Univerzita Pardubice
Fakulta elektrotechniky a informatiky

Python

Distanční opora

doc. Ing. Michael Bažant, Ph.D.

Financováno Evropskou unií
NextGenerationEU
Národní plán obnovy
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

1 Úvod do programování v jazyce Python

Studijní cíl

Tento první blok celého kurzu je zaměřen na zvládnutí základních prvků jazyka Python.

Doba nutná k nastudování 2– 2,5 hodiny

Python je všestranný a široce používaný programovací jazyk známý svou jednoduchostí a čitelností. Jedná se o vysokoúrovňový interpretovaný jazyk, který získal popularitu mezi vývojáři díky svému snadnému použití, velkému množství snadno dostupných knihoven a frameworků a širokému uplatnění v různých oblastech. V tomto bloku se budeme zabývat historií jazyka Python, populárními integrovanými vývojovými prostředími (IDE) pro vývoj v jazyce Python a některými základními příkazy.

1.1 Historie jazyka Python

Python byl vytvořen Guidem van Rossumem a poprvé byl uveden v roce 1991. Guido van Rossum měl za cíl vytvořit jazyk, který bude klást důraz na čitelnost kódu a snadné použití. Název "Python" byl inspirován britskou komediální skupinou Monty Python.

Python prošel několika hlavními verzemi, přičemž Python 2.x (oficiální podpora ukončena v roce 2020) a Python 3.x jsou nejznámější. Python 3.x přinesl významné vylepšení a změny v jazyce, které ho učinily konzistentnějším a robustnějším. Python 2.x je nyní považován za zastaralý, a komunita silně doporučuje vývojářům používat Python 3.x pro všechny nové projekty. Zásadní rozdíly mezi verzí 2.x a 3.x jsou následující:

- Pro příkaz print je nutné ve verzi 3.x používat dvojici kulatých závorek, tedy print(„Hello“). V rámci verze 2.x bylo možné realizovat tento příkaz: print „Hello“.
- Zásadní rozdíl při dělení celých čísel (ve verzi 2.x je výsledkem celé číslo, ve verzi 3.x je výsledkem reálné číslo).
- Podpora Unicode ve verzi 3.x, ve verzi 2.x je textový řetězec ukládán pomocí ASCII.
- Iterování objektů ve verzi 3.x v běžně používaných metodách (např. range, map, filter apod.) místo návratu seznamu ve verzi 2.x.

1.2 Integrovaná vývojová prostředí (IDE) pro Python

Integrované vývojové prostředí (IDE) jsou softwarové aplikace, které poskytují nástroje a funkce pro usnadnění vývoje software. Python má řadu populárních IDE, z nichž každé nabízí specifické funkce, které vyhovují různým potřebám. Zde jsou některá z nejběžněji používaných IDE pro vývoj v jazyce Python.

1.2.1. PyCharm

PyCharm, vyvinutý společností JetBrains, je široce používané IDE pro vývoj v vazyce Python. Nabízí funkce jako analýza kódu, ladění, správa verzí a širokou škálu doplňků. PyCharm je k dispozici jak ve verzi zdarma (Community Edition), tak ve verzi placené (Professional Edition). Edice Professional disponuje značným množstvím funkcí navíc oproti Community Edition (podpora pro celou žadu frameworků, SQl databází apod.).

1.2.2. Visual Studio Code (VSCode)

Visual Studio Code je open-source textový editor vyvinutý společností Microsoft. Je vysoce přizpůsobitelný pomocí rozšíření, což ho činí oblíbenou volbou pro vývojáře v nejrůznějších programovacích jazycích a také v jazyce Python. Díky rozšíření pro Python nabízí VSCode funkce jako doplňování kódu, ladění, integraci s Git apod.

1.2.3. Jupyter Notebook

Jupyter Notebook je open-source webová aplikace, která umožňuje vytvářet a sdílet dokumenty obsahující živý kód, rovnice, vizualizace a text. Je zvláště užitečný pro data science (datovou vědu) a interaktivní programování. Výhodou je integrace s webovým prohlížečem a tak lze pracovat a zobrazovat výsledky programů přímo v tomto běžně používaném SW.

1.2.4. Spyder

Spyder je IDE navržené speciálně pro vědecké a datové analýzy v jazyce Python. Poskytuje interaktivní vývojové prostředí s funkcemi jako je inspekce proměnných, ladění a integrace s vědeckými knihovnami jako NumPy a SciPy.

1.2.5. IDLE (Python's Integrated Development and Learning Environment)

IDLE je základní IDE, které je dodáváno s jazykem Python jako součást. Jedná se o velmi jednoduché vývojové prostředí, což ho činí dobrou volbou pro začátečníky.

1.3 Základní pravidla pro konvence kódu v jazyce Python

V následujícím textu jsou uvedena pouze základní pravidla pro zápis zdrojového kódu. Bližší informace jsou uvedeny v dokumentu PEP 8, přičemž první verze tohoto dokumentu vznikla v roce 2001 – Style Guide for Python Code - PEP 8 – Style Guide for Python Code | peps.python.org

1.3.1. Odsazování

Používáme 4 mezery pro odsazení kódu (nikoliv tabulátory). Odsazení by mělo být konzistentní v rámci celého kódu.

1.3.2. Mezery kolem operátorů

Mezery vkládáme také kolem operátorů (=, +, -, *, / atd.).

```
# dobre:
x = 10
y = x * 2

# spatne:
x=10
y = x*2
```

1.3.3. Doporučené délky řádků

Snažíme se udržovat řádky s max. délkou 79 znaků. Pro dlouhé řádky lze použít zpětné lomítko (\) nebo závorky.

1.3.4. Prázdné řádky

Prázdné řádky používáme ke zdůraznění oddělení funkcí nebo tříd. Mezi jednotlivými bloky kódu by měly být prázdné řádky pro lepší čitelnost.

1.3.5. Komentáře

Komentáře přidáváme ke složitým částem kódu pro vysvětlení toho, co kód dělá. Komentáře by měly být jasné a stručné.

Základním typem komentáře je komentář do konce řádku:

```
pocet_studentu = 10 # promenna pro pocet studentu
```

Další možností je víceřádkový komentář:

```
'''
Toto je
viceradkovy
komentar
'''
```

1.3.6. Názvy proměnných a funkcí

Používáme smysluplné názvy proměnných a funkcí, které jasně popisují jejich účel. Názvy by měly být psány malými písmeny, s podtržítky pro víceslovné názvy (snake_case).

```
# dobre:
pocet_pokusu = 3
def vypocet_souctu(a, b):
    return a + b


# spatne:
pp = 3
def vypocet_sum(a, b):
    return a + b
```

1.3.7. Importy

Importy by měly být umístěny na začátku souboru. Rozdělujeme je do tří skupin: standardní knihovny, knihovny třetí strany a vlastní moduly. Příkaz "import" používáme pro celé moduly a "from module import name" pro konkrétní části, které potřebujeme použít:

```
import os
import sys

from numpy import array, linspace
from muj_modul import funkce_a, funkce_b
```

1.3.8. Ošetření výjimek

Bloky try a except používáme k zachycení a zpracování výjimek tam, kde je to vhodné.

```
try:
    # Kod, ktery muze vyvolat vyjimku
    vysledek = deleni(5, 0)
except ZeroDivisionError:
    print("Nelze delit nulou.")
```

1.4 Spuštění Python kódu

Při používání programovacího jazyka Python je důležité vědět jaký způsob je využíván pro spuštění kódu. Při jistém nadhledu můžeme uvažovat následující kroky vedoucí ke spuštění kódu zapsaného pomocí jazyka Python:

1. **Zdrojový kód:** Python kód napíšeme do textového souboru s příponou .py, jako je například my_program.py. Tento soubor obsahuje instrukce, které má interpret jazyce Python provést.

2. **Tokenizace a analýza**: Když spustíme vlastní skript Python, interpret jazyce Python načte zdrojový kód a rozdělí ho na řadu tokenů (jako jsou klíčová slova, identifikátory a symboly) a poté tyto tokeny analyzuje do abstraktního syntaktického stromu (AST). Tento proces kontroluje, zda kód neobsahuje syntaktické chyby.

3. **Kompilace bytecode**: Python interpret kompiluje zdrojové soubory do bytecode, který je alokován v paměti. Bytecode může být také uložen v podobě pyc souboru pro rychlejší spouštění. Bytecode je reprezentace kódu nižší úrovně, která je nezávislá na platformě a může být spuštěna virtuálním strojem Python (PVM).

4. **Spuštění:** Interpret jazyce Python, nebo konkrétněji Python Virtual Machine (PVM), provádí bytecode řádek po řádku. Interpretuje každý příkaz a provádí zadané operace, jako je přiřazení proměnných, volání funkcí a příkazy toku řízení (if, for, while atd.).

5. **Správa paměti:** Python spravuje paměť automaticky. Přiděluje paměť pro proměnné a datové struktury a uvolňuje paměť, která se již nepoužívá a to prostřednictvím procesu známého jako garbage collection.

6. **Standardní knihovna a moduly:** Python poskytuje rozsáhlou standardní knihovnu a umožňuje importovat externí moduly a balíčky. Když importujeme modul, Python spustí kód v tomto modulu a zpřístupní jeho funkce a třídy pro použití v námi vytvořeném programu.

7. **Zpracování chyb a výjimek:** Pokud během spuštění programu dojde k chybě (např. dělení nulou nebo chyba názvu), Python vyvolá výjimku. Tyto výjimky lze zpracovat pomocí bloků try a except pro záchyt výjimek (viz další výukové bloky v tomto kurzu).

8. **Ukončení:** Když váš program dosáhne konce svého provádění nebo narazí na neošetřenou výjimku, tak se ukončí a všechny přidělené prostředky se uvolní.

9. **Čištění:** Python před ukončením programu provede všechny nezbytné úkoly čištění, jako je uzavření souborů a uvolnění zdrojů.

Je důležité si uvědomit, že Python je interpretovaný jazyk, což znamená, že zdrojový kód je spouštěn přímo interpretem jazyce Python. Existují však také nástroje jako py2exe, cx_Freeze nebo PyInstaller, které dokážou převést skripty Python do samostatných spustitelných souborů pro distribuci, takže to vypadá, jako by byl kód spouštěn bez potřeby samostatného interpretu jazyce Python nainstalovaného v systému uživatele.

1.5 Základní příkazy v jazyce Python

Nyní se podívejme na některé základní příkazy v jazyce Python.

1.5.1. Tisk do konzole

Můžete použít funkci **`print()`** k zobrazení textu nebo hodnot do konzole.

```
print("Ahoj, Pythone!")
```

1.5.2. Deklarace proměnných

V jazyce Python můžeme deklarovat proměnné bez explicitního určení jejich datových typů.

```
x = 10          # cele cislo (Integer)
y = 3.14        # desetinne cislo (Float)
name = "Alice" # retezec (String)
```

V jazyce Python neurčujeme při deklaraci explicitně datový typ proměnné. Python je dynamicky typovaný jazyk, což znamená, že datový typ proměnné je určen za běhu na základě hodnoty, která je jí přiřazena (zásadní rozdíl oproti jazykům jako jsou Java, C++ apod).

Python poskytuje širokou škálu vestavěných datových typů a datových struktur, včetně celých čísel, reálných čísel, řetězců, seznamů, slovníků, sad a dalších. Samozřejmě lze snadno vytvářet vlastní datové typy a datové struktury např. pomocí tříd.

Pokud potřebujeme provést kontrolu typu nebo převod typu, Python nabízí funkce jako isinstance(), int(), float(), str() a další pro práci s datovými typy podle potřeby. Například:

```
x = 10
if isinstance(x, int):
print("x is an integer")


y = "3.14"
float_y = float(y)  # prevod string na float
print(float_y)      # vystup: 3.14
```

I pokud v jazyce Python explicitně nedeklarujeme datové typy pro proměnné, můžeme s datovými typy pracovat a manipulovat podle potřeby pomocí vestavěných funkcí a vlastností jazyce Python.

V jazyce Python můžeme pomocí typových anotací poskytovat hint pro datové typy, které označují očekávané datové typy proměnných, parametrů funkcí a návratových hodnot. I když hints pro typy nejsou vynucovány za běhu, mohou být užitečné pro dokumentaci, srozumitelnost kódu a pro statické analytické nástroje (jako jsou linters nebo type checkers), aby se včas zachytily potenciální problémy související s datovým typem.

Chcete-li přidat hint na datový typ, použijeme dvojtečku (:) následovanou požadovaným datovým typem za názvem proměnné nebo parametru. Například:

```
# hint pro promennou
x: int = 10

# hint pro parametry metod a navratovou hodnotu
def add(a: int, b: int) -> int:
    return a + b
```

Ve výše uvedeném příkladu jsme použili hints k označení, že se očekává, že proměnná x bude celé číslo, a funkce add přebírá dva celočíselné parametry (a, b) a vrací celé číslo.

1.6 Shrnutí

V této lekci jsou uvedeny základní koncepty programování pomocí vyššího programovacího jazyka Python v podobě představení základních možností vývoje pomocí:

- běžně používaných vývojových prostředí,
- doporučených syntaktických pravidel pro zápis zdrojového kódu,
- způsobu spuštění zapsaného Python kódu,
- přehledu základních příkazů.

1.7 Otázky na procvičení

1. Jaké základní verze jazyka Python se běžně používají a jaký je mezi nimi základní rozdíl?
2. Jaká platí základní syntaktická pravidla pro zápis programů v jazyce Python?
3. Jaká jsou běžně používaná vývojová prostředí pro jazyk Python?
4. Jaký způsobem zapisujeme deklaraci proměnných v jazyce Python?
5. Jaký je základní postup v jazyce Python od zápisu zdrojového kódu po jeho spuštění?

1.8 Odkazy a další studijní materiály

1. PEP 8 – Style Guide for Python Code | peps.python.org

Financováno Evropskou unií
NextGenerationEU
Národní plán obnovy
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

2 Základní datové typy a datové struktury

Studijní cíl

Tento blok je zaměřen na problematiku standardních datových typů a datových struktur v jazyce Python.

Doba nutná k nastudování 2– 2,5 hodiny

2.1 Datové typy

V jazyce Python jsou:

- veškeré **datové typy implementovány jako objekty**,
- všechny objekty jsou **určitého typu**, což určuje množinu operací, které mohou být realizovány,

Typ objektu také určuje, zda **může být uložená hodnota změněna (mutable) nebo nemůže být změněna (immutable).**

Python je **silně typový jazyk** – typ objektu se nemění i když samotnou hodnotu měnit lze.

Přehled vestavěných datových typů:

- int
- float
- str
- bool
- None

Příklad deklarace proměnné datového typu int:

```
>>> a = 7
```

Datový typ deklarované proměnné lze zjistit několika způsoby, nejjednodušší je s využitím funkce type():

```
>>> type(a)
<class 'int'>

>>> type(99.9)
<class 'float'>

>>> type(,abc')
<class 'str'>
```

2.2 Rezervovaná slova

Rezervovaná slova mají definovaný význam a nelze je využít pro jiné účely, např. jména proměnných. Seznam rezervovaných slov:

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

2.3 Základní operátory

Jazyk Python podporuje standardní aritmetické operace jako sčítání, odčítání, násobení a dělení.

```
>>> 5 + 9
14
>>> 100 - 7
93
>>> 4 - 10
-6
>>> 6 * 7
42
```

Při dělení celých čísel je výsledkem reálné číslo:

```
>>> 9 / 5
1.8
```

Celočíselné dělení:

```
>>> 9 // 5
1
```

Dělení nulou:

```
>>> 5 // 0
ZeroDivisionError: division by zero
```

Zbytek po dělení:

```
>>> 9 % 5
4
```

2.4 Převod mezi datovými typy

Velmi často je nutné realizovat datové konvenze. Základní způsobu převodu mezi datovými typy jsou uvedeny v této kapitole.

2.4.1. Převod na int

Pro převod libovolného datového typu na celé číslo používáme funkci int(), která ponechá celou část čísla a zbytek zůstane nevyužitý.

```
>>> int(True)
1
>>> int(False)
0
>>> int(98.6)
98
>>> int(1.0e4)
10000
>>> int('99')
99

>>> int('99 bottles')
ValueError: invalid literal for int() with base 10: ,99 bottles'
```

2.4.2. Převod na float

Velmi podobně jako u celých čísel pracujeme s čísly reálnými.

```
>>> float(True)
1.0
>>> float(False)
0.0
>>> float(9)
98.0
```

2.5 Textové řetězce

V jazyce Python jsou textové řetězce sekvencí jednotlivých znaků a v jazyce Python jsou řetězce neměnné (immutable).

Možnosti pro tvorbu řetězců, které jsou rovnocenné:

```
>>> 'Python'
,Python'
>>> "Python"
,Python'
Spojování řetězců:
>>> bottles = 99
>>> base = ''
>>> base += 'current inventory: '
>>> base += str(bottles)
>>> base
'current inventory: 99'
```

2.5.1. Převod datových typů na text

Pro převod libovolného datového typu na textový řetězec používáme funkci str():

```
>>> str(98.6)
'98.6'
>>> str(True)
'True'
```

2.5.2. Přístup k jednotlivým znakům řetězce

K jednotlivým znakům přistupujeme pomocí indexů:

```
>>> letters = 'abcdefghijklmnopqrstuvwyz'
>>> letters[0]
'a'
>>> letters[1]
'b'
>>> letters[-1]
'z'
>>> letters[100]
IndexError: string index out of range
```

Z důvodu immutable u string není možná změna znaku v řetězci"

```
>>> name = 'Adam'
>>> name[0] = ,P`
TypeError: ,str` object does not support item assignment
```

Místo této operace je nutné použít jiný přístup, např. pomocí metody replace() kdy vznikne kopie původního objektu na níž dojde k požadované záměně:

```
>>> name = 'Adam'
>>> name.replace('A', 'P')
Pdam
```

2.5.3. Funkce len()

Slouží pro zjištění znaků v textovém řetězci:

```
>>> name = 'Adam'
>>> len(name)
4
```

2.5.4. Metoda split()

Specifická metoda pouze pro datový typ string.

```
>>> names = 'Adam, Vit'
>>> names.split(',')
[,Adam', ,Vit']
```

2.5.5. Metoda join()

Je opakem metody split, slouží ke spojování řetězců.

Samozřejmě je k dispozici celá řada dalších funkcí a metod pro práci s textovými řetězci, bohužel je představení všech nad rámec tohoto textu. Je ale velmi žádoucí si vyzkoušet i tyto užitečné metody: startswith(), endswith(), find(), rfind(), count() isalnum(), strip(), calitalize(), title(), upper(), lower(), swapcase(), center(), ljust(), rjust(), replace().

2.6 Datové struktury

Vestavěné datové struktury, které uchovávají elementy libovolného typu (libovolný objekt) v jazyce Python jsou:

- list (mutable)
- tuple (immutable)
- dictionary
- set

2.6.1. List (seznam)

Slouží k uchování elementů v určeném pořadí, jsou odděleny čárkou a uzavřeny pomocí hranatých závorek. Prvky listu nemusí být unikátní.

```
>>> empty_list = []
>>> weekday = [,Monday', ,Tuesday', ,Wednesday']
```

Stejně jako u datového typu str lze aplikovat funkci len() pro zjištění počtu prvků seznamu.

```
>>> empty_list = []
>>> weekday = [‚Monday‘, ‚Tuesday‘, ‚Wednesday‘]
>>> len(weekday)
3
```

Seznamy lze:

- modifikovat pomocí indexu prvku
- doplňovat o další prvek na konci seznamu – metoda `append()`
- spojovat pomocí metody `extend()` nebo operátoru +=
- přidávat prvek na daný index pomocí metody `insert()`
- mazat prvek na daném indexu pomocí metody `del()`
- mazat prvek s danou hodnotou pomocí metody `remove()`
- vyzvednout a smazat prvek v jednom kroku pomocí metody `pop()`
- získat index prvku s danou hodnotou pomocí metody `index()`
- testovat, zda se v seznamu nachází prvek s danou hodnotou pomocí operátoru in
- zjišťovat počet výskytů dané hodnoty v seznamu pomocí metody `count()`
- třídit prvky in place pomocí metody `sort()` nebo vracet kopii setříděného seznamu pomocí metody `sorted()`

2.6.2. Tuple

Tyto datové struktury jsou velmi podobné seznamu, jen jsou neměnné (immutable), tedy nelze přidávat, mazat nebo měnit prvky. V zásadě jde tedy o konstantní seznam.

```
>>> empty_tuple = ()
>>> empty_tuple
()
```

Rozdíly tuple oproti seznamu:

- tuple zabírá méně místa v paměti
- nelze omylem změnit prvky
- tuple lze využít jako klíče pro dictionary

Vzhledem k tomu, že seznamy a dictionary jsou podstatně častěji používané, tak jim je v tomto materiálu věnováno více prostoru.

2.6.3. Dictionary

Python slovník (dictionary) je kolekce prvků, která nám umožňuje ukládat data ve formě klíč: hodnota. Pořadí položek v dictionary není rozhodující a pro práci s jednotlivými položkami dictionary je rozhodující unikátní klíč.

Vytvoření slovníku s hodnotami:

```
>>> country_capitals = {"Německo": "Berlín", "Kanada": "Ottawa",
"Anglie": "Londýn"}
>>> print(country_capitals)
{'Německo': 'Berlín', 'Kanada': 'Ottawa', 'Anglie': 'Londýn'}
```

Pro přidání hodnoty do dictionary:

```
>>> print(country_capitals["Německo"])
Berlín
```

Pro přidání hodnoty do dictionary:

```
>>> country_capitals["Německo"] = "Hamburk"
```

Pro smazání hodnoty v dictionary:

```
>>> del country_capitals["Anglie"]
```

Iterace přes klíče a hodnoty:

```
>>> for key in country_capitals.keys():
>>>     print(key)
```

2.6.4. Set

Set (množina) je kolekce unikátních prvků v neutříděném pořadí (není určeno žádné konkrétní pořadí prvků).

Set je mutable (lze tedy modifikovat), ale nemůže obsahovat mutable prvky typu list, set, dictionary.

Základní vlastnosti set jsou následující:

- Unikátnost: každý prvek v set je unikátní.
- Neutříděnost: prvek v set nemá definované své pořadí v set.
- Neměnnost: prvek v set nemůže být změněn.

2.7 Shrnutí

V tomto výukovém bloku jsou uvedeny základní datové typy, datové struktury a další podstatné mechanismy s tímto tématem související v rámci vyššího programovacího jazyka Python:

- Datové typy: int, float, str, bool, None.
- Datové struktury: list, tuple, dictionary, set
- Rezervovaná slova
- Konverze mezi datovými typy
- Textové řetězce a základní operace s textovými řetězci

2.8 Otázky na procvičení

1. Jaký je rozdíl mezi datovými strukturami list a tuple?
2. K čemu slouží dictionary a jaké operace nad ním jsou k dispozici?
3. Jaké jsou možnosti pro konverzi mezi datovými typy?
4. Jaké základní metody jsou k dispozici nad datovým typem str?
5. K čemu slouží set a jaké operace nad ním jsou k dispozici?

2.9 Odkazy a další studijní materiály

[1] 5. Data Structures — Python 3.12.3 documentation

Financováno Evropskou unií
NextGenerationEU
Národní plán obnovy
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

3 Operátory, příkazy, funkce, notace

Studijní cíl

Tento blok je zaměřen na problematiku příkazů, operátorů, funkcí a notací v jazyce Python.

Doba nutná k nastudování 2– 2,5 hodiny

V programování jsou operátory základními symboly, které provádějí operace s proměnnými a hodnotami a umožňují úkoly, jako jsou aritmetické výpočty, logická porovnávání a další užitečné manipulace.

3.1 Operátory

V této tabulce jsou uvedeny základní operátory pro jazyk Python, jejich význam a příklad s výsledkem:

Operátor	Význam	Příklad	Výsledek
+	Součet	5 + 8	13
-	Rozdíl	90 – 10	80
*	Součín	4 – 7	28
/	Dělení na úrovni reálných čísel	7 / 2	3.5
//	Dělení na úrovni celých čísel	7 // 2	3
%	Zbytek po dělení	7 % 3	1
**	Exponent	3 ** 4	81
==	Je rovno	color == „red“	True, False
!=	Není rovno	color != „red“	True, False
<, <=, >, >=	Relační operátory		
in	Je členem		
and, or	Logické operátory		
not	Logická negace		

3.2 Příkazy

Mezi základní příkazy patří podmíněné příkazy a příkazy cyklu.

3.2.1. Podmíněné příkazy

Pro podmíněnou logiku můžeme použít **`if`**, **`elif`** a **`else`**, přičemž části elif a else jsou volitelné.

```
if x > 5:
    print("x je větší než 5")
elif x == 5:
    print("x je rovno 5")
else:
    print("x je menší než 5")
```

3.2.2. Cyklus while

Nejjednodušším typem cyklu v jazyce Python je while, který probíhá dokud platí podmínka definovaná za klíčovým slovem while.

```
count = 1
while count <= 5:
    print(count)
    count += 1
```

3.2.3. Cyklus for

V jazyce Python se velmi často používají iterátory, pomocí kterých procházíme kolekce bez znalosti jejich rozsahu a implementace.

Tedy místo tohoto přístupu:

```
weekdays = [,Monday‘, ,Tuesday‘, ,Wednesday‘]
current = 0
while current < len(weekdays):
    print(weekdays[current])
    current += 1
```

Je doporučovaný tento přístup:

```
for weekday in weekdays:
    print(weekday)
```

Pro cyklus s daným opakováním dle definovaného číselného rozsahu můžeme použít tyto možnosti:

```
for i in range(10):
    print(i)
for i in range(3, 11):
    print(i)
```

3.2.4. Příkaz break

Pokud chceme cyklus ukončit při výskytu určité podmínky tak používáme příkaz break:

```
while True:
    if color == "red"
        break
    print(color)
```

3.2.5. Příkaz continue

Pokud chceme v rámci cyklu skočit na další iteraci bez ukončení cyklu, je k dispozici příkaz continue:

```
while True:
    if color == "red"
        continue
    print(color)
```

3.3 Funkce

V jazyce Python lze kód zapisovat pomocí malých fragmentů pro vykonání pouze velmi omezených úloh. Cílem je i tyto fragmenty zapisovat tak, aby je bylo možné používat opakovaně. K tomu slouží funkce, které umožňují vkládat libovolný počet vstupních parametrů a vracet libovolný počet a typ návratových hodnot.

Pro definici funkce používáme rezervované slovo def, názvy funkcí musí splňovat stejné podmínky jako názvy proměnných (začínají písmenem nebo symbolem „_“):

```
>>> def do_nothing():
        pass


>>> def vypis_text():
        print('text')
```

3.3.1. Poziční parametry

Nejvíce používanými parametry jsou poziční parametry kdy jsou hodnoty argumentů zkopírované do hodnot parametrů v daném pořadí.

```
>>> def menu(polevka, hlavni_chod):
        return {'polevka': polevka, 'hlavni_chod': hlavni_chod}
>>> obed = menu("cockova", "ryba")
```

3.3.2. Pojmenované argumenty

Pro předcházení případných problémů s pozicemi parametrů je možné argumenty pojmenovat:

```
>>> menu(hlavni_chod='ryba', polevka='cockova')
```

Oba uvedené přístupy lze v zásadě libovolně kombinovat.

3.3.3. Default argumenty

Lze definovat default argumenty pro případy kdy je to vhodné.

```
def menu(polevka, hlavni_chod, dezert='pudink'):
     return    {'polevka':    polevka,    'hlavni_chod':    hlavni_chod,
               'dezert': dezert}
```

Pokud použijme skutečnou hodnotu argumentu, bude tato použita místo default hodnoty.

3.4 Slices

Slice notace se používá k extrahování částí sekvencí (jako jsou seznamy, řetězce nebo tuple).

Je k dispozici 5 základních slice konstruktů:

- [:] extrahuje celou sekvenci
- [start:] definujeme začátek
- [:end] definujeme konec (minus 1)
- [start:end] definujeme začátek i konec (minus 1)
- [start:end:step] definujeme začátek i konec (minus 1) s definovaným krokem

3.5 Comprehensions

Kromě slices nabízí Python několik dalších zápisů a funkcí, které zvyšují jeho expresivitu a všestrannost. Zde jsou některé z nich.

3.5.1. Comprehensions pro seznamy

Comprehension pro seznamy poskytují stručný způsob vytváření seznamů na základě existujících seznamů (nebo jiných iterovatelných objektů).

Syntaxe: [výraz for položku in iterable if podmínka].

Příklad:

```
numbers = [1, 2, 3, 4, 5]
squares = [x**2 for x in numbers]
```

Výsledek:

```
[1, 4, 9, 16, 25]
```

3.5.2. Comprehensions pro dictionary

Podobně jako comprehension pro seznamy, comprehension pro slovníky umožňuje vytvářet slovníky kompaktním způsobem.

Syntaxe:

```
{klíč: hodnota for položka in iterable if podmínka}
```

Příklad:

```
names = ['Alice', 'Bob', 'Charlie']
name_lengths = {name: len(name) for name in names}
```

Výsledek:

```
{'Alice': 5, 'Bob': 3, 'Charlie': 7}
```

3.6 Shrnutí

V tomto výukovém bloku jsou uvedeny operátory používané v programovacím jazyce Python, základní příkazy, práce s funkcemi a jejich parametry a slice a comprehension.

3.7 Otázky na procvičení

1. Jaké typy cyklů jsou k dispozici v jazyce Python?
2. Jaké části má podmíněný příkaz v jazyce Python?
3. Jaký je rozdíl mezi příkazem break a continue?
4. Jaké typy parametrů jsou k dispozici v jazyce Python?
5. Kde lze a s jakou výhodou využít comprehensions?

3.8 Odkazy a další studijní materiály

Financováno Evropskou unií
NextGenerationEU
Národní plán obnovy
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

4 Moduly, balíčky

Studijní cíl

Tento blok je zaměřen na problematiku modulů a balíčků v jazyce Python.

Doba nutná k nastudování 2– 2,5 hodiny

4.1 Moduly

Pro větší programy je vhodné a výhodné členění kódu do více souborů. Pod pojmem **modul** rozumíme soubor s Python kódem.

4.1.1. Příkaz import

Nejjednodušším způsobem použití modulu je pomocí příkazu import a názvu modulu, přičemž název modulu je název souboru a neuvádíme příponu .py.

Příklad hlavního programu (`meteo.py`):

```
import report

pocasi = report.get_pocasi()
print('Dnes je: ', pocasi)
```

V modulu report (`report.py`) musí být k dispozici funkce get_pocasi():

```
def get_pocasi():
    from random import choice
    mozne_pocasi=['dest', 'snih', 'slunecno', 'oblacno']

    return choice(mozne_pocasi)
```

Spuštění programu je možné tímto způsobem (pokud jsou oba soubory ve stejném adresáři):

```
$ python meteo.py
```

4.1.2. Příkaz import s pojmenováním

Pokud z nějakého důvodu potřebujeme změnit jméno pro importovaný modul, můžete to realizovat tímto způsobem s tím, že název můžeme volit libovolný, ideálně dle konvencí jazyka Python (v tomto případě „pocasi_report“):

```
import report as pocasi_report
```

4.1.3. Příkaz import s výběrem importovaných prvků

Pokud potřebujeme importovat jen vybrané části z celého modulu, lze to realizovat tímto snadným způsobem:

```
from report import get_pocasi
pocasi = get_pocasi()

print('Dnes je: ', pocasi)
```

Je samozřejmě možná také kombinace s pojmenováním:

```
from report import get_pocasi as g_poc
pocasi = g_poc()

print('Dnes je: ', pocasi)
```

4.2 Balíčky

Organizované větší množství modulů v určité hierarchii v jazyce Python označujeme jako balíčky. Pro ukázku rozšiřme předchozí příklad do více souborů – pro předpověď počasí budeme mít 2 moduly – denní a týdenní s tím, že každý z nich disponuje funkcí `dej_predpoved()`. Týdenní předpověď počasí vrátí předpověď pro dalších 7 dnů.

Pro správnou funkčnost těchto souborů musí být v uvažovaném adresáři alespoň prázdný soubor `__init__.py`. Za tohoto předpokladu bude Python uvažovat adresář za balíček.

Hlavní program v souboru `pocasi.py`:

```
from predpovedi import denni, tydenni

print('Predpoved pro dnesni den: ', denni.dej_predpoved())

for cislo, predpoved in enumerate(tydenni.dej_predpoved(), 1):
    print(cislo, predpoved)
```

Modul pro denní předpověď v souboru `denni.py`:

```
def dej_predpoved():
    return 'stejne jako vcera'
```

Modul pro týdenní předpověď v souboru `tydenni.py`:

```
def dej_predpoved():
    return   ['snih',   'vetrno',   'teplo',   'zima',   'dest',   'mlha,
             'slunecno']
```

4.3 Standardní knihovny

Jedním z častých tvrzení o jazyce Python je, že má celou řadu programovacích problémů již vyřešenou, tedy značnou knihovnu modulů, které provádějí mnoho užitečných úkolů a jsou vedeny odděleně pro snadné použití. Vždy když chceme napsat nějaký kód v jazyce Python, je výhodné si ověřit, zda neexistuje standardní modul, který již řeší danou problematiku. Často se ve standardní knihovně setkáme s celou řadou velmi zajímavých řešení.

4.3.1. OrderedDict

V části kde jsme se zabývali dictionary je uvedeno, že pořadí v dictionary není důležité a tedy ani předvídatelné. Tedy že klíče definované v pořadí a, b, c mohou být vráceny jako c, b, a apod.

Pokud bychom potřebovali mít garantované pořadí jako při definici dictionary, tak můžeme využít ze standardní knihovny OrderedDict() a z tuple vytvořit OrderedDict:

```
from collections import OrderedDict
country_capitals = OrderedDict([
    ("Německo", "Berlín"),
    ("Kanada", "Ottawa"),
    ("Anglie", "Londýn")
    ])

for stat in country_capitals:
    print(stat)
```

4.3.2. Funkce pprint()

Prozatím jsme pro výpis výsledků používali funkci print(), jejíž výstupy ale nejsou vždy úplně přehledné. Pro dosažení přehlednějšího výpisu lze použít např. pretty printer, tedy pprint(). Porovnejte oba způsoby výpisu, viz příklad níže.

```
from collections import OrderedDict
country_capitals = OrderedDict([
    ("Německo", "Berlín"),
    ("Kanada", "Ottawa"),
    ("Anglie", "Londýn")
  ])


>>> print(country_capitals)

>>> pprint(country_capitals)
```

4.3.3. Součet pomocí Counter

Standardní knihovna disponuje čítačem, který realizuje počítadlo:

```
>>> from collections import Counter
>>> mesta = ['Praha', 'Pardubice', 'Brno', 'Pardubice']
>>> mesta_counter = Counter(mesta)
>>> mesta_counter
```

Výsledek předchozího příkladu:

```
Counter({'Praha': 1, 'Pardubice': 2, 'Brno': 1})
```

Metoda most_common() vrací všechny prvky v sestupném pořadí:

```
>>> mesta_counter.most_common()
```

Výsledek předchozího příkazu:

```
[('Pardubice', 2), ('Praha', 1), ('Brno', 1)]
```

Counter má také implementované operace pro součet, rozdíl a množinové operace průnik (&) a sjednocení (|). Vyzkoušejte si je v rámci procvičení.

4.3.4. Knihovna itertools

Knihovna itertools obsahuje funkce zaměřené na iteraci při průchodu kolekcemi. Funkce vždy vrací element při každém průchodu cyklem for … in a jednotlivé mezivýsledky jsou uplatněny v rámci výpočtu. Dostupných funkcí je celá řada, zde jsou uvedeny pouze základní možnosti:

```
>>> import itertools
>>> for item in itertools.accumulate([1, 2, 3, 4]):
>>>     print(item)
```

Výsledek předchozího příkladu – je realizován součet prvků:

```
1
3
6
10
```

Do metody accumulate lze přidat jako druhý argument funkci, která má být uplatněna – zde součin aktuálního prvku s předchozím výsledkem.

```
>>> import itertools

>>> def multiply(a, b):
        return a * b

>>> for item in itertools.accumulate([1, 2, 3, 4], multiply):
>>>     print(item)
```

Výsledek předchozího příkladu – je realizován součet prvků:

```
1
2
6
24
```

4.4 Shrnutí

V tomto výukovém bloku je vysvětlena problematika modulů společně s příkazem import, problematika balíčků a standardních knihoven.

4.5 Otázky na procvičení

1. Co rozumíme pod pojmem modul v jazyce Python?
2. Co rozumíme pod pojmem balíček v jazyce Python?
3. K čemu slouží příkaz import a jaké varianty příkazu import jsou k dispozici?
4. K čemu slouží standardní knihovny?
5. Jaké příklady standardních knihoven znáte?

4.6 Odkazy a další studijní materiály

Financováno Evropskou unií
NextGenerationEU
Národní plán obnovy
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

5 Objektově orientované programování

Studijní cíl

Tento blok je zaměřen na problematiku objektově orientovaného programování v jazyce Python.

Doba nutná k nastudování 2– 2,5 hodiny

5.1 Třídy

Definice třídy je v programovacím jazyce Python velmi snadná, pass indikuje, že je třída prázdná:

```
>>> class Person():
        pass
```

Na základě takto definované třídy můžeme snadno vytvořit objekt:

```
>>> osoba = Person()
```

5.1.1. Konstruktor

Pro inicializaci hodnot objektů se standardně používají konstruktory, také v jazyce Python je samozřejmě možnost definovat konstruktor, který v nejjednodušší podobě vypadá takto:

```
>>> class Person():
        def __init__(self):
            pass
```

V jazyce Python je zajímavostí povinný argument `self`, který specifikuje, že se jedná o referenci na daný konkrétní objekt. Tento argument by měl vždy být na prvním místě v seznamu argumentů.

Následující konstruktor již více odpovídá reálnému kódu, kde je kromě argumentu self je také argument pro jméno osoby.

```
>>> class Person():
        def __init__(self, name):
            self.name = name
```

Na základě takto definovaného konstruktoru můžeme vytvořit objekt s definovaným jménem:

```
>>> osoba = Person('Jiri Novak')
```

5.1.2. Přístup k instančním proměnným

Pro přístup k instančním proměnným používáme tečkovou notaci, viz následující příklad:

```
>>> print('jmeno osoby: ', osoba.name)
```

5.2 Dědičnost

Pro vytvoření třídy na základě dříve definované třídy s možností rozšíření nebo změn je samozřejmě k dispozici dědičnost. Takto vytvořená třída automaticky využívá kód z děděné třídy. Potomka vytvoříme tak, že při definici třídy za název třídy do závorky zapíšeme označení předka.

Dědičnost v jazyce Python zapíšeme tímto způsobem, pro víceslovné názvy používáme CamelCase:

```
>>> class Auto():
        pass

>>> class NakladniAuto(Auto):
        pass
```

Následně můžeme vytvořit objekty tímto způsobem:

```
>>> auto = Auto()
>>> nakladni_auto = NakladniAuto():
```

5.2.1. Definice metod

Definice metod v rámci třídy je stejná jako definice konstruktoru, jen nejsme nijak svázáni při volbě názvu metody.

```
>>> class Auto():
        def pozdrav():
            print('Jsem auto')
```

5.2.2. Překrývání metod

Samozřejmě je také k dispozici překrývání metod

```
>>> class Auto():
        def pozdrav():
            print('Jsem auto')

>>> class NakladniAuto():
        def pozdrav():
            print('Jsem nakladni auto')
```

Volání připravených metod na objektech:

```
>>> auto = Auto()
>>> nakladni_auto = NakladniAuto()

>>> auto.pozdrav()
Jsem auto

>>> nakladni_auto.pozdrav()
Jsem nakladni auto
```

5.2.3. Volání metod předka

K volání prvků předka slouží klíčové slovo super:

```
>>> class Auto():
        def __init__(self, name):
            self.name = name

>>> class NakladniAuto():
        def __init__(self, name, pocet_naprav):
            super().__init__(name)
            self.pocet_naprav = pocet_naprav
```

5.2.4. Přístupová práva

V jazyce Python jsou všechny atributy a metody veřejné a pokud bychom potřebovali omezit přístup, tak musíme využít property nebo dekorátor.

Řešení pomocí property vypadá následovně:

```
>>> class Auto():
        def __init__(self, name):
            self.hidden_name = name
        def get_name(self):
            return self.hidden_name
        def set_name(self, input_name)
            print('v metode set')
            self.hidden_name = input_name
        name = property(get_name, set_name)
```

Následně při pokusu o přímý přístup k atributu name dojde k zavolání metody set_name():

```
>>> auto = Auto('Skoda Octavia')
>>> auto.name = 'Skoda Fabia'
v metode set
```

Stejného výsledku dosáhneme s využitím dekorátoru:

```
>>> class Auto():
        def __init__(self, input_name):
            self.hidden_name = input_name
        @property
        def name(self):
            print('v metode get')
            return self.hidden_name
        @name.setter
        def name(self, input_name)
            print('v metode set')
            self.hidden_name = input_name
```

Následně při pokusu o přímý přístup k atributu name dojde k zavolání metody set_name():

```
>>> auto = Auto('Skoda Octavia')
>>> auto.name = 'Skoda Fabia'
v metode set

>>> auto.name
v metode get
Skoda Fabia
```

5.2.5. Zabezpečení přístupu pomocí jmenné konvence

Jazyk Python disponuje také jmennou konvencí pro omezení přístupu k atributům a to pomocí názvu začínajícího na „__“

```
>>> class Auto():
        def __init__(self, input_name):
            self.__name = input_name
        @property
        def name(self):
            print('v metode get')
            return self.__name
        @name.setter
        def name(self, input_name)
            print('v metode set')
            self.__name = input_name
```

Následně při pokusu o přímý přístup k atributu name dojde k zavolání metody set_name():

```
>>> auto = Auto('Skoda Octavia')
>>> auto.name
v metode get
Skoda Octavia

>>> auto.name = 'Skoda Fabia'
v metode set

>>> auto.name
'Skoda Fabia'


>>> auto.__name
AttributeError: 'Auto' object has no attribute '__name'
```

5.2.6. Proměnné a metody třídy

V jazycích jako je C++ nebo Java používáme označení statické proměnné nebo statické metody. V jazyce Python toto označení nepoužíváme, ale tento koncept jaké k dispozici jak pro proměnné (zde proměnná pocet_aut), tak metody (zde metoda get_pocet_aut).

```
>>> class Auto():

        pocet_aut = 0

        def __init__(self, name):
            self.name = name
            Auto.pocet_aut += 1

        @classmethod
        def get_pocet_aut(cls):
            return cls.pocet_aut

>>> a1 = Auto('Skoda Octavia')
>>> a2 = Auto('Skoda Octavia')
>>> a3 = Auto('Skoda Octavia')

>>> Auto.get_pocet_aut()
3
```

5.3 Shrnutí

V tomto výukovém bloku je představen koncept objektově orientovaného programování v jazyce Python se všemi podstatnými prvky (konstruktory, metody, dědičnost, volání metod předka, přístupová práva, proměnné a metody třídy).

5.4 Otázky na procvičení

1. Jak v jazyce Python definujeme třídy a jaké prvky může obsahovat?
2. Jaká platí pravidla pro zápis konstruktorů v jazyce Python?
3. Jakým způsobem zapisujeme dědičnost v jazyce Python?
4. Jakým způsobem můžeme volat metodu předka v jazyce Python?
5. Jaké jsou možnosti omezení přístupu k atributům a metodám objektu v jazyce Python?
6. Jak lze definovat proměnné a metody třídy v jazyce Python?

5.5 Odkazy a další studijní materiály

[1] https://docs.python.org/3/tutorial/classes.html

Financováno Evropskou unií
NextGenerationEU
Národní plán obnovy
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

6 Objektově orientované programování – magické metody, asociace

Studijní cíl

Tento blok je zaměřen na problematiku objektově orientovaného programování v jazyce Python s důrazem na magické metody.

Doba nutná k nastudování 2– 2,5 hodiny

6.1 Magické metody

Magické metody jsou speciální metody, které umožňují přizpůsobit chování definovaných tříd. Názvy takových metod začínají a končí pomocí sekvence „__“. V minulém bloku jsme se již s jednou takovou metodou setkali, šlo o metodu „`__init__()`“. Podobným způsobem můžeme definovat celou řadu dalších metod.

Následující příklad definuje metodu equals pro porovnání dvou slov, přičemž považujeme dvě slova za stejná i v případě pokud mají různě velká písmena.

```
>>> class Slovo():
        def __init__(self, text):
            self.text = text

        def equals(self, slovo2):
            return self.text.lower() == slovo2.text.lower()


>>> prvni = Slovo('ha')
>>> druhe = Slovo('HA')
>>> treti = Slovo('eh')

>>> prvni.equals(druhe)
True
```

Pro výše uvedené řešení jsme museli definovat metodu equals a také tuto metodu zavolat při použití (porovnání slov). Samozřejmě by bylo zajímavější řešení pokud by stačilo použít operátor porovnávající slova „==“ tak, jak běžně porovnáváme vestavěné datové typové v jazyce Python.

K tomuto účelu nám poslouží tzv. magická metoda „`__eq__()`“:

```
>>> class Slovo():
        def __init__(self, text):
            self.text = text

        def __eq__(self, slovo2):
            return self.text.lower() == slovo2.text.lower()
```

Následně takto připravenou metodu vyzkoušíme:

```
>>> prvni = Slovo('ha')
>>> druhe = Slovo('HA')
>>> treti = Slovo('eh')

>>> prvni == druhe
True
```

Pro přehled magických metod **pro porovnávání objektů** nám poslouží následující tabulka:

`__eq__(self, other)`	`self == other`
`__ne__(self, other)`	`self != other`
`__lt__(self, other)`	`self < other`
`__gt__(self, other)`	`self > other`
`__le__(self, other)`	`self <= other`
`__ge__(self, other)`	`self >= other`

Pro přehled magických metod **pro matematické operace** nám poslouží následující tabulka:

`__add__(self, other)`	`self + other`
`__sub__(self, other)`	`self - other`
`__mul__(self, other)`	`self * other`
`__floordiv__(self, other)`	`self // other`
`__truediv__(self, other)`	`self / other`

`__mod__(self, other)`	`self % other`
`__pow__(self, other)`	`self ** other`

Pro přehled magických metod **pro další operace** nám poslouží následující tabulka, z níž __str__() slouží k textové reprezentaci objektu, k interaktivnímu interpreteru __repr__() a __len__() pro vrácení počtu prvků objektu.

`__str__(self)`	`str(self)`
`__repr__(self)`	`repr(self)`
`__len__(self)`	`len(self)`

Pro bližší představu o těchto dalších operacích nám poslouží následující příklad:

```
>>> class Slovo():
        def __init__(self, text):
            self.text = text

        def __eq__(self, slovo2):
            return self.text.lower() == slovo2.text.lower()

        def __str__(self):
            return self.text
        def __repr__(self):
            return 'Slovo("'self.text'")'

>>> prvni = Slovo('ha')
>>> prvni
Slovo(„ha“)
>>> print(prvni)
ha
```

6.2 Asociace

Pro vytvoření asociační vazby použijeme následující zápis:

```
>>> class Volant():
        def __init__(self, nazev):
            self.nazev = nazev
        def __str__(self):
            return self.nazev

>>> class Motor():
        def __init__(self, nazev):
            self.nazev = nazev
        def __str__(self):
            return self.nazev


>>> class Auto():
        def __init__(self, volant, motor):
            self.volant = volant
            self.motor = motor
        def about(self):
            print('Toto auto ma ', volant, ' volant a ', motor, ' motor')


>>> volant = Volant('V1')
>>> motor = Motor('M1')

>>> auto = Auto(volant, motor)
>>> auto.about()
Toto auto ma  V1  volant a  M1  motor
```

6.3 Shrnutí

V tomto výukovém bloku byly představeny magické metody pro přizpůsobení chování objektů i s příklady. Dalším demonstrovaným tématem je asociace mezi objekty.

6.4 Otázky na procvičení

1. Jaké magické metody lze využít pro porovnání objektů?
2. Jaké magické metody lze využít pro matematické operace?
3. Jaké magické metody lze využít pro textovou reprezentaci objektu?
4. Jaký je rozdíl mezi magickou metodou str() a repr()?
5. K čemu slouží asociace mezi objekty a jak ji lze zapsat v jazyce Python?

6.5 Odkazy a další studijní materiály

[1] https://docs.python.org/3/library/operator.html#operator.eq

Financováno Evropskou unií
NextGenerationEU
Národní plán obnovy
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

7 Výjimky, testování

Studijní cíl

Tento blok je zaměřen na problematiku výjimek a jednotkového testování v jazyce Python.

Doba nutná k nastudování 2– 2,5 hodiny

7.1 Výjimky

V jazyce Python, stejně jako i v některých jiných vyšších programovacích jazycích, se uplatňuje používání výjimek pokud je spuštěn kód, při němž dojde k výskytu chyby. Již i v předchozích výukových blocích jsme se s výskytem setkali (např. při práci se seznamem).

Je dobrou praxí zachytávat výjimky všude tam, kde hrozí riziko výskytu výjimky. Tento přístup je vhodný s ohledem na poskytnutí informace uživateli – že došlo k chybě v programu a také program ukončit vhodným způsobem, nikoliv pádem programu.

Pokud dojde k výskytu výjimky v nějaké funkci, a tato výjimka není přímo ve funkci zachycena, je tato výjimka předávána dále směrem k volajícím funkcím dokud není zachycena odpovídajícím handlerem. Pokud nevytvoříme vlastní handler, Python vypíše chybovou zprávu a doplňkové informace o místu výskytu chyby a dojde k zastavení běhu programu:

```
>>> short_list = [1, 2, 3]
>>> position = 5
>>> short_list[position]
...
IndexError: list index out of range
```

Je zřejmé, že je výhodnější tento potenciálně problematický kód obalit vlastním kódem pro možnost zachycení výjimky:

```
>>> short_list = [1, 2, 3]
>>> position = 5
>>> try:
        short_list[position]
    except:
        print('Je nutne zadat pozici mezi 0 a ', len(short_list)-1)
```

Po spuštění získáme tento výstup:

```
Je nutne zadat pozici v listu mezi 0 a 2
```

7.1.1. Definice vlastních výjimek

V jazyce Python je výjimka objektem, proto je pro vytváření vlastních výjimek nutné definovat vlastní třídu, která je potomkem třídy Exception a takto definovanou výjimku můžeme následně použít

```
>>> class MyException(Exception):
        pass
```

Zde mějme příklad na procházení listu se slovy, u kterých musí být použita pouze písmena malé abecedy. Pokud dojde k výskytu slova nesplňujícího tuto podmínku, dojde k zachycení výjimky:

```
>>> class UppercaseException(Exception):
        pass


>>> words = [‘one’, ‘two’, ‘THREE’]
>>> for word in words:
        if word.isupper():
            Raise UppercaseException(word)
```

7.2 Jednotkové testování

Z ostatních předmětů již známe význam a problematiku testování, v této kapitole si ukážeme jak realizovat jednotkové testy v jazyce Python.

Ve standardních knihovnách jazyka Python jsou dva testovací balíčky:

- unittest
- doctest

7.2.1. Testování pomocí unittest

Jako jednoduchý příklad zvolme metodu, která převádí zadaný text na text kde každé slovo začíná velkým písmenem a tento kód mějme uložen v souboru „kap.py“:

```
def preved_na_kapitalky(text):
    return text.capitalize()
```

Pro otestování vytvoříme soubor „test_kap.py“ s následujícím obsahem, přičemž metoda setUp() je volána před realizací každé testovací metody a metoda tearDown() po každé testovací metodě. Význam těchto metod je alokovat a uvolňovat externí zdroje jako např. přístup k datům v DB apod.

```
import unittest
import kap

class TestKap(unittest.TestCase):

    def setup(self):
        pass

    def tearDown(self):
        pass

    def test_jedno_slovo(self):
        text = "slovo"
        vysledek = kap.preved_kapitalky(text)

        self.assertEqual(result, "Slovo")

    def test_vice_slov(self):
        text = "vice slov v testu"
        vysledek = kap.preved_na_kapitalky(text)

        self.assertEqual(vysledek, "Vice Slov V Testu")


if __name__ == '__main__':
    unittest.main()
```

Po spuštění testu získáme následující výsledek, přičemž problém je ten, že metoda capitalize() vždy mění písmeno na velké pouze v prvním slově daného textového řetězce:

```
Ran 2 tests in 0.005s

FAILED (failures=1)

Vice Slov V Testu != Vice slov v testu

Expected :Vice slov v testu
Actual   :Vice Slov V Testu
<Click to see difference>

Traceback (most recent call last):
  File "test_kap.py", line 23, in test_vice_slov
    self.assertEqual(vysledek, 'Vice Slov V Testu')
AssertionError: 'Vice slov v testu' != 'Vice Slov V Testu'
- Vice slov v testu
?      ^    ^^^
+ Vice Slov V Testu
?      ^    ^^^
```

Změníme tedy metodu pro převod tak, aby využívala metodu title() místo původně uplatněné capitalize():

```
def preved_na_kapitalky(text):
    return text.title()
```

Po této změně získáváme následující výsledek testu:

```
Ran 2 tests in 0.003s

OK

Process finished with exit code 0
```

7.2.2. Testování pomocí doctest

Testování pomocí doctest probíhá přímo v metodě se zapsanou logikou a to v rámci dokumentačních komentářů kam zapisujeme jak volané funkce, tak očekávané hodnoty. Pokud test dopadne v pořádku, tak doctest nevrací žádnou informaci:

```
def preved_na_kapitalky(text):
    """
    >>> preved_na_kapitalky('slovo')
    'Slovo'
    >>> preved_na_kapitalky('vice slov v testu')
    'Vice Slov V Testu'
    """
    from string import capwords
    return capwords(text)

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Pokud bychom chtěli získat bližší informace v případě kladných výsledků testů, je nutné spustit kód s přepínačem „-v“ (verbose):

```
>>> kap_doctest.py -v
Trying:
    preved_na_kapitalky('slovo')
Expecting:
    'Slovo'
ok
Trying:
    preved_na_kapitalky('vice slov v testu')
Expecting:
    'Vice Slov V Testu'
ok
1 items had no tests:
    __main__
1 items passed all tests:
   2 tests in __main__.preved_na_kapitalky
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

7.3 Shrnutí

V tomto výukovém bloku je představen koncept výjimek a jednotkového testování včetně příkladů s využitím dvou balíčků – unittest a doctest.

7.4 Otázky na procvičení

1. Jaký koncept využíváme pro definici vlastních výjimek?
2. Jak lze ošetřit výskyt výjimky v Python kódu?
3. Jaké balíčky lze využít pro jednotkové testování Python kódu?
4. Jaký přístup využíváme při jednotkovém testování pomocí balíčku unittest?
5. Jaký přístup využíváme při jednotkovém testování pomocí balíčku doctest?

7.5 Odkazy a další studijní materiály

[1] https://docs.python.org/3/library/exceptions.html#Exception

Financováno Evropskou unií
NextGenerationEU
Národní plán obnovy
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

8 Správa balíčků pomocí Pip, virtuální prostředí

Studijní cíl

Tento blok je zaměřen na problematiku virtuálního prostředí v jazyce Python.

Doba nutná k nastudování 2– 2,5 hodiny

8.1 Správa balíčků pomocí Pip

Pip (Pip installs packages) je systém pro správu balíčků napsaný v jazyce Python. Používá se k instalaci a správě softwarových balíků. Python Software Foundation doporučuje používat pip pro instalaci aplikací vytvořených v jazyce Python a jejich závislostí během nasazení. Pip je nezbytnou součástí ekosystému jazyka Python a umožňuje vývojářům snadno spravovat knihovny a závislosti, které nejsou zahrnuty ve standardní knihovně jazyka Python. Alternativou je easy_install, ale preferovaným nástrojem je pip.

8.2 Virtuální prostředí

Virtuální prostředí řeší typický problém: umožňuje více Python projektům s požadavky na různé verze závislostí koexistovat v jednom operačním systému. Typicky spravujeme v rámci operačního systému větší počet projektů a některé z nich běžně vyžadují různé verze stejných knihoven. Právě problémy tohoto typu řešíme pomocí virtuálního prostředí.

Virtuální prostředí je tedy absolutně izolované prostředí pro každý projekt a toto izolované prostředí je v zásadě adresářem, který obsahuje kopii všech souborů potřebných pro běh Python programu – spustitelný Python, kopii standardních Python knihoven, kopii pip atd. Pokud pomocí pip nainstalujeme balíčky v rámci virtuálního prostředí, tak instalace těchto balíčků zůstává pouze v daném virtuálním prostředí bez ovlivnění jiných virtuálních prostředí.

8.2.1. Instalace virtuálního prostředí

Pokud již v rámci operačního systému není nástroj pro správu virtuálních prostředí k dispozici, tak je jeho instalace velmi snadná.

V zásadě pouze balíčky pip a virtualenv jsou jediné dva balíčky, které by měly být instalovány v rámci operačního systému globálně (tedy bez virtuálního prostředí).

Všechny ostatní balíčky instalujeme vždy v rámci virtuálního prostředí pro předcházení problémů se závislostmi na různých verzí balíčků.

Samotná instalace je takto snadná:

```
$ sudo pip install virtualenv
```

8.2.2. Vytvoření virtuálního prostředí

Pro vytvoření virtuálního prostředí je nezbytné disponovat nástrojem virtualenv. Následně se stačí přemístit do adresáře projektu kde má být virtuální prostředí vytvořeno a následně virtuální prostředí vytvořit – zadat název adresáře, v němž mají být soubory virtuálního prostředí vytvořeny (zde **env**).

```
$ cd myproject/
$ virtualenv env
```

Pro potřeby verzovacích systémů je vhodné adresář env ignorovat, tedy nezahrnovat pod správu.

8.2.3. Použití virtuálního prostředí

Nejjednodušším způsobem jak začít používat vytvořené virtuální prostředí je jeho aktivace pomocí připraveného skriptu activate, který se nachází v adresáři env/bin. Skript spustíme pomocí tohoto příkazu v adresáři projektu:

```
$ source env/bin/activate
```

Pokud potřebujeme ukončit práci ve virtuálním prostředí, tak je připraven skript deactivate:

```
$ deactivate
```

8.2.4. Instalace většího počtu závislostí

V případě většího množství závislostí je vhodné a výhodné tyto závislosti spravovat na úrovni textového souboru se zapsanými závislostmi. Tento textový soubor má název „requirements.txt“ a jeho obsah typicky vypadá následovně:

```
###### Požadované balíčky bez specifikace verze ######
nose
nose-cov
beautifulsoup4
#
###### Požadované závislosti se specifikací verzí ######
#   See https://www.python.org/dev/peps/pep-0440/#version-specifiers
docopt == 0.6.1             # Version Matching. Must be version 0.6.1
keyring >= 4.1.1            # Minimum version 4.1.1
coverage != 3.5             # Version Exclusion. Anything except version 3.5
Mopidy-Dirble ~= 1.1        # Compatible release. Same as >= 1.1, == 1.*
```

U zapsaných závislostí vidíme jak variantu bez specifikace závislostí, tak závislosti s definovanými verzemi s různými požadavky na verze.

Pomocí následujícího příkazu dojde k instalaci požadovaných závislostí z textového souboru:

```
$ pip install -r requirements.txt
```

8.2.5. Vytvoření seznamu závislostí

Pomocí příkazu pip freeze získáme seznam balíčků a jejich verzí nainstalovaných v daném virtuálním prostředí a ve formátu, který je běžně využíván v textovém souboru requirements.txt:

```
$ pip freeze
agate==1.6.0
agate-dbf==0.2.0
agate-excel==0.2.1
agate-sql==0.5.2
```

Tento seznam může být také uložen v textovém souboru requirements.txt, čehož dosáhneme pomocí následujícího příkazu:

```
$ pip freeze > requirements.txt
```

8.3 Shrnutí

V tomto výukovém bloku je představen koncept správy balíčků pomocí systému Pip a virtuálních prostředí, která je velmi výhodné využívat.

8.4 Otázky na procvičení

1. K čemu slouží virtuální prostředí v jazyce Python?
2. Jak lze spravovat balíčky a jejich verze v rámci Python projektů?
3. Jak nainstalovat, vytvořit a začít používat virtuální prostředí v Python projektu?
4. Jak lze vytvořit seznam závislostí pro daný projekt?
5. Jak lze nainstalovat hromadně všechny definované závislosti v projektu?

8.5 Odkazy a další studijní materiály

[1] https://docs.python.org/3/library/venv.html#index-0

Financováno Evropskou unií
NextGenerationEU
Národní plán obnovy
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

9 Databázové aplikace

Studijní cíl

Tento blok je zaměřen na problematiku vývoje databázových aplikací v jazyce Python.

Doba nutná k nastudování 2– 2,5 hodiny

V jazyce Python je k dispozici několik možných přístupů pro přístup k datům, která jsou uložena v databázi. Cílem tohoto bloku je představit základní možnosti v této oblasti.

9.1 DB API

DB API (Application Programming Interface) je množina funkcí, které můžeme využívat pro přístup k vybraným službám. DB API je standard v jazyce Python pro přístup k relačním databázovým systémům. Tímto způsobem lze vytvořit aplikaci, která je připravena na přístup k datům v několika databázových systémech (podobně jako např. v Java JDBC).

Mezi hlavní funkce DB API patří:

- connect() – vytvoření připojení k DB. Pomocí argumentů lze zadat další údaje jako např. uživatelské jméne, heslo, adresu serveru, případně další údaje.
- cursor() – vytvoření objektu typu kurzor pro správu dotazů.
- execute() – spuštění jednoho příkazu v DB
- fetchone(), fetchmany(), fetchall() – získání výsledků z operace execute()

9.1.1. SQlite – vytvoření tabulky

Pro jednoduchou ukázku vytvoření databáze a databázové tabulky s definovanými sloupci je výhodné použit SQlite, které je již svým způsobem vestavěné v jazyce Python. Obdobným způsobem bychom ale

mohli pracovat i s jinými DBMS, např. MySQL, PostgreSQL s tím, že dialekt jednotlivých systémů se ale samozřejmě odlišuje. Následující kód všechny tyto operace zabezpečí pro SQlite:

```
import sqlite3

conn = sqlite3.connect('studenti.db')
curs = conn.cursor()
curs.execute('''CREATE TABLE student
    (id INT PRIMARY KEY,
     jmeno VARCHAR(50),
     rocnik_zahajeni INT)''')
```

9.1.2. SQlite – vložení dat

Pro vložení dat do tabulky použijeme kurzor a jeho metodu execute() společně s příkazem SQL pro vkládání dat:

```
curs.execute('INSERT INTO student VALUES (1, "Jiri Novak", 2024)')
```

9.1.3. SQlite – vložení dat pomocí parametrizovaného dotazu

Bezpečnějším způsobem pro vkládání dat do databáze je tzv. parametrizovaný dotaz, kde značně omezíme práci s uvozovkami a také máme možnost předcházet bezpečnostním hrozbám v podobě SQL injection apod.

```
ins = 'INSERT INTO student (id, jmeno, rocnik_zahajeni) VALUES (?, ?, ?)'
curs.execute(ins, (1, "Jiri Novak", 2024))
```

9.1.4. SQlite – získání dat

Pro získání dat z tabulky použijeme kurzor a jeho metodu execute() společně s příkazem SQL pro získání dat:

```
curs.execute('SELECT * from student')
```

9.1.5. SQlite – ukončení připojení

Při ukončení připojení je vhodné ukončit práci s kurzorem i s připojením k DB:

```
curs.close()
conn.close()
```

9.2 SQLAlchemy

Jak již bylo uvedeno, každý databázový systém disponuje odlišným dialektem v rámci svých příkazů. Existuje celá řada knihoven, které se snaží o odstínění těchto odlišností, přičemž nejvíce populární z nich je SQLAlchemy.

V době psaní tohoto textu je byla aktuální verze SQLAlchemy 2.0.29 a tak jsou všechny příklady vytvořené právě s využitím této verze.

Nejedná se o standardní knihovnu a tak je nutné ji nainstalovat, ideálně do virtuálního prostředí, pomocí tohoto příkazu:

```
$ pip install sqlalchemy
```

S SQLAlchemy lze pracovat na několika úrovních:

- Nejnižší úroveň slouží pro zabezpečení přístupu do databáze, spouštění SQL příkazů apod. Tato úroveň je velmi podobná výše představenému DB-API.
- Další úroveň spočívá v SQL expression language, což je SQL builder pro Python
- Nejvyšší úroveň je ORM (objektově-relační mapování), která využívá SQL expression language a propojuje aplikaci s relačně orientovanými datovými strukturami.

9.2.1. SQLAlchemy – připojení

Pro připojení k vybrané DB je nutné uplatnit tzv. connection string, který vypadá následovně. Ovladač obvykle není nutné instalovat, je k dispozici v rámci instalace SQL Alchemy.

```
dialect + driver :// user : password @ host : port / dbname
```

Jednotlivé části connection string mají následující význam.

dialect – typ databáze

driver – ovladač pro zvolenou databázi

user, password – autentizace uživatele

host, port – umístění DB serveru v síti

dbname – k jaké databázi se připojit na DB serveru

Pro volbu dialect a ovladače slouží následující tabulka:

dialect	**driver**
sqlite	pysqlite (nemusí se uvádět)
mysql	mysqlconnector
mysql	pymysql
mysql	oursql
postgresql	psycopg2
postgresql	pypostgresql

9.2.2. SQLAlchemy – spouštění SQL příkazů pomocí SQL Expression Language

Z následujícího příkladu je zřejmé, že SQLAlchemy je výhodné používat pro abstrakci nad různými DBMS. Bez ohledu na použitý DBMS budou příkazy pro definici tabulky a sloupců vždy stejné, přičemž v tomto příkladu využíváme SQL Expression Language:

```
import sqlalchemy as sa

engine = sa.create_engine('sqlite://')

meta = sa.MetaData()

studenti = sa.Table('student', meta,
    sa.Column('id', sa.Integer, primary_key=True),
    sa.Column('jmeno', sa.String),
    sa.Column('rocnik_zahajeni', sa.Integer)
)

meta.create_all(engine)
```

Pro vložení dat do tabulky použijeme následující příkazy:

```
from sqlalchemy import insert

stmt = insert(studenti).values(id = 1, jmeno = "Jiri Novak",
rocnik_zahajeni = 2024)

with engine.connect() as conn:
    result = conn.execute(stmt)
    conn.commit()
```

Pro získání záznamů z DB použijeme tyto příkazy:

```
stmt = select(studenti)

with engine.connect() as conn:
    for row in conn.execute(stmt):
        print(row)
```

A výsledkem je výpis záznamů z tabulky:

```
(1, 'Jiri Novak', 2024)
(2, 'Karel Novy', 2023)
```

9.2.3. SQLAlchemy – objektově relační mapování

V předchozí kapitole byla věnována pozornost prostřední vrstvě přístupu SQLAlchemy s využitím SQL Expression language, přičemž objektově-relační mapper (ORM) také využívá SQL Expression language jen s tím rozdílem, že se snaží tuto vrstvu standardně skrývat.

V zásadě nadefinujeme třídy a ORM řeší jakým způsobem dostat data do databáze a následně z databáze.

Práce s SQLAlchemy a ORM vypadá následovně. DeclarativeBase odkazuje na kolekci MetaData, která je vytvořena automaticky za předpokladu, že neposkytneme nějakou externí explicitně. Kolekce MetaData je přístupná prostřednictvím DeclarativeBase.MetaData, pokud bychom chtěli znát detaily. Když vytváříme nové mapované třídy, každá z nich bude odkazovat na tabulku v této kolekci MetaData.

```
from sqlalchemy.orm import DeclarativeBase

class Base(DeclarativeBase):
    pass
```

Následně již můžeme přizpůsobit třídu pro automatické relační mapování s tím, že každý objekt bude záznamem v DB:

```
from sqlalchemy.orm import mapped_column
from sqlalchemy.orm import Mapped


class Student(Base):
    __tablename__ = "student"
    id: Mapped[int] = mapped_column(primary_key=True)
    jmeno: Mapped[str] = mapped_column(nullable=False)
    rocnik_zahajeni: Mapped[int]

    def __init__(self, id, jmeno, rocnik_zahajeni):
        self.id = id
        self.jmeno = jmeno
        self.rocnik_zahajeni = rocnik_zahajeni

    def __repr__(self):
        return "<Student({}, {}, {})>".format(self.id, self.jmeno,
                 self.rocnik_zahajeni)
```

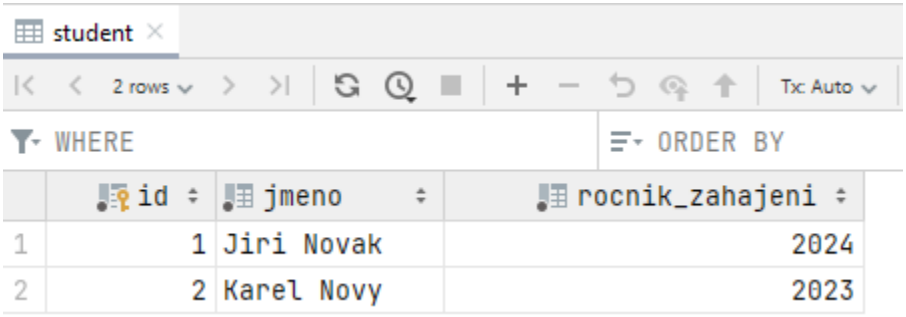
Pokud máme třídu pro ORM připravenou, vytvoříme databázový soubor, připojíme se k němu a pomocí session přidáme objekty do databáze:

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///studenti.db', echo=True)
Base.metadata.create_all(engine)

student1 = Student(1, "Jiri Novak", 2024)
student2 = Student(2, "Karel Novy", 2023)

Session = sessionmaker(bind = engine)
session = Session()

session.add(student1)
session.add(student2)
```

Následně zbývá realizovat commit do databáze a objekty jsou v DB uloženy:

```
session.commit()
```

Data máme v DB uložena v tabulce „student“:



Pro ověření přítomnosti dat v DB lze také použít příkazovou řádku následujícím způsobem:

```
$ sqlite3 studenti.db
$ .tables
student

$ select * from student;
1|Jiri Novak|2024
2|Karel Novy|2023
```

9.3 Shrnutí

V tomto výukovém bloku je představen koncept programování databázových aplikací s využitím dvou přístupů – DB API a SQLAlchemy.

9.4 Otázky na procvičení

1. Jaký je rozdíl při použití přístupů DB API a SQLAlchemy?
2. Jaké jsou hlavní funkce při využití DB API?
3. Jaké základní úrovně pro práci s DB lze využít v SQLAlchemy?
4. K čemu slouží ORM v SQLAlchemy?
5. Jaké RDBMS lze využít v rámci SQLAlchemy?

9.5 Odkazy a další studijní materiály

[1] https://docs.python.org/3/library/sqlite3.html

[2] https://www.sqlalchemy.org/

Financováno Evropskou unií
NextGenerationEU
Národní plán obnovy
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

10 Webové aplikace

Studijní cíl

Tento blok je zaměřen na problematiku vývoje webových aplikací v jazyce Python.

Doba nutná k nastudování 2– 2,5 hodiny

Pro základní vývoj webových aplikací jsou ve standardní knihovně jazyka Python dva základní balíčky:

- `http`, který obsahuje základní prvky pro http:
  - `client`
  - `server`
  - `cookies` a `cookiejar` pro správu a ukládání dat
- `urllib`, který funguje nad http a obsahuje:
  - `request`
  - `response`
  - `parse` pro zpracování jednotlivých částí URL

Jako jednoduchá ukázka možností těchto knihoven nám poslouží následující příklad, jehož výsledkem je objekt obsahující response ze serveru:

```
>>> import urllib.request as ur

>>> url = 'http://www.google.com'
>>> conn = ur.urlopen(url)
>>> print(conn)
<http.client.HTTPResponse object at 0x0000021B19654E80>
```

Pro zobrazení dat v response použijeme metodu read():

```
>>> data = conn.read()
>>> print(data)
```

V tomto příkladu došlo k otevření TCP/IP připojení k serveru, byl realizován HTTP request a byla vrácena HTTP response. Response obsahuje, kromě samotných dat ze stránky, také další data jako např. status kód (zde 200 – tedy OK):

```
>>> print(conn.status)
200
```

10.1 Balíček requests

Pro pokročilejší práci s request je výhodné nainstalovat balíček requests, který obsahuje celou řadu modulů, které ve standardní knihovně urllib nejsou zastoupeny.

Intalace, ideálně ve virtuálním prostředí, je velmi snadná:

```
$ pip install requests
```

Obdoba předchozího příkladu s využitím requests vypadá následovně s tím, že výpis textu nám poskytne obsah stránky:

```
>>> import requests

>>> url = 'http://www.google.com'
>>> resp = requests.get(url)
>>> resp
<Response [200]>

>>> print(resp.text)
```

10.2 Web servery

Vývojáři webových aplikací si jazyk Python velmi oblíbili pro psaní webových aplikací na straně serveru. To vedlo k vývoji celé řadě frameworků a knihoven právě pro vývoj webových aplikací.

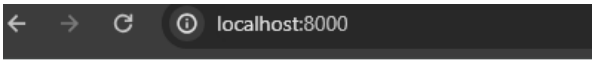
Součástí instalace Python je také jednoduchý web server, jehož spuštění je takto snadné:

```
$ python -m http.server
```

Po spuštění tohoto příkazu získáme následující informaci:

```
Serving HTTP on :: port 8000 (http://[::]:8000/) ...
```

Po zadání této adresy do adresního řádku prohlížeče získáme obsah aktuálního adresáře:



Directory listing for /

Současně také dojde v terminálu k zobrazení informace o přístupu na server:

```
Serving HTTP on :: port 8000 (http://[::]:8000/) ...
::1 - - [27/Jan/2024 11:53:37] "GET / HTTP/1.1" 200 -
```

10.3 Nástroje pro vývoj webových aplikací v jazyce Python

Pro jazyk Python je k dispozici celá řada knihoven a frameworků pro vývoj webových aplikací, které zabezpečují správu requestů a response s tím, že mezi základní součásti patří:

- Routes – interpretace URL
- Templates – kompilace dat ze serveru do podoby HTML
- Autentizace a autorizace – práce s uživateli, jejich hesly, oprávněními apod.
- Session – ukládání uživatelských dat během používání webové aplikace

Mezi velmi populární frameworky v jazyce Python pro vývoj webových aplikací patří zejména:

- Bottle – micro web framework
- Flask – micro web framework
- Django – high-level web framework
- Pyramid – micro web framework
- web2py – full-stack framework
- CherryPy – micro web framework

10.4 Základy Flask

Flask patří mezi micro frameworky, kde je ale zabezpečena snadná rozšiřitelnost v podobě např. Facebook autentizace, integrace databáze atd.

Cílem této kapitoly není představit kompletní příručku nebo tutoriál pro Flask, takové materiály jsou k dispozici přímo na stránce projektu (https://flask.palletsprojects.com/en/3.0.x/). Cílem této kapitoly je na jednoduchém příkladu ukázat základní principy vývoje s využitím tohoto frameworku.

Flask také obsahuje WSGI knihovnu (specifikace web serveru) a jinja2 template knihovnu.

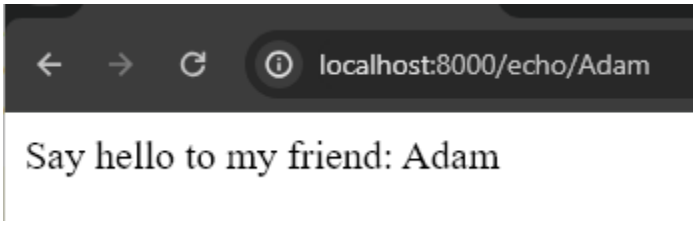
Instalace, ideálně ve virtuálním prostředí, je velmi snadná:

```
$ pip install flask
```

Příklad velmi jednoduché webové aplikace je uveden zde:

```
from flask import Flask

app = Flask(__name__, static_folder='.', static_url_path='')

@app.route('/')
def home():
    return app.send_static_file('index.html')

@app.route('/echo/<thing>')
def echo(thing):
    return "Say hello to my friend: %s" % thing

app.run(port=8000, debug=True)
```

Argument debug vytváří debug stránku v případě HTTP error, výsledek zadání níže uvedené URL je zde:



Pro ukázku Jinja2 šablon nám poslouží následující drobná úprava spočívající v uplatnění funkce **render_template()** a vytvoření jednoduchého HTML souboru umístěného **do adresáře „templates“**.

```
from flask import Flask, render_template
@app.route('/echo/<thing>')
def echo(thing):
    return render_template('flask1.html', thing = thing)
```

Soubor „flask1.html“ umístěný v adresáři „templates“:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Flask Example</title>
</head>
<body>
    Say hello to my friend:{{ thing }}
</body>
</html>
```

Výsledem je stejná response jako v předchozím příkladu s tím rozdílem, že nyní jde o dynamicky generovaný obsah s využitím šablony.

10.5 Shrnutí

V tomto výukovém bloku je představena problematika vývoje webových aplikací, přičemž jsou v tomto bloku představeny základní balíčky pro tuto oblast – http, urllib, requests a následně základy populárního frameworku Flask se systémem šablon Jinja2.

10.6 Otázky na procvičení

1. K čemu slouží balíček http?
2. K čemu slouží balíček urllib?
3. K čemu slouží balíček requests?
4. Jaké znáte frameworky pro vývoj webových aplikací v jazyce Python?
5. Jaký je význam šablonovacích systémů?

10.7 Odkazy a další studijní materiály

[1] https://flask.palletsprojects.com/en/3.0.x/

Financováno Evropskou unií
NextGenerationEU
Národní plán obnovy
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

11 Balíčky pro získání rozsáhlých dat

Studijní cíl

Tento blok je zaměřen na problematiku balíčků pro získání rozsáhlých dat v jazyce Python.

Doba nutná k nastudování 2– 2,5 hodiny

Jazyk Python je velmi často využíván pro zpracování rozsáhlých dat (big data). Způsobů, jak rozsáhlá data získat je celá řada, v tomto bloku jsou uvedeny pouze základní možnosti. Získání dat z databází je uvedeno v bloku 9 tohoto kurzu, v tomto bloku tedy budou uvedeny možnosti získání dat z textových souborů a z online zdrojů.

11.1 Získání dat z textových souborů

Pro čtení dat z textového souboru je nutné nejprve vytvořit objekt reprezentující textový soubor:

```
# 'r' - pro cteni
soubor_pro_cteni = open('soubor.txt', 'r')

# 'w' - pro zapis
soubor_pro_zapis = open('soubor.txt', 'w')

# 'a' - pro pridani na konec souboru
soubor_pro_doplneni = open('soubor.txt', 'a')
```

Poměrně často dochází k opomenutí uzavření souboru po dokončení práce se souborem. Proto je bezpečnější používat následující přístup pro automatické uzavření souboru po dokončení operací se souborem:

```
with open('soubor.txt', 'r') as f:
    data = funkce_pro_ziskani_dat_ze_souboru(f)

# v teto casti kodu je jiz soubor uzavren a pracujeme pouze s daty
zpracovani_dat(data)
```

Pro přečtení celého obsahu textového souboru můžeme použít tento kód:

```
with open('soubor.txt', 'r') as f:
    for line in f:
        print(line)
```

Pro přečtení celého csv souboru s následujícími hodnotami:

```
6/20/2014;AAPL;0.91
6/20/2014;MSFT;41.68
6/20/2014;FB;64.5
6/19/2014;AAPL;91.86
6/19/2014;MSFT;41.51
6/19/2014;FB;64.34
```

Použijeme tento kód:

```
import csv

with open('soubor_ceny.txt', 'r') as f:
    reader = csv.reader(f, delimiter=';')

    for row in reader:
        date = row[0]
        symbol = row[1]
        price = float(row[2])

        print(row)
```

Pokud máme k dispozici data včetně záhlaví hodnot, je výhodnější použít DictReader s tím, že každý řádek bude reprezentován dictionary:

```
datum;symbol;cena
6/20/2014;AAPL;0.91
6/20/2014;MSFT;41.68
6/20/2014;FB;64.5
6/19/2014;AAPL;91.86
6/19/2014;MSFT;41.51
6/19/2014;FB;64.34
```

```
import csv

with open('soubor_ceny_zahlavi.txt', 'r') as f:
    reader = csv.DictReader(f, delimiter=';')

    for row in reader:
        date = row['datum']
        symbol = row['symbol']
        price = float(row['cena'])

        print(row)
```

11.2 Získání dat z API, JSON

Značná část webových aplikací a služeb poskytuje pro přístup k datům API (Application Programming Interface), kde jsou data přístupná v definovaných strukturách. Nejčastějším formátem takto zpřístupněných dat je JSON (JavaScript Object Notation), což je formát velmi podobný datové struktuře dictionary v jazce Python.

Příklad dat ve formátu JSON:

```
{ "title" : "Data Science Book",
  "author" : "Josef Novák",
  "publicationYear" : 2024,
  "topics" : [ "data", "science", "data science"] }
```

Parsování JSON dat je následně takto snadnou záležitostí:

```
import json
serialized = """{ "title" : "Data Science Book",
                  "author" : " Josef Novák",
                  "publicationYear" : 2024,
                  "topics" : [ "data", "science", "data science"] }"""

deserialized = json.loads(serialized)

print(deserialized)
```

Velmi často je také nutné data ukládat do JSON, tedy převod z objektu do JSON. K tomu slouží funkce dumps():

```
data_json = json.dumps(deserialized)
print(data_json)
```

11.3 Získání dat pomocí Pandas

Kromě výše uvedených přístupů je k dispozici celá řada knihoven pro získávání a zpracování dat. Jednou z velmi populárních knihoven je Pandas, což je nástroj se značným množstvím tříd a funkcí.

11.3.1.Pandas DataFrame

Jádrem knihovny Pandas je třída DataFrame pro organizaci dat v tabulkové podobě (rozdělení do sloupců, záhlaví sloupců, indexace sloupců apod.).

Pro práci s Pandas je nutná jednoduchá instalace, opět nejlépe ve virtuálním prostředí:

```
$ pip install pandas
```

Pro vytvoření DataFrame nám poslouží následující příklad kde pomocí listu definujeme data pro DataFrame, označení záhlaví sloupce a také popis (index) jednotlivých dat:

```
import pandas as pd

df = pd.DataFrame([10, 20, 30, 40],
                  columns = ['cisla'],
                  index = ['a', 'b', 'c', 'd'])
print(df)
```

Získáme následující výstup:

```
   cisla
a     10
b     20
c     30
d     40
```

11.3.2.Získání dat pomocí Pandas

Výhodou Pandas je také možnost získání dat z celé řady datových formátů kam patří:

- SQL – pd.read_sql()
- XLS, XLSX – pd.read_excel()
- JSON – pd.read_json()
- HTML – pd.read_html()

Na tomto příkladu vidíme, že využití Pandas je pro čtení z CVS souborů jednodušší variantou dříve představených přístupů:

```
import pandas as pd

df = pd.read_csv('soubor_ceny.cvs')
print(df)
```

Výsledkem této sekvence příkazů je tento výstup:

```
    6/20/2014;AAPL;0.91
0  6/20/2014;MSFT;41.68
1     6/20/2014;FB;64.5
2  6/19/2014;AAPL;91.86
3  6/19/2014;MSFT;41.51
4    6/19/2014;FB;64.34
```

Podobným způsobem můžeme také data zapsat do CSV souborů nebo i do jiných formátů, např.:

```
df.to_csv('ceny.csv', sep=';')
```

Výsledkem je soubor na disku s názvem „ceny.csv“ s tímto obsahem:

```
;"datum;symbol;cena"
0;"6/20/2014;AAPL;0.91"
1;"6/20/2014;MSFT;41.68"
2;"6/20/2014;FB;64.5"
3;"6/19/2014;AAPL;91.86"
4;"6/19/2014;MSFT;41.51"
5;"6/19/2014;FB;64.34"
```

Obdobným způsobem bychom získali data i z jiných zdrojů, např. SQL, XLS.

11.4 Shrnutí

V tomto výukovém bloku jsou představeny základní možnosti pro získání rozsáhlých z různých zdrojů, např. z textových souborů, z API ve formátu JSON a také pomocí k tomu určené knihovny Pandas (data z SQL, XLS, JSON, HTML apod.).

11.5 Otázky na procvičení

1. Jakým způsobem provedeme přečtení textového souboru?
2. Jakým způsobem provedeme zápis do textového souboru?
3. Jakým způsobem lze získat data z API?
4. Jaká je výhoda při získání dat pomocí knihovny Pandas?
5. Jaké formáty dat jsou podporovány v rámci knihovny Pandas?

11.6 Odkazy a další studijní materiály

[1] https://pandas.pydata.org/

Financováno Evropskou unií
NextGenerationEU
Národní plán obnovy
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

12 Balíčky pro zpracování a analýzu rozsáhlých dat

Studijní cíl

Tento blok je zaměřen na problematiku balíčků pro získání zpracování a analýzu dat v jazyce Python.

Doba nutná k nastudování 2– 2,5 hodiny

Jazyk Python je velmi často využíván pro zpracování rozsáhlých dat (big data) a v tomto bloku jsou uvedeny základní možnosti pro tuto oblast.

V předchozím bloku jsme si představili základní koncept Pandas DataFrame a jeho dostupné základní operace pro čtení a zápis dat. V této kapitole se zaměříme na zpracování dat.

12.1 Zpracování dat pomocí Pandas

Práce s DataFrame je následně stejná jako s jakýmkoliv jiným objektem v jazyce Python, využíváme tedy proměnné a metody DataFrame:

```
import pandas as pd

>>> df = pd.DataFrame([10, 20, 30, 40],
                      columns = ['cisla'],
                      index = ['a', 'b', 'c', 'd'])
>>> df


   cisla
a     10
b     20
c     30
d     40


>>> df.index
index(['a', 'b', 'c', 'd'], dtype='object')
```

Získání sloupců a indexu:

```
>>> df.columns
Index(['cisla'], dtype='object')
```

Získání hodnot pro index „c“:

```
>>> df.loc['c']
Name: c, dtype: int64
```

Získání součtu pro sloupec:

```
>>> df.sum()
cisla    100
dtype: int64
```

Aplikace funkce apply pro výpočet nové hodnoty a uplnění druhé mocniny původních hodnot:

```
>>> df_mocnina = df.apply(lambda x: x ** 2)
>>> df_mocnina
cisla
a    100
b    400
c    900
d   1600
```

Rozšíření dat o další sloupec:

```
>>> df.['floats'] = (1.5, 2.5, 3.5, 4.5)
df

   cisla  floats
a     10     1.5
b     20     2.5
c     30     3.5
d     40     4.5
```

Výpis pouze vybraného sloupce:

```
>>> df['floats']
a    1.5
b    2.5
c    3.5
d    4.5
```

Přídání další sloupce může být realizováno také s přidáním dalšího DataFrame:

```
>>> df['jmena'] = pd.DataFrame(['Pavel', 'Lenka', 'Josef', 'Eliska'],
                               index = ['d','a','b','c']))
>>> df
   cisla  floats   jmena
a     10     1.5   Lenka
b     20     2.5   Josef
c     30     3.5  Eliska
d     40     4.5   Pavel
```

Pro přidání dalšího řádku lze zvolit tento postup:

```
>>> novy_zaznam = pd.DataFrame([{'cisla': 100, 'floats': 5.75,
                                 'jmena': 'Jiri'}])
>>> df3 = pd.concat([df, novy_zaznam], ignore_index = True)

>>> print(df3)
   cisla  floats   jmena
0     10    1.50   Lenka
1     20    2.50   Josef
2     30    3.50  Eliska
3     40    4.50   Pavel
4    100    5.75    Jiri
```

12.2 Numerické výpočty s NumPy

NumPy se primárně používá pro numerické výpočty a vědecké s tím, že poskytuje vysoce výkonná vícerozměrná pole. Pole NumPy jsou homogenní (všechny prvky stejného typu) a vícerozměrné. NumPy je paměťově efektivní díky reprezentaci založené na poli, funguje dobře pro velké datové sady a je ideální pro numerické výpočty, lineární algebru a maticové operace.

NumPy se velmi často používá ve vědeckém výzkumu, strojovém učení a numerických simulacích.

Pro generování matice s reálnými čísly s normálním rozdělením s rozměrem 9 řádků a 4 sloupce volíme tyto příkazy:

```
>>> import numpy as np
>>> np.random.seed(100)
>>> a = np.random.standard_normal((9, 4))
>>> a
[[-1.74976547  0.3426804   1.1530358  -0.25243604]
 [ 0.98132079  0.51421884  0.22117967 -1.07004333]
 [-0.18949583  0.25500144 -0.45802699  0.43516349]
 [-0.58359505  0.81684707  0.67272081 -0.10441114]
 [-0.53128038  1.02973269 -0.43813562 -1.11831825]
 [ 1.61898166  1.54160517 -0.25187914 -0.84243574]
 [ 0.18451869  0.9370822   0.73100034  1.36155613]
 [-0.32623806  0.05567601  0.22239961 -1.443217  ]
 [-0.75635231  0.81645401  0.75044476 -0.45594693]]
```

S celou řadou výhod můžeme kombinovat NumPy s Pandas:

```
>>> df = pd.DataFrame(a)
>>> df
          0         1         2         3
0 -1.749765  0.342680  1.153036 -0.252436
1  0.981321  0.514219  0.221180 -1.070043
2 -0.189496  0.255001 -0.458027  0.435163
3 -0.583595  0.816847  0.672721 -0.104411
4 -0.531280  1.029733 -0.438136 -1.118318
5  1.618982  1.541605 -0.251879 -0.842436
6  0.184519  0.937082  0.731000  1.361556
7 -0.326238  0.055676  0.222400 -1.443217
8 -0.756352  0.816454  0.750445 -0.455947
```

Přejmenování sloupců lze realizovat takto:

```
>>> df.columns=['No1', 'No2', 'No3', 'No4']
>>> df
        No1       No2       No3       No4
0 -1.749765  0.342680  1.153036 -0.252436
1  0.981321  0.514219  0.221180 -1.070043
2 -0.189496  0.255001 -0.458027  0.435163
3 -0.583595  0.816847  0.672721 -0.104411
4 -0.531280  1.029733 -0.438136 -1.118318
5  1.618982  1.541605 -0.251879 -0.842436
6  0.184519  0.937082  0.731000  1.361556
7 -0.326238  0.055676  0.222400 -1.443217
8 -0.756352  0.816454  0.750445 -0.455947
```

Výpočet průměrné hodnoty vybraného sloupce:

```
>>> df['No2'].mean())
0.7010330941456459
```

12.3 Práce s datem a časem

Pandas disponuje, mimo jiné, také operacemi pro práci a datumem a časem. Tato vlastnost je velmi často uváděna jako silná výhoda Pandas. Takto můžeme vygenerovat poslední dny pro 9 měsíců od 1. ledna 2024:

```
>>> dates = pd.date_range('2024-1-1', periods = 9, freq = 'M')
```

A následně tyto datumy přiřadit jako index pro hodnoty v matici:

```
>>> df.index = dates
>>> df
                 No1       No2       No3       No4
2024-01-31 -1.749765  0.342680  1.153036 -0.252436
2024-02-29  0.981321  0.514219  0.221180 -1.070043
2024-03-31 -0.189496  0.255001 -0.458027  0.435163
2024-04-30 -0.583595  0.816847  0.672721 -0.104411
2024-05-31 -0.531280  1.029733 -0.438136 -1.118318
2024-06-30  1.618982  1.541605 -0.251879 -0.842436
2024-07-31  0.184519  0.937082  0.731000  1.361556
2024-08-31 -0.326238  0.055676  0.222400 -1.443217
2024-09-30 -0.756352  0.816454  0.750445 -0.455947
```

12.4 Základní výpočty s DataFrame

S DataFrame lze realizovat celou řadu výpočtů.

Realizace souhrnné statistiky:

```
>>> df.describe()
            No1       No2       No3       No4
count  9.000000  9.000000  9.000000  9.000000
mean  -0.150212  0.701033  0.289193 -0.387788
std    0.988306  0.457685  0.579920  0.877532
min   -1.749765  0.055676 -0.458027 -1.443217
25%   -0.583595  0.342680 -0.251879 -1.070043
50%   -0.326238  0.816454  0.222400 -0.455947
75%    0.184519  0.937082  0.731000 -0.104411
max    1.618982  1.541605  1.153036  1.361556
```

Realizace součtů pro jednotlivé sloupce:

```
>>> df.sum()
No1   -1.351906
No2    6.309298
No3    2.602739
No4   -3.490089
dtype: float64
```

Realizace výpočtů průměrů pro jednotlivé sloupce:

```
>>> df.mean()
No1   -0.150212
No2    0.701033
No3    0.289193
No4   -0.387788
dtype: float64
```

Realizace výpočtů průměrů pro jednotlivé řádky:

```
>>> df.mean(axis=1)
2024-01-31   -0.126621
2024-02-29    0.161669
2024-03-31    0.010661
2024-04-30    0.200390
2024-05-31   -0.264500
2024-06-30    0.516568
2024-07-31    0.803539
2024-08-31   -0.372845
2024-09-30    0.088650
Freq: M, dtype: float64
```

Realizace kumulativních součtů ve sloupcích:

```
>>> df.cumsum()
                No1       No2       No3       No4
2024-01-31 -1.749765  0.342680  1.153036 -0.252436
2024-02-29 -0.768445  0.856899  1.374215 -1.322479
2024-03-31 -0.957941  1.111901  0.916188 -0.887316
2024-04-30 -1.541536  1.928748  1.588909 -0.991727
2024-05-31 -2.072816  2.958480  1.150774 -2.110045
2024-06-30 -0.453834  4.500086  0.898895 -2.952481
2024-07-31 -0.269316  5.437168  1.629895 -1.590925
2024-08-31 -0.595554  5.492844  1.852294 -3.034142
2024-09-30 -1.351906  6.309298  2.602739 -3.490089
```

12.5 Shrnutí

V tomto výukovém bloku je představen základní koncept zpracování rozsáhlých dat v podobě elementárních operací nad získanými daty. V tomto bloku je také stručně představena knihovna NumPy pro rychlé matematické výpočty

12.6 Otázky na procvičení

1. Jaké je základní uspořádání dat v Pandas dataframe?
2. Jakým způsobem lze přidat další sloupec do Pandas dataframe?
3. Jakým způsobem lze přidat další řádek do Pandas dataframe?
4. Jakým způsobem lze přejmenovat sloupce v Pandas dataframe?
5. Jaké základní souhrnné statistiky můžeme snadno získat z Pandas dataframe a jakým způsobem?

12.7 Odkazy a další studijní materiály

[1] https://numpy.org/

Financováno Evropskou unií
NextGenerationEU
Národní plán obnovy
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

13 Balíčky pro vizualizaci rozsáhlých dat

Studijní cíl

Tento blok je zaměřen na problematiku balíčků pro vizualizaci dat v jazyce Python.

Doba nutná k nastudování 2– 2,5 hodiny

Jazyk Python je velmi často využíván jak pro zpracování rozsáhlých dat (viz předchozí kapitola), tak i jejich vizualizaci. V tomto bloku jsou uvedeny základní možnosti pro vizualizaci dat, přičemž základním balíčkem pro vizualizaci dat je balíček matplotlib. Existuje celá řada alternativních nebo i rozšiřujících balíčků (např. plotly), ale práce s nimi je nad rámec tohoto výukového materiálu.

13.1 Balíček matplotlib

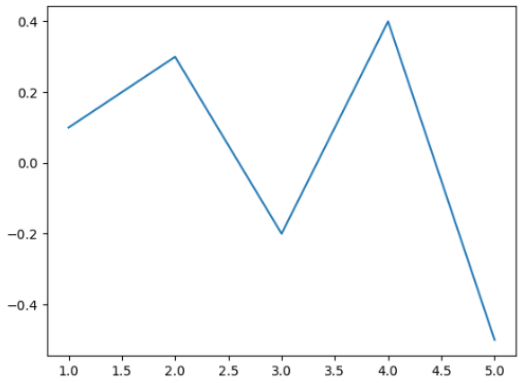
Balíček matplotlib je považován za nejběžněji používaný prostředek pro vizualizaci s ohledem na robustnost a spolehlivost – je velmi dobře integrován s NumPy a pandas a datovými strukturami, které jsou v těchto knihovnách používané. Balíček matplotlib umožňuje tvorbu výstupů v rastrové podobě (PNG, JPG apod.).

13.1.1. Jednorozměrné datové řady

Nejjednodušším způsobem použití je vykreslení bodů pro odpovídající souřadnice [x, y] následujícím způsobem (default způsob vykreslení je pomocí spojnicového grafu):

```
>>> import matplotlib.pyplot as plt

>>> x = [1, 2, 3, 4, 5]
>>> y = [0.1, 0.3, -0.2, 0.4, -0.5]

>>> plt.plot(x, y)
>>> plt.show()
```

Výstupem je tento graf, kde chybí název grafu, popis os s jednotkami na osách, definice rozsahu os, legenda v případě více datových řad atd. Tyto nedostatky budou postupně doplněny v tomto výukovém bloku.

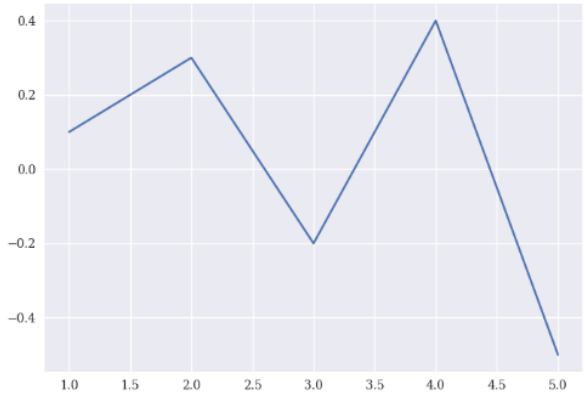


V balíčku Matplotlib je k dispozici celá řada typů grafů [1] a také stylů [2].

Pro specifické nastavení stylu, fontu a dalších atributů lze využít následující možnosti konfigurace:

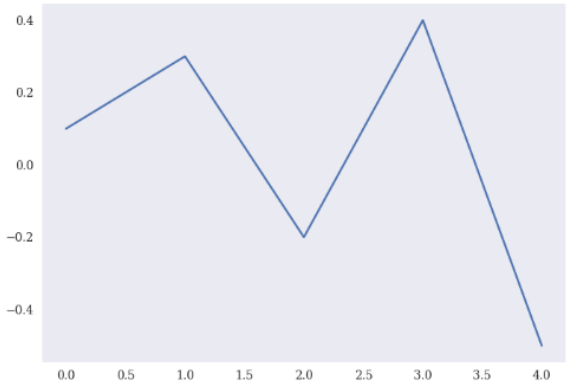
```
>>> import matplotlib as mpl
>>> import matplotlib.pyplot as plt

>>> plt.style.use('seaborn-v0_8')
>>> mpl.rcParams['font.family'] = 'serif'
```

Po nastavení těchto parametrů získáváme následující výstup, kde je použit definovaný font a styl grafu:



Pro vykreslení výstupu lze použít také zápis bez definice hodnot pro osu x, jejíž rozsah je nastaven automaticky:

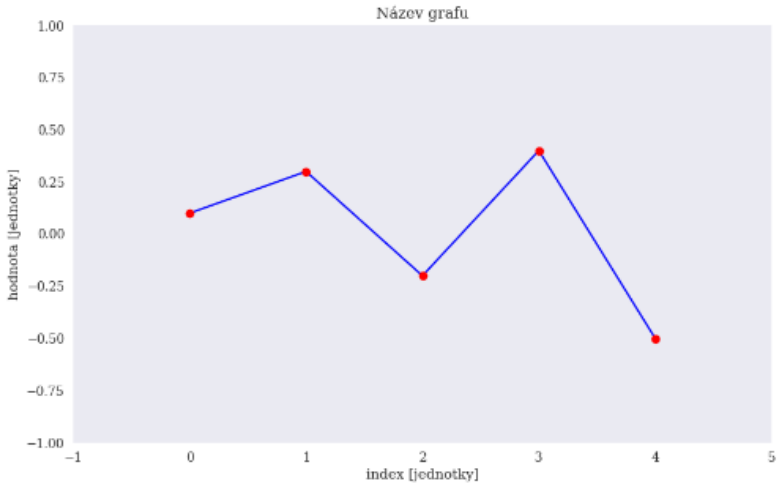
```
>>> x = [1, 2, 3, 4, 5]
>>> y = [0.1, 0.3, -0.2, 0.4, -0.5]

>>> plt.plot(y)
>>> plt.grid(False)
>>> plt.show()
```



Pro nastavení vlastností výstupu lze využít celou řadu nastavení, velmi důležité je nastavení os k čemuž slouží plt.axis() s následujícími možnostmi: off (vypnutí vodících čar a popisů os), [xmin, xmax, ymin, ymax] (pro nastavení limitů pro obě osy) apod.

Alternativně lze nastavit rozsah hodnot pro osy také s využitím metod xlim() a ylim(), popis os pomocí metod xlabel(), ylabel(), název grafu pomocí metody title() a velikost grafu pomocí metody figure(). V metodě plot lze také, kromě samotných hodnot, definovat formát vykreslení hodnot (např. 'b', lw=1.5 pro modrou úsečku s tloušťkou 1.5 bodu a 'ro' pro vykreslení červených bodů). Všechny tyto možnosti dokumentuje následující příklad:

```
>>> plt.figure(figsize=(10, 6))
>>> plt.plot(y, 'b', lw=1.5)
>>> plt.plot(y, 'ro')
>>> plt.grid(False)
>>> plt.xlim(-1, 5)
>>> plt.ylim(-1, 1)
>>> plt.xlabel('index [jednotky]')
>>> plt.ylabel('hodnota [jednotky]')
>>> plt.title('Název grafu')

>>> plt.show()
```

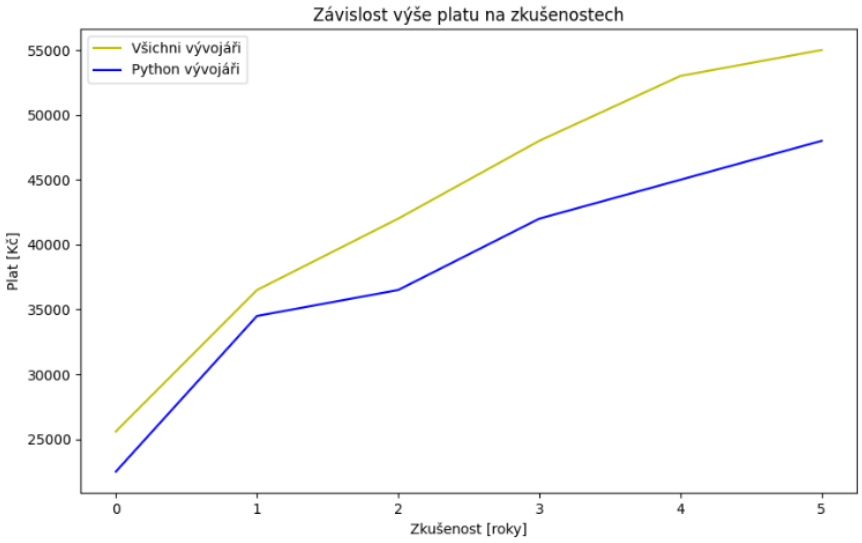


V [3] jsou uvedeny další možnosti nastavení vykreslování dat v grafu (nastavení barev, datových bodů atd.).

13.1.2. Vícerozměrné datové řady

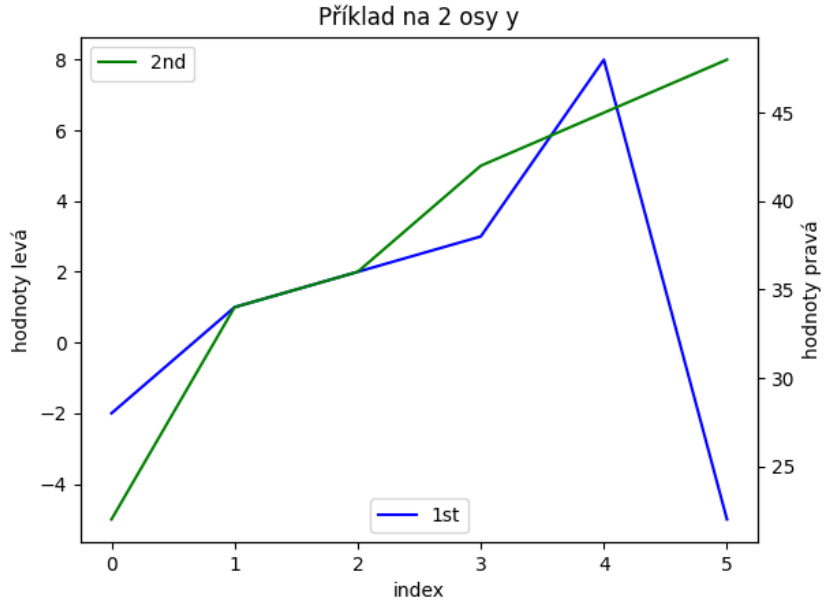
Práce s jednorozměrnými datovými řadami je spíše výjimečná, podstatně běžnější je zpracování vícerozměrných datových setů. Práce s nimi je v zásadě stejná jako s jednorozměrnými datovými řadami, což je zřejmé z následujícího příkladu, kdy je při práci s více datovými řadami nutné doplnit do grafu legendu pro odlišení řad s tím, že parametr loc určuje umístění legendy v grafu (0 pro umístění do nejvíce volného prostoru).

```
>>> import matplotlib.pyplot as plt

>>> x_experience_years = [0, 1, 2, 3, 4, 5]

>>> salary_python = [25600, 36500, 42000, 48000, 53000, 55000]
>>> salary_devs = [22500, 34500, 36500, 42000, 45000, 48000]

>>> plt.figure(figsize=(10, 6))
>>>  plt.plot(x_experience_years,  salary_python,  'y',  label='Python
vývojáři')
>>>  plt.plot(x_experience_years,  salary_devs,  'b',  label='Všichni
vývojáři')
>>> plt.xlabel('Zkušenost [roky]')
>>> plt.ylabel('Plat [Kč]')
>>> plt.title('Závislost výše průměrného platu na zkušenostech')
>>> plt.legend(loc=0)

>>> plt.show()
```



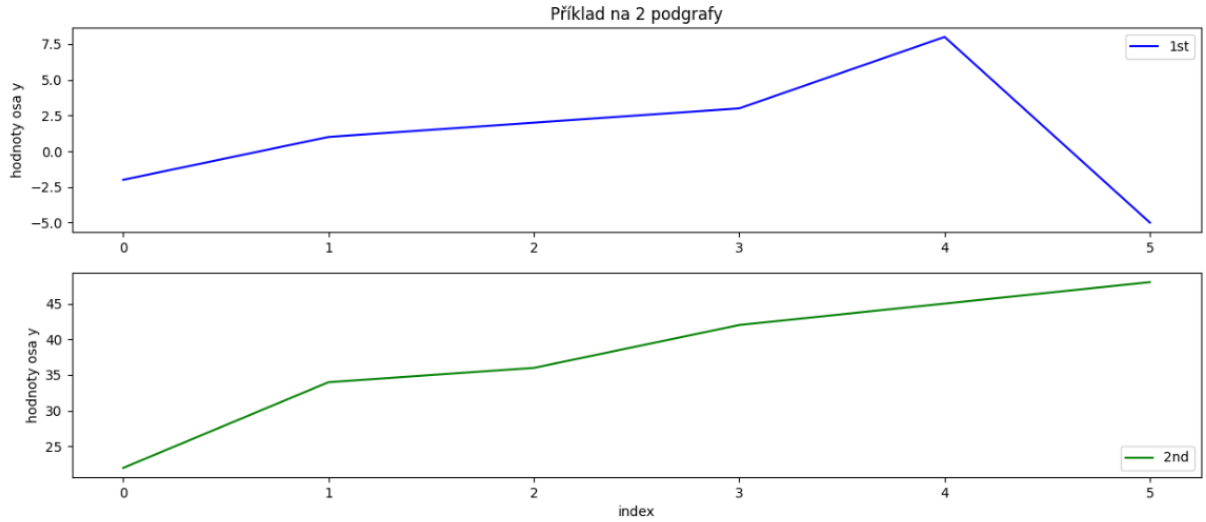
Velmi často se setkáme se situací, kdy je vhodné do jednoho grafu umístit dvě datové řady, přičemž rozsah těchto datových řad je velmi rozdílný. K tomuto účelu je možné přidat rozsah a legendu pro druhou osu y. K řešení využijeme metodu subplots() a twinx():

```
>>> data_1 = [-2, 1, 2, 3, 8, -5]
>>> data_2 = [22, 34, 36, 42, 45, 48]

>>> plt.figure(figsize=(15, 6))
>>> fig, ax1 = plt.subplots()
>>> plt.plot(data_1, 'b', label='1st')
>>> plt.legend(loc=8)
>>> plt.xlabel('index')
>>> plt.ylabel('hodnoty levá')
>>> plt.title('Příklad na 2 osy y')

>>> ax2 = ax1.twinx()
>>> plt.plot(data_2, 'g', label='2nd')
>>> plt.legend(loc=0)
>>> plt.ylabel('hodnoty pravá')

>>> plt.show()
```



Alternativně lze vykreslit obě části grafu zvlášť s využitím metody subplot() a definice počtu řádků subgrafů(2), počtu sloupců subgrafů(1) a indexu v takto definovaném uspořádání:

```
>>> data_1 = [-2, 1, 2, 3, 8, -5]
>>> data_2 = [22, 34, 36, 42, 45, 48]

>>> plt.figure(figsize=(15, 6))
>>> plt.subplot(211)
>>> plt.plot(data_1, 'b', label='1st')
>>> plt.legend(loc=1)
>>> plt.ylabel('hodnoty osa y')
>>> plt.title('Příklad na 2 podgrafy')

>>> plt.subplot(212)
>>> plt.plot(data_2, 'g', label='2nd')
>>> plt.legend(loc=4)
>>> plt.ylabel('hodnoty osa y')
>>> plt.xlabel('index')

>>> plt.show()
```



Obdobným způsobem lze realizovat nejrůznější kombinace podgrafů různých typů.

13.2 Shrnutí

V tomto výukovém bloku jsou uvedeny základní možnosti pro vizualizaci dat ve formě 2D grafů s využitím balíčku matplotlib s konfigurací:

- Vzhledu grafu
- Formátu os grafu
- Legendy grafu
- Podgrafů

13.3 Otázky na procvičení

1. S jakými balíčky standardně spolupracuje balíček matplotlib?
2. Která metoda slouží k vykreslení dat do grafu?
3. Jak realizovat vykreslení dat i s datovými body do grafu?
4. Jakým způsobem je možné nastavit dvě osy y s různými rozsahy hodnot?
5. Jaký je význam podgrafů a jak je lze realizovat s pomocí balíčku matplotlib?

13.4 Odkazy a další studijní materiály

[1] Plot types — Matplotlib 3.8.3 documentation
[2] Style sheets reference — Matplotlib 3.8.4 documentation
[3] matplotlib.pyplot.plot — Matplotlib 3.8.4 documentation

UNIVERZITA PARDUBICE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Vytvořeno v rámci projektu **Digitalizace studijních Agend, Nové TechnologiE, systémy a přístupy k výuce na UPCE**, reg. č. NPO_UPCE_MSMT-16591/2022.

Financováno Evropskou unií
NextGenerationEU

Národní plán obnovy

MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY