

Část E

Přílohy

a seznamy

A	Konfigurace ve Windows	322
A.1	Definice substituovaných disků	322
A.2	Nastavování zástupce spouštějícího IDLE	323
B	Použité funkce ze standardní knihovny	325
B.1	Zabudované funkce	325
B.2	Speciální metody	328
B.3	Metody třídy dir	329
B.4	Metody posloupností – Sequence	329
B.5	Metody třídy list	329
B.6	Metody třídy str	330
C	Konvence pro psaní programů v Pythonu	331
C.1	Uspořádání kódu	331
C.2	Jmenné konvence	332
C.3	Dokumentační komentáře (PEP 257)	333
D	Vývojová prostředí pro psaní programů v Pythonu	334
D.1	Jupyter Notebook a JupyterLab	334
D.2	Visual Studio Code	335
D.3	PyCharm	335
	Seznam programů	336
	Seznam obrázků	340
	Seznam tabulek	341
	Seznam odboček – podšeděných bloků	342

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 vězte, 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 vytvořit 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. poklepnáním 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 nastavenou 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 substituuovat 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.

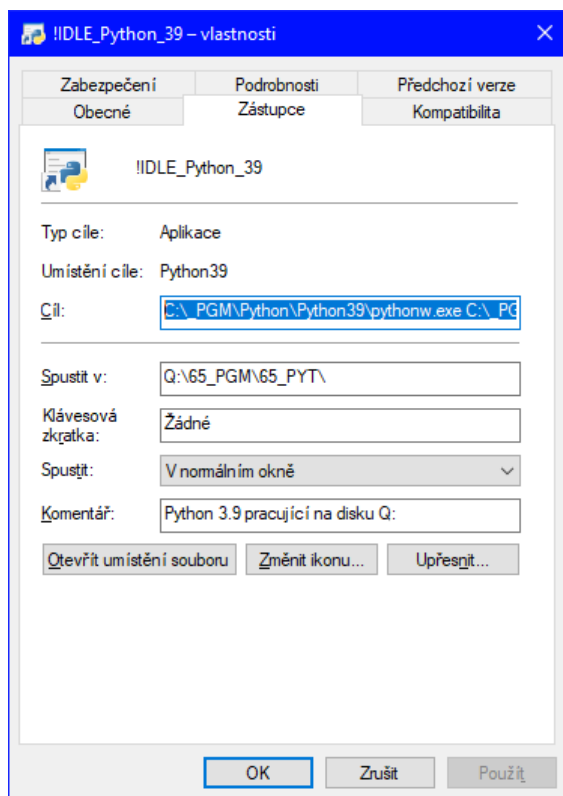
A.2 Nastavování zástupce spouštějícího IDLE

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 spouštění skriptu 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**. Od této chvíle můžete IDLE spouštět poklepáním na daného zástupce.

Příloha B

Použité funkce ze standardní knihovny

B.1 Zabudované funkce

Ležaté hranaté závorky v následujících hlavičkách funkcí označují nepovinné součásti.

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 [8.1](#) na straně [105](#).

dict(x)

Vytvoří slovník z prvků dodaných generátorem zadaným v argumentu. Podrobněji viz [Vytvoření prostřednictvím konstruktorů](#) na straně [133](#).

divmod(x, y, /)

Očekává jako argumenty dvě (ne komplexní) čísla a vrátí dvojici hodnot obsahující jejich celočíselný podíl a zbytek po dělení. Pro celočíselné argumenty platí, že výsledek je $(x//y, x\%y)$. Pro reálné argumenty je to trochu složitější – podrobnosti najdete v dokumentaci.

enumerate(iterable, start=0)

Očekává jako argument zdroj sekvenčních dat. Vrátí instanci třídy `enumerate` generující posloupnost dvojic, v nichž je nultým prvkem klíč a prvním prvkem příslušná hodnota.

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.

float(x=0, /)

Vrátí reálné číslo vytvořené ze zadaného čísla či stringu. Je-li argumentem string, požaduje, aby vyhovoval pravidlům pro zápis reálného čísla, na něž jej převede. Jsou ale povoleny pouze úvodní a závěrečné bílé znaky.

frozenset(/iterable/)

Vrátí nový objekt typu `frozenset`, volitelně s prvky převzatými z iterable. O typu `frozenset` jsme hovořili v podkapitole [10.3 Druhy kontejnerů](#) na straně [131](#).

hash(obj, /)

Vrátí heš-hodnotu objektu (má-li ji), což je celé číslo, které daný objekt do jisté míry charakterizuje. Podrobněji jsme o heš-kódu hovořili v pasáži [Hešovatelné objekty](#) na straně [130](#).

help(/objekt/)

Funkce slouží k získání nápovědy. Protože je ale výklad rozsáhlejší, je mu věnována samostatná podkapitola [Získání nápovědy – dokumentace](#) na straně [86](#).

chr(i, /)

Očekává celé číslo a vrátí string tvořený znakem s daným kódem. Je to vlastně inverzní funkce k funkci `ord()`.

```
>>> chr(0x20ac) #Aby se znak zobrazil, musí být definován v použitém fontu
'€'
>>> chr(0x1D120) #Houslový klíč není v použitém fontu => zobrazil se jako kód
'\\U0001d120'
```

id(object, /)

Vrátí celé číslo sloužící k identifikaci zadaného objektu. Toto číslo můžeme chápat jako virtuální adresu objektu. Objekty se stejným `id` považuje systém za totožné. Rozdílné objekty musí být na různých místech paměti, a proto musejí mít i rozdílná `id`.

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 ni uživatel zapíše to, k čemu byl vyzván, a stiskne ENTER. Vrátí uživatelem zadaný text.

```
>>> input("Zadej číslo: ")
Zadej číslo: 123
'123'
```

int(x=0, base=10, /)

Prvním argumentem je objekt, který lze převést na celé číslo. Je-li argumentem reálné číslo, vrátí celé číslo představující jeho celou část. Je-li argumentem string, požaduje, aby vyhovoval pravidlům pro zápis celého čísla v desítkové soustavě, na něž jej převede. Přitom jsou povoleny pouze úvodní a závěrečné bílé znaky. Není-li zadán argument, vrátí nulu.

```
>>> int(3.9)    #Při převodu se nezaokrouhluje, ale usekává se desetinná část
3
>>> int(-3.9)   #Stejně je tomu i při převodu záporného čísla
-3
```

Je-li zadán druhý argument, pak v prvním argumentu očekává string reprezentující celé číslo zadané v číselné soustavě definované druhým argumentem, který musí být z rozsahu $2 \leq \text{base} \leq 36$. V zadaném čísle jsou povolena zpřehledňovací podtržítka. Číselné soustavy se základem větším než 10 používají v roli dalších číslic písmena A-Z, resp. a-z. Velikost použitých písmen přitom nehraje roli.

len(obj, /)

Očekává objekt s více prvky a vrací počet těchto prvků. Je-li argumentem string, vrátí počet jeho znaků.

```
>>> len("")
0
>>> len('12345')
5
```

list(x)

Vytvoří seznam z prvků dodaných generátorem zadaným v argumentu. Podrobněji viz [Vytoáření prostřednictvím konstruktorů](#) na straně 133.

object()

Vytvoří novou instanci třídy `object`.

ord(c)

Při zadání řetězce představujícího jeden znak vrátí celé číslo představující kód zadaného znaku v sadě *Unicode*. Například `ord('a')` vrátí celé číslo 97 a `ord('€')` (znak eura) vrátí 8364. Je to inverzní funkce k funkci `chr()`.

print(argumenty)

Očekává seznam čárkami oddělených výrazů, jejichž hodnoty má vytisknout na standardní výstup. Podrobnosti viz [6.6 Funkce print\(\) a její parametry](#) na straně 90.

range(stop)**range(start, stop[, step])**

Očekává celočíselné argumenty a vrátí objekt, který se používá jako zdroj posloupnosti celých čísel začínající číslem **start**, pokračující čísla 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)**.

reload(/modul/); přesněji importlib.reload(/modul/)

Znovu načte zadaný modul. Podrobněji viz [5.6 Oprava načteného modulu](#) na straně [79](#).

repr(x)

Vrátí string představující systémový podpis svého argumentu.

set(x)

Vytvoří množinu prvků dodaných generátorem zadaným v argumentu. Podrobněji viz [Vytváření prostřednictvím konstruktorů](#) na straně [133](#).

str(object='')

Převeď zadaný argument na text představující jeho uživatelský podpis.

super(/type/, object-or-type//)

Vytvoří instanci třídy **super** schopnou delegovat požadavek na přístup k zadanému atributu na správného rodiče. Použijete-li bezparametrickou verzi, dodá potřebné argumenty překladač. Volání **super()** je pak ekvivalentní s voláním

```
super(type(self), self)
```

tuple(x)

Vytvoří n-tici prvků dodaných generátorem zadaným v argumentu. Podrobněji viz [Vytváření prostřednictvím konstruktorů](#) na straně [133](#).

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 [Metody repr \(\)](#) a [str \(\)](#) na straně [170](#).

`__str__()`

Vrátí uživatelský podpis své instance. Podrobnosti viz [Metody `repr\(\)` a `str\(\)`](#) na straně [170](#).

B.3 Metody třídy `dir`

`fromkeys(iterable[, value])`

Vytvoří slovník se zadanými klíči a jednotně inicializovanými hodnotami.

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, vyvolá výjimku `ValueError`.

`sorted(x)`

Vrátí nově vytvořený seznam, obsahující seřazená data dodaná zdrojem předaným jako argument. Podrobněji je funkce rozebrána v [\[13\]](#).

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 [20.7](#) na straně [253](#).

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 20.3 na straně 248.

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 řádce 6 ve výpisu 20.9 na straně 255.

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 20.3 na straně 248.

Příloha C

Konvence pro psaní programů v Pythonu

Oficiální konvence pro psaní kódu jsou včetně demonstračních ukázek podrobně popsány v dokumentu [PEP 8](https://www.python.org/dev/peps/pep-0008/)³² na adrese <https://www.python.org/dev/peps/pep-0008/>. Konvence pro psaní dokumentačních komentářů jsou v dokumentu [PEP 257](https://www.python.org/dev/peps/pep-0257/), 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.

C.1 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)
```

³² 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 odděľujte dvěma řádky, definice metod uvnitř třídy odděľujte 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 *double underscores*) by měly být definovány na počátku modulu za dokumentačním komentářem, ale před importy.
- Pro trojitě ohraničení stringů používejte trojitě 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*).

C.2 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 znaky **l** (malé L), **0** (velké o) nebo **I** (velké i). Některé fonty neumí odlišit znaky **l-l-I** (jedna – L – I) a **0-0** (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.

C.3 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.
- Trojitě uvozovky používejte, i když se komentář vejde na řádek, jak je tomu např. ve výpisu [6.2](#) na straně [86](#). 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 vypsát všechny třídy, výjimky a funkce daného modulu.
- Dokumentační komentář třídy by měl vypsát její chování, seznam veřejných metod a atributů včetně instančních.
- Dokumentační komentář funkce by měl vypsát její funkci, argumenty a návratovou hodnotu, vedlejší efekty a případná omezení.

Příloha D

Vývojová prostředí pro psaní programů v Pythonu

Následující přehled stručně představuje tři nejpoužívanější IDE podporující vývoj programů v *Pythonu*. Zájemce o přehled širšího spektra prostředí odkážu na stránku https://en.wikipedia.org/wiki/Comparison_of_integrated_development_environments#Python, kde najdete stručné porovnání 21 IDE.

D.1 Jupyter Notebook a JupyterLab

Project Jupyter je nezisková organizace zabývající se vývojem open-source softwaru, otevřených standardů a služeb pro interaktivní výpočetní techniku. Její název vznikl jako akronym složený z názvů tří primárně podporovaných jazyků: *Julia*, *Python* a *R*. Nabídka se postupně rozšířila, takže v současné době podporuje 40 programovacích jazyků.

Jupyter Notebook je webová aplikace pro vytváření a sdílení výpočetních dokumentů. Nabízí jednoduché, zjednodušené prostředí zaměřené na dokumenty označované jako poznámkové bloky (notebooks) *Jupyter*.

JupyterLab je uživatelské rozhraní nové generace. Jeho flexibilní rozhraní umožňuje uživatelům konfigurovat a uspořádat pracovní postupy v oblasti analýzy dat, vědeckých výpočtů, počítačové žurnalistiky a strojového učení. Modulární design usnadňuje definici rozšíření, která dále rozšiřují a obohacují jeho funkcionalitu.

S oběma se můžete seznámit na adrese <https://jupyter.org/try>. Podpora prostředí *Jupyter* je začleněna do obou následně zmiňovaných vývojových prostředí.

D.2 Visual Studio Code

Prostředí *Visual Studio Code* je relativně mladé. *Microsoft* uvedl jeho předběžnou verzi označovanou jako *public preview* v dubnu 2015 jako open-source projekt vytvářený v jazycích *JavaScript* a *TypeScript*. Finální verze vyšla rok poté.

Prostředí má nepřeberné množství pluginů pro nejružnější jazyky včetně *Pythonu*. *Microsoft* je označuje jako nejrozšířenější vývojové prostředí, což je vzhledem k množství podporovaných jazyků docela pochopitelné. Mezi programátory *Pythonu* je populární zejména u těch, kteří mají s tímto prostředím zkušenosti z vývoje v jiných programovacích jazycích.

Prostředí můžete stáhnout na adrese <https://code.visualstudio.com/>.

D.3 PyCharm

Těm, kteří touží po něčem maximálně dokonalém, bych doporučil vývojové prostředí *PyCharm* vyvinuté pražskou firmou *JetBrains*. Verze *Professional Edition* je sice placená (studenti řady škol ji ale mají zdarma), nicméně existuje i verze označená *Community Edition*, která je zdarma a pro látku probíranou v obou příručkách bohatě stačí.

Firma vznikla v roce 2001 a stala se známou především kvůli vývojovému prostředí *IntelliJ IDEA*, které je zdaleka nejpoužívanějším prostředím pro vývoj v jazyku *Java*, *Kotlin* a řadě dalších. Nabízí i plugin pro vývoj v jazyku *Python*.

V lednu 2011 firma vydala speciální vývojové prostředí určené především pro vývoj v jazyku *Python* a pojmenovala je *PyCharm*. Toto prostředí se poměrně rychle stalo nejpoužívanějším vývojovým prostředím profesionálních programátorů v *Pythonu*.

Zvládnutí tohoto prostředí je sice poněkud náročnější než zvládnutí *IDLE*, ale zejména při vývoji rozsáhlejších projektů se vám tato námaha bohatě vrátí.

Obě verze (*Community* a *Professional*) vývojového prostředí *PyCharm* si můžete stáhnout na adrese <https://www.jetbrains.com/pycharm/download>.

Seznam programů

Výpis 1.1:	Komunikace s interpretem spuštěným v konzolovém okně	35
Výpis 2.1:	Komunikace s interpretem spuštěným v konzolovém okně	42
Výpis 2.2:	Komunikace s interpretem v příkazovém okně IDLE	43
Výpis 3.1:	Počáteční mezery a komentáře	44
Výpis 3.2:	Možnosti zápisu celých čísel	45
Výpis 3.3:	Zadávaní a zobrazování reálných čísel	46
Výpis 3.4:	Zápis jednořádkového textu	47
Výpis 3.5:	Ve stringu představuje znak # sám sebe	47
Výpis 3.6:	Zápis víceřádkového textu	48
Výpis 3.7:	Definice a použití proměnných	52
Výpis 3.8:	Současné přiřazení skupiny hodnot proměnným	53
Výpis 3.9:	Nebezpečné změny hodnot	56
Výpis 4.1:	Použití funkcí	59
Výpis 4.2:	Hodnota None a její zobrazení	60
Výpis 4.3:	Reakce na použití objektu ... (Ellipsis)	61
Výpis 4.4:	Použití formátovaných stringů	62
Výpis 4.5:	Více příkazů versus více výrazů na jednom řádku	63
Výpis 4.6:	Složené přiřazovací operátory	65
Výpis 4.7:	Ukázka použití funkce input() a implicitní proměnné	65
Výpis 4.8:	Typy objektů, s nimiž jsme se doposud seznámili	66
Výpis 4.9:	Demonstrace práce s atributy	68
Výpis 5.1:	Importování modulů	72
Výpis 5.2:	Uložení odkazu na modul pod jiným názvem	74
Výpis 5.3:	Import zadaných objektů ze zadaného modulu	75
Výpis 5.4:	Modul m05a_module_demo sloužící k demonstraci chování Pythonu při zavádění modulu	76
Výpis 5.5:	Použití modulu m05a_module_demo	78
Výpis 5.6:	Upravené řádky v modulu m05a_module_demo	79
Výpis 5.7:	Načtení upravené verze modulu m05a_module_demo	80
Výpis 6.1:	Definice funkce demo() a její použití	85
Výpis 6.2:	Definice funkce pozdrav()	86
Výpis 6.3:	Upravená definice funkce pozdrav()	87
Výpis 6.4:	Definice a použití funkce bmi	88
Výpis 6.5:	Používání pozičních a pojmenovaných argumentů	89
Výpis 6.6:	Možné podoby definice prázdné funkce	91
Výpis 6.7:	Definice funkce demonstrující použití anotací	92
Výpis 7.1:	Vytvoření světa s definovanými počty značek a umístění robota	97
Výpis 7.2:	Ovládání vytvořeného robota	100
Výpis 7.3:	Použití některých zjišťovacích funkcí	101

Výpis 7.4:	Demonstrace vlivu skrývání	102
Výpis 7.5:	Definice funkcí <code>turn_about()</code> a <code>step_back()</code> demonstrujících vnořené použití skrývání	103
Výpis 8.1:	Převod libovolné hodnoty na logickou	105
Výpis 8.2:	Porovnání hodnot versus porovnání objektů	107
Výpis 8.3:	Chování logických operátorů	109
Výpis 8.4:	Funkce <code>free_before()</code> zjišťující, zda je před robotem volno	109
Výpis 8.5:	Podmíněný výraz a jeho použití	110
Výpis 8.6:	Začlenění nuly	111
Výpis 8.7:	Jednoduchý podmíněný příkaz a jeho použití	112
Výpis 8.8:	Použití podmíněného příkazu ke zvýraznění zapnutí a vypnutí zobrazení	113
Výpis 8.9:	Definice a test funkce <code>are_2_markers()</code> využívající zanořování bloků	113
Výpis 8.10:	Ukázka použití úplného podmíněného příkazu	115
Výpis 8.11:	Ukázka použití rozšířeného podmíněného příkazu	116
Výpis 8.12:	Ukázka použití přepínače příkazu	117
Výpis 9.1:	Definice funkce <code>to_wall()</code> využívající cyklus <code>while</code>	119
Výpis 9.2:	Definice funkce <code>find_treasure()</code> hledající s robotem poklad ze značek	120
Výpis 9.3:	Definice funkce <code>markers_2_wall()</code> používající cyklus s podmínkou uprostřed	121
Výpis 9.4:	Definice funkce <code>goto_next_marker()</code> demonstrující použití cyklu s koncovou podmínkou	122
Výpis 9.5:	String použitý jako objekt a jako zdroj hodnot	123
Výpis 9.6:	Posloupnosti reprezentované objekty typu <code>range</code>	124
Výpis 9.7:	Posloupnosti reprezentované objekty typu <code>enumerate</code>	124
Výpis 9.8:	Definice funkce <code>put_n()</code> demonstrující použití jednoduchého cyklu <code>for</code>	125
Výpis 9.9:	Definice funkce <code>put_n_beside()</code> demonstrující možnost zadání potřebných funkcí pomocí proměnných	126
Výpis 9.10:	Definice funkce <code>test_put_n()</code> testující obě výše definované funkce	126
Výpis 9.11:	Definice funkce <code>factorial()</code> demonstrující použití rekurze	127
Výpis 9.12:	Definice funkce <code>put_marker_at_wall()</code> řešené pomocí rekurze	128
Výpis 10.1:	Vytváření kontejnerů prostřednictvím literálů	132
Výpis 10.2:	Vytváření kontejnerů prostřednictvím konstruktorů	134
Výpis 10.3:	Vytváření kontejnerů prostřednictvím generátorové notace	135
Výpis 10.4:	Funkce <code>mocniny()</code> vracející generátor pro tvorbu kontejnerů	137
Výpis 11.1:	Získání prvku z kontejneru pomocí indexu	139
Výpis 11.2:	Procházení kontejnerem pomocí příkazu <code>for</code>	140
Výpis 11.3:	Cyklus <code>for</code> s více parametry	141
Výpis 11.4:	Pohledy a jejich reflexe změn v „pozorovaném“ slovníku	142
Výpis 11.5:	Další možnosti práce s jednotlivými prvky kontejnerů	144
Výpis 11.6:	Použití hvězdičkového parametru a argumentu	146
Výpis 11.7:	Použití dvouhvězdičkového parametru a argumentu	147
Výpis 12.1:	Nástin syntaxe příkazů pro zachycení a ošetření výjimky	151
Výpis 12.2:	Definice modulu <code>m12a_Demo_výjimky</code> sloužícího k demonstraci zachycení a ošetření výjimky	152
Výpis 12.3:	Záznam importu modulu <code>m12a_Demo_výjimky</code> a volání jeho funkcí	154
Výpis 12.4:	Definice funkce <code>factorial()</code> demonstrující použití kontrolních tisků	157
Výpis 13.1:	Definice demonstrační třídy <code>DemoClass</code>	167
Výpis 13.2:	Vytvoření instancí třídy <code>DemoClass</code> a práce s nimi	172
Výpis 13.3:	Definice prázdné třídy	173
Výpis 14.1:	Definice tříd <code>Matka</code> a <code>Dcera</code> v modulu <code>m14a_MDPV</code>	182

Výpis 14.2:	Záznam vytvoření instancí tříd Matka a Dcera a volání jejich metod	183
Výpis 14.3:	Definice tříd Přítelkyně a VnučkaDP v modulu m14a_MDPV	184
Výpis 14.4:	Záznam vytvoření instancí tříd VnučkaDP a VnučkaPD z modulu m14a_MDPV	185
Výpis 14.5:	AHA-příklad s definicí hierarchie abstraktních a konkrétních tříd	187
Výpis 14.6:	Definice protokolu IKarel v initoru balíčku robotcz	189
Výpis 14.7:	Definice a použití třídy Josef implementující protokol IKarel	189
Výpis 15.1:	Ukázka přidání definovaných funkcí jako metod třídy Karel	192
Výpis 15.2:	Ukázka přidání metod třídy Karel jako funkcí aktuálního modulu	193
Výpis 15.3:	Použití příkazu global	195
Výpis 15.4:	Zadání zjišťovací vlastnosti pomocí dekorátoru	198
Výpis 15.5:	Prvky standardních n-tic nemají jména	200
Výpis 15.6:	Pojmenované n-tice	201
Výpis 15.7:	Definice třídy Direction4 z balíčku robotcz	202
Výpis 16.1:	Šablona initorů balíčků v souboru __init__Template.py	205
Výpis 16.2:	Initor balíčku pkg	206
Výpis 16.3:	Definice modulu pkg.pkg2a.importing	207
Výpis 16.4:	Import modulu pkg.pkg2a.importing	207
Výpis 16.5:	Definice modulu m16a_script	209
Výpis 16.6:	Reakce na spuštění modulu m16a_script v interaktivním režimu	209
Výpis 16.7:	Reakce na spuštění modulu m16a_script v příkazovém okně Windows	209
Výpis 16.8:	Modul main v balíčku demo	210
Výpis 16.9:	Vytvoření samostatné aplikace pomocí modulu zipapp a její spuštění	211
Výpis 16.10:	Definice modulu __main__ v souboru 70_Demo_App.pyz	212
Výpis 19.1:	Definice třídy ScenarioStep v modulu scenarios	239
Výpis 19.2:	Podpis instancí třídy ScenarioStep definované podle výpisu 19.1	240
Výpis 19.3:	Změna systémového podpisu instancí třídy ScenarioStep	240
Výpis 19.4:	Podpis kroku scénáře po změně metody __repr__()	240
Výpis 19.5:	Definice počátku n-tice reprezentující „šťastný scénář“ vytvářené hry	241
Výpis 19.6:	Definice funkce simulate_simple() v modulu scenarios	242
Výpis 19.7:	Spuštění metody simulate_simple()	242
Výpis 19.8:	Definice funkce simulate_with_state() v modulu scenarios	243
Výpis 19.9:	Začátek simulace vypsané metodou simulate_with_state()	243
Výpis 20.1:	Definice funkce execute_command() v modulu __init__	245
Výpis 20.2:	Upravená definice třídy Place v modulu world	247
Výpis 20.3:	Definice funkce test_scenario() v modulu scenarios	248
Výpis 20.4:	Definice metody ERROR() v modulu scenarios	249
Výpis 20.5:	Import modulu scenarios spojený se spuštěním testu	250
Výpis 20.6:	Definice upravené metody execute_command() v modulu __init__	252
Výpis 20.7:	Definice funkce execute_command() v modulu actions	253
Výpis 20.8:	Definice metody _execute_empty_command() v modulu actions	254
Výpis 20.9:	Definice funkce _execute_standard_command() v modulu actions	255
Výpis 20.10:	Import modulu scenarios	256
Výpis 21.1:	Doplnění definice funkce _initialize() v modulu actions o inicializaci modulu world	258
Výpis 21.2:	Definice třídy ANamed v modulu world	259
Výpis 21.3:	Upravená definice třídy Item v modulu world	260
Výpis 21.4:	Upravená definice třídy Place v modulu world	261

Výpis 21.5:	Definice atributu NAME_2_PLACE v modulu world.....	261
Výpis 21.6:	Inicializační funkce modulu world.....	262
Výpis 21.7:	Konec zprávy o průběhu testu aplikace	262
Výpis 21.8:	Definice třídy Action v modulu actions	264
Výpis 21.9:	Definice abstraktní třídy AItemContainer v modulu world.....	265
Výpis 21.10:	Upravené definice initoru třídy ANamed v modulu world.....	267
Výpis 21.11:	Upravené definice třídy Place v modulu world.....	267
Výpis 21.12:	Upravená definice třídy Bag v modulu world.....	268
Výpis 21.13:	Definice funkce take() v modulu actions a zařazení nové akce do slovníku NAME_2_ACTION.....	269
Výpis 21.14:	Závěr zprávy o průběhu testu	270
Výpis 21.15:	Definice funkce put v modulu actions spolu s aktualizovanou podobou inicializace slovníku NAME_2_ACTION	270
Výpis 21.16:	Definice funkce goto v modulu actions.....	271
Výpis 21.17:	Definice funkce end v modulu actions.....	271
Výpis 22.1:	Začátek definice třídy Scenario v modulu scenarios.....	274
Výpis 22.2:	Začátek definice chybového scénáře v modulu scenarios	276
Výpis 22.3:	Definice slovníku scénářů a dalších funkcí v modulu scenarios.....	277
Výpis 22.4:	Konec zprávy po zadání trojitého testu	278
Výpis 22.5:	Konec zprávy o průběhu testu po opravě reakce na špatný start	280
Výpis 22.6:	Konec zprávy o průběhu testu po opravě reakcí na nezadaný argument.....	281
Výpis 22.7:	Upravená definice třídy Item v modulu world.....	282
Výpis 22.8:	Definice metody try_add() ve třídě Bag v modulu world	283
Výpis 22.9:	Definice funkce take() v modulu actions.....	284
Výpis 22.10:	Definice funkce help v modulu actions.....	284
Výpis 23.1:	Definice funkce run() v modulu __init__.....	286
Výpis 23.2:	Spuštění hry v interaktivním režimu	287
Výpis 23.3:	Definice funkce multirun() v modulu __init__	288
Výpis 23.4:	Definice funkce current_state() v modulu __init__.....	289
Výpis 23.5:	Upravená definice funkce execute_command() v modulu __init__	289
Výpis 23.6:	Definice funkcí main() a help() v modulu __init__.....	290
Výpis 24.1:	Definice třídy Console v modulu __init__	293
Výpis 24.2:	Test argumentu příkazového řádku vkládaný do funkce multirun v modulu __init__	294
Výpis 24.3:	Upravené definice funkcí run a multirun v modulu __init__	294
Výpis 24.4:	Definice modulu gui_0.....	299
Výpis 24.5:	Spuštění hry v grafickém režimu.....	300
Výpis 24.6:	Definice modulu __main__ v kořenovém balíčku vytvářené aplikace.....	300
Výpis 24.7:	Definice tříd PlaceName a PlaceDescription v modulu texts	303
Výpis 24.8:	Definice tříd ItemName a ItemWeight v modulu texts.....	303
Výpis 24.9:	Počátek definice slovníku NAME_2_PLACE v modulu world	304
Výpis 24.10:	Konec definice třídy ActionMessages v modulu texts.....	304
Výpis 24.11:	Definice funkce take() v modulu actions.....	305

Seznam obrázků

Obrázek 1.1: Okno s dokumentací platformy	34
Obrázek 1.2: Spuštění interpretu z příkazového řádku Windows	34
Obrázek 2.1: Okno vývojového prostředí IDLE se zadáním z výpisu 1.1	38
Obrázek 2.2: Okno vývojového prostředí IDLE	38
Obrázek 5.1: Okno světa želvy po provedení zadaných příkazů	73
Obrázek 5.2: Okno světa želvy po provedení zadaných příkazů	75
Obrázek 7.1: Svět robotů vytvořený programem z výpisu 7.1 doplněný o robota příkazy z výpisu	98
Obrázek 7.2: Svět robota po provedení akcí z výpisu 7.2	99
Obrázek 7.3: Okno s chybovým hlášením po provedení akcí z výpisu 7.2	100
Obrázek 8.1: Postup zpracování jednoduchého podmíněného příkazu	111
Obrázek 8.2: Vývojový diagram úplného podmíněného příkazu	114
Obrázek 8.3: Vývojový diagram zobrazující postup vykonávání rozšířeného podmíněného příkazu	115
Obrázek 9.1: Bludiště, v němž robot hledá poklad	120
Obrázek 11.1: Indexování jednotlivých znaků textového řetězce – stringu	143
Obrázek 14.1: Diamantový problém	184
Obrázek 17.1: Prvky používané při tvorbě UML diagramů	223
Obrázek 18.1: Diagram tříd aktuálního stavu aplikace zkopírovaného do balíčku game_ver.v1a_basic_architecture	234
Obrázek 19.1: Diagram tříd aktuálního stavu aplikace zkopírovaného do balíčku game_ver.v1a_basic_architecture	244
Obrázek 20.1: Dotaz systému, má-li ukončit běžící program	251
Obrázek 20.2: Diagram tříd aktuálního stavu aplikace zkopírovaného do balíčku game_ver.v1b_test_start	257
Obrázek 21.1: Diagram tříd aktuálního stavu aplikace zkopírovaného do balíčku game_ver.v1c_world	263
Obrázek 21.2: Diagram tříd aktuálního stavu aplikace zkopírovaného do balíčku game_ver.v1d_actions	272
Obrázek 22.1: Diagram tříd aktuálního stavu aplikace zkopírovaného do balíčku game_ver.v1e_mistakes	285
Obrázek 23.1: Diagram tříd aktuálního stavu aplikace zkopírovaného do balíčku game_ver.v1f_application	291
Obrázek 24.1: Dialogové okno s výzvou k zadání údajů	298
Obrázek 24.2: Upozornění na chybné zadání číselné hodnoty	298

Seznam tabulek

Tabulka 3.1:	Klíčová slova jazyka Python.....	50
Tabulka 11.1:	Tabulka různých způsobů vykrajování	143
Tabulka 17.1:	Porovnání metody shora dolů a zdola nahoru	222

Seznam odboček – podšeděných bloků

Odbočka – podšeděný blok.....	23
Historie robota Karla	96
Zvláštnosti programových kontejnerů	129
Co to je h-objekt.....	228
Návrhový vzor Fasáda.....	296



Python vznikl jako jazyk, který má laikům usnadnit vstup do světa programování. Jeho obliba a s ní související oblast použití se postupně neustále rozšiřují a podle většiny ukazatelů je dnes nejpoužívanějším programovacím jazykem vůbec.

Předkládaná příručka seznámí čtenáře s tímto jazykem a postupně ho naučí používat Python pro tvorbu stále dokonalejších programů. První část představí základy jazyka, v druhé se naučíte vytvářet jednoduché programy, třetí vás připraví pro tvorbu větších aplikací a ve čtvrté části se na příkladu tvorby jednoduché, nicméně netriviální aplikace seznámíte se základními pravidly vytváření komplexnějších programů. Při vývoji aplikace si současně osvojíte zásady moderního programování a naučíte se je naplňovat v praxi.

Tato kniha se od běžných učebnic a kurzů odlišuje: neomezuje se na výklad toho, jak navržený program zakódovat v jazyku Python, ale naučí čtenáře programy především samostatně navrhovat a také je rozchodit. Navazuje na ni publikace *Python – Kompletní příručka pro verzi 3.10*, která probírá možnosti jazyka ještě více do hloubky.



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 429 Kč / SK 17,88 €

