',
          'Chaloupka, kde bydlí babička',
          ('Temný_les', ),
          ('Postel', 'Stůl', 'Babička', ),
         ),
    Place('Jeskyně',
          'Jeskyně, kde v zimě přespává medvěd',
          ('Temný_les', ),
          (),
         ),
)}
```

14.7 Vytvoření prostorů hry

Vypadá to, že třída `Bag` prozatím žádnou úpravu nepotřebuje, takže máme všechny třídy připravené. Nyní bychom tedy měli vytvořit svět hry.

Začneme tím, že vytvoříme všechny pokoje a uložíme je do kontejneru. Optimální by asi bylo, kdyby tímto kontejnerem byl slovník, abychom mohli pokoje oslovovat prostřednictvím jejich názvu. Jenom nesmíme zapomínat na to, že fungování aplikace nesmí záviset na velikost zadaných znaků, takže bychom názvy v klíčích měli hned převést na malá písmena.

Definici atributu nazvaného `_NAME_2_PLACE` a vytvořeného podle uvedených zásad si můžete prohlédnout ve výpisu 14.4.

14.8 Inicializace aktuálního prostoru a test

Nyní již zbývá inicializovat atribut `current_place` výchozím prostorem. Upravíme proto inicializační příkaz do tvaru

```
current_place = _NAME_2_PLACE['domeček']
```

a můžeme spustit test. Vynechám-li standardní začátek a úspěšné zpracování nultého příkazu, tak zprávu o testu reakce na první příkaz najdete ve výpisu 14.5.

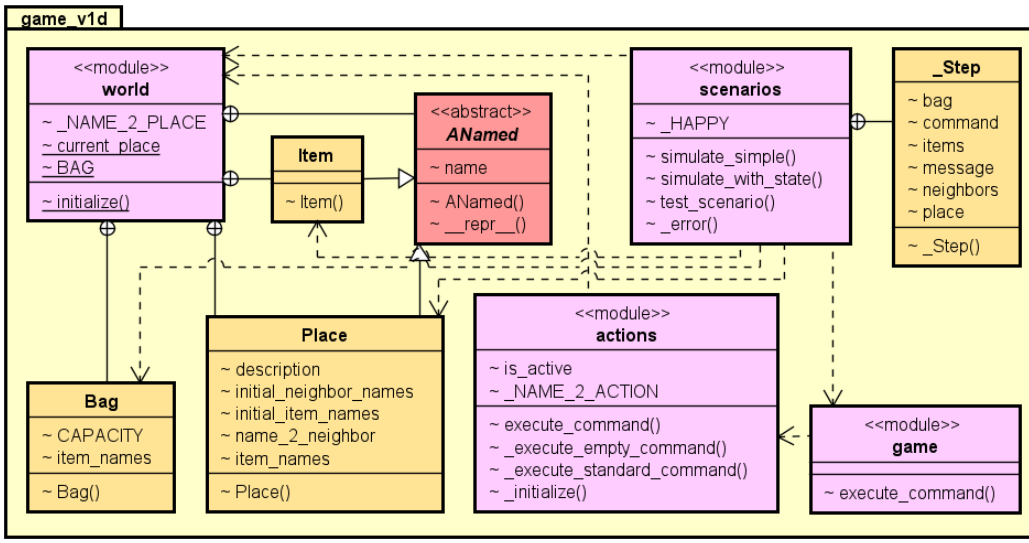
Výpis 14.5: *Konec zprávy o průběhu testu aplikace v balíčku `game_v1d`*

```
1.
------------------------------
Vezmi víno
------------------------------
Tento příkaz neznám: vezmi
============================================================

V 1. kroku neodpovídá: odpověď hry
   Očekáváno: Karkulka dala do košíku objekt: Víno
   Obdrženo:  Tento příkaz neznám: vezmi
Stisk klávesy Enter aplikaci ukončí
>>>
```

Dozvěděli jsme se, že hra nezná příkaz `vezmi`. V příští kapitole ho tedy nadefinujeme.

14.9 Shrnutí

Aktuální diagram tříd balíčku `game_v1d` najdete na obrázku 14.3. Zdrojové kódy odpovídající výše popsanému stavu vývoje aplikace najdete v doprovodných programech v balíčku `game.game_v1d`. Přehled výpisů programů v této kapitole pak v souborech s názvem `m14__create_world`.



Obrázek 14.3:
Diagram tříd modulu `game_v1d`

Kapitola 15
Příkazy Vezmi a Polož

Co se v kapitole naučíte
V této kapitole budeme pokračovat ve vývoji aplikace. Test nás dovede k vývoji příkazů pro práci s h-objekty. Přitom se ukáže jako užitečné zavedení dalšího společného rodiče.

15.1 Balíček game_v1e

Abyste si mohli později snáze připomenout, co se v průběhu kapitoly změnilo, vytvořil jsem opět nový balíček – podle očekávání je to balíček `game.game_v1e`. V něm budou zpočátku kopie modulů z balíčku `game.game_v1d`. Ty pak v průběhu kapitoly upravíme.

15.2 Obecná akce

Ve výpisu 14.5 nám testovací program oznamuje, že hra nezná příkaz `vezmi`, což je podnět k tomu, abychom začali programovat jednotlivé akce. Pojďme si ale nejprve zopakovat, co o akcích doposud víme:

- Každá akce musí mít svůj jedinečný název.
- Akce musejí mít definován stručný popisek, který bude používat nápověda.
- Příkazy sestávají z názvu akce následovaného případnými argumenty.
- Výsledkem akce je změna stavu hry a její návratovou hodnotou je string, který se uživateli vypíše jako odpověď na zadaný příkaz.

To je dostatek důvodů pro to, aby mělo smysl definovat pro ně nějakou obecnou, abstraktní akci jako společného rodiče, který bude mít na starosti to, co je pro všechny akce společné. Nazveme ji `_AAction`. Jak vyplývá z názvu, definujeme ji jako nezveřejňovanou, a stejné to bude i s jejími potomky, protože není žádný důvod pro to, aby o nich věděl

okolní svět. Stačí, když o nich bude vědět správce akcí, kterým je modul `actions` a který má na starosti komunikaci s okolním světem. Možnou definici třídy `_AAction` najdete spolu s přidaným importem ve výpisu 15.1.

Jak komentář naznačuje, její initor uloží název a popis vytvářené akce. Třída současně definuje abstraktní metodu `execute()`, která obdrží zadané argumenty a vrátí odpověď hry. To je metoda, kterou jsme ve výpisu 12.5 na straně 169 použili v definici funkce `_execute_standard_command()`, čímž jsme naplánovali, že ji každá akce musí mít. Tím, že jsme v rodičovské třídě definovali tuto metodu jako abstraktní, jsme zařídili, že při každé definici její dceřiné třídy interpret zkontroluje, zda daná třída tuto metodu definuje.

Výpis 15.1: *Definice abstraktní třídy `_AAction` v modulu `actions` v balíčku `game_v1e`*

```
from abc    import ABCMeta, abstractmethod

class _AAction(world.ANamed, metaclass=ABCMeta):
    """Společná rodičovská třída všech akcí."""

    def __init__(self, name: str, description: str):
        """Zapamatuje si název vytvářené akce a její stručný popis."""
        super().__init__(name)
        self.description = description

    @abstractmethod
    def execute(self, arguments: list[str]) -> str:
        """Realizuje reakci hry na zadání daného příkazu.
        Počet argumentů je závislý na konkrétní akci.
        """
```

15.3 Společný rodič batohu a prostorů

Nyní bychom měli začít přemýšlet nad definicí příkazu `Vezmi`, který má odebrat h-objekt z aktuálního prostoru a přesunout jej do batohu. Jistě si ale vybavíte, že v základní nabídce je i příkaz `Polož`, který naopak odebere h-objekt z batohu a přidá jej do prostoru.

Souhrnně řečeno: oba tyto objekty bychom mohli považovat za kontejnery h-objektů, které lze do nich přidat, anebo je z nich odebrat. Koledují si tak o společného rodiče, jehož bychom mohli nazvat `AItemContainer`, který tyto operace definuje, a naše třídy je od něj již pouze zdědí. Možná definice této třídy je ve výpisu 15.2.

Initor

Při konstrukci initoru si musíme uvědomit, že třída `Place` je již potomkem třídy `ANamed` a že jí chceme zadat dalšího rodiče. Vzpomeneme si proto na pasáž *Návrh třídy s více*

bezprostředními rodiči na straně 178 a definujeme initor podle výše uvedeného doporučení, tj. s dvouhvězdičkovým parametrem a s voláním rodičovského initoru, přestože je třída bezprostředním potomkem třídy `object`.

Výpis 15.2: *Definice abstraktní třídy `AItemContainer` v modulu `world` v balíčku `game_v1e`*

```
class AItemContainer(metaclass=ABCMeta):
    """Společný rodič tříd, jejichž instance mohou obsahovat h-objekty."""

    def __init__(self, initial_item_names: tuple[str], **kwds):
        """Rodič si zapamatuje název vytvářeného kontejneru a uloží názvy
        výchozí sady obsažených objektů v kontejneru pro inicializaci.
        """
        super().__init__(**kwds)
        # Počáteční názvy se ukládají včetně prefixů, aby se při
        # inicializaci daly použít jako argumenty initoru
        self.initial_item_names = tuple(initial_item_names)

    def initialize(self) -> None:
        """Připraví počáteční obsah kontejneru na počátku hry tak,
        že vytvoří a zapamatuje si objekty zadaných počátečních názvů.
        """
        # Následující dva atributy se vytvářejí až při první inicializaci
        self.items      = [Item(name) for name in self.initial_item_names]
        # Názvy se ukládají převedené na malá písmena
        self.item_names = [item.name.lower() for item in self.items]

    def add_item(self, item: Item) -> None:
        """Přidá zadanou položku do seznamu a její název převedený
        na malá písmena uloží do seznamu názvů.
        """
        # Obě se uloží na konec seznamu, takže budou mít shodný index
        self.item_names.append(item.name.lower())
        self.items     .append(item)

    def remove_item(self, item_name: str) -> Item:
        """Vyjme položku se zadaným názvem ze seznamu položek a vrátí ji
        jako svoji funkční hodnotu. Není-li položka v seznamu, vrátí None.
        Při rozpoznávání názvů nesmí záležet na velikosti písmen.
        """
        try:
            # Zapamatované názvy jsou malými písmeny => musím převést zadaný
            index = self.item_names.index(item_name.lower())
        except ValueError:
            return None
        # Název i odpovídající objekt mají v seznamech shodné indexy
        self.item_names.pop(index)
        result = self.items.pop(index)
        return result
```

Všimněte si, že initor pouze uloží názvy objektů do atributu `initial_item_names` a žádné h-objekty nevytváří. Ty se vytvoří až při inicializaci. Je sice pravda, že se tak

vytvářejí při každém startu hry znovu, ale to je stále jednodušší než dohledávat, kam se v průběhu hry zatoulaly, a vracet je do původního kontejneru.

Kdyby to byly objekty, jejichž vytvoření je drahé (např. zabere hodně času), dalo by se o tom uvažovat. Ale u tak levně vytvořitelných objektů, jakými jsou instance třídy `Item`, je efektivnější je vytvořit znovu.

Inicializace

Podívejme se na inicializaci (řádky 12–19). Vytvářejí se při ní dva paralelní seznamy: v prvním budou názvy převedené na malá písmena (proč, to jsme rozebírali v pasáži *Proč na malá písmena* na straně 159), ve druhém pak příslušné objekty.

Dva seznamy vytváříme proto, že v daném kontejneru smí být několik předmětů se stejným názvem, což brání použití slovníku. Když ale budeme do obou seznamů názvy a příslušné h-objekty shodně přidávat a zase je z nich odebírat, bude v nich mít vždy název i odpovídající h-objekt stejný index, přes nějž k nim můžeme přistupovat.

Za pozornost stojí i to, že pro uchování názvů v průběhu hry nepoužíváme n-tici `initial_item_names`, ale vytváříme pro ně seznam `item_names`. Je to proto, že v průběhu hry se obsah tohoto seznamu mění a až hra skončí a budeme ji chtít znovu spustit bez vypnutí aplikace, musíme mít někde uloženy informace potřebné pro korektní nastavení počátečního stavu.

Přidání položky

Metoda `add()` sloužící pro přidání položky do kontejneru je definována na řádcích 21–27. Všimněte si, že jak název (kvůli hledání shody převedený na malá písmena), tak příslušný h-objekt ukládá na konec seznamu, takže pak budou mít oba stejné indexy, jak jsem se zmiňoval před chvílí.

Odebrání položky

Metoda `remove()` slouží k odebrání položky. Nejprve na řádku 36 zjišťuje index zadaného názvu v seznamu názvů. Pokud se v kontejneru položka se zadaným názvem vyskytuje, tak na řádku 40 zadaný název vyhodí ze seznamu názvů a na dalším řádku vyhodí odpovídající položku ze seznamu položek. Vyhazovanou položku si zapamatuje a vrátí jako svoji funkční hodnotu.

Pokud zadaný název v seznamu názvů není, tak metoda `index()` volaná na řádku 36 vyhodí výjimku. V takovém případě metoda `remove()` vrátí hodnotu `None`.

15.4 Nebezpečí degenerovaných objektů

Na první pohled by se mohlo zdát, že paralelní zpracování názvů a odpovídajících objektů je zbytečně složité a že by měly stačit operace se samotnými názvy. Vždyť h-objekt si přece o sobě nic jiného než svůj název nepamatuje. Takovéto zjednodušující snahy bývají nebezpečné. V našem prozatímním řešení si nic jiného nepamatují, ale jenom proto, že prozatím řešíme velice zjednodušené zadání.

Když si zadání přečtete, zjistíte, že některé objekty nemají být přenositelné. Kdo si pak bude pamatovat, který název reprezentuje nepřenositelný objekt? Obecně se ukazuje, že bývá nejlepší, když si všechny informace o objektu pamatuje o sobě objekt sám.

Existují sice speciální problémy, které je výhodnější řešit trochu jinak, ale to nejsou problémy, které by se řešily v začátečnických kurzech. V naprosté většině případů je nejvýhodnější z objektově orientovaného paradigmatu neutíkat.

15.5 Úprava initoru třídy ANamed

Initor třídy `ANamed` prozatím využívá toho, že třída je bezprostředním potomkem třídy `object` a nevolá rodičovský initor. Jak jsme si ale řekli v pasáži *Návrh třídy s více bezprostředními rodiči* na straně 178, stane-li se taková třída společným rodičem, je vhodné volání rodičovského initoru doplnit spolu s přidáním dvojhvězdičkového parametru. Upravenou definici si můžete prohlédnout ve výpisu 15.3.

Výpis 15.3: *Upravené definice initoru třídy `ANamed` v modulu `world` v balíčku `game_v1e`*

```
1 def __init__(self, name, **kwds):
2     """Zapamatuje si jméno dané instance."""
3     super().__init__(**kwds)
4     self.name = name
```

Tato změna se naštěstí nedotkne definic initorů ve třídách, u nichž je třída `ANamed` jejich jediným rodičem. Musíme ji mít na mysli pouze u třídy `Place`, kde je pouze jedním z rodičů, takže v ní musíme předávané argumenty zadávat jako pojmenované – viz řádek 9 ve výpisu 15.4.

15.6 Upravené definice prostorů a batohu

Po definici společného rodiče prostorů a batohu musíme příslušně upravit jejich definice. Jejich možná verze je ve výpisu 15.4.

Oproti předchozí verzi se změnily hlavičky, kde je nyní zadán předek, přičemž třída `Place` má již druhého. Je na to třeba myslet v definici jejího initoru, v němž musíme postupně volat initory obou předků a každému předat ty správné argumenty.

Rodičovský initor musí volat i initor třídy `Bag`. Protože ve hře, která je ve scénáři a podle níž aplikace testujeme, je na počátku batoh prázdný, musí se rodičovskému initoru předat prázdná n-tice. Pamatujte na to, že ji nemůžeme nahradit tím, že nezadáme nic. Pak bychom totiž zadali pouze jeden argument, ale rodičovský initor má dva parametry, takže by se překladač vzbouřil.

A když jsme u úprav třídy batohu, tak si vzpomeneme, že hrajeme hru o Červené Karkulce, které se vejdou do košíku pouze dva předměty, příslušně tedy upravíme kapacitu batohu.

Výpis 15.4: *Upravené definice tříd `Place` a `Bag` v modulu `world` v balíčku `game_v1e`*

```
class Place(ANamed, AItemContainer):
    """Instance reprezentují prostory hry."""

    def __init__(self, name: str, description: str,
                 initial_neighbor_names: tuple[str],
                 initial_item_names:     tuple[str]):
        """Vytvoří prostor se zadaným názvem a stručným popisem, napojený
        zpočátku na zadané sousedy a obsahující zadané h-objekty."""
        super().__init__(name=name, initial_item_names=initial_item_names)
        self.description            = description
        self.initial_neighbor_names = initial_neighbor_names
        self.name_2_neighbor        = dict.fromkeys(initial_neighbor_names)


class Bag(AItemContainer):
    """Instance třídy reprezentuje batoh."""

    CAPACITY = 2    # Maximální kapacita batohu

    def __init__(self):
        """Vytvoří batoh."""
        super().__init__(())
```

15.7 Definice akce `Vezmi`

Společného rodiče akcí i třídy, s jejichž instancemi bude daný prostor pracovat, jsme definovali. Nyní se můžeme pokusit definovat akci `vezmi`, o níž jsme se z poslední zprávy testeru (viz výpis 14.5 na straně 187) dozvěděli, že ji naše aplikace ještě nezná.

Co se má stát při této akci? Má se odebrat h-objekt z aktuálního prostoru a vložit do batohu. Předběžný návrh třídy najdete ve výpisu 15.5. Initor je triviální, takže se soustředíme na metodu `execute()`, která bude do jisté míry vzorem pro tyto metody v následujících třídách akcí.

Metoda se nejprve podívá na svůj argument, který by měl být v první položce obdržené kolekce (v nulté položce je název spouštěné akce). Tímto argumentem je v tomto případě h-objekt nacházející se v aktuálním prostoru.

Metoda požádá na řádku 13 o jeho odebrání z aktuálního prostoru, načež se zeptá, jestli tam h-objekt se zadaným názvem vůbec byl. Vrátila-li metoda `remove_item()` hodnotu `None`, bude podmínka nepravdivá a metoda vrátí oznámení, že zadaný předmět v prostoru není. Pokud však h-objekt v prostoru byl, nechá jej vložit do košíku a vrátí zprávu s příslušným oznámením.

Výpis 15.5: *Definice třídy `_Take` v modulu `actions` v balíčku `game_v1e`*

```
1  class _Take(_AAction):
2      """Přesune h-objekt z aktuálního prostoru do košíku.
3      """
4      def __init__(self):
5          super().__init__("Vezmi",
6                  "Přesune zadaný předmět z aktuálního prostoru do košíku.")
7
8      def execute(self, arguments: list[str]) -> str:
9          """Ověří existenci zadaného h-objektu v aktuálním prostoru
10         a je-li tam, přesune jej do košíku.
11         """
12         item_name = arguments[1]
13         item      = world.current_place.remove_item(item_name)
14         if not item:
15             return 'Zadaný předmět v prostoru není: ' + item_name
16         world.BAG.add_item(item)
17         return 'Karkulka dala do košíku objekt: ' + item.name
```

15.8 Definice akce Polož

Když už jsme nadefinovali třídu `_Take`, jejíž instance realizuje akci `Vezmi`, bylo by vhodné nadefinovat hned i třídu pro akci `Polož`, protože bude prakticky stejná, pouze se v ní prohodí dodavatel a odběratel. Nazveme ji `_Put` a její možnou definici si můžete prohlédnout ve výpisu 15.6.

15.9 Spuštění testu

Když se nyní pokusíme spustit test, opět se dozvíme, že příkaz `Vezmi` program nezná. Samozřejmě, protože jsme jej ještě nezadali do slovníku `_NAME_2_ACTION`. Musíme upravit definici tohoto atributu do tvaru:

```
_NAME_2_ACTION = {'polož': _Put(),  'vezmi': Take(), }
```

Když nyní spustíme test, skončí vyhozením výjimky, která na posledních třech řádcích chybové zprávy tvrdí:

```
  File "Q:\65_PGM\65_PYT\game\game_v1e\scenarios.py", line 185, in test_scenario
    if compare_containers(step.items, current_place.item_names):
AttributeError: 'Place' object has no attribute 'item_names'
```

Výpis 15.6: *Definice třídy `_Put` v modulu `actions` v balíčku `game_v1e`*

```
1  class _Put(_AAction):
2      """Přesune h-objekt z košíku do aktuálního prostoru.
3      """
4      def __init__(self):
5          super().__init__('Polož',
6                  'Přesune zadaný předmět z košíku do aktuálního prostoru.')
7
8      def execute(self, arguments: list[str]) -> str:
9          """Ověří existenci zadaného h-objektu v košíku a je-li tam,
10         vyjme jej z košíku a přesune do aktuálního prostoru.
11         """
12         item_name = arguments[1]
13         item      = world.BAG.remove_item(item_name)
14         if item:
15             world.current_place.add_item(item)
16             return 'Karkulka vyndala z košíku objekt: ' + item.name
17         else:
18             return 'Zadaný předmět v košíku není: ' + item_name
```

Z ní se dozvídáme, že instance třídy `Place` nemají atribut nazvaný `item_names`. Nemají, protože jej initor nevytvořil – ten vytváří pouze atribut `initial_item_names`. Atribut `item_names` se vytváří až v průběhu první inicializace (viz řádek 19 ve výpisu 15.2 na straně 191), ale ta ještě neproběhla.

Musíme doplnit inicializaci jednotlivých objektů světa hry. V modulu `world` proto upravíme funkci `initialize()` do tvaru ve výpisu 15.7.

Nyní už test úspěšně provede první tři kroky (nultý až druhý) a zastaví se na třetím kroku se zprávou, jejíž konec najdete ve výpisu 15.8.

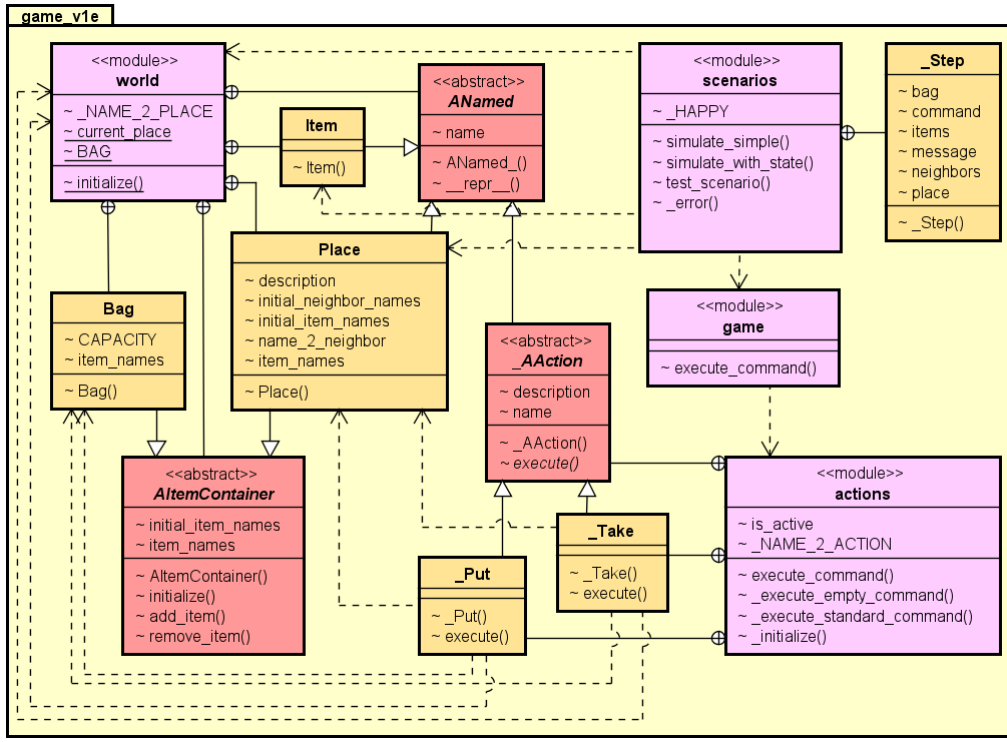
Výpis 15.7: *Upravená definice metody `initialize()` v modulu `world` v balíčku `game_v1e`*

```
1  def initialize() -> None:
2      """Inicializuje batoh a prostory."""
3      for p in _NAME_2_PLACE.values():
4          p.initialize()
5      BAG.initialize()
```

Výpis 15.8: *Konec zprávy o průběhu testu aplikace v balíčku `game_v1e`*

```
1  3.
2  ------------------------------
3  Jdi LES
4  ------------------------------
5  Tento příkaz neznám: jdi
6  ============================================================
7
8  V 3. kroku neodpovídá: odpověď hry
9     Očekáváno: Karkulka se přesunula do prostoru:
10 Les s jahodami, malinami a pramenem vody
11    Obdrženo:  Tento příkaz neznám: jdi
12 Stisk klávesy Enter aplikaci ukončí
13 >>>
```

15.10 Shrnutí



Obrázek 15.1:
Diagram tříd modulu `game_v1e`

Aktuální diagram tříd balíčku `game_v1e` najdete na obrázku 15.1. Zdrojové kódy odpovídající výše popsanému stavu vývoje aplikace najdete v doprovodných programech v balíčku `game.game_v1e`. Přehled výpisů programů v této kapitole pak v souborech s názvem `m15__take_put_actrions`.

Kapitola 16
Rozběhnutí aplikace

Co se v kapitole naučíte
V této kapitole naši aplikaci doplníme o poslední dvě akce. Přitom objevím nepřesnosti v testech, takže příslušně upravíme nejen testy, ale i naši aplikaci. Na konci se tak aplikace konečně rozběhne.

16.1 Balíček `game_v1f`

Jak jste už jistě očekávali, pro usnadnění rekapitulace provedených změn začnu novou kapitolu opět s novým balíčkem – podle očekávání balíčkem `game.game_v1f`. Zpočátku v něm budou kopie modulů z balíčku `game.game_v1e`, které budeme v průběhu kapitoly postupně upravovat.

16.2 Definice třídy `_GoTo`

Ve výpisu 15.8 nám testovací program oznamuje, že aplikace ještě nezná příkaz `Jdi`. Možnou definici třídy nazvané `_GoTo`, jejíž instance bude tento problém řešit, najdete ve výpisu 16.1.

Jak jsme si říkali v podkapitole *15.9 Spuštění testu* na straně 195, definice třídy daných akcí ještě nestačí. Aby akce byla použitelná, musíme ji ještě spolu s jejím názvem přidat do slovníku `_NAME_2_ACTION`.

Když nyní spustíme test, tak se zase o něco posuneme dopředu, protože se právě vytvořená akce ve správnou chvíli opravdu spustí. Spustí se, ale jak ukazuje výpis 16.2. s koncem záznamu o testu, zhavaruje na tom, že les nebyl nalezen mezi sousedy domečku.

Výpis 16.1: *Definice třídy `_GoTo` v modulu `actions` v balíčku `game_v1f`*

```
class _GoTo(_AAction):
    """Přesune Karkulku do zadaného sousedního prostoru.
    """
    def __init__(self):
        super().__init__("jdi",
                         "Přesune Karkulku do zadaného sousedního prostoru.")

    def execute(self, arguments: List[str]) -> str:
        """Ověří, že zadaný prostor patří mezi sousedy aktuálního
        prostoru, Karkulku do něj přemístí a vrátí příslušnou zprávu.
        Není-li cílový prostor sousedem, vrátí příslušné oznámení.
        """
        destination_name = arguments[1]
        try:
            destination = world.current_place \
                               .name_2_neighbor[destination_name.lower()]
        except KeyError:
            return ('Do zadaného prostoru se odsud nedá přejít: '
                    + destination_name)
        world.current_place = destination
        return ('Karkulka se přesunula do prostoru: ' +
                destination.description)
```

Výpis 16.2: *Závěr testu*

```
3.
------------------------------
Jdi LES
------------------------------
Do zadaného prostoru se odsud nedá přejít: les
============================================================

V 3. kroku neodpovídá: odpověď hry
   Očekáváno: Karkulka se přesunula do prostoru:
Les s jahodami, malinami a pramenem vody
   Obdrženo:  Do zadaného prostoru se odsud nedá přejít: les
Stisk klávesy Enter aplikaci ukončí
>>>
```

16.3 Upravujeme zprávu o chybě

Někteří z vás již možná tuší, kde je chyba, ale jistě se shodneme na tom, že by se mnohem lépe dohledávala, kdyby byla zpráva o chybě podrobnější. Možná si vzpomenete, že jsme po definici šťastného scénáře definovali dvě simulační metody, přičemž ta druhá (viz výpis 9.5 na straně 137) zobrazovala podrobné informace o očekávaném stavu hry po provedení daného kroku.

Definice funkce current_state() v modulu game

Vedle popisu stavu požadovaného scénářem bychom potřebovali popsat i aktuální stav hry. Můžeme sice tento popis naprogramovat ve funkci _error(), ale to by byla škoda, protože už by se k němu nikdo jiný nedostal. Lepší by bylo, kdybychom v modulu game definovali funkci, která na požádání vrátí zprávu o aktuálním stavu rozehrané hry. Funkci jsem nazval current_state a její definici si můžete prohlédnout ve výpisu 16.3.

Výpis 16.3: *Definice funkce current_state() v modulu game v balíčku game_v1f*

```
def current_state() -> str:
    """Vrátí string s popisem aktuálního stavu, tj. s názvy
    aktuálního prostoru, jeho sousedů a h-objektů a obsahu batohu.
    """
    cp = world.current_place
    result = f'Aktuální prostor: {cp}\n' \
             f'Sousedé prostoru: {list(cp.name_2_neighbor.values())}\n' \
             f'Předměty v prostoru: {cp.items}\n' \
             f'Předměty v batohu:   {world.BAG.items}\n'
    return result
```

Všimněte si, že hra na rozdíl od kroku scénáře nepovažuje za součást stavu svoji poslední odpověď. Pokud bychom chtěli začlenit odpověď a případně i poslední příkaz do aktuálního stavu, museli bychom pro ně definovat nějaké atributy, v nichž bychom si je pamatovali. Máte-li zájem, můžete definovat metodu vracející rozšířenou informaci, která bude tyto dva texty obsahovat.

Nová definice funkce _error() v modulu scenarios

Změnili jsme zdroje informací, musíme změnit i definici funkce oznamující chybu. Nová verze funkce _error() je ve výpisu 16.4. Jak vidíte, oproti své předchozí definici (viz např. výpis 11.5 na straně 158) se příliš nezměnila, avšak poskytuje výrazně podrobnější informace.

Výpis 16.4: *Definice funkce _error() v modulu scenarios v balíčku game_v1f*

```
def _error(step: _Step, reason: str, answer: str) -> None:
    """Ohlásí zprávu o chybě a ukončí aplikaci."""
    msg = f'V {step.index}. kroku neodpovídá: {reason}\n' \
          f'Očekáváno:\n{step}' \
          f'Obdrženo:\n{answer}\n{_single_line}\n' \
          f'{game.current_state()}'
    print( msg )
    input('Stisk klávesy Enter aplikaci ukončí')
    exit(1)
```

Úprava testovací funkce

Metoda se zdánlivě moc nezměnila, ale ve skutečnosti se změnila dost podstatně, protože má jiné parametry, a tím pádem i jiné rozhraní, kterému se všichni, kdo ji používají, musejí přizpůsobit. Nezbývá nám tedy než upravit také testovací funkci. Změněnou část, kterou vidíte ve výpisu 16.5, můžete porovnat s původní definicí ve výpisu *11.4* na straně 157.

Jak vidíte, funkci jsem netiskl celou (její definice má hříšných 45 řádek), ale zobrazuji zde pouze část, která se změnila. Jak se můžete přesvědčit, změnily se pouze argumenty zadávané funkci `_error()`.

Výpis 16.5: *Upravená definice části funkce `test_scenario()` v modulu `scenarios` v balíčku `game_v1f`*

```
if step.message != answer:
    _error(step, 'odpověď hry', answer)
current_place = world.current_place
if  step.place != current_place.name:
    _error(step, 'aktuální prostor', answer)
if compare_containers(step.neighbors,
                      current_place.name_2_neighbor.keys()):
    _error(step, 'aktuální sousedé', answer)
if compare_containers(step.items, current_place.item_names):
    _error(step, 'objekty v prostoru', answer)
if compare_containers(step.bag, world.BAG.item_names):
    _error(step, 'objekty v batohu', answer)
```

Úprava ošetření vyhozené výjimky

Při úpravě testovací funkce bychom mohli upravit i její reakci na vyhození výjimky při vykonávání zadaného příkazu. Když totiž hra vyhodí výjimku, tak vám testovací funkce neprozradí o aktuálním stavu skoro nic. Jenom vám oznámí, že program vyhodil výjimku.

O stavu hry se v takové situaci nemáme šanci cokoliv dozvědět, ale mohli bychom alespoň prozradit očekávaný stav, z nějž by bylo možné leccos odvodit.

16.4 Nový test

Spustíme-li po všech těchto úpravách znovu test, obdržíme zprávu, jejíž konec je ve výpisu 16.6. Zpráva naznačuje, že domeček nemá žádné sousedy, přesněji že sousedem je objekt `None`.

Výpis 16.6: *Konec zprávy o průběhu testu*

```
3.
------------------------------
Jdi LES
------------------------------
Do zadaného prostoru se odsud nedá přejít: les
============================================================

V 3. kroku neodpovídá: odpověď hry
Očekáváno:
3.
Jdi LES
------------------------------
Karkulka se přesunula do prostoru:
Les s jahodami, malinami a pramenem vody
------------------------------
Aktuální prostor: Les
Sousedé prostoru: ('Domeček', 'Temný_les')
Předměty v prostoru: ('Maliny', 'Jahody', 'Studánka')
Předměty v batohu:   ('Bábovka', 'Víno')
============================================================
Obdrženo:
Do zadaného prostoru se odsud nedá přejít: les
------------------------------
Aktuální prostor: Domeček
Sousedé prostoru: [None]
Předměty v prostoru: [Stůl, Panenka]
Předměty v batohu:   [Víno, Bábovka]

Stisk klávesy Enter aplikaci ukončí
```

16.5 Inicializace sousedů

Prozatím jsme při inicializaci prostorů používali metodu zděděnou od třídy `AItemContainer`, která ale inicializovala pouze h-objekty v daném prostoru. Musíme ji proto ve třídě `Place` přebít. Možnou podobu této definice najdete ve výpisu 16.7.

Všimněte si, že definice začíná voláním rodičovské inicializační metody inicializující h-objekty, za níž přidává inicializaci sousedů.

Navíc by bylo vhodné odstranit falešnou inicializaci atributu `name_2_neighbor` z initoru. Atribut se tak vytvoří až při první inicializaci a my budeme mít jistotu, že jej nikdo nepoužil před tím, než byl inicializován.

Když po této opravě spustíme test, zjistíme, že byly úspěšně provedeny všechny kroky s výjimkou posledního, u nějž hra oznámila, že příkaz `Konec` nezná.

Výpis 16.7: *Definice metody `initialize()` ve třídě `Place` v modulu `world` v balíčku `game_v1f`*

```
def initialize(self) -> None:
    """Inicializuje prostor na počátku hry.
    """
    AItemContainer.initialize(self)     # Inicializuje h-objekty
    # Inicializuje sousedy, atribut se vytváří až při první inicializaci
    self.name_2_neighbor = {name.lower() : _NAME_2_PLACE[name.lower()]
                            for name in self.initial_neighbor_names}
```

16.6 Akce Konec

Zbývá tedy definovat akci `Konec`, která se s uživatelem rozloučí a převede hru do neaktivního stavu, z nějž ji bude možno opět spustit. Předpokládám, že už jste tak zkušení, že byste tuto třídu dokázali bez problému definovat sami. Pro jistotu ale uvádím její definici ve výpisu 16.8.

Výpis 16.8: *Definice třídy `_End` v modulu `actions` v balíčku `game_v1f`*

```
class _End(_AAction):
    """Ukončuje hru a převede ji do pasivního stavu.
    """
    def __init__(self):
        """Jen si u rodiče zapamatuje svůj název a popis."""
        super().__init__("Konec", "Ukončení hry")

    def execute(self, arguments: str) -> str:
        is_active = False
        return 'Ukončili jste hru.\nDěkujeme, že jste si zahráli.'
```

Abyste tuto akci zprovoznili, nesmíte ji zapomenout přidat k ostatním akcím do slovníku `_NAME_2_ACTION`.

Když nyní spustíte test, celá hra proběhne podle zadaného scénáře a testovací funkce se s vámi rozloučí větou:

```
Hra úspěšně otestována podle šťastného scénáře
```

16.7 Shrnutí

Zdrojové kódy odpovídající výše popsanému stavu vývoje aplikace najdete v doprovodných programech v balíčku `game.game_v1f`. Přehled výpisů programů v této kapitole je v souborech s názvem `m16_application_start`. Architekturu jsme doplnili jen nepatrně, a proto diagram tříd neuvádím.

Kapitola 17
Co nám ještě chybí

Co se v kapitole naučíte
V této kapitole znovu projdeme zadání, abychom zjistili, co vše ještě není naprogramováno. Definujeme další test, který bude ověřovat korektnost reakce aplikace na chyby v uživatelových zadáních.

17.1 Nesplněné body zadání

Aplikaci jsme rozchodili, i když prozatím pouze z hlediska testovacího programu. Nyní bychom se měli podívat na zadání (najdete je v podkapitole 7.2 Zadání na straně 103) a zjistit, které jeho body ještě nejsou splněny. Při prohlídce najdeme následující nesplněné body:

- *Množství h-objektů, které se do batohu vejdou, je omezené.*
  Batoh má sice definovanou maximální kapacitu, ale nikde se nekontroluje, zda hráč nepřidává h-objekt, který se již do batohu nevejde.
- *Některé z h-objektů je možné přenášet, jiné ne.*
  Hra prozatím nijak nereaguje na možnost, že by hráč chtěl přenést nepřenositelný h-objekt.
- *Hra musí umožňovat předčasné ukončení a opakované spuštění.*
  Myslíme si, že jsme ji tak naprogramovali, ale nikde jsme nezkontrolovali, že to opravdu funguje.
- *Musí být schopna poskytnout nápovědu.*
  Neexistuje akce pro získání nápovědy s přehledem příkazů a jejich významu.
- *Hra musí umět korektně reagovat na chybně zadané příkazy uživatele.*
  Nepřemýšleli jsme nad tím, kde všude by uživatel při zadávání příkazů mohl udělat chybu.

Nový balíček game_v1g

Jak je vidět, zbývá nám toho ještě dost. Proto opět vytvořím nový balíček – tentokrát balíček `game.game_v1g`, do nějž zkopíruju obsah balíčku `game.game_v1f` a příslušně upravím soubor `__init__.py` s initorem balíčku.

Nový scénář

Připomenu vám pasáž *Programování řízené testy* na straně 129, v níž jsem vás seznámil s metodikou TDD, která doporučuje, abychom vždy nejprve napsali testy, a teprve pak začali programovat aplikaci. Takto jsme doposud naši aplikaci navrhovali a doufám, že sami cítíte, že jsme si tím ušetřili mnohé bloudění. Prostě jsme vždy jen spustili test a on nám řekl, kde máme v návrhu aplikace pokračovat.

Měli bychom proto začít tím, že bychom vedle šťastného scénáře, v němž se všechno dařilo, definovali ještě chybový scénář, v němž bychom postupně předvedli všechna chybná zadání, která by uživatel mohl zadat, doplněná o žádost o nápovědu, jež ve šťastném scénáři nebyla potřeba.

17.2 Třída Scenario

Potřebujeme další scénář, možná několik scénářů. Scénáře se nám začínají množit. Měli bychom si proto vzpomenout na podkapitolu *7.4 Správci skupin objektů* na straně 108 a definovat pro naše scénáře správce. Pro jejich rozlišení by však bylo vhodné, kdyby měl každý scénář své jméno, které není závislé na názvu proměnné, do níž je odkaz na něj uložen.

V modulu `scenarios` proto definujeme třídu `Scenario`, jejímiž instancemi budou jednotlivé scénáře. Ty si budou pamatovat svůj název, stručný popis a testující posloupnost kroků. Začátek její definice by tak mohl odpovídat výpisu 17.1.

Výpis 17.1: *Začátek definice třídy `Scenario` v modulu `scenarios` v balíčku `game_v1g`*

```
1  class Scenario(world.ANamed):
2      """Instance reprezentují jednotlivé scénáře hry.
3      Každý scénář má svůj název, stručný popis toho, co se jím testuje,
4      a n-tici kroků, jejichž příkazy se postupně zadávají hře.
5      """
6      def __init__(self, name: str, description: str, steps: Tuple[_Step]):
7          super().__init__(name)
8          self.description = description
9          self.steps = steps
10
11     def __repr__(self) -> str:
12         return f'{self.name} scenario\n{self.description}\n'
```

Ve chvíli, kdy máme definovanou třídu scénářů, by bylo vhodné, kdyby simulační a testovací funkce přestaly být řadovými funkcemi modulu, ale staly se metodami scénářů. Funkcím by přibyl parametr self a v jejich těle bychom (koho) proměnnou `_HAPPY_SCENARIO` nahradili (kým) proměnnou `self`. Třída `Scenario` by tak měla definovány metody:

- `__init__( self, name: str, description: str, steps: Tuple[_Step])`,
- `__repr__(self)`,
- `simulate_simple(self)`,
- `simulate_with_state(self)`,
- `test(self)`.

Funkci `_error()` bychom nechali dále definovanou samostatně, protože k vykonání zadané činnosti žádnou instanci ani třídu nepotřebuje.

17.3 Chybový scénář

Třídu scénářů máme definovanou, takže se můžeme pustit do definice chybového scénáře.

Společný startovní krok

Předpokládám, že je vám jasné, že každý scénář bude začínat stejným startovním krokem. Máme-li proto dodržet zásadu DRY (viz *DRY – bez kopií* na straně 93), měli bychom definovat startovní krok separátně (dejme tomu jako atribut `_START_STEP`) a v každém scénáři použít tento atribut jako startovní krok aplikace. Tím současně zabezpečíme to, že se jednotlivé startovní kroky nebudou lišit.

Co vše se má zkontrolovat

Napíšeme si seznam věcí, které se mají při testu podle testovacího scénáře zkontrolovat. Bylo by přitom vhodné uspořádat na závěr jejich pořadí tak, abychom při postupném zavádění opravného kódu nemuseli zbytečně přebíhat mezi různými místy v kódu. Postupně bychom tedy měli zkontrolovat:

1. Správnou inicializaci, tj. možnost opakovaného spuštění.
2. Správnou reakci na pokus o chybné spuštění.
3. Odmítnutí prázdného příkazu po spuštění hry.
4. Žádost o zvednutí předmětu bez zadání argumentu.
5. Žádost o zvednutí nepřítomného předmětu.

6. Žádost o zvednutí nezvednutelného objektu.
7. Žádost o přidání objektu do plného batohu.
8. Žádost o položení předmětu bez zadání argumentu.
9. Žádost o položení předmětu, jenž není v batohu.
10. Žádost o přejití do dalšího prostoru bez zadání argumentu.
11. Žádost o přejití do nesousedního prostoru.
12. Žádost o nápovědu.

Začátek definice chybového scénáře se zadanými prvními třemi kroky by mohl vypadat např. tak, jak naznačuje výpis 17.2. Všimněte si, že před vlastní definicí se reinicializuje atribut `last_index`. Scénář tak bude mít na počátku dva kroky s indexem 0, protože startovní krok je již definován (a má index 0) a zadává se pouze odkazem, takže inkrementovaný index bude mít až následující krok.

Předpokládám, že si z výpisu 17.2 dokážete sami odvodit, jak by se měla upravit definice šťastného scénáře. Kdo tápe, může se podívat do definice modulu `scenarios` v balíčku `game_v1g`.

Výpis 17.2: *Začátek definice chybového scénáře v modulu* `scenarios` *v balíčku* `game_v1g`

```
_Step.last_index = -1
_MISTAKE = Scenario('MISTAKE',
    'Základní chybový scénář demonstrující průběh hry, v němž hráč\n'
    'nejprve zkusí zadat neprázdný příkaz, pak korektně hru spustí,'
    'postupně zadá všechny chybné verze příkazů, požádá o nápovědu'
    'a nakonec hru předčasně ukončí.',
    steps = (
        _Step('START',
              'Prvním příkazem není startovací příkaz.\n' +
              'Hru, která neběží, lze spustit pouze startovacím příkazem.',
              'Prostor',
              ('Sousedé', ),
              ('H-objekty', ),
              ('Batoh', ),
              ),
        _START_STEP
        ,
        _Step('',
              'Prázdný příkaz lze použít pouze pro start hry',
              'Domeček',
              ('Les', ),
              ('Bábovka', 'Víno', 'Stůl', 'Panenka', ),
              (),
              ),
```

Nekorektní spuštění

Možná, že někoho zarazí nesmyslné hodnoty aktuálního prostoru, jeho sousedů a h-objektů a obsahu batohu. Musíte si ale uvědomit, že zde testujeme reakci na zadání neprázdného příkazu hře, která neběží, a proto její stav není definován a vůbec by proto neměl být kontrolován. Mohl jsem tedy zadat prázdný řetězec a prázdné n-tice, ale pak by hrozilo nebezpečí, že bych se přehlédl a zadal např. špatný počet n-tic. Takto mi to připadalo přehlednější.

17.4 Dodatečné definice testů

Scénáře se nám „namnožily“, takže bychom měli uvažovat o kontejneru, v němž bychom je všechny uchovávali. Jako optimální mi připadá definovat slovník, který bychom nazvali např. `_SCENARIO`. Definoval jsem jej jako nezveřejňovaný, aby někdo omylem nezměnil jeho obsah.

Výpis 17.3: *Definice dalších funkcí v modulu* `scenarios` *v balíčku* `game_v1g`

```
# Slovník s definovanými scénáři klíčovanými svým názvem
_NAME_2_SCENARIO = {_HAPPY.name: _HAPPY,
                    _MISTAKE.name: _MISTAKE,
                   }

def get() -> Tuple[Scenario]:
    """Vrátí kolekci definovaných scénářů."""
    return tuple(_SCENARIO.values())

def show() -> None:
    """Vytiskne indexy, názvy a stručné popisy doposud definovaných scénářů.
    """
    ss = get()
    for i in range(len(ss)):
        print(f'{i}. {ss[i]}')

def test_named(*names: Tuple[str]) -> None:
    """Postupně spustí testy podle scénářů se zadanými názvy."""
    for name in names:
        _SCENARIO[name].test()

def test_indexed(*indexes: Tuple[int]) -> None:
    """Postupně spustí testy podle scénářů se zadanými indexy."""
    ss = get()
    for index in indexes:
        ss[index].test()

################################################################################
print('\nDefinované scénáře:\n');  show()
print(f'===== Modul {__name__} ===== STOP')
```

Navíc jsem definoval ještě několik funkcí, které umožní operativně získat přehled o definovaných scénářích a především spustit posloupnost testů podle zadaných scénářů. Přehled jejich definic spolu s definicí zmíněného slovníku najdete ve výpisu 17.3.

Všimněte si, že v definicích testů jsou použity hvězdičkové parametry, takže můžeme zadat libovolný počet argumentů a funkce obdrží jediný argument, kterým bude n-tice zadaných argumentů.

Na konci modulu je místo dříve spouštěného testu výpis definovaných scénářů, aby se vývojář mohl sám rozhodnout, podle kterých bude testovat. Přestože uznávám, že zadávání názvů je pro mnohé přehlednější, tak musím přiznat, že preferuji stručnost a při tak malém počtu objektů mi připadá stejně přehledné zadávat scénáře podle jejich indexu v n-tici. Proto je k dispozici jak funkce umožňující zadat scénáře jejich názvy, tak funkce pro jejich zadání prostřednictvím jejich indexů.

17.5 Opakované spuštění

Seznam v pasáži *Co vše se má zkontrolovat* na straně 207 začíná požadavkem na prověření toho, že hra je schopna opakovaného spuštění. To bychom mohli prověřit např. tak, že necháme dvakrát za sebou otestovat hru podle šťastného scénáře, který již máme vyzkoušený.

Výpis 17.4: *Import modulu* `scenarios` *v balíčku* `game_v1g` *a zadání trojitého testu*

```
================================ RESTART: Shell ================================
>>> from game.game_v1g import scenarios
##### game.game_v1g - Balíček s verzí hry na konci 17. kapitoly
      po splnění dalších požadavků zadání
===== Modul game.game_v1g.scenarios ===== START
===== Modul game.game_v1g.world ===== START
===== Modul game.game_v1g.world ===== STOP
===== Modul game.game_v1g.actions ===== START
===== Modul game.game_v1g.actions ===== STOP
===== Modul game.game_v1g.game ===== START
===== Modul game.game_v1g.game ===== STOP

Definované scénáře:

0. HAPPY scenario
Základní úspěšný scénář demonstrující průběh hry, v němž hráč
nezadává chybné příkazy a směřuje chytře k zadanému cíli.

1. MISTAKE scenario
Základní chybový scénář demonstrující průběh hry, v němž hráč
nejprve zkusí zadat neprázdný příkaz, pak hru korektně spustí,
postupně zadá všechny chybné verze příkazů, požádá o nápovědu,
a nakonec hru předčasně ukončí.

===== Modul game.game_v1g.scenarios ===== STOP
>>> scenarios.test_indexed(0, 0, 1)
```

Začátek seance najdete ve výpisu 17.4. Na prvním řádku je zadán import, na posledním řádku je zavolána testovací metoda. Následující výpis průběhu testu jsem již vynechal.

Opakované spuštění

Reakci na zadaný test neuvádím, ale prozradím, že skončí zprávou

```
Prázdný příkaz lze použít pouze pro start hry
```

kterou hra dává pouze v případě, že proměnná `is_active` má hodnotu `True`. Když se ale podíváme na definici třídy `_End` definující reakci na závěrečný příkaz (viz výpis 16.8 na straně 203), vidíme, že proměnná `is_active` je nastavena ne `False`.

Jenomže to je právě omyl. Připomenu vám podkapitolu *10.9 Použití nelokálních proměnných* na straně 150, kde jsem vysvětloval, že první přiřazení definuje v *Pythonu* novou proměnou v aktuálním jmenném prostoru. Potřebujeme-li ve funkci přiřazovat hodnotu jiné proměnné než té lokální, musíme použít příkaz `global` nebo `nonlocal`. A to jsme zde neudělali, takže jsme nepřiřadili hodnotu globální proměnné, ale vytvořili jsme lokální proměnnou, kterou nikdo nepotřeboval. Vložení příkazu

```
global is_active
```

na počátek těla metody `execute()` tuto chybu napraví.

Oprava inicializace

Restartem systému a opětným spuštěním předchozího testu se dozvíme, že (obdrženou zprávu jsem zestručnil a trochu upravil):

```
V 0. kroku neodpovídá: aktuální prostor
Očekáváno: Domeček
Obdrženo:  Chaloupka
```

Jinými slovy: aktuální prostor v proměnné `current_place` se neinicializoval a v proměnné zůstal odkaz na poslední prostor z minulého běhu hry. Přesuneme proto inicializaci aktuálního prostoru do metody `initialize()` v modulu `world`. Nesmíme ale zapomenout zadat příkaz

```
global current_place
```

protože jinak bychom, jak už víme, inicializovali lokální proměnnou této funkce.

Po dalším restartu a testu již proběhnou oba šťastné scénáře a program se zasekne na úvodním kroku chybového scénáře testujícího nekorektní start hry zadáním neprázdného příkazu.

17.6 Příkazy break a continue

Než se pustíme dál, musím vás seznámit s příkazy `break` a `continue`, které umožňují standardní běh cyklu modifikovat.

Příkaz break

Příkaz `break` použijete v okamžiku, kdy potřebujete uprostřed těla cyklu náhle cyklus opustit a pokračovat příkazem za tělem cyklu.

Ve výpisu 17.5 je definována funkce `demo_break()`, která prochází kroky šťastného scénáře a hledá krok, v němž by měl aktuální prostor více jak dva sousedy. Jakmile jej najde, vyskočí z cyklu a pokračuje prováděním příkazů za cyklem.

Zkuste si vyzkoušet, jak se bude funkce chovat, když změníte podmínku na řádku 5 a budete hledat kroky s jiným počtem sousedů.

Výpis 17.5: *Použití příkazu `break`*

```
>>> \
def demo_break() -> None:
    """Najde krok, v němž má aktuální prostor více jak dva sousedy."""
    for step in scenarios._HAPPY.steps:
        if len(step.neighbors) > 2:
            break
    print(f'{step.index}. {step.command}\n{step.neighbors}')

>>> demo_break()
4. Jdi temný_les
('Les', 'Jeskyně', 'Chaloupka')
>>>
```

Příkaz continue

Příkaz `continue` použijete naopak v okamžiku, kdy zjistíte, že už není potřeba provádět zbylé příkazy těla cyklu, a chcete pokračovat dalším během cyklu.

Ve výpisu 17.6 je definována funkce `demo_continue()`, která prochází kroky chybového scénáře a hledá kroky, jejichž příkaz není prázdný, ale nemá žádné argumenty. Prochází krok za krokem, když krok nevyhovuje, přejde na další krok cyklu. Když krok vyhovuje, vytiskne jej a pokračuje v cyklu.

Zkuste si vyzkoušet, jak se bude funkce chovat, když na řádku 4 změníte procházený scénář nebo když budete hledat kroky s jiným počtem argumentů.

Výpis 17.6: *Použití příkazu* `continue`

```
>>> \
def demo_continue() -> None:
    """Vypíše indexy a příkazy kroků bez argumentů."""
    for step in scenarios._MISTAKE.steps:
        if (len(step.command.split()) != 1):
            continue
        print(f'{step.index}. {step.command}')

>>> demo_continue()
0. START
2. Vezmi
8. POLOŽ
10.   Jdi
12. ?
13. KONEC
>>>
```

17.7 Nekorektní spuštění

Vrátíme se ale k naší rozdělané aplikaci. Jak jsem řekl, poslední test se zasekl na úvodním kroku chybového scénáře testujícího nekorektní start hry zadáním neprázdného příkazu. Test tvrdí, že neodpovídá aktuální prostor, ale my jsme si v pasáži *Nekorektní spuštění* na straně 209 řekli, že u hry, která neběží, nemá smysl kontrolovat stav.

Musíme proto upravit definici metody `test()`, aby v případě, že hra není aktivní, zkontrolovala pouze odpověď na zadaný příkaz. Před zjišťování aktuálního prostoru (ve výpisu 11.4 na straně 157 je na řádku 45) proto vložíme příkaz

```
if not actions.is_active:
    continue
```

po němž se rovnou spustí testování dalšího kroku.

Test ukončení hry

Tady bychom si ale mohli uvědomit, že by se po provedení celého scénáře měla hra vrátit zpět do neaktivního stavu. Pokud by tak nestalo (a je jedno, jestli by to bylo kvůli chybě ve scénáři nebo ve hře), nemohl by se pak korektně spustit další scénář. Proto za cyklus, ale před závěrečný tisk zprávy o úspěšném testu podle zadaného scénáře (ve výpisu 11.4 na straně 157 je na řádku 34) vložíme příkaz

```
if actions.is_active:
    _error(step, 'stav hry po testu',
           'Po ukončeném testu má být hra neaktivní')
```

Tímto příkazem vkládáme hře „do úst“ falešnou odpověď, ale na druhou stranu takovou, aby vývojář, který si ji přečte, věděl, kde má hledat chybu.

17.8 Nezadané argumenty

Po opravě opět standardně restartujeme interpret, importujeme testovací balíček a spustíme test. Konec zprávy o průběhu testu je ve výpisu 17.7. Za tímto textem následuje chybové hlášení, ale to již není zobrazeno, protože je nepotřebujeme.

Výpis 17.7: *Konec zprávy o průběhu testu po opravě reakce na příkaz ukončení v balíčku* `game_v1g`

```
 1 2.
 2 ------------------------------
 3 Vezmi
 4 ------------------------------
 5 Při vykonávání příkazu byla vyhozena výjimka:
 6  list index out of range
 7 Očekávaný stav po kroku č. 2.
 8 Vezmi
 9 ------------------------------
10 Nevím, co mám zvednout.
11 Je třeba zadat název zvedaného předmětu.
12 ------------------------------
13 Aktuální prostor: Domeček
14 Sousedé prostoru: ('Les',)
15 Předměty v prostoru: ('Bábovka', 'Víno', 'Stůl', 'Panenka')
16 Předměty v batohu:   ()
17 ============================================================
```

Ze zadaného příkazu i očekávané odpovědi je zřejmé, že tento krok scénáře má testovat reakci hry na opomenutí uživatele, který nezadal požadovaný argument. Otevřeme-li definici třídy `_Take` reprezentující odpovídající akci (viz výpis 15.5 na straně 195), zjistíme, že metoda `execute()` opravdu netestuje počet prvků seznamu `arguments`. Nezadá-li proto uživatel požadovaný argument, bude se metoda na řádku `11` snažit získat neexistující prvek seznamu.

Oprava je jednoduchá: na začátek těla metody `execute()` vložíme podmíněný příkaz ověřující počet prvků v seznamu:

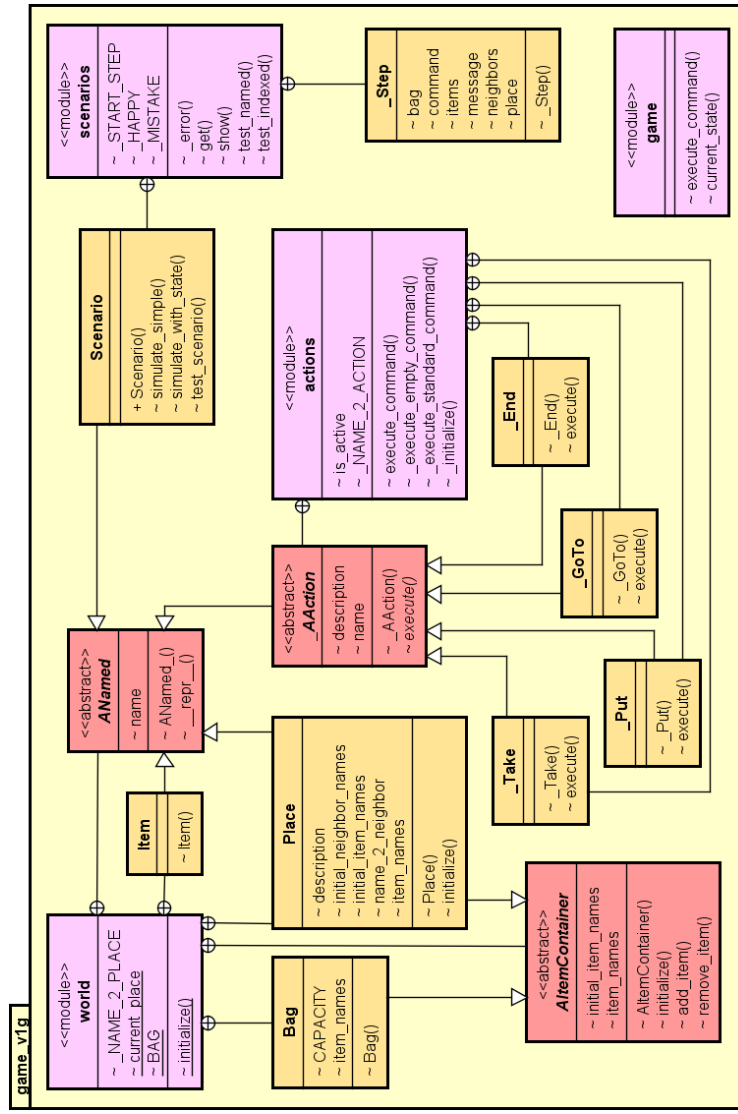
```
if len(arguments) < 2:
    return ('Nevím, co mám zvednout.\n'
            'Je třeba zadat název zvedaného předmětu.')
```

A protože obdobný problém budou mít i akce realizující pokládání h-objektu a přechod do sousedního prostoru, vložíme odpovídající úvodní příkazy i tam.

Při následujícím spuštění se test zarazí až ve čtvrtém kroku chybového scénáře, v němž Karkulka na požádání vloží do košíku stůl. To je trochu složitější problém, tak si jej ponecháme do příští kapitoly.

17.9 Shrnutí

Aktuální diagram tříd balíčku `game_v1g` najdete na obrázku 17.1. Zase maličko nakynul, tak jsem jej raději otočil. Zdrojové kódy odpovídající výše popsanému stavu vývoje aplikace najdete v doprovodných programech v balíčku `game.game_v1g`. Přehled výpisů programů z této kapitoly je v souborech s názvem `m17_remaining`.



Obrázek 17.1:
Diagram tříd modulu `game_v1g`

Kapitola 18
Batoh a nápověda

Co se v kapitole naučíte
V této kapitole vás nejprve seznámím s vykrajováním obsahu posloupností, což vzápětí využijeme při návrhu dalších akcí a jejich reakcí. Na konci kapitoly bude naše aplikace procházet testy podle šťastného i chybového scénáře.

18.1 Vykrajování (slicing)

Než se pustíme do vylepšování naší aplikace, opět přidám trochu teorie a informace o dalších možnostech jazyka *Python*.

O použití indexů při získávání prvků posloupností jsme hovořili v podkapitole *8.5 Získání prvku z kontejneru* na straně 122. Nyní své vědomosti o možnostech používání číselných indexů poněkud rozšíříme.

Tehdy jsem vám ale ještě neprozradil, že *Python* umožňuje indexovat posloupnosti i od konce. Index `-1` označuje poslední prvek zadané posloupnosti (stringu, seznamu, n-tice, …), index `-2` předposlední prvek a tak dále, až výraz `a[-len(a)]` představuje počáteční prvek posloupnosti `a`.

Velmi příjemným syntaktickým rozšířením indexace je vykrajování částí posloupností. Při něm se v hranatých závorkách za názvem posloupnosti zadají indexy prvního a „po-posledního“ znaku požadované podposloupnosti, oddělené vzájemně dvojtečkou. Není-li zadán index prvního znaku, dosadí se nula, není-li zadán index „po-posledního“ znaku, dosadí se délka posloupnosti. Bude-li proto v indexových závorkách pouze dvojtečka, vykrojí se celý řetězec.

V tabulce 18.1 najdete ukázky různých vykrojení částí řetězce `"Python"` uloženého v proměnné `python`. V levém sloupečku je zadávaný výraz, ve sloupečcích nadepsaných 0 až 5 jsou znaky s odpovídajícím indexem v původním řetězci, přičemž znaky, které se „neprobojují“ do výsledku, jsou podšeděny. V pravém sloupečku pak vidíte výsledek vyhodnocení výrazu z levého sloupečku.

Tabulka 18.1: *Tabulka různých způsobů vykrajování*

Výraz	0	1	2	3	4	5	Výsledek
`python[ 2: 5]`	P	y	t	h	o	n	`"tho"`
`python[ 2:  ]`	P	y	t	h	o	n	`"thon"`
`python[  : 2]`	P	y	t	h	o	n	`"Py"`
`python[ 2:-1]`	P	y	t	h	o	n	`"tho"`
`python[-4:-1]`	P	y	t	h	o	n	`"tho"`
`python[  :-2]`	P	y	t	h	o	n	`"Pyth"`
`python[-2:  ]`	P	y	t	h	o	n	`"on"`
`python[  :  ]`	P	y	t	h	o	n	`"Python"`
	-6	-5	-4	-3	-2	-1	

Na obrázku 18.1 je pak znázorněno používání „vykrajovacích indexů“, u nichž bychom si mohli přestavit, že indexují pozice mezi znaky.

```
  0   1   2   3   4   5   <-- Indexování od počátku
 -6  -5  -4  -3  -2  -1   <-- Indexování od konce
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
0   1   2   3   4   5   6 <-- Vykrajování od počátku
-6  -5  -4  -3  -2  -1     <-- Vykrajování od konce
```

Obrázek 18.1:
Indexování jednotlivých znaků textového řetězce – stringu

Opět bych chtěl připomenout, že stringy zde zastupují obecnou posloupnost a že stejně bychom indexovali pozice v seznamu, n-tici či libovolné jiné posloupnosti.

18.2 Nový balíček `game_v1h`

Stejně jako v řadě předchozích kapitol i zde začnu s novým balíčkem, abyste mohli později snadno zkontrolovat, co se v průběhu kapitoly změnilo. Nyní to bude (podle očekávání) balíček `game.game_v1h`, kam jsem zkopíroval soubory z balíčku `game.game_v1g`. V dalších úpravách budu pokračovat v tomto balíčku.

18.3 Nezvednutelné h-objekty

V minulé kapitole se při závěrečném spuštění test zarazil ve čtvrtém kroku chybového scénáře, v němž Karkulka na požádání vložila do košíku stůl, avšak správně měla hra odpovědět:

```
Zadaný předmět nelze zvednout: Stůl
```

Znamená to, že u vytvářených objektů musíme zadat, zda je možné je zvednout. Tady jsme omezeni tím, že nemůžeme tuto informaci zadat initoru v separátním argumentu, protože při definici prostorů v modulu `world` se zadávají pouze názvy h-objektů v daném prostoru. Máme tedy dvě možnosti:

- Upravit způsob zadávání h-objektů tak, aby bylo možné vedle názvu předat i další informace – např. přenositelnost daného h-objektu. Tyto informace by se pak předávaly initoru h-objektů v separátních argumentech.
- Ponechat stávající způsob, ale vložit potřebnou informaci do názvu objektu a upravit definici jeho initoru tak, aby si ji z názvu vypreparoval.

Doufám, že se mnou budete souhlasit, že druhá možnost působí dojmem, že bude vyžadovat méně úprav ve stávajícím programu a že i výsledný program bude přehlednější. Proto se vydáme touto cestou.

Doplníme v zadání názvy h-objektů o dodatečnou předponu, která ponese doplňkové informace – v našem případě informaci o přenositelnosti objektu. Bude-li počátečním znakem podtržení, bude objekt přenositelný, bude-li jím znak `#` (mříž), bude h-objekt nepřenositelný.

Upravená definice initoru h-objektů by pak mohla vypadat například jako ve výpisu 18.1. Všimněte si konstanty `HEAVY`, která obsahuje příznak, že objekt není přenositelný. Její hodnota se proto zadává na řádku `14` jako váha nepřenositelného objektu.

Teoreticky by zde stačilo číslo o jedničku větší, než je kapacita batohu, ale na tu se nemůžeme odvolávat. Batoh obsahuje h-objekty, tj. položky třídy `Item`, takže by vznikla definice kruhem, a té se vždy chceme vyvarovat.

Předpona může mít širší význam

V současné chvíli indikuje předpona pouze přenositelnost objektu. Nicméně její význam může být širší. V jedné hře indikovala to, zda je daný nápoj alkoholický, v další to, zda se dá daný předmět přenášet pouze oběma rukama (a uživatel je proto musí mít volné) atd.

Výpis 18.1: *Upravená definice třídy `Item` v modulu `world` v balíčku `game_v1h`*

```
1  class Item(ANamed):
2      """Instance reprezentují h-objekty v prostorech hry a v batohu."""
3
4      HEAVY: int = 999    # Váha nepřenositelného h-objektu
5
6      def __init__(self, name: str):
7          """Vytvoří h-objekt se zadaným názvem. Podle prvního znaku
8          pozná přenositelnost objektu a nastaví jeho váhu.
9          Zbylé znaky argumentu si zapamatuje jako jeho název.
10         """
11         prefix    = name[0]   # Předpona indikující další vlastnosti
12         real_name = name[1:]  # Název h-objektu používaný v příkazech
13         super().__init__(real_name)
14         self.weight = 1 if prefix == '_' else Item.HEAVY
```

Úprava metody _Take.execute()

Nyní je ještě třeba upravit definici metody execute() ve třídě _Take. Úprava je jednoduchá a určitě byste ji zvládli sami. Stačí před příkaz pro přidání předmětu do batohu (ve výpisu 15.5 na straně 195 je na řádku 16) vložit podmíněný příkaz:

```
if item.weight == world.Item.HEAVY:
    world.current_place.add_item(item)
    return 'Zadaný předmět nelze zvednout: ' + item.name
```

Před vlastním oznámením nezvednutelnosti zadaného h-objektu nesmíme zapomenout h-objekt do aktuálního prostoru opět vrátit, protože součástí testu přítomnosti daného h-objektu v aktuálním prostoru je jeho odebrání z prostoru.

18.4 Konečná kapacita batohu

Po předchozí úpravě metody pro odebrání h-objektu se test opět posune o kousek dál. Tentokrát se zastaví na sedmém kroku chybového scénáře, kde po Karkulce chceme, aby dala do košíku panenku. Kapacita košíku jsou dva předměty, takže by hra měla podle scénáře odpovědět

```
Zadaný předmět si již do košíku nevejde: Panenka
```

Místo toho ji však do košíku přidá. Nezbyde nám než ještě jednou upravit metodu _Take.execute(). Ve chvíli, kdy zjistíme, že se daný h-objekt nachází v aktuálním prostoru a je přenositelný, jej nemůžeme rovnou přidat, ale musíme provést pouhý pokus o přidání. Ten se povede pouze v případě, že se přidávaný předmět do batohu vejde.

Metoda try_add()

Bylo by proto vhodné definovat pro batoh metodu, kterou bychom mohli nazvat try_add a která bude mít na starosti právě onen pokus. Vrátí pak logickou hodnotu informující o tom, zda se pokus podařil. Možná definice této metody je ve výpisu 18.2.

Výpis 18.2: *Definice metody try_add() ve třídě Bag v modulu world v balíčku game_v1h*

```
1 def try_add(self, item: Item) -> bool:
2     """Pokusí se vložit do batohu zadaný objekt a vrátí informaci
3     o tom, zda se tam ještě vešel.
4     """
5     if len(self.items) + item.weight  > Bag.CAPACITY:
6         return False
7     self.add_item(item)
8     return True
```

Konečná verze metody _Take.execute()

Nyní bychom tedy měli mít k dispozici vše potřebné pro úpravu metody `_Take.execute()` do konečné podoby. V ní se o přidání h-objektu do batohu pouze pokusíme, a pokud se povede, vrátíme zprávu o provedené akci. Pokud se nepovede, vrátíme odebraný h-objekt zpět do aktuálního prostoru a vrátíme zprávu o tom, že se do batohu nevejde. Možnou konečnou podobu metody si můžete prohlédnout ve výpisu 18.3.

Výpis 18.3: *Definice metody `execute()` ve třídě `_Take` v modulu `actions` v balíčku `game_v1h`*

```
def execute(self, arguments: list[str]) -> str:
    """Ověří existenci zadaného h-objektu v aktuálním prostoru
    a je-li tam, přesune jej do košíku.
    """
    if len(arguments) < 2:
        return ('Nevím, co mám zvednout.\n'
                'Je třeba zadat název zvedaného předmětu.')
    item_name = arguments[1]
    item      = world.current_place.remove_item(item_name)
    if not item:
        return 'Zadaný předmět v prostoru není: ' + item_name
    if item.weight == world.Item.HEAVY:
        world.current_place.add_item(item)
        return 'Zadaný předmět nelze zvednout: ' + item.name
    if world.BAG.try_add(item):
        return 'Karkulka dala do košíku objekt: ' + item.name
    else:
        world.current_place.add_item(item)
        return 'Zadaný předmět se již do košíku nevejde: ' + item.name
```

Ověřovací test

Tentokrát test prošel až do 12. kroku, kde narazil na žádost o nápovědu, a tu hra ještě vypsat neumí. Pojďme na ni.

18.5 Nápověda

Získání nápovědy by nemělo být složité. Modul `actions` si ve slovníku `_NAME_2_ACTION` pamatuje všechny příkazy a každý příkaz si v atributu `description` pamatuje svoji stručnou charakteristiku. Stačí proto použít jednoduchý cyklus, jak je naznačeno ve výpisu 18.4.

Výpis 18.4: *Definice třídy `_Help` v modulu `actions` v balíčku `game_v1h`*

```
1  class _Help(_AAction):
2      """Zobrazí definované akce a jejich popisy.
3      """
4      def __init__(self):
5          super().__init__('?',
6                           'Zobrazí seznam dostupných akcí spolu'
7                           's jejich stručnými popisy.')
8
9      def execute(self, arguments: list[str]) -> str:
10         """Zobrazí definované akce a jejich popisy."""
11         result = 'Hra umožňuje zadat následující příkazy:\n\n'
12         for a in _NAME_2_ACTION.values():
13             result += f'{a.name}\n{a.description}\n\n'
14         return result
```

Výsledek testu

Spustíme-li nyní test, hra sice na žádost o nápovědu správně zareaguje, ale tester s její odpovědí není spokojen, protože tato odpověď je mnohem delší než ta, která je uvedena ve scénáři.

18.6 Úprava testovací funkce

Bylo by asi vhodné upravit test tak, že se u odpovědi porovná s plánovanou odpovědí pouze začátek skutečné odpovědi. Bude-li skutečná odpověď delší, můžeme prohlásit, že její zbytek obsahuje dodatečné informace, které není třeba prověřovat.

V hlavičce podmíněného příkazu testujícího odpověď hry (ve výpisu 11.4 na straně 157 na řádku 32) bychom mohli původní podmínku

```
step.message != answer
```

změnit na

```
step.message != answer[:len(step.message)]
```

Po této úpravě celý test úspěšně skončí.

18.7 Rozšíření výstupu

Předchozí úprava testovací funkce se může hodit i v situaci, kdy se rozhodneme vylepšit výstup hry a doplnit do něj např. informace o aktuálním stavu hry poskytované metodou `current_state()`. Toto doplnění můžeme navíc vázat na stav atributu modulu, který nazveme např. `print_state`.

Ve výpisu 18.5 je zobrazena definice proměnné `print_state` a upravená definice funkce `execute_command()`.

Výpis 18.5: *Upravená definice metody `execute_command()` v modulu `game` v balíčku `game_v1h`*

```
print_state = False

def execute_command(command: str) -> str:
    """Zpracuje zadaný příkaz a vrátí string se zprávou pro uživatele."""
    message = actions.execute_command(command) \
            + '\n' + 30*'-' \
            + (('\n' + current_state()) if print_state else '')
    return message
```

18.8 Shrnutí

Zdrojové kódy odpovídající výše popsanému stavu vývoje aplikace najdete v doprovodných programech v balíčku `game.game_v1h`. Přehled výpisů programů v této kapitole je v souborech s názvem `m18_bag_help`. Architekturu jsme opět rozšířili jen nepatrně, a proto diagram tříd neuvádím.

Kapitola 19
Spustitelná aplikace

Co se v kapitole naučíte
V této kapitole se seznámíte s cyklem `while`, který vzápětí použijeme při vybavování aplikaci uživatelským rozhraním. Od té chvíle ji bude moci používat nejen testovací program, ale i řadový uživatel. Poté se naučíte převést vyvinutou aplikaci do podoby, ve které se s ní budete moci pochlubit svým známým.

19.1 Příkaz `while` – cyklus

Na začátku kapitoly začneme opět troškou teorie. Občas potřebujeme, aby se nějaká činnost opakovala. Algoritmické konstrukce realizující opakování označujeme jako cykly.

Prozatím jsme pro naprogramování opakované činnosti používali cyklus `for` označovaný často také jako *cyklus s parametrem*, kde parametrem cyklu byly hodnoty odvozené z hodnot produkovaných zadaným zdrojem. Jinou často používanou konstrukcí je cyklus `while`, který se zapisuje podobně jako podmíněný příkaz a liší se pouze úvodním klíčovým slovem a samozřejmě způsobem provedení. Syntaxe tohoto příkazu je následující:

```
while vstupní_podmínka : tělo_cyklu
```

Příkaz `while` se vykonává tak, že se nejprve vyhodnotí vstupní podmínka, a je-li pravdivá, provede se tělo cyklu a běh programu se vrací na test vstupní podmínky. Tělo příkazu se provádí tak dlouho, dokud je vstupní podmínka pravdivá.

Jakmile se vstupní podmínka vyhodnotí jako nepravdivá, provádění těla se ukončí. Je-li nepravdivá hned při prvním testu, neprovede se tělo cyklu ani jednou.

Ve výpisu 19.1 jsou ukázky dvou funkcí využívajících cyklus `while`. Na řádcích 2-7 je definována funkce `countDown`, která odpočítává čas do symbolického výbuchu. Na řádcích 12-18 je pak definována funkce `factW` vracející faktoriál zadaného čísla.

Výpis 19.1: *Ukázky definic funkcí používajících cyklus* `while`

```
>>> \
def countDown(n) -> None:
    """Vytiskne zadaný počet čísel - a po nich slovo BUM."""
    while n>0:
        print(n, end='-')
        n -= 1
    print('BUM')

>>> countDown(5)
5-4-3-2-1-BUM
>>> \
def factW(n):
    """Faktoriál počítaný pomocí cyklu while."""
    f = 1       # Hodnota vracená v případě, když n <= 1
    while n > 1:
        f *= n
        n -= 1
    return f

>>> factW(5)
120
>>>
```

19.2 Nekonečný cyklus

Cyklus `while` definuje ve své hlavičce podmínku, po jejímž nesplnění se cyklus ukončí. Občas je ale výhodné, aby tato podmínka byla vždy pravdivá a cyklus nikdy neskončil. Nejjednodušší nekonečný cyklus realizujeme příkazem:

```
while True : pass
```

Výpis 19.2 naznačuje možnou definici funkce obsahující nekonečný cyklus (řádky 2-5). Po zavolání této funkce v IDLE se s vámi prostředí přestane bavit. Dovolí vám sice psát, ale na zadávané texty nebude reagovat (to výpis nezobrazuje, zkuste si to sami). Zareaguje až na stisk CTRL+C, který násilně přeruší chod programu, a ten vám to samozřejmě oznámí (řádky 8-13).

Nekonečný cyklus se v programech používá častěji, než by se na první pohled zdálo. Nekonečným cyklem je např. samotný běh operačního systému a řada dalších činností. Většinou ale daný cyklus není doopravdy nekonečný, ale je v něm někde definována podmínka, po jejímž splnění jej program opustí.

Jak uvidíte za chvíli, prostřednictvím nekonečného cyklu bude nakonec definována i funkce realizující běh naší hry.

Výpis 19.2: *Nekonečný cyklus*

```
>>> \
def infinite() -> None:
    """Spustí prázdný nekonečný cyklus, který bude nutné přerušit zvenku."""
    while True: pass
    print('Hotovo')

>>> infinite ()
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    infinite()
  File "<pyshell#10>", line 4, in infinite
    while True: pass
KeyboardInterrupt
>>>
```

19.3 Logické operátory a operace

Název podkapitoly označuje probírané operátory jako logické a jako takové jsou definovány ve většině programovacích jazyků. *Python* ale definuje tyto tři operátory poněkud nestandardně:

`not` Unární operátor, který se jako jediný chová jako logický operátor a vrací negaci logické hodnoty svého operandu. Výsledkem operace je vždy logická hodnota. Výraz `not X` se interpretuje: „*není pravda, že X*“.

`and` Binární operátor, jehož klasická verze vrátí `True` právě tehdy, mají-li oba jeho operandy hodnotu `True`. Výraz `X and Y` se interpretuje: „*X a zároveň Y*“.

Jazyk *Python* však provádí tuto operaci následovně: je-li logická hodnota levého operandu `False`, vrátí skutečnou hodnotu daného operandu. Je-li jeho logická hodnota `True`, vrátí skutečnou hodnotu pravého operandu.

Převedeme-li vrácenou hodnotu na logickou, tak nám vyjde totéž co v jiných jazycích. Jenomže v *Pythonu* bývá často výsledkem logických operací hodnota jiného typu, kterou musíme jako logickou teprve interpretovat.

`or` Binární operátor, jehož klasická verze vrátí `False` právě tehdy, mají-li oba jeho operandy hodnotu `False`. Výraz `X or Y` se interpretuje: „*X nebo Y*“.

I tuto operaci provádí jazyk *Python* nestandardně: je-li logická hodnota levého operandu `True`, vrátí skutečnou hodnotu daného operandu. Je-li jeho logická hodnota `False`, vrátí skutečnou hodnotu pravého operandu.

Popsané chování demonstruje výpis 19.3. Předpokládám, že vše je vysvětleno v komentářích a demonstrační příklady další výklad nepotřebují.

Programátory se zkušenostmi z jiných jazyků může zaskočit, že logický výraz může vrátit hodnotu libovolného typu.

Výpis 19.3: Chování logických operátorů

```
>>> a = 5 > 3          #Výsledek porovnání (logická hodnota) ukládám do proměnné
>>> a
True
>>> not a
False
>>> '' and 'cokoliv'  #Levý operand je False => jeho hodnota je výsledkem
''
>>> 1 and ''          #Levý operand je True => výsledkem je hodnota pravého
''
>>> '' or 'cokoliv'   #Levý operand je False => výsledkem je hodnota pravého
'cokoliv'
>>> 1 or ''           #Levý operand je True => jeho hodnota je výsledkem
1
>>> #V následujícím příkladu jsou hodnoty prvních tří operandů False =>
>>> not a or 0 or "" or "poslední" # => výsledkem je hodnota posledního
'poslední'
>>>
```

19.4 Balíček game_v1i

Jak už jistě očekáváte, na počátku kapitoly začneme s novým balíčkem. Nyní to bude (podle očekávání) balíček `game.game_v1i`, do nějž překopíruju soubory z balíčku `game.game_v1h`. V dalších úpravách budeme pokračovat zde.

19.5 Jednoduché textové uživatelské rozhraní

Jediný, kdo si mohl naši hru doposud zahrát, byl testovací program. Když jsme s jeho pomocí dovedli aplikaci do stavu, že funguje podle zadání (alespoň si to myslíme), měli bychom ji doplnit o modul realizující komunikaci s uživatelem.

Možností, které nám *Python* pro tento účel poskytuje, je mnoho. My ale začneme tou nejjednodušší, kterou je komunikace prostřednictvím příkazového řádku. Tu jsme si už vyzkoušeli např. ve výpisu 3.3 na straně 59.

Mnozí možná budou tvrdit, že v současné době je textové rozhraní příkazového řádku přežitek. Těm bych odpověděl, že podle zásad agilního programování bychom měli na počátku rozchodit to nejjednodušší řešení. Grafické rozhraní bychom mohli naprogramovat v dalším kole, i když jeho netriviální výklad by zabral další učebnici. Bude-li zájem, mohu ji připravit.

Ve výpisu 19.4 je definice funkce `run()`, jejímž zavoláním spustíte hru pro uživatele komunikujícího s programem prostřednictvím příkazového řádku. Funkce si připraví prázdný string jako příkaz spouštějící hru a pak v nekonečném cyklu (řádek 4) vždy zavolá funkci `execute_command()`, které předá v argumentu zadaný příkaz (řádek 5). Poté vytiskne odpověď hry (řádek 6) a zjišťuje (řádek 7), zda poslední zadaný příkaz hru neukončil. Pokud ano, opustí nekonečný cyklus (řádek 8) – a tím funkce končí. Pokud ne, zeptá se uživatele na další příkaz (řádek 11) a tělo cyklu jede znovu od počátku.

Jak je v komentáři k příkazu `break` naznačeno, v tomto případě by asi bylo lepší zadat na řádku 8 příkaz `return`, protože s ukončením hry končí současně i běh dané funkce. Příkaz `break` jsem použil jenom proto, abych vám připomenul jeho funkci.

Výpis 19.4: *Definice metody `run()` v modulu `game` v balíčku `game_v1i`*

```
1  def run() -> None:
2      """Spustí hru ovládanou z příkazového řádku."""
3      command = ''
4      while True:     # Nekonečný cyklus hraní hry
5          message = execute_command(command)
6          print(message)
7          if not actions.is_active:
8              break    # Hra je ukončena => vyskakujeme z cyklu ---------->
9              # Předchozí příkaz by mohl být i return, což by bylo lepší,
10             # protože v danou chvíli hra beztak končí
11         command = input('Zadejte příkaz: ')
```

19.6 Možnost opakovaného spouštění

Někdy je výhodné mít možnost po ukončení jednoho běhu hru opět spustit, aniž bychom museli spouštět znovu celou aplikaci. Možnost takto koncipované komunikace s uživatelem demonstruje funkce `multirun()`, jejíž definici najdete ve výpisu 19.5.

Tato funkce používá dokonce dva do sebe vnořené nekonečné cykly. Vnější cyklus definovaný na řádcích 3–13 umožňuje opakované spouštění hry. Nejprve na řádku 4 spustí jeden běh hry a po jeho skončení se zeptá uživatele, zda by si nechtěl zahrát ještě jednou.

Získání odpovědi na tuto otázku má na starosti druhý nekonečný cyklus definovaný na řádcích 5–10. Nejprve se na řádku 6 uživatele zeptá, zda by si nechtěl zahrát ještě jednou, a poté analyzuje, zda ze zadané odpovědi umí poznat, jak se uživatel rozhodl. Test na řádku 7 zjišťuje, zda uživatel zadal neprázdný string začínající některým ze zadaných znaků.

Pokud ano, považuje odpověď za korektní, vyskočí z vnitřního cyklu (řádek 8) a pokračuje podmíněným příkazem na řádku 11, v němž zjišťuje, pro co se uživatel rozhodl. Zjistí-li, že chce končit, opustí na řádku 12 i vnější cyklus a zakončí běh funkce tiskem na řádku 14, v němž uživateli ještě jednou poděkuje za hru.

Nechce-li už skončit, vrací se na začátek těla cyklu na řádku 4 a hru zavoláním funkce `run()` znovu spouští.

Pokud test na řádku 7 zjistí, že uživatel nezadal korektní odpověď, z níž by bylo možné zjistit jeho přání, upozorní jej na korektní podobu odpovědi a vrací se na začátek vnitřního cyklu na řádku 6.

Výpis 19.5: *Definice metody `multirun()` v modulu `game` v balíčku `game_v1i`*

```
1  def multirun() -> None:
2      """Umožní opakované spuštění hry ovládané z příkazového řádku."""
3      while True:
4          run()
5          while True:
6              answer = input('Chcete si zahrát ještě jednou (A/N): ').strip()
7              if (len(answer) > 0)   and   (answer[0] in '01ANan'):
8                  break       # Výskok z vnitřního cyklu    ---------->
9              print('Odpověď musí začínat některým ze znaků "01ANan"\n'
10                   'Zkuste odpovědět znovu.')
11         if answer[0] in '0Nn':
12             break       # Výskok z vnějšího cyklu    ---------->
13         print('\nDobře, zahrajeme si ještě jednou.\n')
14     print('\nJeště jednou děkuji za hru.\nNa shledanou.')
```

19.7 Kontrolní tisky

Při zavádění modulů jsme doposud používali kontrolní tisky, které nás informovaly o postupu, v jakém jsou moduly zaváděny. Když už jsme aplikaci rozchodili, mohli bychom je smazat, nebo alespoň zakomentovat, aby uživatele nerušily. Problém nastane v okamžiku, kdy se v aplikaci objeví chyba, a my budeme potřebovat kontrolní tisky opět oživit. Výhodnější než zakomentovat je proto pouze je deaktivovat.

Konstanta `__debug__`

Python definuje logickou konstantu `__debug__`, jejíž hodnota `True` oznamuje, že se program nachází v ladicím režimu, a budou se proto aktivovat některé ladicí tisky a další konstrukce. Problém je v tom, že tato konstanta je implicitně nastavena na `True` a lze ji shodit pouze zadáním argumentu `-O` (O = optimalizovat) při spouštění *Pythonu*.

Při spouštění aplikace poklepáním ale nelze *Pythonu* zadávat argumenty příkazového řádku. Chceme-li proto umožnit uživateli tento způsob spouštění aplikace, musíme zvolit jinou cestu – takovou, při níž můžeme ovlivnit podrobnost kontrolních tisků přímo v kódu programu.

Mohli bychom sice definovat zástupce spouštěného programu, který by potřebný argument dodal, ale příprava takovéto aplikace již přesahuje začátečnické zaměření této učebnice.

Alternativní postup

Alternativní možností je definovat obdobnou globální konstantu ve speciálním modulu definovaném v kořenovém balíčku. Nazvěme modul `dbg` a definujme v něm proměnnou

```
DBG = 0      # Nastavení hladiny kontrolních tisků
```

Upravíme-li pak ve všech modulech počáteční kontrolní tisky do tvaru (využívám možnost definovat celý složený příkaz na jednom řádku, o níž jsem se zmiňoval v poznámce na straně 55):

```
from dbg import DBG
if DBG>0: print(f'===== Modul {__name__} ===== START')
```

a závěrečné kontrolní tisky do tvaru

```
if DBG>0: print(f'===== Modul {__name__} ===== STOP')
```

tak se kontrolní tisky přestanou tisknout. Jakmile však objevíme chybu a budeme si chtít připomenout, v jakém pořadí se jednotlivé balíčky a moduly zaváděly, stačí změnit hodnotu v modulu `dbg` a tisky se při dalším běhu opět objeví.

Pomocí hodnoty této konstanty bychom dokonce mohli nastavovat podrobnost kontrolních tisků.

Dokonalejší postup

Programátoři označují provádění kontrolních tisků jako tzv. logování. Protože mají na jeho univerzalitu poměrně vysoké nároky, bývají pro ně definovány speciální knihovny. V *Pythonu* se pro tyto účely používá modul `logging`. Postup, který jsem v předchozí pasáži naznačil, je jeho slabý odvar.

Výklad použití modulu `logging` již překračuje rámec této učebnice. Začnete-li však vytvářet rozsáhlejší aplikace, měli byste se s ním seznámit

19.8 Přímé spuštění zadaného skriptu

Máte-li kód, který chcete spustit, v jediném souboru, můžete tento soubor přímo spustit jako skript. Oproti interaktivnímu režimu se v takovém případě většinou rovnou spouští naprogramovaná akce, kdežto v interaktivním režimu se modul pouze importuje, čímž se zpřístupní jeho atributy, abychom je mohli následně použít.

Aby bylo možné daný skript spouštět jak samostatně, tak v rámci konverzace v interaktivním režimu, je třeba umět poznat, v jakém režimu je modul zaváděn.

Rozpoznání režimu, v němž byl modul spuštěn

V pasáži *Vše je součástí nějakého modulu* na straně 78 jsme si řekli, že v interaktivním režimu se modul, jehož součástí je vše, co v rámci konverzace vytvoříme, jmenuje

`__main__`. To je v *Pythonu* povinné jméno hlavního modulu, a tím je v případě samostatně spuštěné aplikace její kořenový balíček. Z toho vyplývá:

- V interaktivním režimu je třeba modul zavést explicitním zadáním některé z verzí příkazu `import`, přičemž modul vystupuje pod jménem importovaného souboru.
- Spouštíte-li skript tvořený jediným modulem, tak se tento modul stává hlavním modulem a jmenuje se proto `__main__`. Z toho vyplývá, že mají-li být skripty použitelné jak přímo, tak v interaktivním režimu, je vhodné v nich definovat podmíněný příkaz:

```
if __name__ == '__main__':
    # Kód, který se spustí, je-li daný modul spouštěn přímo
```

- Spustitelné aplikace tvořené větším počtem modulů (zanedlouho si ukážeme, jak se taková aplikace vytváří) musejí mít definován modul `__main__`, jehož zdrojový soubor musí být definován v kořenovém balíčku.

Demonstrace

Pro demonstraci pravidla o spouštění jednoduchého skriptu jsem definoval modul `m19a_script`, jehož zdrojový kód si můžete prohlédnout ve výpisu 19.6. Vedle úvodního dokumentačního komentáře modul obsahuje úvodní a závěrečný kontrolní tisk, který nám prozradí aktuální jméno daného modulu. Mezi nimi je pak podmíněný příkaz, který z aktuálního jména modulu odvodí, jak byl daný modul spuštěn, a tuto informaci vytiskne.

Zprávu modulu po jeho spuštění v interaktivním režimu si můžete prohlédnout ve výpisu 19.7. Jak vidíte, modul má název shodný s názvem souboru se zdrojovým textem.

Zprávu modulu po jeho spuštění v příkazovém panelu *Windows* si můžete prohlédnout ve výpisu 19.8. Tentokrát obdržel modul od systému název `__main__`.

Spuštění modulu poklepáním vám neukážu, protože to bychom museli dovybavit modul grafickým uživatelským rozhraním, jehož použití jsem vám prozatím nevysvětloval.

Výpis 19.6: *Definice modulu* `m19a_script`

```
"""
Modul pro demonstraci rozpoznání režimu, v němž je spouštěn.
"""
print(f'===== Modul {__name__} ===== START')

if __name__ == '__main__':
    print(f'Skript je spouštěn ze systému')
else:
    print(f'Skript je spouštěn v interaktivním režimu')

print(f'===== Modul {__name__} ===== STOP')
```

Výpis 19.7: *Reakce na spuštění modulu `m19a_script` v interaktivním režimu*

```
>>> import m19a_script
===== Modul m19a_script ===== START
Skript je spouštěn v interaktivním režimu
===== Modul m19a_script ===== STOP
>>>
```

Výpis 19.8: *Reakce na spuštění modulu `m19a_script` v příkazovém okně Windows*

```
Microsoft Windows [Version 10.0.18362.900]
(c) 2019 Microsoft Corporation. Všechna práva vyhrazena.

Q:\65_PGM\65_PYT>m19a_script.py
===== Modul __main__ ===== START
Skript je spouštěn ze systému
===== Modul __main__ ===== STOP

Q:\65_PGM\65_PYT>
```

19.9 Vytvoření spustitelné aplikace

Pojďme si ukázat, jak ze současné podoby naší hry vytvořit samostatnou aplikaci. Postup není složitý.

Ukážeme si nejprve postup, kdy pro vytvoření aplikace použijeme modul `zipapp`, jenž je součástí standardní knihovny. Postupujte následovně:

1. Vytvořte složku pro zdrojové kódy budoucí aplikace a nazvěte ji např. `Game`.
2. Do této složky zkopírujte složku `game_v1i` a soubor `dbg.py` z kořenového balíčku.
3. Přesuňte se do složky, v níž jste vytvořili složku `Game`.
4. Z příkazového řádku zadejte příkaz

   ```
   python -m zipapp Game -m game_v1i.game:run
   ```

   Tím spustíte modul `zipapp`, zadáte mu název složky se svým programem a žádáte ho, aby vytvořil ZIP-soubor pojmenovaný stejně jako zadávaná složka (tj. `Game.zip`).

   Zadáním argumentu `-m` následovaného cestou k funkci `run` žádáte, aby modul `zipapp` vytvořil v kořenovém balíčku modul `__main__`, který spustí funkci `run()` v modulu `game_v1.game`. Všimněte si, že na rozdíl od kódu, kde se všude používají tečky, je zde před názvem zadávané funkce dvojtečka.

Možná jste si všimli, že při vývoji aplikace jsme ukládali balíčky s jednotlivými verzemi jako podbalíčky balíčku `game`, kdežto zde jsem tento společný rodičovský balíček zrušil. To si můžeme dovolit proto, že jsme všechny importy uvnitř naší aplikace zadávali relativně (viz pasáž *Relativní import* na straně 113).

Po provedení zadaného příkazu se vedle složky se zdrojovým programem (v našem případě složka `Game`) objeví stejnojmenný soubor s příponou `pyz` (*Python* zip). Pokud na něj poklepete nebo zadáte jeho název jako příkaz v příkazovém řádku, tak se program spustí – vyzkoušejte si to. Záznam seance, při níž jsem aplikaci vytvořil, spustil a hned ukončil, najdete ve výpisu 19.9.

Výpis 19.9: *Vytvoření samostatné aplikace pomocí modu `winapp` a její spuštění*

```
Microsoft Windows [Version 10.0.18362.900]
(c) 2019 Microsoft Corporation. Všechna práva vyhrazena.

Q:\65_PGM\65_APP>python -m zipapp Game -m game_v1i.game:run

Q:\65_PGM\65_APP>Game.pyz
Vítejte!
Toto je příběh o Červené Karkulce, babičce a vlkovi.
Svými příkazy řídíte Karkulku, aby donesla věci babičce.
Nebudete-li si vědět rady, zadejte znak ?.
------------------------------
Zadejte příkaz: vezmi víno
Karkulka dala do košíku objekt: Víno
------------------------------
Zadejte příkaz: konec
Ukončili jste hru.
Děkujeme, že jste si zahráli.
------------------------------

Q:\65_PGM\65_APP>
```

Soubor typu `pyz`

Pojďme se podívat, co jsme to vytvořili, a nahlédněme do útrob souboru `Game.pyz`. Vzhledem k tomu, že je to klasický ZIP-soubor, tak by vám měla toto nahlížení umožnit většina správců souborů včetně prostoduchého průzkumníka ve *Windows*.

Při prohlídce zjistíte, že v archivu jsou pouze kopie zdrojových souborů a že v kořenové složce přibyl soubor `__main__.py`. Podíváte-li se do něj, najdete kód z výpisu 19.10.

Výpis 19.10: *Definice modulu `__main__` v souboru `Game.pyz`*

```
# -*- coding: utf-8 -*-
import game_v1i.game
game_v1i.game.run()
```

První řádek pouze standardním způsobem oznamuje, že soubor je kódován v UTF-8, což je sice u souboru, který používá pouze ASCII, nedůležité, ale dokumentace naznačuje, že by tam tento řádek měl být.

19.10 Argumenty příkazového řádku

Řada aplikací je schopna modifikovat svoji činnost na základě argumentů zadávaných v příkazovém řádku při spouštění aplikace. Tyto argumenty jsou uloženy v seznamu, na nějž odkazuje proměnná `argv` v modulu `sys`. V nultém prvku tohoto seznamu je název spouštěného skriptu či aplikace a v dalších prvcích jsou pak jednotlivé stringy zadávané jako argumenty.

Tato proměnná se ale uplatní pouze tehdy, když skript či aplikaci spouštíte z příkazového řádku. Mohli bychom proto do modulu `game` doplnit funkci nazvanou `main`, která by tyto argumenty analyzovala a podle nich nastavila chování naší aplikace.

Doplnění modulu `game`

Pojďme rozšířit modul `game` o funkce, které vylepší chování naší aplikace. Ve výpisu 19.11 je na řádcích **1–15** definována funkce `main()`, která argumenty příkazového řádku analyzuje a podle nich rozhodne, jak bude aplikace pokračovat.

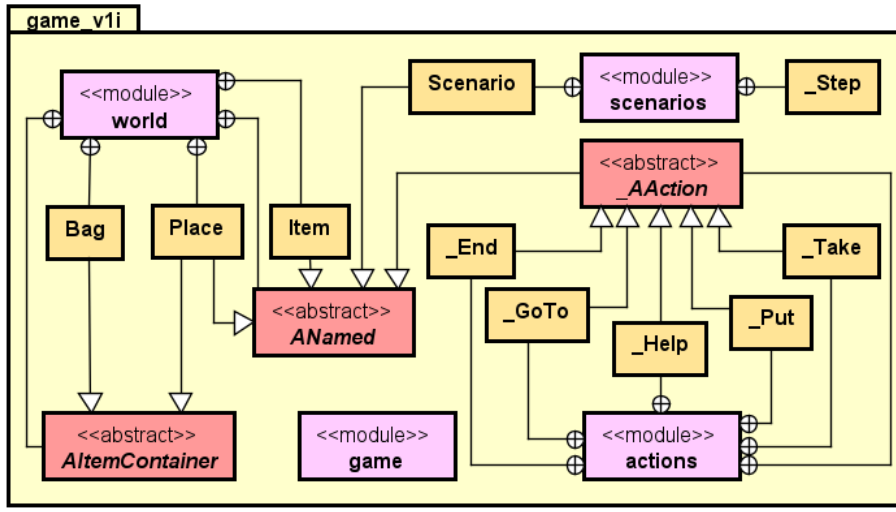
Zde bychom neměli zapomínat na možnost zadat žádost o nápovědu, která bývá podle konvencí zadávána argumentem `-h` a seznámí uživatele se základním ovládáním dané aplikace. Funkce `help()`, která požadovanou nápovědu vypíše, je definována na řádcích **18–25**.

Výpis 19.11: *Definice funkcí `main()` a `help()` v modulu `game` v balíčku `game_v1i`*

```
1  def main() -> None:
2      """Spustí aplikaci podle zadaných argumentů příkazového řádku."""
3      args = sys.argv
4      if DBG>=0: print(f'main - argumenty příkazového řádku: {args}')
5      if '-h' in args:
6          help()
7          return
8
9      global print_state
10     print_state = ('-s' in args)
11
12     if '-m' in args:
13         multirun()
14     else:
15         run()
16
17
18 def help() -> None:
19     """Zobrazí nápovědu k aplikaci."""
20     print('Aplikace představuje jednoduchou konverzační hru.\n'
21           'Při spuštění můžete zadat následující argumenty:\n'
22           '-h  Spustí tuto nápovědu\n'
23           '-m  Umožní zahrát si hru po skončení znovu\n'
24           '-s  Součástí odpovědi hry na zadání příkazu bude informace\n'
25           '    o aktuálním stavu světa hry')
```

19.11 Shrnutí

Aktuální diagram tříd balíčku `game_v1i` najdete na obrázku 19.1. Tentokrát již nezobrazuje atributy a metody, ale ukazuje pouze přehled architektury, v němž jsou zobrazeny jen názvy tříd a případné stereotypy. Zdrojové kódy odpovídající výše popsanému stavu vývoje aplikace najdete v doprovodných programech v balíčku `game.game_v1i`. Přehled výpisů programů v této kapitole je v souborech s názvem `m19_ui`.



Obrázek 19.1:
Stručný diagram tříd modulu `game_v1i`

Část D
Vylepšujeme aplikaci

Ve čtvrté, závěrečné části si nejprve ukážeme, jak lze nahradit literály konstantami a jaké to může mít výhody při případných dalších úpravách a vylepšování programu. Poté si ukážeme, jak nahradit současné textové uživatelské rozhraní jednoduchým grafickým, a na závěr si povíme, kde můžete získávat další znalosti.

Kapitola 20
Převod literálů na konstanty

Co se v kapitole naučíte
Tato kapitola vám předvede, jak je možné převést roztroušené literály na konstanty a jaké to může mít výhody pro další vylepšování programu. Poté vás seznámím s několika náměty, jak program dále vylepšit.

20.1 Magické hodnoty

V pasáži *DRY – bez kopií* na straně 93 jsem vám říkal, že by se v programech neměly vyskytovat tzv. magické hodnoty zadávané jako literály, ale že bychom měli vždy definovat nějaké konstanty, které bychom pak v programu používali místo těchto literálů.

Je to krajně důležité v okamžiku, kdy danou hodnotu používáme na více místech, ale bývá to vhodné i tehdy, objeví-li se daná hodnota v programu jen jednou. Pak totiž v případě potřeby danou hodnotu změnit nemusím v programu hledat místo, kde je použito, ale budu opravu zanášet na známém místě, kde jsou definovány příslušné konstanty.

V našem programu máme prozatím definovány pouze dvě takové konstanty, a to váhu nepřenositelného h-objektu a kapacitu batohu. To jsou ale obě číselné konstanty. O možnosti uložení použitých stringových literálů do konstant jsme prozatím vůbec neuvažovali.

Takový převod má jednu nevýhodu: názvy použitých konstant nikdy nebudou tak „výmluvné“ jako vlastní texty, takže pochopení pročítaného programu je pak o něco náročnější.

Na druhou stranu má ale takový převod řadu nejrůznějších výhod, které onu nevýhodu převáží:

- Můžeme koncentrovat všechny texty na jednom místě, kde se snáze kontrolují a modifikují.
- Snáze se zanášejí nejrůznější opravy.
- Lze snadno definovat různé jazykové verze.

Nevím, jestli jste se pokoušeli definovat vlastní verzi hry, ale při používání literálů, jak jsme to dělali doposud, vám hrozí dvě věci:

- Při kopírování stejného textu mezi různými místy programu dochází k nepřesnostem, způsobeným většinou špatným zkopírováním krajních znaků, zejména jsou-li to mezery.
- Objevíte-li v některém z textů chybu, musíte najít všechny jeho výskyty a všude ji správně opravit.

To nám při používání konstant odpadne, protože za nás řadu kontrol provede již překladač.

20.2 Modul textových konstant

Existuje několik způsobů, jak potřebné konstanty vhodně definovat. V prvním přiblížení bychom mohli zmínit dva:

- Definovat separátní modul, do nějž bychom příslušné definice vložili jako zdrojový program.
- Uložení textů do souboru a jejich načtení vhodným programem, který je současně uloží do kontejneru, z nějž je pak budou jednotlivé části číst.

Vzhledem k tomu, že v *Pythonu* jsou programy stejně uloženy ve zdrojovém tvaru, tak dám přednost prvnímu způsobu, protože nám pak odpadne programování spojené s načítáním jednotlivých konstant a jejich následnou kontrolou, protože to za nás provede překladač.

Pro definici textových konstant jsem proto zavedl nový modul pojmenovaný `texts`. V něm jsem pro jednotlivé skupiny konstant definoval třídy a příslušné konstanty jsem pak zavedl jako jejich třídní atributy.

20.3 Konstanty související s prostory

Pro konstanty související s prostory jsou definovány dvě třídy. Atributy třídy `PlaceName` uchovávají názvy jednotlivých prostorů a atributy třídy `PlaceDescription` jejich stručné popisy (viz výpis 20.1).

Atributy odpovídající jednotlivým prostorům se v obou třídách jmenují stejně, čímž se zjednoduší orientace v názvech konstant. To, jestli se v daném okamžiku použije název nebo popis, se odvodí od použité kvalifikace.

Výpis 20.1: *Definice tříd `PlaceName` a `PlaceDescription` v modulu `texts` v balíčku `game_v2a`*

```
class PlaceName:
    """Názvy prostorů hry"""
    house   = 'Domeček'
    wood    = 'Les'
    forest  = 'Temný_les'
    cottage = 'Chaloupka'
    cave    = 'Jeskyně'

class PlaceDescription:
    """Stručné popisy prostorů hry"""
    house   = 'Domeček, kde bydlí Karkulka'
    wood    = 'Les s jahodami, malinami a pramenem vody'
    forest  = 'Temný_les s jeskyní a číhajícím vlkem'
    cottage = 'Chaloupka, kde bydlí babička'
    cave    = 'Jeskyně, kde v zimě přespává medvěd'
```

Výpis 20.2: *Definice tříd `ItemName` a `ItemWeight` v modulu `texts` v balíčku `game_v2a`*

```
class ItemName:
    """Názvy h-objektů použitých ve hře."""
    cake        = 'Bábovka'
    wine        = 'Víno'
    table       = 'Stůl'
    doll        = 'Panenka'
    wolf        = 'Vlk'
    raspberry   = 'Maliny'
    strawberry  = 'Jahody'
    well        = 'Studánka'
    bed         = 'Postel'
    granny      = 'Babička'

class ItemWeight:
    """Názvy h-objektů doplněné o informační prefixy."""
    light = '_'  # Prefix označující běžný h-objekt
    heavy = '#'  # Prefix označující nezvednutelný h-objekt
    cake        = light + ItemName.cake
    wine        = light + ItemName.wine
    table       = heavy + ItemName.table
    doll        = light + ItemName.doll
    wolf        = heavy + ItemName.wolf
    raspberry   = light + ItemName.raspberry
    strawberry  = light + ItemName.strawberry
    well        = heavy + ItemName.well
    bed         = heavy + ItemName.bed
    granny      = heavy + ItemName.granny
```

20.4 Konstanty související s h-objekty

Pro konstanty související s h-objekty jsou definovány opět dvě třídy. Atributy třídy `ItemName` uchovávají názvy jednotlivých h-objektů a atributy třídy `ItemWeight` pak před názvy doplní příslušné prefixy (viz výpis 20.2). Ty se použijí při vytváření příslušných h-objektů a běžně ve všech ostatních situacích.

Odpovídající atributy se opět jmenují v obou třídách stejně, takže to, zda se použije prostý název, nebo název s předponou, se určí prostřednictvím kvalifikace.

20.5 Definice světa hry

V modulu `world` se změní závěrečná definice světa ve slovníku `_NAME_2_PLACE` a přibydou příslušné importy. Ve výpisu 20.3 jsou tyto importy uvedeny a ze slovníku jsou zde definice prvních dvou prostorů. Všimněte si, že odkazy na importované třídy jsou v proměnných s dvoupísmennými názvy, aby se odkazy na konstanty trochu zpřehlednily.

Výpis 20.3: *Definice slovníku `_NAME_2_PLACE` v modulu `world` v balíčku `game_v2a`*

```
1  from .texts import PlaceName         as PN
2  from .texts import PlaceDescription  as PD
3  from .texts import ItemWeight        as IW
4
5  _NAME_2_PLACE = {p.name.lower() : p for p in (
6      Place(PN.house,
7            PD.house,
8            (PN.wood,),
9            (IW.cake, IW.wine, IW.table, IW.doll, ),
10          ),
11     Place(PN.wood,
12           PD.wood,
13           (PN.house, PN.forest,),
14           (IW.raspberry, IW.strawberry, IW.well, ),
15         ),
```

V modulu `world` byla provedena ještě jedna změna, a to v initoru třídy `Item`. Zde byl v posledním příkazu (ve výpisu 18.1 na straně 218 na řádku 14) nahrazen literál odkazem na konstantu, takže řádek má nyní tvar

```
self.weight = 1 if prefix == IW.light else Item.HEAVY
```

20.6 Další úpravy

Obdobně je třeba projít i zbylé moduly a nahradit v nich literály příslušnými konstantami. Pro lepší představu je ve výpisu 20.4 uveden konec definice třídy `ActionMessages` s definicemi konstant se zprávami vracenými jednotlivými příkazy.

Výpis 20.4: *Konec definice třídy `ActionMessages` v modulu `texts` v balíčku `game_v2a`*

```
class ActionMessages:
    """Zprávy vydávané při zpracování příkazů."""

...# Vynechané příkazy

    take_no    = ('Nevím, co mám zvednout.\n'
                  'Je třeba zadat název zvedaného předmětu.')
    take_wrong = 'Zadaný předmět v prostoru není:'
    take_heavy = 'Zadaný předmět nelze zvednout:'
    take_full  = 'Zadaný předmět se již do košíku nevejde:'
    take_done  = 'Karkulka dala do košíku objekt:'
```

Ve výpisu 20.5 je pak uvedena upravená definice metody `execute()` ve třídě `_Take`. Z výpisu si jistě domyslíte, že třída `ActionMessages` byla importována příkazem

```
from .texts import ActionMessages as AM
```

Výpis 20.5: *Definice metody `execute()` ve třídě `_Take` v modulu `actions` v balíčku `game_v2a`*

```
def execute(self, arguments: list[str]) -> str:
    """Ověří existenci zadaného h-objektu v aktuálním prostoru
    a je-li tam, přesune jej do košíku.
    """
    if len(arguments) < 2:
        return AM.take_no
    item_name = arguments[1]
    item      = world.current_place.remove_item(item_name)
    if not item:
        return f'{AM.take_wrong} {item_name}'
    if item.weight == world.Item.HEAVY:
        world.current_place.add_item(item)
        return f'{AM.take_heavy} {item.name}'
    if world.BAG.try_add(item):
        return f'{AM.take_done} {item.name}'
    else:
        world.current_place.add_item(item)
        return f'{AM.take_full} {item.name}'
```

20.7 Shrnutí

Zdrojové kódy odpovídající výše popsanému stavu vývoje aplikace najdete v doprovodných programech v balíčku `game.game_v2a`. Přehled výpisů programů v této kapitole je v souborech s názvem `m20_constants`. Architekturu jsme nijak výrazně neměnili, takže diagram tříd nezobrazuji.

Kapitola 21
Primitivní GUI

Co se v kapitole naučíte
V této kapitole si ukážeme, jak je možné upravit architekturu aplikace, aby umožnila nastavit, zda bude komunikace s uživatelem probíhat v příkazovém řádku, anebo v dialogových oknech. Postupně se dozvíte základní informace o grafické knihovně `tkinter` a naučíte se používat jejích služeb při komunikaci s uživatelem prostřednictvím jednoduchých dialogových oken. Na konci vám pak ukáže, jak vytvořit samostatnou aplikaci, která ke svému spuštění nepotřebuje doprovod okna příkazového řádku.

21.1 Balíček game_v2b

Jak název kapitoly napovídá, chystám se vám ukázat, jak rozšířit naši aplikaci o grafické uživatelské rozhraní, které bývá většinou označované zkratkou GUI. Obdobně jako v předchozích kapitolách i tentokrát vytvořím v balíčku `game` nový podbalíček. Nazvu jej `game_v2b` a zkopíruji do něj obsah předchozího balíčku `game_v2a`. Kromě toho v něm vytvořím nový, prozatím prázdný modul `gui_0`. V něm budeme později definovat funkce potřebné pro realizaci plánovaného GUI.

21.2 Změna architektury

Doposud funkce komunikovaly s uživatelem jenom prostřednictvím standardního vstupu a výstupu. Budeme-li chtít rozšířit naši aplikaci o možnost komunikace prostřednictvím GUI, budeme muset umožnit, aby bylo možné vyměnit objekty, jejichž prostřednictvím komunikujeme.

Jednou z možností je definovat proměnnou, v níž bude uložen odkaz na objekt, jehož prostřednictvím s uživatelem komunikujeme. Doposud jsme využívali zabudované funkce, ale nyní budeme muset vše zabalit do speciálního objektu, který bude mít tuto komunikaci na starosti.

Třída Console

Jednou z možností je definovat v modulu `game` třídu nazvanou např. `Console` a v ní definovat metody, jejichž zavoláním bychom mohli dosáhnout obdobné funkce. Rozhodl jsem se definovat tři metody:

- **`get_command(answer: str) -> str`**
  Funkce realizující příkazy na řádcích **6** a **11** ve výpisu 19.4 na straně 227, tj. zobrazení odpovědi hry a převzetí dalšího příkazu.
- **`show_message(message: str) -> None`**
  Předchozí funkce vždy očekává další příkaz. Po ukončení hry se ale už žádný příkaz nezadává. Proto je potřeba ještě funkce, která vytiskne závěrečnou zprávu, na níž už uživatel neodpovídá, a to má na starosti funkce `show_message()`.
- **`ask_question(question: str) -> bool`**
  Poslední funkce má na starosti získání odpovědi na dotaz, zda si uživatel přeje hrát ještě jednou, realizovaný doposud příkazy na řádcích **5-10** ve výpisu 19.5 na straně 228.

Definici třídy `Console` si můžete prohlédnout ve výpisu 21.1. Jak vidíte, metody jsou definovány jako statické (použil jsem u nich dekorátor `@staticmethod` – viz pasáž *Dekorátory* na straně 72), protože vše zvládne objekt třídy sám a žádné instance k tomu nepotřebuje.

Výpis 21.1: *Definice třídy `Console` v modulu `game` v balíčku `game_v2b`*

```
1  class Console:
2      """Definuje základní funkce pro standardní vstup a výstup."""
3
4      @staticmethod
5      def get_command(answer: str) -> str:
6          """Zobrazí uživateli zadanou odpověď hry,
7          vyzve jej k zadání dalšího příkazu a vrátí zadaný příkaz.
8          """
9          return input(f'answer\n----------\n{GM.enter_cmd} ')
10
11     @staticmethod
12     def show_message(message: str) -> None:
13         """Zobrazí uživateli zadanou zprávu."""
14         print(message)
15
16     @staticmethod
17     def ask_question(question: str) -> bool:
18         """Položí uživateli zadanou otázku a čeká na odpověď ANO/NE."""
19         while True:
20             answer = input(question).strip()
21             if (len(answer) > 0)   and   (answer[0] in '01ANan'):
22                 break       # ---------->
23             print(GM.multi_wrong)
24         return answer[0] in '1Aa'
```

Atribut io

V modulu game definujeme atribut io, který bude obsahovat odkaz na objekt, jenž zprostředkuje komunikaci s uživatelem a který inicializujeme odkazem na třídu Console. Ve funkci main() (viz výpis 19.11 na straně 233) pak za sekci rozhodující o výpisu nápovědy k celé aplikaci (ve výpisu řádky 5-7) přidáme sekci z výpisu 21.2. V té zjistíme, byl-li zadán argument -d (zkratka z *dialog*), a pokud ano, tak do atributu io vložíme odkaz na modul gui_0, v němž požadované funkce naprogramujeme tak, aby využívaly výše zmíněná dialogová okna.

Výpis 21.2: *Upravené definice funkcí run a multirun v modulu game v balíčku game_v2b*

```
1 if '-d' in args:
2     global io
3     from . import gui_0
4     io = gui_0
```

Úprava funkcí run() a multirun()

Zavedení objektu, který bude mít na starosti zprostředkování vstupu a výstupu, si vyžádá drobnou úpravu funkce run() definované ve výpisu 19.4 na straně 227 a multirun() definované ve výpisu 19.5 na straně 228.

Ve funkci run() je třeba odlišit předání odpovědi hry, po němž očekáváme zadání příkazu, a čisté předání závěrečné odpovědi. Novou podobu definice si můžete prohlédnout ve výpisu 21.3 na řádcích 4-12.

Ve funkci multiru() je třeba změnit realizaci dotazu na opakované spuštění hry a reakce na obdrženou odpověď. Novou podobu této funkce najdete v témže výpisu na řádcích 14-22.

21.3 Knihovna tkinter

Aplikace je tedy připravena na plánované rozšíření. Pojďme se nyní podívat na to, jaké nástroje nám k tomu *Python* nabízí.

Python definuje ve standardní knihovně balíček tkinter obsahující sadu modulů definujících základní nástroje pro vytváření GUI. Protože se jedná o relativně ucelenou sadu, bývá často označován jako knihovna. O tomto balíčku budu tedy v dalším výkladu hovořit i já.

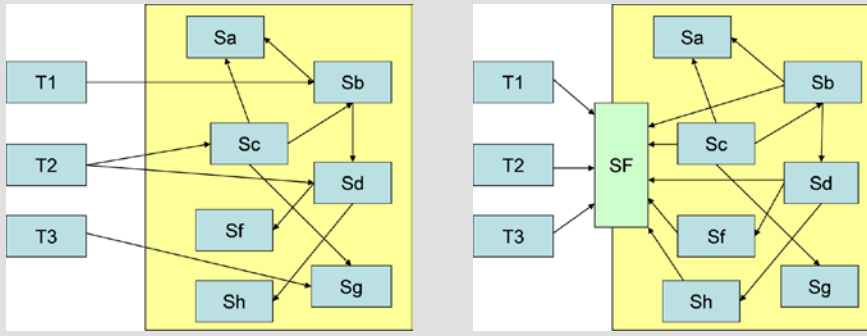
Knihovna tkinter představuje v *Pythonu* de facto standard pro tvorbu GUI. Časem vznikla řada dalších knihoven a frameworků, které jsou možná z toho či onoho pohledu výhodnější, ale hlavní výhodou knihovny tkinter je to, že je integrální součástí standardní knihovny, takže ji najdete prakticky všude, kde je instalovaný *Python*. Za pomoci této knihovny je naprogramováno i prostředí IDLE, které zde používám, anebo želví grafika, s níž jsme se setkali v podkapitole 5.2 *Příkaz import* na straně 80.

Výpis 21.3: *Upravené definice funkcí `run` a `multirun` v modulu `game` v balíčku `game_v2b`*

```
"""Objekt zprostředkovávající vstup a výstup."""
io = Console

def run() -> None:
    """Spustí hru ovládanou z příkazového řádku."""
    command = ''
    while True:     # Nekonečný cyklus hraní hry
        answer = execute_command(command)
        if not actions.is_active:
            io.show_message(answer)
            return    # Hra je ukončena => vyskakujeme z cyklu ==========>
        command = io.get_command(answer)

def multirun():
    """Umožní opakované spuštění hry ovládané z příkazového řádku."""
    while True:
        run()
        answer = io.ask_question(GM.multi_more)
        if not answer:      # Už nechce pokračovat
            print(f'\n{GM.multi_no}\n')
            return          # ==========>
        print(f'\n{GM.multi_yes}\n')
```

Návrhový vzor Fasáda

Návrhový vzor *Fasáda* použijeme ve chvíli, kdy nějaký systém začíná být pro své uživatele příliš složitý vzhledem k oblasti úloh, které chtějí s jeho pomocí řešit. Doporučuje nahradit sadu rozhraní jednotlivých subsystémů sjednoceným rozhraním zastupujícím celý systém. Definuje tak rozhraní vyšší úrovně, které usnadní využívání podsystémů. Cílem je zjednodušit rozhraní celého systému a snížit počet tříd, s nimiž musí uživatel přímo či nepřímo komunikovat.



Používá se např. u grafických knihoven, kdy uživatelům, kteří nepotřebují vytvářet složitě strukturovaná okna, nabídne sadu funkcí vyvolávajících předpřipravená jednoúčelová okna.

Některé učebnice a kurzy doporučují používat některou jinou knihovnu (a je jich k dispozici celá řada), ale všechny ostatní knihovny musíte nejprve stáhnout a instalovat. Jejich nadstavbové možnosti a schopnosti proto využijete pouze v případě, kdy potřebujete vytvářet složitě strukturovaná GUI s různými speciálními komponentami. Pro běžná standardní GUI se jejich nasazení nevyplatí.

Nejjednodušší způsob definice GUI je využit předdefinované funkce v knihovně `tkinter`, které fungují jako fasáda (viz podšeděný blok *Návrhový vzor Fasáda*) pro ty, kteří potřebují jenom něco sdělit uživateli, anebo od něj získat nějakou jednoduchou informaci.

21.4 Modalita dialogových oken

Než se ale pustíme do návrhu našeho GUI, měli bychom si nejprve ujasnit pojem modality. Dialogová okna, jejichž prostřednictvím uživatel komunikuje s aplikací, můžeme rozdělit do dvou skupin:

- **Modální okna** zastaví jakoukoliv další komunikaci s aplikací do doby, než je uživatel řádně ukončí. Modální bývají např. okna pro otevření či uložení souboru. Dokud nezadáte, jaký soubor chcete otevřít, resp. uložit, aplikace se s vámi „nebaví“.

  To, že se zastaví komunikace s uživatelem, ale neznamená, že se zastaví běh celé aplikace. Ta může dále běžet včetně případného zobrazování průběžných výsledků. Můžete např. spustit nějakou animaci a otevřít dialogové okno s výzvou: *„Až se pokocháš, stiskni OK!“*. Uživatel si může například prohlížet běžící animaci a stiskem OK pak celou aplikaci zavřít.
- **Nemodální okna** sice vyskočí, ale aplikace s uživatelem dále komunikuje. Nemodální bývají v řadě aplikací okna pro vyhledávání a nahrazování textu. Poté, co zadáte příkaz, který okno otevře, můžete odskočit do aplikace, tam např. vybrat do schránky text, který chcete vyhledat, vrátit se do okna, a tam jej zadat. Nemodální okno může zůstat otevřené a vy můžete dále s aplikací pracovat.

21.5 Primitivní dialogová okna

Řekli jsme si, že nejjednodušší způsob definice GUI je využit předdefinované funkce v knihovně `tkinter`, jež fungují jako fasáda pro ty, kdo potřebují jenom něco sdělit uživateli, anebo od něj získat nějakou jednoduchou informaci. Tyto funkce jsou ve dvou modulech:

Parametr **options

Všechny metody, které budu dále uvádět, mají dvojhvězdičkový parametr options deklarující, že zde můžete nastavit hodnoty některých parametrů. Teoreticky bych je nemusel uvádět, protože je nebudeme používat. V dokumentaci ale uveden je, i když tam není uvedeno, jaké konkrétní parametry zde můžete inicializovat – to si musíte zjistit jinde. Nechtěl jsem ale, abyste mne později osočovali, že vám předávám zkreslené informace.

Modul tkinter.messagebox

Modul messagebox nabízí dvě sady funkcí otevírajících modální okna. První sada obsahuje funkce:

```
showinfo   (title=None, message=None, **options)
showwarning(title=None, message=None, **options)
showerror  (title=None, message=None, **options)
```

Ty otevřou okno se zprávou zadanou v parametru message, v jehož titulkové liště zobrazí text v parametru title. Poté počkají, až přečtení zprávy potvrdíte stiskem **OK**, a okno zavřou. Okna otevíraná jednotlivými funkcemi z této sady se liší pouze ikonou charakterizující typ zprávy.

Funkce druhé sady po vás chtějí odpověď na položenou otázku a liší se tím, která tlačítka zobrazí. Jejich parametry jsou shodné s parametry funkcí z první sady, tak je uvedu pouze jmény:

- Funkce **askquestion** otevře v českých *Windows* okno s tlačítky **Ano** a **Ne**, po jejich stisku vrátí string 'yes' nebo 'no'.
- Funkce **askokcancel** otevře v českých *Windows* okno s tlačítky **OK** a **Zrušit**, po jejich stisku vrátí hodnoty True nebo False.
- Funkce **askretrycancel** otevře v českých *Windows* okno s tlačítky **Opakovat** a **Zrušit**, po jejich stisku vrátí hodnoty True nebo False.
- Funkce **askyesno** otevře v českých *Windows* okno s tlačítky **Ano** a **Ne**, po jejich stisku vrátí hodnoty True nebo False.
- Funkce **askyesnocancel** je z nich nejrafinovanější. V českých *Windows* otevře okno s tlačítky **Ano**, **Ne** a **Zrušit**, po jejich stisku vrátí hodnoty True, False nebo None. Toto okno má na rozdíl od předchozích dokonce v záhlaví i zavírací tlačítko, po jehož stisku vrátí (stejně jako po stisku tlačítka **Zrušit**) hodnotu None.

Parametr **options

Všechny uvedené metody mají dvojhvězdičkový parametr options, o němž jsem se prozatím nezmiňoval. Jak jsme si řekli v pasáži *Dvouhvězdičkový parametr* na straně 125, tento parametr umožňuje inicializovat řadu proměnných. Jenom musíme vědět, jak se jmenují. V naší primitivní úloze by mělo smysl maximálně nastavovat parametr

`parent`, jímž se zadává odkaz na rodičovské okno, o kterém budeme hovořit za chvíli v pasáži *21.6 Rodičovské okno*, ale ani to nebude potřeba.

Modul `tkinter.simpledialog`

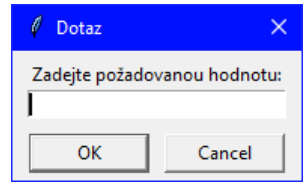
Modul `simpledialog` definuje funkce, jež se vás zeptají na konkrétní hodnotu, kterou pak vrátí. Otevřou dialogové okno s textem parametru `title` v záhlaví. V okně bude výzva v parametru `prompt` a pod ní vstupní textové pole, kam má uživatel zapsat svoji odpověď.

Předpokládám, že si typ zjišťované hodnoty odvodíte z názvu funkce, takže opět uvádím jen signatury:

```
askfloat  (title, prompt, **kw)
askinteger(title, prompt, **kw)
askstring (title, prompt, **kw)
```

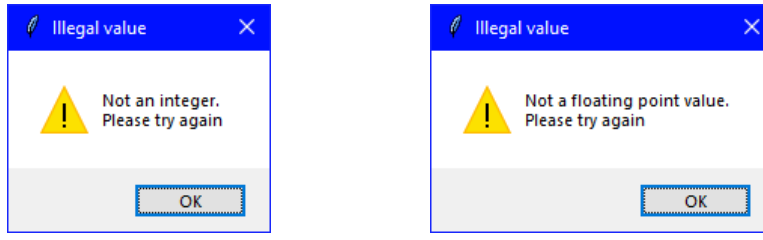
Pod vstupním textovým polem jsou tlačítka **OK** a **Cancel** – viz obrázek 21.1 ukazující reakci systému na dvojici příkazů (nezkoušejte to hned, za chvíli se dozvíte, že ještě musí něco předcházet):

```
>>> from tkinter import simpledialog as sd
>>> sd.askstring('Dotaz', 'Zadejte požadovanou hodnotu:')
```



Obrázek 21.1:
Dialogové okno s výzvou k zadání údajů

Zavře-li uživatel okno nebo stiskne tlačítko **Cancel**, funkce vrátí hodnotu `None`. Zadá-li uživatel při zadávání čísel nepřevoditelný text nebo nechá-li pole prázdné, systém jej na to upozorní dialogovým oknem z obrázku 21.2 a po jeho potvrzení požaduje nové zadání. Funkce `askstring()` samozřejmě proti prázdnému poli neprotestuje a vrátí prázdný string.



Obrázek 21.2:
Upozornění na chybné zadání číselné hodnoty

21.6 Rodičovské okno

Zatím to vypadá jednoduše, ale úplně jednoduché to zase není. Výše uvedené funkce totiž vyžadují, aby před jejich spuštěním již bylo otevřeno nějaké okno, které bude považováno za jejich rodičovské okno. To vytvoříte např. příkazy

```
>>> import tkinter
>>> root = tkinter.Tk()
```

Jakmile máme okno vytvořené, můžeme zadávat výše uvedené příkazy a začnou se otevírat jejich okna. Pokud totiž předchozím funkcím nenastavíme rodičovské okno vytvářených dialogů, použijí to existující.

Schování okna

To, že při otevírání dialogového okna musí existovat nějaké rodičovské okno, ještě neznamená, že toto okno musí být vidět. Vy je můžete vytvořit, umístit a pak odebrat (i když ve skutečnosti se spíš jen „zneviditelní“).

Tohoto „zneviditelnění“ budoucího rodičovského okna bychom dosáhli zadáním příkazu

```
>>> root.withdraw()
```

21.7 Modul dialogových oken

Nyní už bychom měli vědět vše, abychom mohli naprogramovat modul `gui_0`, který bude obsahovat nultou verzi grafického uživatelského rozhraní. Definici tohoto modulu s vynechanými kontrolními tisky najdete ve výpisu 21.4.

Jak jsem naznačil, v originálním modulu najdete kontrolní tisky umístěné vždy před příkazem otevírajícím dialogové okno a za ním. Pomocí nich můžete ověřit chování volaných funkcí.

V definici bych jen upozornil na řádek 15, v němž je definována reakce na zavření dialogového okna bez zadání odpovědi. Příkaz nahradí hodnotu `None` prázdným stringem, protože na hodnotu `None` naše hra reagovat neumí a zbytečně by vyhodila výjimku.

21.8 Přímé spuštění aplikace

Vyzkoušejte si chod programu jak při spuštění ze systémového okna IDLE, tak přímo z příkazového řádku. Nezapomeňte, že pro spuštění okenního režimu je třeba zadat argument `-d`.

Takovéto spouštění má ale jednu nepříjemnou vlastnost: otevírá se při něm okno příkazového řádku. Vy byste přitom byli určitě rádi, kdybyste mohli spouštět svoji „okenní“ aplikaci bez této zátěže.

Výpis 21.4: *Definice modulu `gui_0` v balíčku `game_v2b`*

```
"""
Sada funkcí umožňujících použít GUI v aplikaci
navržené původně pro příkazový řádek.
"""

import tkinter
from   tkinter import messagebox   as mb, \
                      simpledialog as sd

def get_command(answer: str) -> str:
    """Zobrazí uživateli zadanou odpověď hry,
    vyzve jej k zadání dalšího příkazu a vrátí zadaný příkaz.
    """
    command = sd.askstring('Zadání příkazu', answer)
    if command == None: command = ''
    return command


def show_message(message: str) -> None:
    """Zobrazí uživateli zadanou zprávu."""
    mb.showinfo('Poděkování', message)


def ask_question(message: str) -> bool:
    """Položí uživateli zadanou otázku a čeká na odpověď ANO/NE."""
    answer = mb.askyesno('Dotaz', message)
    return answer


################################################################################

parent = tkinter.Tk()           # Vytvoří rodičovské okno
parent.withdraw()               # a zhasne je
```

Jednou z možností je definovat v kořenovém balíčku vlastní verzi modulu `__main__` balíčku – např. podle výpisu 21.5 (modul opět předpokládá, že balíček `game_v2b` je umístěn přímo v kořenovém balíčku). Nezapomeňte do kořenového balíčku vytvářené aplikace zkopírovat také modul `dbg`, s jehož přítomností nyní naše moduly počítají.

Výpis 21.5: *Definice modulu `__main__` v kořenovém balíčku vytvářené aplikace*

```
# -*- coding: utf-8 -*-

import sys
sys.argv.append('-d')

import game_v2b.game as game
game.main()
```

Jak jste si jistě všimli, tato definice nepožaduje pro spuštění režimu s dialogovými okny zadání argumentu `-d`, protože jej mezi argumenty příkazového řádku dodá sama. Když vytvoříte vlastní verzi modulu `__main__`, nebudete již po aplikaci `zipapp` požadovat jeho vytvoření a nebudete jí proto zadávat argument `-m` následovaný odkazem na hlavní metodu aplikace.

Aplikace sice spouští dialogový režim bez nutnosti zadat argument `-d`, ale to lze napravit. Stačí změnit přípony vytvořeného souboru z `pyz` na `pyw`. Při správné instalaci *Pythonu* se aplikace s touto příponou spouští bez otevírání příkazového okna.

21.9 Shrnutí

Zdrojové kódy odpovídající výše popsanému stavu vývoje aplikace najdete v doprovodných programech v balíčku `game.game_v2b`. Přehled výpisů programů v této kapitole je v souborech s názvem `m21_dialogs`. Architekturu jsme nijak výrazně neměnili, takže diagram tříd nezobrazuji.

Kapitola 22
Kudy dál

Co se v kapitole naučíte
Tato kapitola obsahuje pouze souhrn námětů pro další vylepšování aplikace pro ty, kdo by chtěli navrhnout vlastní, lepší konverzační hru. Pak přidá několik námětů pro učitele a na závěr pár adres, kde byste se mohli o *Pythonu* dozvědět leccos zajímavého.

22.1 Další vylepšování

Před dalšími náměty na rozšíření bych chtěl upozornit, že cílem výkladu nebylo vyvinout hru, ale na příkladu vývoje hry vám předat základní programátorské znalosti a dovednosti. Následující poznámky jsou proto míněny jako náměty těm, kteří chtějí tuto aplikaci využít k dalšímu zlepšování svých dovedností. Budou proto spíše jen náznakové.

Následující tipy a náměty navíc vnímejte i tak, že se na nich ukáže, jak lze vhodným návrhem architektury připravit aplikaci tak, aby se maximálně usnadnilo její další rozšiřování a vylepšování. Berte přitom naši hru jako reprezentanta obecné aplikace.

22.2 Přehled námětů

Hra, kterou jsme doposud vytvořili, je velmi prostoduchá. Nicméně vytvořený program by mohl být základem her mnohem dokonalejších. Současná koncepce nám umožňuje využít program pro definici zcela jiné hry. Pojďme si projít některé náměty.

Převod pod kvalitní grafické uživatelské rozhraní

Po primitivním GUI z minulé kapitoly je to jistě to první, co vás napadne. Nicméně popis tohoto tématu by spolu s výkladem potřebných programátorských obratů vydal sám o sobě na samostatnou publikaci. Na toto místo se už potřebný výklad nevejde.

Změna světa hry

Veškerá definice světa hry je založena na zadání skupin textových řetězců a jejich uložení v nezveřejňovaném atributu `_NAME_2_PLACE` v modulu `world`.

Mohli bychom například doplnit v modulu `world` funkci `set_world`, která by tento atribut inicializovala a současně by mohla inicializovat i batoh, tj. nastavit jeho kapacitu a počáteční obsah. Při tomto nastavování by mohla kontrolovat, zda je vše zadáváno korektně, a včas upozorňovat na případné chyby.

Mezi prostory mohou být takové, které na počátku nemají daný prostor za svého souseda. Sousedem se stane, až otevřete spojovací dveře, které budou v aktuálním prostoru prezentovány jako h-objekt. Teprve po otevření či odemčení dveří se potenciální sousední prostor stane skutečným sousedem, do kterého lze přejít standardním příkazem pro přesun.

Studenti jsou nápaditi. Setkal jsem se i s hrou, kde se dveře otevřely až poté, co se v jiném prostoru nanosila do nějaké kádě voda, a ta svou vahou posunula západku, která bránila průchodu do daného sousedního prostoru.

Zdokonalení h-objektů

Doposud jsme v prefixu názvu h-objektů pouze oznamovali, zda daný objekt je či není přenositelný. Mohli bychom ale zadat i jemnější dělení. Budou-li např. náš batoh představovat ruce, mohli bychom rozdělit předměty na ty, k jejichž přenesení stačí jedna ruka, a na ty těžší, k jejichž přenesení jsou potřeba obě ruce.

Vhodným nastavením prefixu můžete zadat i další charakteristiky, např. že se jedná o alkoholický nápoj, který proto smějí konzumovat pouze dospělé osoby.

H-objekty – prostory

Mezi h-objekty mohou být i takové, které jsou současně prostory, do nichž se musíte dostat speciálním příkazem. V aktuálním prostoru může být např. truhla, do níž se dostanete zadáním příkazu `otevři`. Pak se truhla stane aktuálním prostorem, z nějž můžete h-objekty odebírat, nebo je do něj naopak ukládat.

Hra může také vyžadovat, abyste k otevření truhly použili klíč, který musíte nejprve najít a případně se jej proti nějakému odporu zmocnit.

Rozšiřování sady příkazů

Tím se dostáváme k rozšiřování sady příkazů. V tomto případě je nejvhodnější definovat každou novou akci jako potomka abstraktní třídy `_AAction` z modulu `actions`. Nebylo by ale vhodné doplňovat další definice přímo do modulu `actions`, lepší je pro ně vyhradit nějaký samostatný rozšiřující modul.

Jako součást akcí je často vhodné definovat nějaké příznaky, které informují o splnění určitých podmínek, např. jestli je již truhla odemčená, abychom ji mohli otevřít, jestli je hlídač podplacený, aby nás nechal utéct apod.

Rozhovor

Speciálním druhem akce je taková, při níž zapředeme rozhovor s nějakým h-objektem – osobou či věcí, od níž něco potřebujeme. Jednotlivé věty rozhovoru nejsou příkazy, ale na druhou stranu rozhovor může být prokládán příkazy. Můžeme např. oslovit kouzelného dědečka, od něj se dozvíme, že má hlad, my mu dáme buchtu, on nám dá píšťalku, kterou uspíme psa, jenž hlídá princeznu, apod.

22.3 Tipy pro učitele

Zde na chvíli odbočím od výkladu programování. Tvorba vlastní konverzační hry může být docela dobrý závěrečný test při výuce programování. Její základní výhoda tkví v tom, že můžeme na počátku po studentech chtít, aby každý definoval svůj vlastní scénář, v němž bude muset rozšířit základní modul o několik vlastních příkazů. Jednotlivé scénáře se musejí dostatečně lišit.

Takovéto zadání je výhodné v tom, že jako závěrečnou práci dělají z jistého pohledu všichni totéž, ale přeci jenom každý něco jiného.

Navíc je možné poměrně jednoduše kontrolovat výsledek. K tomu je vhodné rozšířit definici instancí třídy `Step` o atribut, v němž lze definovat typ daného kroku a podle něj i zkontrolovat, zda byl daný krok správně definován – např. zda se při reakci na příkaz pro zvednutí předmětu nezměnil aktuální prostor apod.

Kromě toho lze poměrně snadno definovat testovací program, který prověří, zda scénáře studentů odpovídají zadaným požadavkům: minimální požadovaný počet prostorů, rozšiřujících příkazů, kroků úspěšného scénáře, zda jsou v chybovém scénáři otestována všechna možná chybná zadání uživatele apod.

22.4 Další ukázkové příklady

Chystám se do *Pythonu* převést hru, kterou jsem používal jako demo ve svých kurzech *Javy*. Až na to najdu čas, tak bych ji zveřejnil na stránkách knihy jako samostatný doprovodný program. Ten by měl obsahovat i rozšíření, o nichž jsem před chvílí hovořil v poznámkách pro učitele.

Kromě toho se na stránkách knihy chystám uvádět i různé zajímavé náměty studentů jako inspiraci pro ty, kdo by chtěli vytvořit vlastní konverzační hru s nestandardním námětem.

22.5 Další zdroje

Paralelně vedle této knihy vydalo nakladatelství *Grada Publishing* další dvě publikace na blízká témata: příručku *Python – Kompletní příručka jazyka pro verzi 3.9*, kterou jsem napsal já, a knihu *Python – práce s knihovnami* s přehledem nejdůležitějších knihoven a možností jejich využití, kterou napsal Slavoj Písek. Jestli se v záplavě požadavků na další knihy objeví prostor, chtěl bych napsat ještě samostatnou příručku o tvorbě GUI pomocí integrované knihovny *Tkinter*, s níž jsem vás v minulé kapitole seznámil, a knihu o návrhových vzorech v jazyce *Python*, kterou jsem sliboval v kapitole o OOP.

Vedle knih je k dispozici i nepřeberná řada webů zaměřených na jazyk *Python*, mezi nimi pak i takové, které se s vámi baví česky či slovensky. Z nich bych jmenoval následující (neuvádím stránky, které nabízejí pouze kurzy):

- https://python.cz/,
- https://python.sk/,
- https://www.py.cz/FrontPage,
- https://www.root.cz/n/python/,
- https://www.itnetwork.cz/python,
- https://pyvec.org/.

Určitě nejsou všechny – ve výčtu najdete jen ty, o něž jsem na svých toulkách zakopl.

Část E
Přílohy

Přílohy obsahují některé pomocné texty. První seznamuje uživatele *Windows* s některými možnostmi instalace doprovodných programů, druhá obsahuje souhrn informací o funkcích a metodách standardní knihovny, které jsou použity v doprovodných programech.

Příloha A Konfigurace ve Windows

Co se v kapitole naučíte
Tato příloha vás seznámí s některými operacemi, které je vhodné provést při instalaci doprovodných programů v prostředí *Windows*.

A.1 Definice substituovaných disků

V operačním systému *Windows* můžete používat 26 logických disků – pro každé písmeno abecedy jeden. Většina uživatelů však používá pouze zlomek tohoto počtu. *Windows* umožňují použít volná písmena pro tzv. **substituované disky**, což jsou složky, které se rozhodnete vydávat za logický disk. Protože o této možnosti většina uživatelů neví, a přitom je to funkce velice užitečná, ukážu vám, jak ji můžete využít.

Substituované disky se definují pomocí příkazu:

```
SUBST název_disku substituovaná_složka
```

Nejjednodušší způsob, jak definovat ve *Windows* např. substituovaný disk `P:`, je vložit do složky, kterou budete chtít substituovat jako disk `P:`, dávkový soubor s příkazem k substituci. (Písmeno `P` se pro *Python* hodí nejlépe, ale můžete si vybrat jakékoliv jiné, které je na vašem počítači volné.)

Pokud jste ještě nepracovali s dávkovými soubory, tak věřte, že to jsou obyčejné textové soubory, do nichž zapisujete příkazy pro operační systém. Jejich název může být libovolný, ale musí mít příponu bat (zkratka ze slova batch – dávka). V dávkovém souboru budou následující příkazy (na velikosti písmen nezáleží):

```
SUBST P: /d
SUBST P: .
```

První příkaz má za úkol zrušit případnou doposud nastavenou substituci disku `P:` (není-li v daném okamžiku označený disk substituován, systém vypíše chybovou zprávu, ale jinak se nic nestane), druhý příkaz pak substituuje aktuální složku jako disk `P:`.

Soubor umístíte do složky, z níž budete chtít udělat substituovaný disk. Kdykoliv pak tento dávkový soubor spustíte, dávka substituuje složku, v níž je umístěna, jako příslušný disk. Dávka se přitom spouští obdobně jako aplikace – např. poklepáním na ikonu jejího souboru v *Průzkumníku*.

Kdykoliv od této chvíle budete pracovat s diskem `P:`, budete ve skutečnosti pracovat s obsahem substituované složky. A naopak: cokoliv uděláte s obsahem substituované složky, uděláte zároveň s obsahem disku `P:`.

Budete-li chtít mít danou substituci nastavenu trvale, můžete umístit zástupce dávkového souboru do položky Po spuštění ve startovní nabídce. Protože je v každé verzi operačního systému jinde, bude nejlepší, když ji ve startovní nabídce najdete, klepnete na ni pravým tlačítkem a v následně otevřené místní nabídce zadáte **Otevřít**. Tím otevřete okno průzkumníka s touto složkou. Pak v druhém okně průzkumníka otevřete složku s příslušným dávkovým souborem, uchopíte jeho ikonu PRAVÝM tlačítkem myši, přesunete ji do složky nabídky a pustíte. Otevře se místní nabídka (ta se otevře, pouze pokud přesouváte soubor pravým tlačítkem myši), ve které zadáte, že zde chcete vytvořit zástupce, a tím celý proces končí.

Abyste mohli složku substituovat jako nějaký disk, nesmí váš operační systém používat disk označený tímto písmenem. Písmeno může být použito nejvýše pro jiný substituovaný disk, protože tuto substituci můžete před nastavením nové substituce zrušit (pro tento případ je v dávkovém souboru první příkaz s parametrem `/d` – delete).

Používáte-li operační systém *Windows*, můžete urychlit budoucí vyhledání složky s projekty právě tím, že pro ni zřídíte zvláštní substituovaný disk. Kdykoliv se pak obrátíte na příslušný disk, obrátíte se ve skutečnosti k příslušné složce. Pomocí substituce si tak můžete zkrátit cestu k často používaným složkám.

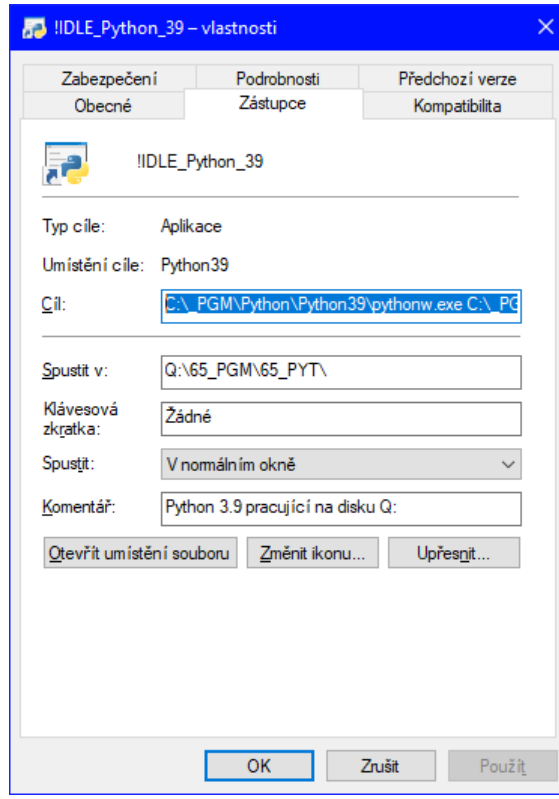
A.2 Nastavování zástupce spouštějícího IDLE

Používáte-li *Windows* a nemáte-li zkušenosti s nastavováním zástupců, doporučuji využít mého zástupce a pouze mu upravit některá nastavení. Postupujte následovně:

1. Otevřete ve svém oblíbeném správci souborů (budu předpokládat, že jím je *Průzkumník* nebo *Total Commander*) složku s doprovodnými programy `65_PYT`.
2. Klepněte na soubor `!IDLE_Python_39.lnk` (používáte-li *Průzkumník*, ten příponu zástupců, tj. `lnk`, nezobrazuje).
3. Stiskněte ALT+ENTER. Otevře se okno z obrázku A.1.
4. V poli **Cíl** je zadáno:

```
C:\_PGM\Python\Python39\pythonw.exe
C:\_PGM\Python\Python39\Lib\idlelib\idle.pyw
```

   První část končící `pythonw.exe` je úplná cesta k programu spouštějícímu *Python*, který nebude otevírat okno příkazového řádku, protože očekává, že spouštěný skript používá GUI.



Obrázek A.1:
Okno zástupce spouštějícího IDLE

Tato část je nepovinná. Máte-li korektně instalovanou pouze jednu verzi *Pythonu*, nemusíte ji vůbec zadávat, protože *Windows* podle přípony spouštěného skriptu poznají, co mají spustit. Já ji zadávat musím, protože mám v době psaní knihy instalovanou vedle stabilního *Pythonu 3.8.3* také beta verzi *Pythonu 3.9*, kterou v této učebnici používám, a potřebuji si být jistý, která verze daný skript spouští.

Potřebujete-li proto (stejně jako já) také zadávat spouštěný program, musíte zde zadat úplnou cestu k tomuto programu na vašem počítači.

5. Druhá část příkazu je již povinná a zadává úplnou cestu ke spouštěnému skriptu, tj. k souboru `idle.pyw`. Vy budete mít spouštěný skript nejspíš jinde, tak zde musíte zadanou cestu příslušně upravit.
6. V poli **Spustit v** je zadána cesta ke složce se zdrojovými soubory doprovodných skriptů. Tu si také upravte podle svého počítače.
7. Když máte vše korektně upraveno, uložte upraveného zástupce stiskem **OK**, a od této chvíle můžete IDLE spouštět poklepáním na daného zástupce.

Příloha B
Použité funkce ze standardní knihovny

B.1 Zabudované funkce

abs(x)
Vrátí absolutní hodnotu zadaného čísla (řádky 1 až 4).

```
>>> abs(-3.14)
3.14
>>> abs(1+1j)      #Absolutní hodnotou komplexního čísla je jeho velikost
1.4142135623730951
```

bool(x)
Vrátí logickou hodnotu, na kterou se daná hodnota v případě potřeby převede – viz např. výpis 10.1 na straně 140.

dict(x)
Vytvoří slovník z prvků dodaných generátorem zadaným v argumentu. Podrobněji viz *Slovník – dict* na straně 121.

eval(výraz[, globals[, locals]])
Vyhodnotí zadaný výraz a vrátí spočtenou hodnotu. Při vyhodnocení může používat zadané globální a lokální proměnné, které však musejí být zadány prostřednictvím slovníku.

help([objekt])
Funkce slouží k získání nápovědy. Protože je ale výklad rozsáhlejší, je mu věnována samostatná podkapitola *Získání nápovědy – dokumentace* na straně 56.

input([výzva])
Může být volána buď bez argumentů, anebo s argumentem představujícím text výzvy. Vypíše na standardní výstup zadanou výzvu (nebyla-li zadána, nevypíše nic) a čeká, až za ní uživatel zapíše to, k čemu byl vyzván, a stiskne ENTER. Vrátí uživatelem zadaný text.

```
>>> input("Zadej číslo: ")
Zadej číslo: 123
'123'
```

len(x)
Vrátí počet prvků zadaného kontejneru.

list(x)
Vytvoří seznam z prvků dodaných generátorem zadaným v argumentu. Podrobněji viz *Seznam – list* na straně 118.

range(stop)
range(start, stop[, step])
Očekává celočíselné argumenty a vrátí objekt, který se používá jako zdroj posloupnosti celých čísel začínající číslem `start`, pokračující čísly zvětšujícími se postupně o `step` a končící posledním číslem menším (při záporné hodnotě `step` větším) než `stop`.

Není-li zadán argument `step`, rostou čísla po jedné. Volání `range(stop)` je ekvivalentní volání `range(0, stop)`, respektive `range(0, stop, 1)`.

print(argumenty)
Očekává seznam čárkami oddělených výrazů, jejichž hodnoty má vytisknout na standardní výstup. Podrobnosti viz *3.6 Funkce print() a její parametry* na straně 61.

reload([modul]); přesněji importlib.reload([modul])
Znovu načte zadaný modul. Podrobněji viz *5.6 Opětovné načtení opraveného modulu* na straně 87.

set(x)
Vytvoří množinu prvků dodaných generátorem zadaným v argumentu. Podrobněji viz *Množiny – set, frozenset* na straně 120.

str([object])
Převede zadaný argument na text. Není-li zadán argument, vrátí prázdný text.

super([type[, object-or-type]])
Vrátí odkaz na správného rodiče – adresáta zprávy, kterou kvalifikuje volání této funkce. Ve většině případů si překladač dokáže své argumenty domyslet. Pokud ne, tak první argument představuje typ, jehož předka hledáme, druhý argument pak instanci, pro níž daný atribut hledáme.

`tuple(x)`
Vytvoří n-tici prvků dodaných generátorem zadaným v argumentu. Podrobněji viz *N-tice – tuple* na straně 119.

`type(object)`
Vrátí typ zadaného argumentu.

B.2 Speciální metody

`__init__()`
Initor instance zodpovídající za její inicializaci.

`__repr__()`
Vrátí systémový podpis své instance. Podrobnosti viz *9.1 Metody `__repr__()` a `__str__()`* na straně 128.

`__str__()`
Vrátí uživatelský podpis své instance. Podrobnosti viz *9.1 Metody `__repr__()` a `__str__()`* na straně 128.

B.3 Metody třídy `dir`

`fromkeys(iterable[, value])`
Vytvoří slovník se zadanými klíči a jednotně inicializovanými hodnotami. Podrobnější výklad najdete v podšeděném bloku *Vytváření slovníků pomocí metody fromkeys()* na straně 186.

B.4 Metody posloupností – Sequence

Z probraných datových typů patří mezi posloupnosti instance tříd `list`, `tupla` a `str`, tj. seznamy, n-tice a stringy.

`index(x [, i [, j ]] )`
Vrátí index prvního výskytu argumentu `x` v zadané posloupnosti. Je-li zadán argument `i`, začne se hledat od tohoto indexu, je-li zadán argument `j`, přestane se hledat před tímto indexem. Pokud hledanou položku v zadané oblasti nenajde, vyhodí výjimku `ValueError`.

B.5 Metody třídy `list`

`append(x)`
Přidá na konec seznamu svůj argument.

`sort(*, key=None, reverse=False)`
Seřadí prvky v seznamu podle jejich velikosti. Prvky ale musejí být vzájemně porovnatelné. Příklad použití je na řádku `4` ve výpisu 12.2 na straně 166.

B.6 Metody třídy `str`

`lower()`
Vrátí kopii svého stringu převedenou na malá písmena podle standardu *Unicode*. Příklad použití je na řádcích `18-19` ve výpisu 11.4 na straně 157.

`split(sep=None, maxsplit=-1)`
Vrátí seznam stringů, z nichž je tvořen oslovený string. Tyto stringy jsou v něm odděleny stringem zadaným v argumentu `sep`. Není-li tento argument zadán, považuje se za oddělovač posloupnost bílých znaků.

Je-li zadán argument `maxsplit`, bude vrácen seznam o nejvýše `maxsplit+1` prvcích. Příklad použití je na řádku `6` ve výpisu 12.5 na straně 169.

`upper()`
Vrátí kopii svého stringu převedenou na velká písmena podle standardu *Unicode*.

`strip([ chars ])`
Vrátí kopii svého stringu s odstraněnými úvodními a závěrečnými znaky vyskytujícími se v zadaném argumentu. Není-li zadán argument, odstraní úvodní a závěrečné bílé znaky. Příklad použití je na řádcích `15-16` ve výpisu 11.4 na straně 157.

Příloha C
Konvence pro psaní programů v Pythonu

Oficiální konvence pro psaní kódu jsou včetně demonstračních ukázek podrobně popsány v dokumentu PEP 8[1] na adrese https://www.python.org/dev/peps/pep-0008/. Konvence pro psaní dokumentačních komentářů jsou v dokumentu PEP 257, který najdete na adrese https://www.python.org/dev/peps/pep-0257/. Zde uvedu jen stručný výčet.

Zájemcům, kteří chtějí rychle ověřit, že nějaký kód vyhovuje těmto konvencím, doporučuji, aby si na adrese https://pep8.readthedocs.io stáhli program nazvaný příhodně `pep8`, jenž dodržování těchto konvencí ověří za vás.

Uspořádání kódu

Hlavní zásadou při tvorbě kódu by měl být fakt, že program se daleko častěji čte, než zapisuje, takže bychom jej měli zapisovat tak, aby se následně dobře četl. Bude-li kód čitelnější, když v daném místě porušíte některou z konvencí, porušte ji. To je obzvláště důležité u programů určených pro výuku.

- Odsazujte o 4 znaky a v textu nepoužívejte tabulátory, ale jen mezery.
- Omezte délku řádků na 79 znaků, u komentářů (i dokumentačních) na 72 znaků.
- U delších řádků vkládejte konec řádku před binární operátor, takže výraz bude na dalším řádku začínat operátorem – např.

```
suma = (první proměnná
        + druhá proměnná)
```

[1] PEP je zkratka z anglického *Python Enhancement Proposal* (doporučení pro vylepšení *Pythonu*). Jsou to dokumenty poskytující informace komunitě *Pythonu* nebo popisující nové funkce, procesy nebo prostředí. Jejich přehled najdete na https://www.python.org/dev/peps/.

- Definice tříd a samostatných funkcí oddělujte dvěma řádky, definice metod uvnitř třídy oddělujte jedním prázdným řádkem.
- Používejte kódování UTF-8, a to bez občas nabízené úvodní deklarace.
- Vkládejte každý import modulu na samostatný řádek, import několika identifikátorů z daného modulu (`from … import …`) je však možné uvést společně.
- Umístěte importy na počátek modulu před definice globálních proměnných.
- Nepoužívejte hvězdičkový import (`from … import *`).
- Identifikátory uvozené a ukončené dvojicí podtržení (tzv. *dunders* jako zkratka z anglického ***d****ouble* ***unders****cores*) by měly být definovány na počátku modulu za dokumentačním komentářem, ale před importy.
- Pro trojité ohraničení stringů používejte trojité uvozovky, abyste byli konzistentní s dokumentačními komentáři podle PEP 257. Jednoduchý ohraničující znak (apostrof či uvozovky) používejte dle svých preferencí, ale buďte konzistentní.
- Mažte závěrečné mezery na koncích řádků. (Některé editory tuto funkci nabízejí.)
- U pojmenovaných argumentů nepoužívejte mezery kolem znaku `=`.
- Nevkládejte více příkazů na jeden řádek.
- U složených příkazů pokračujte na řádku za hlavičkou pouze v případě, že tělo tvoří jeden, jednoduchý příkaz – např.

```
for x in lst: total += x
```

- Nebojte se použít závěrečné čárky v seznamech argumentů, mohou usnadnit přidání dalšího členu – např.

```
FILES = [
    'setup.cfg',
    'tox.ini',
    ]
```

- Komentáře, které neodpovídají kódu, jsou horší než žádné. Udržujte komentáře neustále aktuální (*up-to-date*).

Jmenné konvence

Jmenné konvence bohužel nejsou zcela konzistentní, nicméně existuje několik doporučení, které by měly nové programy dodržovat.

- Ve standardní knihovně musí všechny identifikátory používat pouze znaky `ASCII` a měly by pokud možno používat angličtinu. Doporučuje se toto pravidlo dodržovat i v ostatních programech.
- Názvy viditelné uživateli jako součást API by měly reflektovat spíše užití než implementaci.
- Vyvarujte se názvů tvořených pouze znakem `l` (malé L), `O` (velké o) nebo `I` (velké i). Některé fonty neumí odlišit znaky `1-l-I` (jedna – L – I) a `0-O` (nula – o).

- Používejte ve svém vývojovém prostředí font, který tyto znaky odliší.
- Názvy modulů by měly být krátké a používat jen malá písmena. Použití znaku podtržení se nedoporučuje.
- Názvy tříd by měly používat `VelbloudíNotaci`.
- Názvy funkcí by měly používat jen malá písmena a jednotlivá slova názvu oddělovat podtržítky – např. `long_function_name`.
- První parametr instančních metod by se měl vždy jmenovat `self`.
- První parametr třídních metod by se měl vždy jmenovat `cls`.
- Názvy neveřejných metod a instančních proměnných by měly začínat podtržítkem.
- Názvy konstant by měly používat pouze velká písmena a jednotlivá slova názvu oddělovat podtržítky – např. `LONG_CONSTANT_NAME`.
- U každého atributu (proměnné i metody) se vždy rozmyslete, zda má být veřejný. Neveřejný atribut lze snadno zveřejnit, obrácená operace je po rozšíření dané třídy či modulu již zakázaná.

Dokumentační komentáře (PEP 257)

Dokumentační komentář je stringový literál zadaný jako první příkaz v modulu, třídě, metodě či funkci. Ten se automaticky stane hodnotou atributu `__doc__` daného objektu.

- Dokumentační komentář by měly mít všechny moduly a všechny z něj exportované funkce a třídy.
- Dokumentační komentář balíčku je možné zadat v jeho souboru `__init__.py`.
- Dokumentační komentáře ohraničujte vždy trojitými uvozovkami (`"""`), s případnou předponou `r`, používáte-li v nich zpětná lomítka.
- Trojité uvozovky používejte, i když se komentář vejde na řádek, jak je tomu např. ve výpisu 3.1 na straně 57. Neoddělujte ohraničení mezerami.
- Víceřádkové komentáře zahajte jednořádkovým shrnutím umístěným na stejném řádku s ohraničujícími uvozovkami. Protože může být použito automatickými indexačními programy, je důležité, aby bylo na jednom řádku a aby bylo od dalšího textu odděleno prázdným řádkem. Další řádky komentáře zarovnávejte stejně jako uvozovky na prvním řádku.
- Za dokumentačním komentářem třídy vynechte řádek.
- Dokumentační komentář modulu by měl vypsat všechny třídy, výjimky a funkce daného modulu.
- Dokumentační komentář třídy by měl vypsat její chování, seznam veřejných metod a atributů včetně instančních.
- Dokumentační komentář funkce by měl vypsat její funkci, argumenty a návratovou hodnotu, vedlejší efekty a případná omezení.

Literatura

[1] *Akademický slovník cizích slov*. Praha: Academia, 1995. ISBN 80-200-0497-1.

[2] GAMMA E., HELM R., JOHNSON R., VLISSIDES J.: *Design patterns: elements of reusable object-oriented software.* Boston: Addison-Wesley, 2001. ISBN 0-201-63361-2.

[3] FOWLER M.: *Destilované UML*. Praha: Grada Publishing, 2009. ISBN 80-241-2062-3.

[4] FOWLER M.: *Refactoring: Improving the Design of Existing Code*. Boston: Addison-Wesley Professional, 1999. ISBN 0-201-48567-2.

[5] FOWLER M.: *Refaktoring – Zlepšení existujícího kódu*. Praha: Grada Publishing, 2003. ISBN 80-247-0299-1.

[6] FOWLER M.: *UML Distilled*. Boston: Addison-Wesley Professional, 2003. ISBN 0-321-19368-1.

[7] LEXA I.: *Shora dolů nebo zdola nahoru?* Programování Ostrava, 1987. Dostupné na adrese http://prog-story.technicalmuseum.cz/index.php/m-virtualni-sbirky-tm-v-brne/programovani-a-tvorba-sw-ostrava/1985-1994/1987-programovani-ostrava/2985-1987-shora-dolu-nebo-zdola-nahoru

[8] McLAUGHLIN B. D., POLLICE G. WEST D.: *Head First Object-Oriented Analysis and Design*. O'Reilly Media 2006. ISBN 978-0-596-00867-3.

[9] PECINOVSKÝ R.: *Návrhové vzory – 33 vzorových postupů pro objektové programování.* Brno: Computer Press, a.s. 2007, ISBN 978-80-251-1582-4.

[10] PECINOVSKÝ R.: *Python – Kompletní příručka jazyka pro verzi 3.8.* Praha: Grada Publishing 2019, ISBN 978-80-271-2891-4.

[11] PECINOVSKÝ R.: *Python – Kompletní příručka jazyka pro verzi 3.9.* Praha: Grada Publishing 2020, ISBN 978-80-271-2891-4.

[12] PÍSEK, Slavoj: *Python – práce s knihovnami.* Praha: Grada Publishing 2020. ISBN 978-80-271-0659-2.

Rejstřík

G

H

Ch

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y

Z

Tato publikace uvádí čtenáře do světa programování prostřednictvím jazyka *Python*, který vznikl jako jazyk, jenž má laikům usnadnit vstup do světa programování. Po relativně rychlém úvodu, v němž se čtenář seznámí se základními konstrukcemi jazyka, začne se čtenářem postupně budovat jednoduchou aplikaci – textovou konverzační hru. Kdykoliv se v průběhu budování této aplikace objeví potřeba využít nějakou doposud nevyloženou konstrukci, tak tuto konstrukci vyloží a vzápětí ji v aplikaci použije. Při vývoji aplikace se čtenář současně seznamuje se zásadami moderního programování a učí se je naplňovat v praxi.

Hlavním rozdílem této učebnice od běžných učebnic a kurzů je to, že se neomezuje na výklad toho, jak někým navržený program zakódovat v jazyku *Python*, ale učí čtenáře program samostatně navrhnout a rozchodit.

Osvojené základy umožní čtenáři pokračovat studiem knihy *Python – Kompletní příručka jazyka pro verzi 3.9*, která probírá možnosti jazyka do hloubky. Obě knihy jsou současně podkladem pro doprovodný e-learningový kurz.

Ing. Rudolf Pecinovský, CSc. je absolventem Fakulty Elektrotechnické ČVUT z roku 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 pro naprosté začátečníky. V současné době učí na Vysoké škole ekonomické v Praze, na Fakultě jaderné a fyzikálně inženýrské ČVUT a na Vysoké škole podnikání a práva. Doposud mu vyšlo 60 knih, které byly přeloženy do pěti jazyků. Většina jeho knih je zaměřena na výuku moderního programování a na umění návrhu objektově orientované architektury.

GRADA®

Grada Publishing, a. s.,
U Průhonu 22, 170 00 Praha 7
tel.: +420 234 264 401
e-mail: obchod@grada.cz, www.grada.cz

CZ 349 Kč / SK 14,54 €

9 788027 112371