MASARYKOVA UNIVERZITA
Přírodovědecká fakulta
Ústav matematiky a statistiky

Bakalářská práce

Brno 2023 **Tomáš Salavec**

MASARYKOVA
UNIVERZITA
PŘÍRODOVĚDECKÁ FAKULTA
ÚSTAV MATEMATIKY A STATISTIKY

Python a jeho využití v matematice

Bakalářská práce

Tomáš Salavec

Vedoucí práce: doc. Mgr. Jan Koláček, Ph.D. **Brno 2023**

Bibliografický záznam

Autor:	Tomáš Salavec Přírodovědecká fakulta, Masarykova univerzita Ústav matematiky a statistiky
Název práce:	Python a jeho využití v matematice
Studijní program:	Matematika
Studijní obor:	Matematika-Ekonomie
Vedoucí práce:	doc. Mgr. Jan Koláček, Ph.D.
Akademický rok:	2022/2023
Počet stran:	119
Klíčová slova:	Python; Numpy; Matplotlib; Scipy; Pandas; Sympy; matematika; statistika

Bibliographic Entry

Author:	Tomáš Salavec Faculty of Science, Masaryk University Department of mathematics and statistics
Title of Thesis:	Using Python in mathematics
Degree Programme:	Mathematics
Field of Study:	Mathematics-Economics
Supervisor:	doc. Mgr. Jan Koláček, Ph.D.
Academic Year:	2022/2023
Number of Pages:	119
Keywords:	Python; Numpy; Matplotlib; Scipy; Pandas; Sympy; mathematics; statistics

Abstrakt

V této bakalářské práci se snažíme poskytnout důkladný přehled o možnostech využití programovacího jazyka Python v matematice a statistice. Práce by měla sloužit jako studijní materiál pro studenty matematiky na Přírodovědecké fakultě Masarykovy univerzity v Brně. V práci nejprve v první kapitole představíme stručný úvod do základních funkcí jazyka Python a seznámíme čtenáře s principy programování v Pythonu. Dále se budeme zabývat představením několika nejdůležitějších knihoven používaných pro řešení matematických úloh. Jedná se o knihovnu NumPy sloužící pro vektorizaci výpočtů a efektivní práci s poli, knihovnu SciPy umožňující provádět složitější numerické výpočty, knihovnu SymPy sloužící pro symbolické výpočty, knihovnu Pandas sloužící pro práci s daty uloženými v tabulkách a popisnou statistiku a knihovnu Matplotlib sloužící pro vizualizaci dat. Nakonec krátce představíme i knihovny Scikit-learn a Statsmodels sloužící pro statistické modelování v Pythonu.

Abstract

This bachelor's thesis provides an overview of the use of Python in mathematics and statistics. It will serve as study material for mathematics students at Masaryk University's Faculty of Science in Brno. The thesis covers basic functions of Python, principles of programming, and introduces important libraries for mathematical problem solving such as NumPy for vectorized calculations and arrays, SciPy for numerical calculations, SymPy for symbolic calculations, Pandas for data tables and descriptive statistics, and Matplotlib for data visualization. The thesis also mentions the Scikit-learn and Statsmodels libraries for statistical modeling.

MUNI
SCI

MASARYKOVA UNIVERZITA
PŘÍRODOVĚDECKÁ FAKULTA
KOTLÁŘSKÁ 2, 611 37 BRNO
IČ: 00216224
DIČ: CZ00216224

ZADÁNÍ
BAKALÁŘSKÉ PRÁCE

Akademický rok: 2022/2023

Ústav:	Ústav matematiky a statistiky
Student:	Tomáš Salavec
Program:	Matematika
Specializace:	Ekonomie Matematika

Ředitel *ústavu* PřF MU Vám ve smyslu Studijního a zkušebního řádu MU určuje bakalářskou práci s názvem:

Název práce:	Python a jeho využití v matematice
Název práce anglicky:	Using Python in mathematics
Jazyk závěrečné práce:	čeština

Oficiální zadání:
Popište základní strukturu programovacího jazyka Python. Vysvětlete použití základních knihoven v Pythonu, např. Numpy, Scipy, Matplotly a Pandas. Zkoumanou problematiku demonstrujte na vhodně zvolených příkladech.

Literatura:
VANDERPLAS, Jacob T. *Python data science handbook : essential tools for working with data*. First edition. Tokyo: O'Reilly, 2017. xvi, 529. ISBN 9781491912058.

MCKINNEY, Wes. *Python for data analysis : data wrangling with pandas, NumPy, and IPython*. Second edition. Sebastopol, CA: O'Reilly Media, 2017. xvi, 524. ISBN 9781491957660.

Vedoucí práce:	doc. Mgr. Jan Koláček, Ph.D.
Datum zadání práce:	14. 4. 2022
V Brně dne:	27. 1. 2023

Zadání bylo schváleno prostřednictvím IS MU.

Tomáš Salavec, 18. 10. 2022
doc. Mgr. Jan Koláček, Ph.D., 18. 10. 2022
RNDr. Jan Vondra, Ph.D., 26. 10. 2022

Poděkování

Na tomto místě bych chtěl poděkovat vedoucímu práce doc. Mgr. Janu Koláčkovi, PhD za podporu a časté konzultace při psaní této práce. Dále bych rád poděkoval své rodině, která mě při studiu a psaní mé bakalářské práce také významně podporovala.

Prohlášení

Prohlašuji, že jsem svoji bakalářskou práci vypracoval samostatně pod vedením vedoucího práce s využitím informačních zdrojů, které jsou v práci citovány.

Brno 15. března 2023 .....

Tomáš Salavec

Obsah

Úvod

Programovací jazyk Python je v současnosti jedním z nejpoužívanějších programovacích jazyků na světě. Používá se v nejrůznějších odvětvích. Nepostradatelný je nejen ve vědecké praxi, ale také ve strojovém učení, vývoji webových aplikací, zpracování a analýze dat atd.

V této práci poskytneme čtenáři podrobný přehled o možnostech využití programovacího jazyka Python pro potřeby matematických a statistických výpočtů a zpracování dat. Práce by v budoucnu měla sloužit jako studijní materiál pro potřeby studentů matematiky na Přírodovědecké fakultě Masarykovy univerzity v Brně. Speciálně pak má práce sloužit studentům předmětu Výpočetní matematické systémy. V tomto předmětu se studenti seznamují s možnostmi využití softwaru Matlab a R pro potřeby matematických a statistických výpočtů. Chceme studentům tohoto předmětu, kteří o to budou mít zájem, nabídnout možnost rozšířit si s využitím této práce své znalosti a naučit se programovat v jazyce Python.

V první kapitole se zaměříme na stručný úvod do základních funkcí jazyka Python a seznámíme čtenáře s principy programování v tomto jazyce. Následně se věnujeme představení několika nejdůležitějších knihoven používaných k řešení matematických problémů. Tyto knihovny zahrnují NumPy, která slouží k vektorizaci výpočtů a efektivní práci s poli, SciPy pro provádění složitějších numerických výpočtů, SymPy pro symbolické výpočty, Pandas pro práci s daty uloženými v tabulkách a deskriptivní statistikou a knihovnu Matplotlib pro vizualizaci dat. Nakonec se krátce představí knihovny Scikit-learn a Statsmodels, které se používají k statistickému modelování v Pythonu.

Věříme, že čtenáře naše práce zaujme a obohatí je užitečnými znalostmi, které se jim budou hodit v dalším studiu i budoucí kariéře.

Kapitola 1

Základy jazyka Python

1.1 Základní ovládání

Než začneme programovat v Pythonu je nutné si ho nejprve nainstalovat. Program je možné získat z webových stránek projektu (https://www.python.org/downloads/) [1]. Při psaní této práce jsem používal Python 3.11.0. Budete-li používat některou ze starších verzí programovacího jazyka Python, je možné, že některé zde uváděné postupy nebudou fungovat. Obecně doporučujeme používat nejnovější dostupnou verzi jazyka Python a Python pravidelně aktualizovat. Některé distribuce operačního systému Linux mají v sobě Python zabudovaný. Dále je třeba opatřit si vhodné prostředí, v němž budete psát. Pro výukové účely je vhodné programovat v Notebook, které je možno doinstalovat. Jednodušší příklady v této práci budou prezentovány obvykle tak, jako bychom je psali přímo do konzole IDLE. Výhodou je mimo jiné to, že pro výpis obsahu proměnné není nutné používat příkaz **print**(), ale stačí napsat jen jméno proměnné. Kódy napsané v IDLE konzoli lze poznat tak, že jsou příkazy uvedeny symbolem >>> a vypsané hodnoty nejsou uvedeny žádným symbolem. V příkladech, kde je vhodné zdůraznit, že pracujeme přímo v konzoli (především kvůli odlišení námi napsaného kódu a výstupů) využíváme tento typ formátování. V ostatních příkladech symbol >>> obvykle vynecháváme. Složitější kódy a funkce je samozřejmě nutné uložit a spouštět ze souboru. U složitějších funkcí obvykle nebudou uvedeny jimi vypsané výsledky. Čtenář si ovšem může snadno spustit kód na svém vlastním počítači a výsledky uvidí. Pro pokročilejší projekty je vhodné například prostředí Visual Studio Code, nebo Jupyter Notebook(https://code.visualstudio.com/) [2]. Pokročilejší nástroje, jakým je právě Visual Studio Code, navíc dělají mnoho věcí za uživatele. Typicky napovídají argumenty funkcí a dokončují za uživatele rozepsané příkazy. Pro uživatele, kteří již mají zkušenosti s programováním v jazyce R, může být výhodné používat při programování známé prostředí RStudio. Vytvoření skriptu v jazyce Python v prostředí RStudia je velmi snadné. Stačí v základní nabídce zvolit File » New File » Python Script namísto obvyklého File » New File » R Script. Může se stát, že bude RStudio vyžadovat od uživatele instalaci dodatečných balíčků, které umožní spouštět Pythonovské programy z prostředí RStudia. Tuto instalaci RStudio provede za uživatele. RStudio je možné si nainstalovat přímo z jeho webových stránek (https://www.rstudio.com/products/rstudio/download/). [3]

Po spuštění IDLE se objeví okno IDLE Shell, do něhož je možno psát příkazy a spouštět

z něj uložené programy. Píšeme-li program, obvykle ho chceme spouštět opakovaně proto je vhodné uložit ho v samostatném scriptu. Nový soubor s programem je možno vytvořit přes *File » New File*. Program ukládáme klávesovou zkratkou *CTRL + S* nebo přes grafické rozhraní jako *File » Save File*.

Při psaní kódu v Pythonu je třeba dbát na to, že oproti jiným programovacím jazykům je třeba dodržovat správné odsazování. Odsazení se používá na určení sekvence příkazů spadajících pod danou funkci, podmínku nebo cyklus. Příkazy se píší na samostatné řádky.

Tak jako v jiných programovacích jazycích, není ani v Pythonu nutné pamatovat si dopodrobna všechny příkazy a jejich parametry. Existuje mnoho stránek, kde je možno hledat nápovědu. Pokud si při psaní kódu pouze nejste jistí syntaxí, je možno využít příkazu **help()**. Tento příkaz otevře v terminálu rozhraní nápovědy, kam stačí napsat název příkazu nebo funkce a Python k ní vypíše nápovědu. Pro podrobnější nápovědu je možné se podívat do dokumentace Pythonu (https://docs.python.org/3/).[4]

1.2 Datové typy, proměnné a základní operace s nimi

1.2.1 Proměnné

Tak jako v jiných programovacích jazycích, i v Pythonu jsou základním stavebním kamenem programu proměnné. Proměná je objekt, v němž ukládáme data. Proměnnou vytváříme snadno za použití symbolu =, takto:

```
1
2 x = 3
```

Kód můžeme prokládat komentáři a to buď víceřádkovými nebo jednořádkovými. Komentáře nijak neovlivňují běh programu. Umožňují snadnější orientaci v programu. Začínáte-li s programováním, může se zdát, že jsou komentáře zbytečné, ale obzvláště v delším programu jsou velmi užitečné. Komentáře můžeme vytvářet následujícím způsobem:

```
1 # toto je jednoradkovy komentar
2 x = 3
3
4 """Toto je viceradkovy komentar.
5 Budete ho potkavat casto."""
```

Při pojmenovávání proměnných je dobrým zvykem volit popisné názvy, které stručně vystihují, jaká hodnota je v proměnné uložena. Obzvláště při psaní delšího kódu to usnadňuje orientaci v kódu a zamezuje některým častým chybám. Zde použité pojmenování pomocí jednoho písmene by tedy v delším kódu nebylo vhodné. Pro potřeby této práce je ovšem dostačující, neb příklady jsou krátké.

Proměnné mohou nabývat různých datových typů. Základními datovými typy jsou:

1. int (celá čísla)

2. string (řetězec znaků)
3. boolean (True/False, 1/0)
4. float (racionální číslo)
5. NULL (prázdný datový typ)
6. komplex (komplexní číslo)
7. tuple (N-tice)
8. list (seznam)
9. set (množina)
10. hash (slovník)

Hodnotu uloženou v proměnné můžeme vypsat příkazem **print()** nebo ji použít k dalším výpočtům. Datový typ *NULL* je triviální, vlastnosti ostatních popíšeme níže podrobněji. Prvních pět datových typů jsou tzv. primitivní datové typy. Ostatní jsou složené datové typy, protože jsou složeny z prvků, které jsou primitivními datovými typy.

1.2.2 Celá čísla

S proměnnou s datovým typem *int* můžeme provádět všechny základní algebraické operace. Je ale třeba dávat pozor na to, že například dělení může změnit datový typ proměnné na *float*. Datový typ změní také jiné operace, pokud je alespoň jeden z činitelů *float*. Python je tzv. dynamicky typovaný jazyk. Celá čísla můžeme také porovnávat pomocí operátoru **==**. V Pythonu má operátor **=** jiný význam než operátor **==**. Zatímco = přiřazuje proměnné hodnotu, **==** porovnává hodnotu dvou čísel a vrací Booleovskou hodnotu.

```

#deklarujeme promenou x
x = 3

#deklarujeme promenou y
y = 4

#deklarujeme promenou z, ktere bude mit hodnotu 7
z = x + y

#takto menime hodnotu x
x = x + 3
```

Změnit hodnotu x je možno i pomocí zkrácené notace takto zvýšíme hodnotu x o 3, pak snížíme o 4, vynásobíme 2 a vydělíme 2:

```

x += 3
x -= 4
x *= 2
x /= 2
```

Můžeme definovat i složitější výrazy pomocí závorek:

```
x = ((x + z) * (z - x))
```

Při dělení je možné změnit datový typ, zde na *float*. Tomu lze zabránit např. použitím operátoru // pro celočíselné dělení jako na druhém řádku následujícího kódu::

```

y = y / 5

y = 4 // 3
```

V Pythonu existuje krom základních operací +, -, *, /, // i speciální operace % (modulo) pro zbytek po dělení. Následující kód uloží do proměnné y zbytek po dělení čísla 11 číslem 2 a do proměnné z zbytek po dělení čísla 533 číslem 4:

```

y = 11 % 2

z = 533 % 4
```

Nakonec si můžeme hodnoty uložené v našich proměnných vypsat pomocí příkazu **print()**. Je možno vypsat i více proměnných najednou nebo hodnotu výrazu složeného z těchto proměnných. Můžeme také vypsat hodnotu logických výrazů v našich proměnných. Tyto možnosti demonstruje následující kód:

```
>>> print(x)
24.0
>>> print(z, y)
1 7
>>> print(x + y + z )
32.0
>>> print(x == z)
False
>>> print(x == x)
True
```

V IDLE konzoli můžeme také jednoduše pouze zadat názvy našich proměnných a výrazy ve kterých se vyskytují i bez příkazu **print()** a výsledek bude stejný:

```
>>> x
24.0
>>> z, y
(1, 7)
>>> x + y + z
32.0
>>> x == z
False
>>> x == x
True
```

1.2.3 Řetězce

Řetězec je datový typ představující sekvenci písmen, číslic, mezer a dalších znaků. Řetězec musí být ohraničen jednoduchými nebo dvojitými uvozovkami. Operace + skládá

řetězce za sebe. Operace * skládá řetězce za sebe. Proměnnou každého jiného datového typu můžeme změnit na řetězec pomocí příkazu **str()**. Tyto základní operace s řetězci demonstruje následující příklad:

```
>>> x = "Hello "
>>> x
'Hello'
>>> y = 'World!'
>>> y
'World!'
>>> z = x + y
>>> z
'Hello World!'
>>> u = x * 3
>>> u
'Hello Hello Hello '
>>> p = str(3)
>>> p
'3'
```

Výraz "\n"značí konec řádku. Výrazy oddělené tímto znakem tedy Python vypíše na různé řádky. Jedná se o znak v řetězci. Tento znak přímo souvisí s tzv. regulárními výrazy, které budou probrány později.

Řetězce můžeme porovnávat operátorem == podobně jako čísla. Operátor vrací hodnotu *True*, jsou-li řetězce stejné, nebo *False* v opačném případě. Takto můžeme porovnat, zda-li jsou řetězce x a y z našeho předchozího příkladu stejné:

```
>>> x == y
False
```

Pro práci s řetězci je možné používat metody. O tom, co metody konkrétně znamenají bude pojednáno dále v kapitole o objektech. Prozatím se na ně můžeme dívat jako na speciální funkce vlastní danému datovému typu. Metody můžeme využívat pomocí syntaxe *.nazevmetody(argumenty metody)*. Často používanou metodou je metoda **.split()**. Tato metoda rozdělí řetězec na několik částí podle řetězce, který je předán metodě jako její argument. Metoda vrátí seznam obsahující jednotlivé kusy původního řetězce. Další metody je možno využívat s využitím tzv. "regulárních výrazů". Regulární výrazy budou podrobně probrány později stejně jako metody vážící se k nim.

```
>>> x = "abcd"
>>> xx = x + " efgh"
>>> xx
'abcd efgh'
>>> y = xx.split(" ")
>>> y
['abcd', 'efgh']
```

1.2.4 Racionální čísla

Pro racionální čísla můžeme používat stejné funkce a příkazy jako pro čísla celá. Datový typ *int* je možné změnit na racionální číslo, je-li to třeba, příkazem *float()*. Při porovnávání

racionálních čísel pomocí operátoru == je třeba dbát na nepřesnosti vzniklé numerickým zaokrouhlováním. Čtenáře možná napadne, zda-li v Pythonu existuje datový typ pro reálná čísla. Neexistuje. Počítač není schopen počítat se spojitými veličinami, z principu jeho fungování tedy nutně počítá s diskrétními veličinami. Reálná čísla v Pythonu reprezentovat nejde, pouze čísla racionální a to jen konečně malá. Při řešení většiny praktických problémů to ovšem nepůsobí žádné problémy.

1.2.5 Komplexní čísla

Komplexní čísla v Pythonu vytváříme příkazem **complex(a, b)**, kde a reprezentuje reálnou část komplexního čísla a b imaginární část komplexního čísla. U komplexních čísel je možné používat všechny základní operace tak jako u čísel racionálních. Více operací s komplexními čísly umožňují specifické knihovny, např. knihovna *cmath*.

```
>>> x = complex(4,3)
>>> x
(4+3j)
>>> y = complex(5,6)
>>> y
(5+6j)
>>> z = x + y
>>> z
(9+9j)
>>> u = x / y
>>> u
(0.6229508196721312-0.14754098360655735j)
```

Jak se může čtenář přesvědčit výše, Python značí komplexní jednotku písmenem ”j”namísto běžnějšího písmene ”i”.

1.2.6 N-tice

Datový typ tuple je první z tzv. složených datových typů. To znamená, že prvky každého objektu třídy tuple jsou objekty některého jiného datového typu. Konkrétně tuple jsou nezměnitelné n-tice. Na první pohled mohou připomínat vektory známé z jazyka R. Jejich vlastnosti jsou ovšem značně odlišné. Hlavním rozdílem je neměnitelnost Pythonovských n-tic. Vytváří se pomocí kulatých závorek. Pomocí hranatých závorek je možné získat konkrétní prvky n-tice. Je ovšem třeba dávat pozor na to, že v Pythonu se indexuje od nuly. Tedy první prvek n-tice má index 0. N-tice můžeme skládat za sebe pomocí operace +. Operace * zopakuje n-tici několikrát za sebou. Tedy např. součet dvou dvou-prvkových tuple je čtyřprvková tuple.

```
>>> x = (1,2)
>>> x
(1, 2)
>>> x[0]
1
>>> x[1]
2
>>> y = ("abd", "h")
>>> z = x + y
```

```
>>> z
(1, 2, 'abd', 'h')
>>> u = 5 * x
>>> u
(1, 2, 1, 2, 1, 2, 1, 2, 1, 2)
```

1.2.7 Seznamy

Seznamy jsou dalším ze složených datových typů. Vytváříme je pomocí hranatých závorek. Seznam také můžeme vytvořit pomocí příkazu **list()** z řetězce. Tento příkaz rozloží řetězec na znaky, z nichž každý bude jedním prvkem seznamu. Pomocí příkazu list můžeme na seznamy převádět také jiné datové typy, například množiny. Seznamy mohou obsahovat všechny předchozí datové typy, ale i jiné seznamy. Mohou také obsahovat prvky více různých datových typů, ačkoli tento postup se spíše nedoporučuje. Pomocí seznamů v seznamech je možné vytvářet vícerozměrná pole. (Je také možné pomocí seznamů naprogramovat v Pythonu matice se základními operacemi, k tomu ale Python nabízí lepší nástroje v balíku Numpy, se kterým se seznámíme ve druhé kapitole.). Seznamy můžeme i skládat za sebe pomocí symbolu +. Počet prvků seznamu můžeme zjistit pomocí příkazu **len()**. Základní operace se seznamy ilustruje následující příklad:

```
>>> a = "abcd"
>>> b = list(a)
>>> b
['a'. 'b', 'c', 'd']
>>> bb = [1,2,3,4]
>>> bb[1]
2
>>> bbb = [[1, 2, 3], [5,11, 13]]
>>> bbb[0]
[1, 2, 3]
>>> bbb[0][1]
2
>>> c = b + bb
>>> c
['a', 'b', 'c', 'd', 1, 2, 3, 4]
>>> len(c)
8
```

Do seznamu je oproti N-ticím možné přidávat prvky i je ze seznamu odebírat. Dále je možné seznam seřadit, měnit pořadí prvků atd. K tomu můžeme použít patřičné metody:

append()	přidání prvku na konec seznamu
extend()	přidání více prvků na konec seznamu
pop()	odebrání posledního prvku z konce seznamu
insert()	přidání prvku na konkrétní pozici v seznamu
remove()	odstranění prvního výskytu konkrétního prvku ze seznamu
clear()	odebrání všech prvků ze seznamu
reverse()	obrácení pořadí prvků seznamu
sort()	seřazení prvků seznamu od nejmenšího po největší (číselné datové typy) nebo podle abecedy (řetězce), prvky musí mít stejný datový typ, nelze porovnávat a tedy ani seřadit čísla a řetězce

Využití těchto metod demonstruje následující příklad:

```
>>> b.append("e")
>>> b
['a', 'b', 'c', 'd', 'e']
>>> bb.extend([5,6])
>>> bb
 [1, 2, 3, 4, 5, 6]
>>> b.pop()
>>> b.insert(1, 666)
>>> b.remove("b")
>>> b
['a', 666, 'c', 'd']
>>> seznam_k_serazeni = [1,5,4,5,454,8.5,5,57]
>>> seznam_k_serazeni.sort()
>>> seznam_k_serazeni
[1, 4, 5, 5, 5, 8.5, 57, 454]
>>> seznam_k_serazeni.clear()
>>> seznam_k_serazeni.extend(["bobek", "cecilie", "abraham", "monika
    ", "klarisa"])
>>> seznam_k_serazeni.sort()
>>> seznam_k_serazeni
['abraham', 'bobek', 'cecilie', 'klarisa', 'monika']
```

1.2.8 Množiny

Množina je další ze složených datových typů. Uchovává několik prvků jiného datového typu. Množiny jsou neuspořádané a neindexované. V množině může být každý prvek obsažen nejvýše jednou. Seznam je možné změnit na množinu příkazem **set()**. Pomocí tohoto příkazu je možné snadno odstranit ze seznamu duplicitní prvky tak, že nejprve převedeme

seznam na množinu příkazem **set()** a následně zpět na seznam příkazem **list()**. Úplně novou množinu můžeme vytvořit také pomocí složených závorek. Mnohé metody fungují podobně jako u seznamů. Metodou **add()** přidáváme do množiny nové prvky, metodou **remove()** je odebíráme. Metoda **pop()** odebírá prvek množiny. Užitečná je také metoda **difference()**, která vrací množinu obsahující rozdíl dvou nebo více množin. Následující příklad demonstruje vytváření množin dvěma množnými způsoby:

```
x = {1, 5, 4, 66}

y = [1, 5, 4, 22, 22]

yy = set(y)
```

Takto převedeme množinu *yy* na seznam a uložíme do seznamu *y*, který již neobsahuje duplicitní prvky:

```
y = list(yy)
```

Takto zjistíme množinový rozdíl množiny *yy* a *x* dvěma možnými způsoby:

```
z = yy.difference(x)

zz = x.difference(yy)
```

Takto přidáváme prvek 77 a odebíráme prvkek 4 z množiny x:

```
x.add(77)

x.remove(4)
```

Na množiny můžeme také volat operátor pro množinové sjednocení pomocí znaku `|` a průnik pomocí znaku `&`:

```
>>> x & yy
{1, 4, 5}
>>> x | yy
{1, 66, 4, 5, 22}
```

1.2.9 Slovníky

Slovník v Pythonu je dalším ze složených datových typů, který obsahuje data uložená pod klíči. Hodnoty uložené ve slovníku je možné volat pomocí k nim náležících klíčů, ale ne naopak. Jedná se o měnitelný datový typ, tedy je možné nové hodnoty do slovníku přidat spolu s patřičným klíčem, nebo je i odebrat. Slovník vytváříme pomocí složených závorek. Každou hodnotu ukládáme tak, že nejprve napíšeme klíč, pod kterým bude uložena, následně znak ":"a pak hodnotu, která je pod daným klíčem uložena. Na slovník je také možné převést některé jiné iterovatelné datové typy. Pomocí příkazu **dict()** je možné převést na slovník seznam dvojic, bude jako klíč použit vždy první záznam z každé dvojice a druhý prvek v dvojici bude pod tímto klíčem uložen. Máme-li seznam klíčů a seznam hodnot, je možné tyto dva seznamy převézt na slovník tak, že je nejprve převedeme na seznam dvojic pomocí příkazu **zip()** a následně využijeme příkaz **dict()**. Jednotlivé záznamy ve slovníku oddělujeme čárkou. Hodnoty uložené ve slovníku voláme pomocí hranatých závorek, do nichž zapíšeme klíč, pod nímž jsou uložena data, která chceme volat. Pomocí

hranatých závorek můžeme také hodnotu do slovníku přidat. Hodnoty odebíráme pomocí příkazu **del()**. Počet záznamů ve slovníku zjistíme pomocí příkazu **len()**. Pomocí příkazu **in** můžeme zjistit, zda-li slovník obsahuje daný klíč.

Toto je náš první slovník. Obsahuje pro demonstraci této možnosti klíče i hodnoty různých datových typů (tento postup se ale obecně nedoporučuje):

```
1
2 dict1 = {'kiwi': 'zelena', 1 : '25', (1,2) : [1, 2, 5.5]}
```

Takto voláme konkrétní hodnotu ze slovníku:

```
1 dict1["kiwi"]
```

Takto voláme prvek ze slovníku uložený pod klíčem (1,2). Tento prvek je seznam a z něj voláme 3. prvek:

```
1 dict1[(1,2)][2]
```

Takto přidáme řetězec "world"do slovníku pod klíč "hello":

```
1 dict1["hello"] = "world"
```

Takto odebereme hodnotu uloženou pod klíčem 1:

```
1 del(dict1[1])
```

Takto vytvoříme slovník ze seznamu dvojic:

```
1 seznam1 = [(1, "Hello"), (2,"Beautiful"), (3, "World")]
2
3 dict2 = dict(seznam1)
```

Takto převedeme 2 seznamy na slovník:

```
1 seznam2 = [1, 2, 3, 4]
2 seznam3 = ["a", "b", "c", "d"]
3 dict3 = dict(zip(seznam3, seznam2))
```

Takto otestujeme, zda-li jsou klíče *a* a *h* obsaženy ve slovníku dict3:

```
1 print("a" in dict3, "h" in dict3)
```

1.3 Podmínky a cykly

1.3.1 Podmínky if, elif a else

Často, když píšeme kód, chceme, aby se nějaký kus kódu provedl jen v případě, kdy je splněna předem daná podmínka. K tomu slouží funkce **if**. Hned za slovo **if** píšeme podmínku, jejímž výsledkem musí být logická hodnota, a řádek zakončujeme dvojtečkou. Na další řádek pak píšeme kód, který se provede, pokud bude podmínka splněna. Kód je ale nutno odsadit. Narozdíl od jiných programovacích jazyků Python nepoužívá k tomu, aby určil, která sekvence kódu náleží pod danou podmínku, závorky, nýbrž užívá odsazení. Chceme-li specifikovat, jaký kód se má provést v případě, že podmínka splněna nebude, použijeme značku **else**. Chceme-li, aby měla podmínka více větví, je možné použít výraz **elif**. Syntaxe podmínek tedy vypadá následovně:

```
x = 5
y = 2

if x > y:
    print("x je vetsi nez 2")
    print("x je vetsi nez y")
elif x == y:
    print("x a y jsou stejne")
else:
    print("x je mensi nez y")
```

Kód vrátí následující řetězce:

```
x je vetsi nez 2
x je vetsi nez y
```

Podmínky je možné psát i ve formě složitějších logických výrazů složených z několika podvýrazů. Pro specifikaci logického výrazu můžeme používat i speciální výrazy **and**, **or** a **not**, které mají význam logické konjunkce, disjunkce a negace. Složené logické výrazy složené z několika výrazů můžeme využít například takto:

```

if [(x >= 4) and (x < 56)] or [ (x == y) and (x != 45)]:
    print("tato slozita podminka je splnena")
else:
    print("podminka splnena neni")
```

1.3.2 Podmínky s využitím match

Od verze Pythonu 3.10 je v v Pythonu implementován příkaz **match**, který může být alternativou k podmínkám za použití **if**. Pro čtenáře znalé programovacích jazyků Java nebo Matlab bude zapisování podmínek pomocí **match** jistě povědomé, neb **match** je Pythonovským ekvivalentem příkazu **switch** hojně využívaného při programování v Javě a Matlabu.

Při nejjednodušším možném použití, předáváme výrazu **match** na vstupu nějakou proměnnou a stanovíme jaký kód má Python provést v případě různých hodnot dané proměnné. Jednotlivé hodnoty proměnné nastavujeme pomocí klíčového slova čase, za nějž uvedeme hodnotu proměnné, pro níž chceme provést speciální kód, a pod ní napíšeme kód prováděný v daném případě. Pro určení kódu, který se má provést v případě, že hodnota proměnné neodpovídá žádné z námi specifikovaných hodnot použijme výraz **case other** nebo ekvivalentní výraz **case** _. Můžeme tak například snadno naprogramovat podmínku, která nám na základě proměnné, v níž je uveden pozdrav ”Hello world!”v nějakém jazyce, vypíše, v jakém jazyce je pozdrav napsán:

```
greeting = "Ahoj svete!"

match greeting:
    case "Ahoj svete!":
        print("He's speaking czech.")
    case "Hello world!":
        print("He's speaking english.")
```

```
    case "Hola Mundo!":
        print("He's speaking spanish.")
    case "Hallo Welt!":
        print("He's speaking german.")
    case other:
        print("I don't know that language!")
```

Oproti **switch**, na který jsou asi zvyklí uživatelé programovacího jazyka Java nebo Matlab, je **match** komplexnější. Umožňuje např. použít část zadané proměnné jako identifikátor dané větve kódu a část jako proměnnou v kódu prováděném po identifikaci, takto:

```
greeting = "Hola Mundo"

match greeting.split():
    case ["Hello", *world]:
        print("He's speaking english and the world is world.")
    case ["Hola", *world]:
        print("He's speaking spanish.")
        print("In spanish the word for world is:", *world )
    case ["Hallo", *world]:
        print("He's speaking german.")
        print("In german the word for world is:", *world )
    case other:
        print("I don't know that language!")
```

Příkaz **match** umožňuje psát kód přehledně a úsporně, proto se ho obzvlášť při psaní delších programů vyplatí používat. V kratším kódu je ovšem dostačující větvení kódu i pomocí výrazů **if, elif, else**.

1.3.3 Cyklus for

Cyklus umožňuje opakovat sekvenci příkazů nebo aplikovat sekvenci příkazů na několik prvků seznamu. Indexy, podle kterých iterujeme, mohou také vystupovat jako proměnné v sekvenci příkazů, která se v cyklu opakuje. Cyklus **for** použijeme, víme-li, kolikrát chceme sekvenci příkazů opakovat. Počet opakování je proměnná, která se často nazývá i. Specifikovat ji můžeme ve chvíli, kdy zapisujeme cyklus **for**. Pro specifikaci intervalu, ze kterého chceme vybírat hodnoty proměnné i, použijeme příkaz range(). Tento příkaz může mít buď dva vstupy, kde první z nich udává, od kterého indexu má cyklus **for** začít a druhý, u kterého indexu má skončit. Je třeba dbát na to, že hodnota indexu uvedená jako druhá se již nevyužije, cyklus skončí u hodnoty o jedno nižší. Druhou možností je zadat pouze jednu hodnotu, která udává, kolik opakování se má provést. Zde je třeba nezapomínat na to, že Python indexuje od nuly. Syntaxe cyklu **for** vypadá následovně:

```

for i in range(5):
    print(i)

for i in range(5,10):
    print("this is number:", i)
```

Kód výše vypíše následující výstup:

```
0
1
2
3
4
this is number: 5
this is number: 6
this is number: 7
this is number: 8
this is number: 9
```

Namísto vytváření nové proměnné, podle které budeme iterovat je možné spustit for cyklus také nad prvky nějakého seznamu. Využijeme k tomu výraz **in**. Pomocí stejné syntaxe je také možné procházet postupně všechny znaky v nějakém řetězci. Syntaxe je v tomto případě následující:

```

iteracni_seznam = ["ahoj", "svete", "hello", "world"]

retezec = "Hello world!"

for i in iteracni_seznam:
    print(i)

for j in retezec:
    print(j)
```

Tento kód vypíše následující výstup:

```
ahoj
svete
hello
world
H
e
l
l
o

w
o
r
l
d
!
```

For cykly je možné snadno vkládat do sebe. Uvnitř **for** cyklu se tedy může opakovat jiný **for** cyklus a často je to v průběhu programování výhodné. Tímto způsobem je například možné procházet vícerozměrná pole tak, jako zde:

```

vicerozmerne_pole = [[[5,7], [8, 4]],[[1,2],
                                      [3,4]],[[5,6],[7,8]]]

for i in vicerozmerne_pole:
    for j in i:
```

```
        for m in j:
            print(m)
        print("--")
    print("-------")
```

Výstup, který vypíše tento náš poslední **for** cyklus vypadá následovně:

```
5
7
--
8
4
--
-------
1
2
--
3
4
--
-------
5
6
--
7
8
--
-------
```

1.3.4 Cyklus while

Cyklus **while** využijeme, pokud nevíme přesně, kolikrát chceme danou sekvenci příkazů opakovat, ale chceme, aby se opakovala, dokud platí nějaká logická podmínka. Tímto způsobem můžeme například metodou půlení intervalu počítat druhou odmocninu ze dvou, dokud nedosáhneme požadované přesnosti tak, jako zde:

```

#toto je zkracena notace pro deklaraci promenne
a, b, x = 1, 2, 1.5

#zde deklarujeme promenou urcujici chybu vypoctu
diff = x*x - 2

#takto vypada syntaxe cyklu while
while diff > 0.000000001 :
    x = (a + b)/2

    if x*x < 2:
        a = x
        diff = 2 - x*x
    else:
        b = x
        diff = x*x - 2
```

```
18
19 print(x)
```

Výsledkem kódu je následující číslo, které je dostatečně přesným odhadem odmocniny ze dvou:

```
1 1.4142135623842478
```

Chceme-li z nějakého důvodu ukončit **while** cyklus dříve než nastane podmínka uvedená na začátku, využijeme příkaz **break** například takto:

```
1
2 #cyklus s break
3 n = 40
4
5 while n % 5 != 4:
6     n *= 3
7     if n % 5 == 0:
8         print("!", n)
9         break
10
11 print(n)
```

Výsledkem tohoto kódu je následující výstup:

```
1 ! 120
2 120
```

While cykly můžeme vkládat do sebe. Uvnitř jednoho **while** cyklu tedy můžeme opakovaně spouštět další. Stejně tak může uvnitř **while** cyklu běžet cyklus **for** nebo spouštět **while** cyklus uvnitř **for** cyklu.

1.4 Funkce

Funkce je blok kódu, který se spustí vždy, když je daná funkce volána. Funkce může mít různé parametry, které mohou ovlivňovat běh daného bloku kódu a vystupovat jako argumenty použitých příkazů. Funkce používáme v případě, že se nějaký blok kódu v našem programu často opakuje a nebo se opakuje pouze s odlišnými parametry. Funkci vytváříme pomocí příkazu **def** za nějž napíšeme název funkce a závorku, do níž můžeme zapsat názvy parametrů funkce, pokud nějaké má, a následně dvojtečku. Blok kódu, který se má uskutečnit, zavoláme-li funkci, je nutno odsadit, aby bylo zřejmé, že patří pod danou funkci. Funkci voláme jejím názvem s konkrétními hodnotami parametrů v závorce. Je třeba dbát na to, aby název funkce nebyl shodný s některým zabudovaným příkazem jazyka Python. Python uživateli dovolí přepsat již existující funkci, ale není to dobrá programátorská praxe. Hodnoty proměnných používaných a měněných uvnitř funkce můžeme také vrátit příkazem **return**. Vrácenou hodnotu můžeme uložit do proměnné nebo ji použít jako parametr funkce. Příkaz **return** ukončí běh funkce, kód uvedený uvnitř funkce až po příkazu **return** se tedy již neprovede. Takto můžeme například vytvořit funkci, která počítá pro parametry x a y hodnotu výrazu *xy - x - y*, hodnotu vrátí a následně ji vypíše:

```
#takto definujeme funkci
def calculate_formula(x,y):
    z = x*y - x -y
    print(z)
    return z
```

Zde definujeme několik proměnných a hodnoty funkce calculate_formula() uložíme do proměnných d a e:

```
a, b, c = 7, 8, 9

d = calculate_formula(a,b)

e = calculate_formula(a, c)
```

Takto použijeme funkci jako parametr stejné funkce:

```
f = calculate_formula(calculate_formula(d,c), a)
```

Někdy je zbytečné funkci pojmenovávat a je výhodné využít tzv. *lambda* funkce. Oproti jiným programovacím jazykům mohou být v jazyce Python *lambda* funkce pouze jednořádkové. *Lambda* funkce jsou tzv. anonymní funkce. Používáme je pomocí příkazu *lambda* za nějž napíšeme argumenty funkce, dvojtečku a následně výraz, který bude funkce provádět. Anonymní funkci, která bude počítat hodnotu výrazu *xy - x - y*, můžeme definovat například takto:

```
>>> g = lambda x,y: x*y - x - y
>>> g(a,b)
41
```

Funkce můžeme používat opakovaně na různé prvky iterovatelného datového typu pomocí výše zmíněných cyklů **for** a **while**. Alternativu k tomuto přístupu nabízí funkce **map()**. Často je vhodnější využít funkci **map()** spíše než cykly neb syntaxe je úspornější. Funkce **map()** bere na vstupu dva argumenty. Prvním z nich je funkce, která se má použít na všechny prvky iterovatelného datového typu (často zadána jako nepojmenovaná **lambda** funkce) a druhým argumentem je proměnná iterovatelného datového typu, na jejíž prvky budeme funkci aplikovat. Takto například pomocí funkce **map()** spočítáme druhé mocniny čísel od 14 do 16:

```
h = (14, 15, 16)

k = list(map(lambda x : x*x, h))
```

Funkci **map()** můžeme předat jako argument i více proměnných iterovatelného datového typu. Funkce pak využije vždy dva prvky z obou proměnných se stejným indexem. Prvním argumentem funkce **map()** však vždy musí být funkce, kterou chceme na prvky daných proměnných aplikovat:

```
#takto muzeme pomoci map() secist dva seznamy
l = [4, 5, 44]
```

```
m = [7, 8, 6]

n = list(map(lambda x, y: x + y, m, l))
```

Nakonec kapitoly o funkcích je ještě vhodné definovat pojmy *lokální proměnná* a *globální proměnná*. Globální proměnné jsou proměnné definované v našem kódu mimo těla funkcí. Lokální proměnnou rozumíme proměnnou definovanou uvnitř těla funkce. Má-li taková proměnná stejný název jako nějaká globální proměnná, uvnitř těla funkce je použita proměnná lokální a hodnota globální proměnné se ignoruje. Je-li uvnitř funkce volána nějaká proměnná a není-li v těle funkce definována lokální proměnná se stejným názvem, hledá Python globální proměnnou se stejným názvem. Má-li nějaká lokální proměnná stejný název jako nějaká globální proměnná a je-li uvnitř těla funkce měněna hodnota lokální proměnné, hodnotu uložonou v globální proměnné to nijak neovlivní. Hodnoty globálních proměnných není možné měnit uvnitř funkce. Chceme-li uvnitř funkce měnit hodnotu globální proměnné, můžeme k tomu použít příkaz **global**. Za tento příkaz uvedeme název globální proměnné, kterou má naše funkce měnit. Následující příkazy už mohou hodnotu globální proměnné ovlivnit. Používání globálních proměnných se ovšem obecně nedoporučuje. Na následujících čtyřech funkcích demonstrujeme rozdíl mezi lokálními a globálními proměnnými:

```
x = 4

def func1():
    x = 5
    print(x)

def func2():
    print(x)

def func3():
    x += 4
    print(x)

def func4():
    global x
    x += 4
    print(x)

func1()
func2()
func4()
func3()
```

Funkce *func2* vypíše hodnotu 4, neb v těle této funkce Python nenalezl žádnou proměnnou x a tak začal hledat mezi globálními proměnnými, kde nalezl proměnnou $x = 4$ a její hodnotu vypsal. Funkce *func1* má v sobě definovanou hodnotu lokální proměnnou x a vypíše její hodnotu 5. Funkce *func3* vypíše chybu, neb není možné měnit hodnotu globální proměnné uvnitř funkce. Funkce *func4* vypíše hodnotu 8, protože pomocí příkazu **global** je možné zvýšit hodnotu globální proměnné x o 4 a její hodnotu následně vypsat.

1.5 Objekty, třídy a metody

Python je objektově orientovaný programovací jazyk. Téměř vše v Pythonu jsou vlastně objekty. Objekt je datová entita, která má svoje typické vlastnosti a metody, které na ni můžeme volat. Každý objekt má datovou část, která představuje informace v daném objektu uložené a operace (metody), které umožňují s těmito daty pracovat. Objekty jsou sdruženy do tříd. Každá třída má své specifické vlastnosti a metody. Mezi třídami definujeme dědičné relace. Zjednodušeně je možné prohlásit, že jedna třída dědí metody a vlastnosti některé jiné třídy, navíc může mít vlastní metody a některé metody třídy, od které dědí, může mít definovány jinak.

Obvykle používáme již naprogramované třídy objektů. Takovou třídou jsou například seznamy. Na objekty náležející k takovým již naprogramovaným třídám můžeme volat k nim náležející metody. Takovou již naprogramovanou metodou u seznamů je například metoda **.pop()**, kterou jsme již probrali výše. K většině tříd existuje podrobná dokumentace, ve které je možné dohledat podrobné informace o vlastnostech objektů dané třídy a o metodách, které na ni můžeme volat.

Často je ovšem v průběhu programování potřeba i vytvořit si třídu vlastní. K tomu slouží příkaz **class**, za nějž napíšeme název dané třídy a následně dvojtečku. Pod tento výraz postupně definujeme všechny metody náležející k dané třídě objektů. Metody definujeme pomocí příkazu **def** podobně jako funkce. První metodou musí být vždy speciální metoda **__init__** , která je volána vždy, když je daná třída používána pro vytvoření nového objektu náležejícího k této třídě. Jedná se o jedinou metodu, kterou musí vždy obsahovat každá třída. Takto můžeme například definovat třídu dinosaurů, jako zde:

```
#takto definujeme novou tridu s jedinou metodou
class Dinosaurus:
  def __init__(self, name, age):
    self.name = name
    self.age = age

#zde vytvarime objekt dane tridy
dino1 = Dinosaurus("Stegosaurus", 150000000)

#takto volame jednotlive argumenty objektu nove tridy
print(dino1.name)

print(dino1.age)

#parametry objektu muzeme i menit
dino1.age = 149999999

print(dino1.age)
```

Parametr *self* musí být přítomný jako argument každé metody, kterou budeme definovat. Tomuto parametru nepřiřazujeme při vytváření třídy žádnou hodnotu, vyjadřuje pouze objekt samý a používáme ho jako zástupný symbol pro abstraktní objekt dané třídy.

Třída s jedinou metodou by samozřejmě nebyla příliš užitečná. Proto obvykle chceme definovat více metod, které budou s našimi daty provádět různé zajímavé operace. Pro naši třídu dinosaurus můžeme například definovat metodu, při jejímž volání se náš dinosaurus představí:

```
#takto definujeme novou tridu a jeji metody
class Dinosaurus:
  def __init__(self, name, age):
    self.name = name
    self.age = age
  def predstaveni(self):
    print("I am " + self.name + " and I am a dinosaur!")

#takto pouzivame nami naprogramovanou metodu
dino1.predstaveni()
```

Nyní můžeme chtít vytvořit další třídu pro konkrétní druh dinosaurů, kupříkladu pro terapody. Terapodi jsou dinosauři, tedy mají všechny vlastnosti dinosaurů. Navíc ale chceme u terapodů metodu, která zařídí, aby terapod zařval. Abychom nemuseli u terapodů opakovat definici všech metod, jež náleží dinosaurům, mohou terapodi již nadefinované metody dinosaurů zdědit. Říkáme, že Dinosauři jsou tzv. *Parent class* a terapodi *Child class*. Název sám naznačuje, že *Child class* dědí vlastnosti *Parent class*. Že jedna třída dědí od jiné definujeme syntakticky tak, že při definici třídy, která má dědit, napíšeme do závorky za jméno třídy název třídy, od které tato třída dědí její vlastnosti. Třídu terapodů vytvoříme takto:

```
#takto definujeme novou tridu dedidici od dinosauru
class therapoda(Dinosaurus):
    def roar(self):
        print("Grrrrrr")

#takto vytvarime objekt nove tridy
dino2 = therapoda("T-rex", 66000000)

#nova trida ma vlastnosti tridy od ktere dedi
print(dino2.name, dino2.age)

#nova trida ma i sve vlastni vlastnosti
dino2.roar()
```

Můžeme také chtít vytvořit novou třídu sauropodů dědící od třídy dinosaurů, která má všechny metody dinosaurů, ale má navíc parametr výšky. To znamená, že musíme předefinovat metodu **__init__** tak, aby brala na vstupu o jeden parametr navíc. K tomu můžeme použít v Pythonu zabudovanou funkci **super()**, která využije vlastnosti metody **__init__** nebo i jiné metody definované u rodičovské třídy dinosaurů. Dále můžeme doplnit nové parametry metody **__init__**, které již jsou specifické pro třídu sauropodů. Třídu sauropodů a její specifické nové parametry můžeme naprogramovat, vytvořit a používat takto:

```
#takto vytvorime tridu sauropoda
class sauropoda(Dinosaurus):
```

```
    def __init__(self, name, age, height):
        super().__init__(name, age)
        self.height = height

#nyni muzeme vytvorit objekt tridy sauropoda
dino3 = sauropoda("Brachiosaurus", 150000000, 13)

#trida sauropoda muze uzivat metody dinosauru
print(dino3.name, dino3.age)
dino3.predstaveni()

#trida sauropodu ma navic i vlastni parametry
print(dino3.height)
```

Krom zcela nových metod, které si můžeme sami naprogramovat, existují i speciální metody. Jednu z nich už jsme potkali, jedná se o metodu **__init__**. Společným znakem všech těchto speciálních metod jsou dvě podtržítka na začátku i na konci názvu metody. Tyto metody nejsou volány uživatelem pomocí klasické ”tečkové notace”, nýbrž jsou volány samotným interpreterem jazyka Python. Příkazy, které tyto metody volají tak budou u uživatelem napsané třídy volat tuto speciální metodu, jejíž vlastnosti si u své třídy může uživatel sám nadefinovat. U metody **__init__** to znamená, že si u své třídy zvolíme, jakým způsobem má být nový objekt dané třídy konstruován. Jazyk Python obsahuje 83 takových speciálních metod, které může uživatel u jím napsaných tříd definovat. [5] V této práci nebudeme probírat všechny speciální metody, které může uživatel využívat. Využití speciálních metod demonstrujeme na metodě **__repr__**, která určuje řetězec, který bude reprezentovat objekt, pokusíme-li se s ním jako s řetězcem pracovat. Zkusíme-li například bez předchozího definování metody **__repr__** vypsat objekt ”dino3”z předchozího příkladu pomocí ”print(dino3), získáme něco jako <*__main__.sauropoda object at 0x000002C54D363160*>. Takto náš objekt vnímá Python, pro běžného uživatele však tato informace příliš přínosná nebude. Můžeme tedy definovat pro třídu sauropoda metodu **__repr__** a funkce **print()** bude pro objekty třídy sauropoda vypisovat námi definovaný řetězec. Metodu **__repr__** můžeme pro třídu sauropoda definovat například takto:

```

#takto vytvorime tridu sauropoda
class sauropoda(Dinosaurus):
    def __init__(self, name, age, height):
        super().__init__(name, age)
        self.height = height

    def __repr__(self):
        rep = self.name + " " + str(self.height)
        return rep

#takto vypiseme string reprezentujici objekt dino3
print(dino3)
```

Výsledkem výše vypsaného kódu bude řetězec ”Brachiosaurus 13”, který Python vypíše na obrazovku. Pro objekty třídy sauropoda nově bude Python používat podobné řetězce jako jejich reprezentaci v případě použití příkazů, které s řetězci pracují. Podobně je možné definovat pro třídy objektů vytvořené uživatelem i sčítání, násobení, délku atd. Pro každou

třídu mohou mít tyto operace zcela jiný význam.

1.6 Knihovny, moduly a balíčky

1.6.1 Moduly

Píšeme-li dlouhý program v Pythonu, není vhodné ho celý psát do jednoho skriptu. Definice funkcí, tříd a proměnných můžeme rozčlenit do několika souborů, které následně do hlavního *main* skriptu importujeme. Těmto souborům říkáme *moduly*.

Moduly importujeme pomocí příkazu **import** *název modulu*. Přitom je třeba dávat pozor na to, abychom pracovali nad správným adresářem, kde je daný modul uložen. Modul musí být uložen s příponou *.py* jako všechny soubory obsahující kód v jazyce Python. Je-li modul importován do našeho skriptu, voláme funkce a objekty v něm uložené pomocí zde již dříve použité ”tečkové notace”. Tedy napíšeme název modulu, tečku a následně název funkce nebo jiného objektu, který chceme zavolat. Alternativně můžeme z modulu importovat pouze konkrétní objekty, které pak můžeme volat pouze jejich jménem. Využijeme k tomu syntaxi **from** *název modulu* **import** *objekt*. Voláme-li více objektů, napíšeme je v této syntaxi za příkaz import všechny za sebe oddělené čárkou. Případně můžeme místo jmen objektů použít symbol *, čímž takto importujeme všechny objekty z našeho modulu.

Takto vytvoříme náš první modul, který bude obsahovat funkci faktoriál a funkci počítající největší společný dělitel dvou čísel. Tento kód uložíme do souboru s názvem *modul1.py*:

```
1
2  #funkce pro vypocet faktorialu
3  def fact (n):
4      x = n
5      f = n
6      while x > 1 :
7          x -= 1
8          f = f * x
9      return f
10
11 #funkce pro vypocet nejvetsiho spolecneho delitele
12 def gcd(a, b):
13     r = a%b
14     q = int(a/b)
15
16     while(r!=0):
17         a = b
18         b = r
19         q = int(a/b)
20         r = a - (b * q)
21
22     return(b)
23
24
```

Funkce uložené v souboru *modul1.py* můžeme volat například takto:

```
import modul1

x = modul1.gcd(6,4)

y = modul1.fact(5)

z = modul1.gcd(12, 9)

print(x, y, z)
```

Můžeme také použít volání s využitím příkazu **from** tak, jako zde:

```
from modul1 import gcd, fact

x = gcd(6,4)

y = fact(5)

z = gcd(12, 9)

print(x, y, z)
```

Stejného výsledku bychom v tomto případě dosáhli i nahrazením *gcd, fact* za příkazem **import** symbolem *. Pokud bychom importovali pouze jednu z obou funkcí, kód oznámí chybu, protože druhou z funkcí nebude interpreter Pythonu znát.

Moduly můžeme importovat také pod jiným názvem pomocí příkazu **as**. Tato syntaxe je užitečná pokud má modul dlouhý název, který nechceme často opisovat. Obecně se to nepovažuje za dobrou programátorskou praxi. U některých knihoven (viz. pozdější podkapitola) je však jejich přejmenovávání častou a běžně používanou praxí. Pomocí **as** můžeme náš modul volat např. takto a výsledek bude stejný jako v předchozích případech:

```
import modul1 as mod

x = mod.gcd(6,4)

y = mod.fact(5)

z = mod.gcd(12, 9)

print(x, y, z)
```

1.6.2 Balíčky

Balíčky jsou soubory obsahující více modulů, které mohou být i hierarchicky uspořádané. Jednotlivé moduly z balíčku pak voláme pomocí syntaxe **from** *jmeno_balicku* **import**

jmena_modulu oddělená čárkou. Balíček vytvoříme jednoduše tak, že v adresáři, nad kterým pracujeme, vytvoříme novou složku a do ní uložíme všechny moduly, které má balíček obsahovat a případně další soubory, které budou funkce v našich modulech využívat. Balíček tedy vlastně není nic více než složka s moduly a dalšími soubory, které naše moduly využívají. Z našeho modulu *modul1* můžeme vytvořit balíček např. tak, že funkce gcd a fact uložíme do dvou samostatných modulů, které můžeme nazvat jednoduše *gcd.py* a *fact.py*. Oba tyto moduly uložíme do balíčku *package1* Funkce pak můžeme volat následujícím způsobem a výsledek bude opět stejný jako v předchozích případech:

```
1
2 from package1 import gcd, fact
3
4 x = gcd.gcd(6,4)
5
6 y = fact.fact(5)
7
8 z = gcd.gcd(12, 9)
9
10 print(x, y, z)
```

1.6.3 Knihovny

Knihovny jsou vlastně moduly a balíčky, které za nás už dříve někdo naprogramoval. Můžeme je snadno stáhnout a importovat stejně jako importujeme balíčky a moduly. Některé z nejvýznamnějších knihoven, které využíváme pro matematické a statistické výpočty budou podrobně probrány v následujících kapitolách. V této podkapitole probereme jen několik menších knihoven, kterým není třeba věnovat tolik času, ale mohou být při práci často užitečné. Nejprve je však třeba vědět, jak knihovny stáhnout. Všechny knihovny, které budeme potřebovat můžeme stáhnout pomocí programu *pip*. Jak *pip* stáhnout a používat je možné zjistit na stránkách projektu (https://pip.pypa.io/en/stable/getting-started/). [6]
Konkrétní knihovnu pomocí *pip* instalujeme v systému Windows přes příkazový řádek příkazem **pip install** *název knihovny*. Např knihovnu NumPy, která je hlavním tématem následující kapitoly instalujeme pomocí příkazu:

```
1 pip install numpy
```

Knihovna math: První takovou knihovnou je knihovna *math*. Tato knihovna umožňuje využívat celou řadu základních matematických funkcí jako faktoriál, zaokrouhlování, absolutní hodnoty, odmocňování, umocňování nebo výpočet největšího společného dělitele dvou přirozených čísel. Dále umožňuje pracovat s mnoha matematickými symboly, které základní verze Pythonu neobsahuje, tedy například nekonečno, Eulerovo číslo, Ludolfovo číslo ale i další matematické konstanty. Využití některých základních funkcí z knihovny *math* ilustruje následující příklad:

```
1 import math
2
3 #nejvetsi spolecny delitel
```

```
print(math.gcd(3, 12))

#nejmensi spolecny nasobek
print(math.lcm(3,4))

#kombinacni cislo
print(math.comb(5, 3))

#absolutni hodnota
print(math.fabs(-4))

#faktorial
print(math.factorial(6))

#zaokrouhleni dolu
print(math.floor(4.5))

#zaokrouhleni nahoru
print(math.ceil(4.5))

#umocneni 4 na 3
print(math.pow(4,3))

#umocnovani na e (vraci mantisu a exponent)
print(math.frexp(2))

#druha odmocnina
print(math.sqrt(4))

#dalsi odmocniny (zde treti odmocnina)
print(math.pow(64, 1 / 3))

#nekonecno, pi, e
print(math.inf, math.pi, math.e)

#muzeme take zjistit, zda je cislo konecne
print(math.isfinite(math.inf - 4), math.isfinite(4))

#soucet cisel v iterovatelnem datovem typu
print(math.fsum([1,1,1,2]))

#soucin cisel v iterovatelnem datovem typu
print(math.prod([2,3,5,7]))

#prirozeny logaritmus cisla
print(math.log(40))

#logaritmus ze 64 s bazi 4
print(math.log(64,4))
```

Knihovna *math* nabízí také mnoho dalších užitečných funkcí. Všechny potřebné informace o nich je možné najít v dokumentaci jazyka Python. [4] Některé matematické funkce, které je možné volat pomocí knihovny *math*, je také možné používat v jiné verzi implementované v jiných knihovnách (například knihovna *Numpy*, o níž bude samostatná

kapitola). Je také dobré si povšimnout numerických nepřesností, které vznikají při použití některých matematických funkcí v Pythonu. Tyto nepřesnosti se mohou lišit mezi různými implementacemi v různých knihovnách.

Knihovna re: Další užitečnou knihovnou je knihovna *re*. Tato knihovna slouží pro práci s regulárními výrazy. Regulární výrazy jsou sekvence znaků, které je možné využít s pomocí této speciální knihovny jako argument pro vyhledávání. Hledáme výrazy obsahující nějakou speciální sekvenci znaků. S vyhledanými výrazy je následně možné dále pracovat. Při práci s regulárními výrazy mají některé výrazy speciální význam:

`[]`	množina znaků, které se mohou vyskytnout na dané pozici (například [a-k] značí množinu všech malých písmen od a až po k, dle abecedy, další často používané množiny je možné najít v dokumentaci)
`\`	znak specifikující speciální sekvenci znaků, která má v rámci regulárních výrazů speciální význam (speciální příkazy je možno dohledat v dokumentaci, například znak `\d` značí všechny číslovky), tento znak také umožňuje zbavit některý z jiných speciálních symbolů jeho významu, aby mohl být použit jako běžný znak, který toužíme vyhledat
.	jakýkoli znak krom nového řádku
`^`	značí, že hledaná sekvence znaků má začínat na sekvenci znaků uvedenou po znaku `^`
`$`	značí, že hledaná sekvence znaků má končit na sekvenci znaků uvedenou před znakem `$`
*	značí nula nebo více výskytů dané sekvence znaků
+	značí jeden nebo více výskytů dané sekvence znaků
?	jeden nebo žádný výskyt dané sekvence znaků
`{}`	přesně specifikovaný počet výskytů sekvence znaků (tento počet výskytů zapíšeme mezi složené závorky)
`|`	sekvence znaků před znakem `|` nebo sekvence znaků po něm
()	ohraničení sekvence znaků na níž chceme použít některý z výše zmíněných symbolů

Knihovna *re* umožňuje s regulárními výrazy pracovat pomocí několika speciálních funkcí. Jednou z nich je funkce **findall()**, která najde všechny výrazy v řetězci odpovídající zadanému regulárnímu výrazu. Naopak funkce **search()** vyhledá v řetězci výraz odpovídající zadaným kritériím, ale je-li jich v řetězci více, vrátí pouze první, který nalezne. Je třeba

mít na paměti, že funkce **search()** nevrací přímo řetězec, ale tzv. *Match objekt* obsahující informace o hledaném objektu. Chceme-li vrátit řetězec, musíme na tento objekt ještě použít metodu **.string**. Chceme-li vrátit část řetězce, ve které nastala shoda, použijeme metodu **group()**. Funkce **split()** rozdělí řetězec na více řetězců podle zadaného regulárního výrazu. Při každém výskytu daného regulárního výrazu se řetězec rozdělí. A nakonec funkce **sub()** substituuje námi zadaný regulární výraz řetězcem, který funkci zadáme. Substituuje právě tolik výskytů hledaného řetězce, kolik jí zadáme jako čtvrtý argument. To, že má být hledaný výraz vnímán jako *raw string* signalizujeme znakem r. Využití těchto funkcí z knihovny *re* spolu s regulárními výrazy demonstrují následující příklady:

Takto pomocí funkce **findall()** z knihovny *re* najdeme a vypíšeme všechna slova začínající na k a všechna slova začínající na j z řetězce "kocka kuna krava pes had jeseter lasice":

```
>>> import re
>>> txt = "kocka kuna krava pes had jeseter lasice"
>>> re.findall(r"k[a-z]*", txt)
['kocka', 'kuna', 'krava']
>>> re.findall(r"j[a-z]*", txt)
['jeseter']
```

Takto pomocí funkce **search()** ve stejném řetězci vyhledáme první slovo začínající na k. Speciální příkaz \b značí, že se hledaný vyraz vyskytuje na začátku slova, speciální příkaz \w značí písmena od a do z nebo číslovky od 0 do 9, nebo znak podtržítko (program vypíše řetězec "kočka"):

```
>>> x = re.search(r"\bk\w+", txt)
>>> x.group()
'kocka'
```

Takto rozdělíme řetězec podle všech mezer a výsledná slova python vrátí v seznamu:

```
>>> re.split(" ", txt)
['kocka', 'kuna', 'krava', 'pes', 'had', 'jeseter', 'lasice']
```

Takto substituujeme všechny mezery v řetězci znakem _:

```
>>> y = re.sub(" ", "_", txt, 6)
>>> y
'kocka_kuna_krava_pes_had_jeseter_lasice'
```

Funkce z knihovny *re* mohou být velmi užitečným nástrojem při vyhledávání v datových tabulkách, které budou popsány dále v kapitole pojednávající o knihovně *Pandas*.

Knihovna random: Nakonec poslední knihovnou, kterou probereme v této kapitole, je knihovna *random*. Jedná se o velmi užitečnou knihovnu pro statistické výpočty a především testování statistických funkcí. Tato knihovna umožňuje generování pseudo-náhodných čísel. Zcela náhodná čísla knihovna generovat neumí, ale pseudo-náhodná čísla jsou náhodnému výběru natolik blízko, že je vhodné je použít.

Základní funkcí pro generování náhodných celých čísel je funkce **randint(a, b)**, která vrátí celé číslo z intervalu [a,b]. Další důležitou funkcí je funkce **randrange(start, stop,**

step). Proměnné start, stop a step fungují podobně jako u funkce **range()**, kterou už jsme využívali dříve. Definují množinu celých čísel, z níž má funkce náhodně vybrat číslo, které vrátí. Rozdělení pravděpodobnosti na této množině je rovnoměrné diskrétní.

Takto generujeme náhodné číslo z intervalu [3,16]:

```
import random
random.randint(3,16)
```

Takto náhodně vybíráme jedno z čísel 2,4,6:

```
random.randrange(2, 8, 2)
```

Chceme-li vybrat náhodný prvek z nějakého seznamu, je možné využít funkci **choice()**, která náhodně vybere ze seznamu nějaký prvek opět za využití rovnoměrného diskrétního rozdělení pravděpodobnosti. Pro komplexnější výběry ze seznamu používáme funkci **choices(a,b,c)**, která ze seznamu *a* vybere náhodně *c* prvků. Je-li specifikován další seznam celých čísel *b*, který má stejný počet prvků jako *a*, pak čísla z *b* specifikují váhy prvků, které knihovna *random* přiřadí prvků ze seznamu *a* a na základě toho z tohoto seznamu vybere *c* prvků. Podobně jako funkce **choices()** funguje i funkce **sample()**. Chceme-li přeházet pořadí prvků seznamu, využijeme funkci **shuffle()**. Vybírat z nejrůznějších seznamů, ale i jiných iterovatelných datových typů, můžeme za pomoci těchto funkcí následujícím způsobem:

```
>>> import random
>>> seznam_prvku = ['jablko', 'jahoda', 'mandle', 'bobule']
>>> seznam_vahy = [1,10,12,4]
>>> random.choice(seznam_prvku)
'mandle'
>>> random.choices(seznam_prvku, k=3)
['bobule', 'mandle', 'jahoda']
>>> random.choices(seznam_prvku, seznam_vahy, k=3)
['jablko', 'mandle', 'jahoda']
>>> random.shuffle(seznam_prvku)
>>> seznam_prvku
['mandle', 'bobule', 'jahoda', 'jablko']
>>> random.sample(seznam_prvku, k = 3)
['jablko', 'jahoda', 'mandle']
>>> random.sample(seznam_prvku, k = len(seznam_prvku))
['jablko', 'bobule', 'mandle', 'jahoda']
```

Pro generování reálných náhodných čísel je základní funkcí funkce **random()** z knihovny *random*. Tato funkce generuje pseudo-náhodné reálné číslo z intervalu od 0 do 1. Chceme-li vybrat čísla z jiného intervalu můžeme buď na výsledek funkce **random()** aplikovat nějakou lineární transformaci, nebo použít funkci **uniform(a,b)**, která vybere za využití rovnoměrně spojitého rozdělení jedno reálné číslo z intervalu [a,b]. Pro generování reálných čísel z jiných rozdělení pravděpodobnosti můžeme využít funkce **gauss(mu, sigma)** pro generování čísel z normálního rozdělení se střední hodnotou mu a rozptylem sigma, funkci **betavariate(alpha, beta)** pro rozdělení beta s parametry *alpha* a *beta*, funkci **gammavariate(alpha, beta)** pro rozdělení gamma s parametry *alpha* a *beta* a funkci **expovariate(lambda)** pro exponenciální rozdělení se střední hodnotou *lambda*. Knihovna

random samozřejmě umožňuje i generování pseudo-náhodných čísel z jiných spojitých rozdělení. Konkrétní funkce lze dohledat v dokumentaci. Využití funkcí, které jsme výše popsali, demonstruje následující příklad:

```
>>> random.random()
0.9736402400312913
>>> x = 2 * random.random() + 2
>>> x
2.7715491242205097
>>> y = random.uniform(2,4)
>>> y
2.6863470268191403
>>> random.gauss(0,1)
-0.4761314537771839
>>> random.gauss(3,6)
0.04113184323994368
>>> random.expovariate(5)
0.1271019769388856
>>> random.betavariate(4, 5)
0.6143612788163533
>>>> random.gammavariate(4, 5)
16.900031095435498
```

Je třeba pamatovat na to, že přiřadíme-li proměnné hodnotu za pomoci některé funkce z knihovny *random*, Python si už danou hodnotu pamatuje a hodnota v proměnné je číslo, nikoli objekt třídy *random*. Občas může práce s náhodnými proměnnými tímto způsobem zapříčinit nepříjemné chyby.

1.7 Doplňující příklady ke kapitole

1.7.1 Příklady k procvičení

1. příklad:
   Napište funkci, která pro zadaný seznam čísel vrátí počet čísel z tohoto seznamu, která jsou dělitelná třemi.

2. příklad:
   S využitím funkcí z knihovny *math* napište funkci, která pro zadaná čísla x a y vrátí hodnotu funkce $f(x,y) = x^{12} + log(x-y) + y! - log_2(4xy)$.

3. příklad:
   Naprogramujte funkci *vrh_kostkami()*, která simuluje vrh běžnými šestistěnými hracími kostkami. Bere na vstup jedinou proměnnou n. Následně n-krát simuluje vrh kostkou a vrátí průměr hozených čísel.

4. příklad:
   Naprogramujte funkci, která bere na vstup seznam čísel a vrátí stejný seznam, ze kterého ale nejprve odstraní duplicitní prvky a všechny prvky dělitelné dvěma.

5. příklad:
   Naprogramujte funkci, která vyhledá v řetězci všechny řetězce ”Bc.”a nahradí všechny řetězce `"Bc."` řetězcem `"Mgr."` Zároveň spočítá počet výskytů řetězce `"Bc."` v řetězci a vrátí seznam o dvou prvcích, z nichž prvním prvkem seznamu bude počet výskytů řetězce `"Bc."` a druhým prvkem bude upravený řetězec.

1.7.2 Řešení

1. příklad:

```
def delitelne_tremi(x):
    count = 0
    for i in x:
        if i % 3 == 0:
            count +=1
    return count
```

2. příklad:

```
import math

def my_great_function(x,y):
    z = math.pow(x, 12) + math.log(x - y ,10)
    + math.factorial(y) - math.log(4*x*y, 2)
    return z
```

3. příklad:

```
import random

def vrh_kostkami(n):
    suma = 0
    for i in range(n):
        suma += random.choice([1, 2, 3, 4, 5, 6])
    return suma / n
```

4. příklad:

```
def change_list(x):
    x = list(set(x))
    y = []
    for i in x:
        if i % 2 == 1:
            y.append(i)
    return y
```

5. příklad:

```
def Bc_to_Mgr(txt):
    n = len(re.findall("Bc\.", txt))
   txt2 = re.sub("Bc\.", "Mgr.", txt, len(txt.split()))
   return [n, txt2]
```

Kapitola 2

Knihovna NumPy

2.1 Úvod do NumPy

NumPy je zkratka pro ”Numerical Python”. Jedná se tedy, jak už název napovídá, o knihovnu sloužící pro provádění numerických výpočtů v Pythonu. Jedná se o první knihovnu, které v této práci věnujeme samostatnou kapitolu. Při práci s většinou dalších knihoven budeme využívat objekty z knihovny NumPy, neb se jedná o jednu z nejdůležitějších knihoven pro práci s daty a statistikou v Pythonu.

Některé objekty z knihovny NumPy mají své obdoby i v jiných knihovnách (např. již probrané knihovny *random a math*) nebo dokonce v základním Pythonu. Asi nejdůležitějším z těchto objektů je pole. Vícerozměrná pole jsou extrémně důležitá pro většinu statistických výpočtů. Jako pole polí (”seznamy seznamů”) jsou v Pythonu reprezentovány matice, tenzory apod. Dále se jako vícerozměrná pole ukládají např. obrázky nebo datové tabulky. S takto uloženými daty v zabudovaných vícerozměrných seznamech ovšem Python počítá velmi pomalu. Jedná se o důsledek toho, že je Python dynamicky typovaný jazyk. Každé číslo je uloženo jako ukazatel na danou hodnotu v jazyce C, v němž je napsán i Python, a zároveň záznam o typu objektu a jeho parametrech. Tento způsob ukládání dat je ovšem značně pomalý. Oproti tomu objekty knihovny NumPy jsou napsány v jazyce C a C++ a provádí se s využitím tohoto výrazně rychlejšího programovacího jazyka. Mají také fixní datový typ, který musíme, vytváříme-li objekt předem definovat. NumPy tak umožňuje provádět výpočty výrazně rychleji. [7]

2.2 Datové typy v knihovně Numpy

Při práci s poli z knihovny NumPy není možné používat pole s prvky různých datových typů. Když vytváříme nové pole, je třeba datové typy jeho prvků přesně specifikovat. Přitom nepoužíváme datové typy vlastní Pythonu, které jsme probrali v předchozí kapitole, nýbrž datové typy vlastní jazyku C. Těmito datovými typy jsou: [7]

bool_	booleovský datový typ nabývající hodnot Pravda/Nepravda, True/False, 1/0
int_	defaultní datový typ pro celé číslo (int64 nebo int32)
intc	totožné s int_ jen jiný zápis
intp	celé číslo užívané pro indexování stejné jako intc
int8	Celé číslo z intervalu od -128 po 127
int16	celé číslo z intervalu od -32768 po 32767
int32	celé číslo z intervalu od -2147483648 po 2147483647
int64	celé číslo z intervalu od -9223372036854775808 po 9223372036854775807
uint8	nezáporné celé číslo z intervalu od 0 po 255
uint16	nezáporné celé číslo z intervalu od 0 po 65535
uint32	nezáporné celé číslo z intervalu od 0 po 4294967295
uint64	nezáporné celé číslo z intervalu od 0 po 18446744073709551615
float_	racionální číslo (zkratka pro float64)
float16	racionální číslo s jedním bitem reprezentujícím znaménko, 5 bity pro exponent a 10 bity pro mantisu
float32	racionální číslo s jedním bitem reprezentujícím znaménko, 8 bity pro exponent a 23 bity pro mantisu
float64	racionální číslo s jedním bitem reprezentujícím znaménko, 11 bity pro exponent a 52 bity pro mantisu
complex_	komplexní číslo(zkratka pro complex128)
complex64	komplexní číslo reprezentované dvěma 32 bitovými racionálními čísly
complex128	komplexní číslo reprezentované dvěma 64 bitovými racionálními čísly

Že mají objekty knihovny NumPy fixní datové typy je třeba mít při programování stále na paměti. Je také třeba se před psaním kódu zamyslet nad tím, s jak velkými čísly budeme pracovat. Využití datového typu reprezentovaného menším počtem bitů může urychlit výpočet, ale může být též zdrojem nepříjemných chyb.

2.3 Pole a jejich základní vlastnosti

Pole jsou velmi užitečným druhem objektů. Umožňují pracovat s vektory, maticemi, tenzory atd. Jako pole můžeme též reprezentovat rozsáhlé soubory dat, nebo obrázky. S poli můžeme pracovat už přímo v základním Pythonu jako se seznamy nebo seznamy seznamů. Dále jsou pak implementována různým způsobem v několika různých knihovnách jazyka Python. My se budeme v této práci zabývat poli z knihovny NumPy, která jsou vhodná a nejčastěji užívaná pro potřeby matematických a statistických výpočtů.

Abychom mohli začít používat objekty z knihovny NumPy, nejprve je nutné do našeho programu knihovnu NumPy importovat již dříve užívaným příkazem **import**. Knihovny NumPy je zvykem při importování do našeho programu přejmenovat na *np*. Při práci s obecnou knihovnou se tento postup nepovažuje za dobrou programátorskou praxi, ale pro některé často používané knihovny, jako je NumPy nebo Pandas, je natolik zažitý, že je vhodné knihovny přejmenovávat. Pole vytváříme příkazem **array()** z knihovny NumPy, který voláme na seznam nebo seznam seznamů takto:

```
1
2 import numpy as np
3
4 #takto vytvarime jednorozmerne pole
5 a = np.array([1, 2, 3, 5])
6
7 #takto vytvarime vicerozmerne pole
8 b = np.array([[1, 2, 3], [5,68,7]])
```

V tomto případě jsme v příkladu nespecifikovali datový typ prvků námi vytvářených polí. Python v takovém případě nevrátí chybovou hlášku, ale datový typ prvků pole odvodí z datového typu prvků seznamu, z něhož naše pole vytváříme. Chceme-li datový typ přesně specifikovat, což je často vhodnější přístup, můžeme pole z předchozího příkladu vytvořit s využitím argumentu *type* příkazu **array()** takto:

```
1
2 #takto vytvarime jednorozmerne pole
3 a = np.array([1, 2, 3, 5], dtype = 'float32')
4
5 #takto vytvarime vicerozmerne pole
6 b = np.array([[1, 2, 3], [5,68,7]], dtype = 'int64')
```

Povšimněte si, že využíváme datové typy jazyka C zmíněné v předchozí podkapitole. Tyto datové typ předáváme do argumentu *dtype* jako řetězce ohraničené uvozovkami.

Dále pro některé speciální typy polí, která jsou často používány, existují v NumPy speciální příkazy, které vytváření těchto polí usnadní. Jedná se např. o příkaz **ones()**, který

vytvoří matici daných rozměrů obsahující pouze čísla 1 daného datového typu:

```
one_matrix = np.ones((2,5), dtype = 'float64')
```

Příkaz **zeros()** vytvoří pole dané délky plné nul:

```
zero_array = np.zeros(7, dtype = 'int')
```

Příkaz **full()** vytváří matici daných rozměrů s prvky rovnými předem danému číslu:

```
two_matrix = np.full((2,4), 2, dtype = 'int8')
```

Příkaz **eye** vrací jednotkovou matici daných rozměrů:

```
identity_matrix = np.eye(3)
```

Všechny tyto matice si můžeme vypsat takto:

```
>>> zero_array
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
>>> one_matrix
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
>>> two_matrix
array([[2, 2, 2, 2],
       [2, 2, 2, 2]], dtype='int8')
>>> identity_matrix
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Chceme-li vektor celých čísel z nějakého intervalu, můžeme použít příkaz **arange()**, který funguje podobně jako již dříve zmíněný příkaz **range()**. Dále můžeme použít funkci **linspace()**, které předáme na vstup horní a dolní hranici intervalu a počet ekvidistantních čísel, která chceme generovat:

```
>>> np.arange(1,10)
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(27,32)
array([27, 28, 29, 30, 31])
>>> np.arange(12,35,2)
array([12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34])
>>> np.linspace(0, 1, 6)
array([0. , 0.2, 0.4, 0.6, 0.8, 1. ])
```

Dále můžeme vytvářet pole plná náhodných prvků z různých rozdělení pomocí funkcí z balíku *random* z knihovny NumPy. Ke generování náhodných matic s prvky z intervalu od 0 do 1 a rovnoměrně spojitým rozdělením slouží příkaz **random()**. Ke generování náhodných celých čísel z daného intervalu s rovnoměrným rozdělením použijeme funkci **randint()**. A ke generování náhodných matic z normálního rozdělení slouží funkce **normal()**. V dokumentaci knihovny NumPy je možné najít také funkce generující náhodná pole z jiných rozdělení. NumPy nabízí relativně širokou paletu takových funkcí. [9] Nyní si ukážeme několik příkladů využití náhodných generátorů NumPy polí:

Matici 3x3 s náhodnými prvky od 0 do 1 generujeme takto:

```
mat1 = np.random.random((3,3))
```

Matici 2x4 náhodných celých čísel z intervalu od 15 do 25 generujeme takto:

```
mat2 = np.random.randint(15,25, (2,4))
```

Matici 3x2 čísel vybíraných z normovaného normálního rozdělení generujeme takto:

```
mat3 = np.random.normal(0,1, (3,2))
```

Matici 3x2 čísel vybíraných z normálního rozdělení se střední hodnotou 2 a rozptylem 5 generujeme takto:

```
mat4 = np.random.normal(2,5, (3,2))
```

Ať už pole vytvoříme jakkoli, budeme s ním chtít pravděpodobně dále pracovat a získávat o něm informace. Základními vlastnostmi polí jsou počet dimenzí, který můžeme zjistit pomocí atributu *ndim*, velikost každé z těchto dimenzí, kterou zjišťujeme pomocí atributu *shape* a celkový počet prvků pole, který můžeme zjistit pomocí atributu *size*. Dále můžeme u každého pole zjistit datový typ jeho prvků pomocí atributu *dtype*. Tato operace ovšem vrátí některý z výše zmíněných datových typů vlastních knihovně NumPy. Příkaz **type()**, který jsme již zmínili dříve vrátí, je-li použit na některé z NumPy polí pouze řetězec ”numpy.ndarray”, tedy typ objektu, na nějž byl použit. Ve specifických případech nás také může zajímat, kolik bytů představuje každý prvek pole, což zjistíme pomocí atributu *itemsize* a nebo kolik bytů zabírá celé pole, na což použijeme atribut *nbytes*. U matice mat4, kterou jsme vytvořili v předešlém příkladu, je možno určit tyto informace následujícím způsobem:

```
>>> type(mat4)
<class 'numpy.ndarray'>
>>> mat4.dtype
float64
>>> print("pocet prvku matice: ", mat4.size)
pocet prvku matice:  6
>>> print("velikost dimenzi matice: ", mat4.shape)
velikost dimenzi matice:  (3, 2)
>>> print("pocet dimenzi matice: ", mat4.ndim)
pocet dimenzi matice:  2
>>> print("pocet bytu na celou matici: ", mat4.nbytes)
pocet bytu na celou matici:  48
>>> print("pocet bytu kazdeho prvku: ", mat4.itemsize)
pocet bytu kazdeho prvku:  8
```

Data uložená v nějakém poli můžeme volat příslušnými indexy v hranatých závorkách. Tak jako je v Pythonu obvyklé můžeme i pole z knihovny NumPy indexovat od konce zápornými čísly. Jako vždy je třeba dbát na to, že Python indexuje od nuly. Vícerozměrná pole indexujeme indexy v hranatých závorkách, kde čísla oddělujeme čárkou. Pomocí této notace můžeme polím měnit prvky. Nesmí se ovšem stát, že tímto způsobem do pole vložíme prvek jiného datového typu než mají ostatní prvky pole. Indexování polí z knihovny NumPy tedy funguje velmi podobně, jako indexování seznamů, které jsme si již ukázali v předchozí kapitole. Rozdíly plynou především z toho, že pole v NumPy mají, na rozdíl od seznamů, fixní datový typ. Můžeme také pomocí indexů volat různé podmnožiny původního pole pomocí ”dvojtečkové notace”, se kterou jsme se již dříve setkali. Následující příklad demonstruje, jak je možné pole indexovat:

```
>>> #zde vytvarime pole, se kterymi budeme pracovat
>>> x = np.array([1, 5, 4, 12], dtype = 'float32')
>>> y = np.array([[1,2], [4,8]], dtype = 'float32')
>>> z = np.array([[1, 4, 5], [4, 12, 56], [11, 43, 54]])
>>> #takto vypiseme 2. a 4. prvek x
>>> x[1]
5.0
>>> x[3]
12.0
>>> #takto vypiseme prvek v prvnim radku a druhem sloupci y
>>> y[0,1]
2.0
>>> #takto zmenime hodnotu 3. prvku x na hodnotu 27
>>> x[2] = 27
>>> x
array([ 1.,  5., 27., 12.], dtype='float32')
>>> #takto otestujeme, zda se dva prvky z x rovnaji
>>> x[2] == x[0]
False
>>> #takto vypiseme prvni 3 prvky z x
>>> x[:3]
array([ 1.,  5., 27.], dtype='float32')
>>> #takto vypiseme vsechny prvky z x az na prvni 3
>>>x[3:]
array([12.], dtype='float32')
>>> #takto vypiseme prostredni 2 prvky x
>>> x[1:3]
array([ 5., 27.], dtype='float32')
>>> #zde z x vypiseme kazdy prvek na liche pozici
>>> x[::2]
array([ 1., 27.], dtype='float32')
>>> #zde kazdy z x prvek na sude pozici
>>> x[1::2]
array([ 5., 12.], dtype='float32')
>>> #takto vypiseme prvni radek y
>>> y[0, :]
array([1., 2.], dtype='float32')
>>> #takto vypiseme prvni sloupec z
>>> z[:, 0]
array([ 1,  4, 11])
>>> #takto vypiseme postupne oba hlavni minory matice z
>>> z[0:2, 0:2]
array([[ 1,  4],
       [ 4, 12]])
>>> z[1:, 1:]
array([[12, 56],
       [43, 54]])
```

Pole může být dále užitečné různým způsobem přeskládávat, rozdělovat na menší pole a nebo naopak skládat dohromady. K přearanžování pole používáme metodu **reshape**(), která bere jako argument n-tici, podle které má přearanžovat pole. Můžeme tak například vyrábět matice z vektorů apod. Je třeba dbát na to, aby nové přearanžované pole mělo stejný počet prvků jako pole původní, jinak Python ohlásí chybu. Dalším způsobem, jak měnit rozložení pole je příkaz **newaxis**. Budeme-li indexovat pomocí hranatých závorek a

přidáme dimenzi, na jejíž místo napíšeme **newaxis**, zvýší se dimenze pole o jedna. Pomocí těchto příkazů můžeme provádět užitečné transformace polí tak, jako na následujícím příkladu:

```
>>> x = x.reshape((2,2))
>>> x
array([[ 1.,  5.],
       [27., 12.]], dtype='float32')
>>> z = z.reshape(1,9)[0]
>>> z
array([ 1,  4,  5,  4, 12, 56, 11, 43, 54])
>>> zz = z[:, np.newaxis]
>>> zz
array([[ 1],
       [ 4],
       [ 5],
       [ 4],
       [12],
       [56],
       [11],
       [43],
       [54]])
```

Jednorozměrné pole můžeme rozdělit na více polí pomocí funkce **split()**, která bere na vstup rozdělované pole a seznam indexů, na nichž má být pole rozděleno na více kusů. Pro rozdělování matic na více podmatic můžeme použít funkce **vsplit()** a **hsplit()**, které berou na vstupu matici, kterou mají rozdělit buď vertikálně nebo horizontálně a seznam řádků nebo sloupců, podle kterých mají matici rozřezat na kusy. Tyto metody můžeme aplikovat na naše pole z původního příkladu:

```
>>> z1, z2, z3 = np.split(z, [3,6])
>>> z1, z2, z3
(array([1, 4, 5]), array([ 4, 12, 56]), array([11, 43, 54]))
>>> y1, y2 = np.hsplit(y, [1])
>>> y1
array([[1.],
       [4.]], dtype='float32')
>>> y2
array([[2.],
       [8.]], dtype='float32')
>>>> y11, y22 = np.vsplit(y, [1])
>>> y11
array([[1., 2.]], dtype='float32')
>>> y22
array([[4., 8.]], dtype='float32')
```

Pole je dále možné skládat za sebe nebo vedle sebe a vytvářet tak z více polí jedno nové pomocí funkce **concatenate()**, která skládá pole za sebe podle osy, kterou nastavíme v parametru *axis*, a nebo pomocí funkcí **vstack()** a **hstack()**, které skládají pole vedle sebe nebo resp. pod sebe. Vždy je třeba mít na paměti, že dimenze polí, která skládáme, musí být kompatibilní. Pole, která jsme vytvořili v předchozím příkladu tedy můžeme přeskládat např. takto:

```
>>> z13 = np.concatenate([z1, z3])
>>> z13
```

```
array([ 1,  4,  5, 11, 43, 54])
>>> zzz = np.concatenate([z13, z], axis = 0)
>>> zzz
array([ 1,  4,  5, 11, 43, 54,  1,  4,  5,  4, 12, 56, 11, 43, 54])
>>> yy = np.concatenate([y, y], axis = 0)
>>> yy
array([[1., 2.],
       [4., 8.],
       [1., 2.],
       [4., 8.]], dtype='float32')
>>> yy = np.vstack([y,y])
>>> yy
array([[1., 2.],
       [4., 8.],
       [1., 2.],
       [4., 8.]], dtype='float32')
>>> yyy = np.concatenate([y, y], axis = 1)
>>> yyy
array([[1., 2., 1., 2.],
       [4., 8., 4., 8.]], dtype='float32')
>>> yyy = np.hstack([y,y])
>>> yyy
array([[1., 2., 1., 2.],
       [4., 8., 4., 8.]], dtype='float32')
```

Tento výraz vypíše chybovou hlášku (matice nemají stejný rozměr):

```
yy = np.concatenate([y, z], axis = 0)
```

2.4 Universal functions a vektorizace výpočtů

Jak už bylo zmíněno výše, jedním z hlavních výhod používání knihovny NumPy je možnost provádět výpočty výrazně rychleji, než by to bylo možné v základním Pythonu. Běžné Pythonovské cykly jsou velmi pomalé stejně tak jako iterování přes seznam. Python musí totiž u každého prvku přes který iteruje nejprve zjistit jeho datový typ a následně přeložit kód do jazyka C, v němž je Python napsán a který výpočet provede, a pak přeložit zase zpět. Knihovna NumPy umožňuje tento typ výpočtu vektorizovat. Iterace přes NumPy pole je výrazně rychlejší, neb pole má fixní datový typ a ten se zjišťuje pouze jednou, ne u každého prvku zvlášť. K tomuto typu práce s poli slouží v knihovně NumPy předprogramované tzv. Univerzální funkce (neboli *ufunc*). [7]

Často používané *ufunc* jsou předprogramované v knihovně NumPy. Většinu základních aritmetických operací můžeme volat buď pro danou operaci běžně užívaným symbolem, nebo pomocí speciální funkce, které můžeme přidat další parametry. Hlavní parametry, které u *ufunc* používáme jsou *where*, ve kterém specifikujeme pomocí podmínky nebo booleovského pole, na které prvky pole se má funkce aplikovat, *out*, v němž uvádíme cílový objekt, do něhož se má uložit výsledek a *dtype*, který určuje datový typ výstupního objektu. Některé pokročilejší *ufunc* můžeme volat pouze pomocí názvu funkce. Nejčastěji používanými předprogramovanými *ufunc* v knihovně Python jsou: [8]

1. **add()** - funkce, která sečte prvky dvou polí, které jsou jí předány na vstupu tak, že výsledkem bude pole součtů (též možno zapsat pomocí symbolu + mezi dvěma poli)

2. **subtract()** - odečte sobě odpovídající prvky jednoho pole od prvků druhého pole a vrátí pole rozdílů (též možno zapsat pomocí symbolu - mezi dvěma poli)

3. **multiply()** - vynásobí sobě odpovídající prvky obou polí a vrátí pole násobků (též možno zapsat pomocí symbolu * mezi dvěma poli)

4. **divide()** - dělí prvky prvního pole odpovídajícími prvky druhého pole a vrátí pole podílů (též možno zapsat pomocí symbolu / mezi dvěma poli)

5. **power()** - umocní prvky prvního pole na odpovídající prvky z druhého pole (též možno zapsat pomocí symbolu ** mezi dvěma poli)

6. **mod()** - dělí prvky prvního pole odpovídajícími prvky druhého pole a vrátí pole zbytků po dělení (též možno použít ufunc **remainder()** se stejným výsledkem)

7. **divmod()** - vrátí dvě pole, jedno je výsledkem funkce **divide()** a druhé výsledkem funkce **mod()**

8. **absolute()** - na vstupu bere pole a vrací pole absolutních hodnot jeho prvků

9. **trunc()** - zaokrouhlí všechny prvky pole dolů (totožný výstup s funkcí **fix()** a **floor()**)

10. **around()** - zaokrouhlí všechny prvky pole na daný počet desetinných míst, tento počet desetinných míst bere jako druhý argument

11. **ceil()** - zaokrouhlí všechny prvky pole nahoru

12. **log2()** - aplikuje na všechny prvky pole logaritmus o základu 2 a vrátí pole těchto logaritmů

13. **log10()** - aplikuje na všechny prvky pole logaritmus o základu 10 a vrátí pole těchto logaritmů

14. **log()** - aplikuje na všechny prvky pole logaritmus o základu e (Eulerovo číslo) a vrátí pole těchto logaritmů

15. **sum ()** - sečte všechny prvky pole nebo polí, která dostane na vstupu dohromady a vrátí jedno číslo

16. **cumsum()** - postupně sčítá prvky pole, které dostane na vstupu a vrátí pole kumulativních součtů

17. **prod()** - vynásobí všechny prvky pole nebo polí, která dostane na vstupu dohromady a vrátí jedno číslo (je-li specifikován parametr axis = 1, pak vynásobí mezi sebou prvky všech polí, která dostane na vstupu a vrátí pole o tolika prvcích, kolik bylo seznamů na vstupu)

18. **cumprod()** - postupně násobí prvky pole, které dostane na vstupu a vrátí pole kumulativních násobků

19. **diff()** - vrátí pole rozdílů sousedních prvků pole, které dostal na vstupu, výsledné pole bude mít tedy o 1 méně prvků než pole na vstupu (pomocí parametru n lze tuto operaci provádět opakovaně po sobě a vrátí se až pole vzniklé několikanásobným opakováním funkce **diff()**)

20. **lcm()** - vrátí nejmenší společný násobek všech prvků pole (na pole s více než dvěma prvky je přitom třeba použít metodu **reduce()**, která zajistí opakování funkce, dokud nebude výsledkem jen jedno číslo)

21. **gcd()** - vrátí největší společný dělitel všech prvků pole(na pole s více než dvěma prvky je přitom třeba použít metodu **reduce()**, která zajistí opakování funkce, dokud nebude výsledkem jen jedno číslo)

22. **deg2rad(), rad2deg()** - převede všechny prvky pole ze stupňů na radiány, **rad2deg()** dělá opak (užitečné při práci s trigonometrickými funkcemi)

23. **sin(), coc(), tan(), arcsin(), arccos(), arctan()** - aplikují na každý prvek pole funkci sinus, cosinus, tangents, arcussinus, arcuscosinus nebo arcustangents a vrátí pole výsledků dané operace (funkce počítají s radiány, pokud jsou hodnoty uložené v poli ve stupních, je třeba je nejprve převést)

Následující příklad demonstruje využití některých výše zmíněných funkcí (vzhledem k množství různých *ufunc* nebudeme demonstrovat využití všech, navíc syntaxe je u mnoha z výše zmíněných *ufunc* velmi podobná):

```
>>> a = np.array([[1, 2, 3] , [4, 5, 6]])
>>> b = np.array([[5, 4, 3], [4, 7, 1]])
>>> c = np.array([3.5, 3.666, 5.844, 9.7])
>>> np.add(a, b)
array([[ 6,  6,  6],
       [ 8, 12,  7]])
>>> np.multiply(a, b)
array([[ 5,  8,  9],
       [16, 35,  6]])
>>> np.power(a, b)
array([[    1,    16,    27],
       [  256, 78125,     6]], dtype='int32')
>>> np.divmod(a, b)
(array([[0, 0, 1],
       [1, 0, 6]], dtype='int32'), array([[1, 2, 0],
       [0, 5, 0]], dtype='int32'))
#zaokrouhleni dolu
>>> np.ceil(c)
array([ 4.,  4.,  6., 10.])
>>> np.around(c, 2)
array([3.5 , 3.67, 5.84, 9.7 ])
>>> np.log(c)
array([1.25276297, 1.29910115, 1.76541549, 2.27212589])
>>> np.sum(a)
```

```
21
>>> np.cumsum(a)
array([ 1,  3,  6, 10, 15, 21], dtype='int32')
>>> np.lcm.reduce(a)
array([ 4, 10,  6])
>>> aa = np.rad2deg(a)
>>> aa
array([[ 57.29577951, 114.59155903, 171.88733854],
       [229.18311805, 286.47889757, 343.77467708]])
>>> np.sin(aa)
array([[ 0.67952262,  0.99706973,  0.78348689],
       [ 0.15254773, -0.55965222, -0.97373053]])
```

Zajímavostí knihovny NumPy je, že je mnoho těchto funkcí implementováno nejen jako funkce, ale i jako metody náležící k objektu *numpy.ndarray*. Například následující dva způsoby sčítání prvků pole *a* jsou ekvivalentní a v proměnné *b* i v proměnné *bb* bude uloženo číslo 6:

```

import numpy as np

a = np.array([1, 2, 3])

b = np.sum(a)
bb = a.sum()
```

Krom těchto základních matematických funkcí umožňuje knihovna NumPy používat jako *ufunc* také některé funkce množinové algebry. NumPy pole v takovém případě můžeme používat jako množiny a hledat jejich průniky, sjednocení apod. Jako s množinami ovšem můžeme pracovat pouze s jednodimenzionálními poli. Základními *ufunc* pro práci s množinami předprogramovanými v NumPy jsou: [8]

1. **unique()** - na vstup bere pole a vrátí pole, kde je každý prvek z původního pole zastoupen právě jednou a vytvoří tedy ”množinu (set array)”(má tedy stejný nebo menší počet prvků než původní pole)

2. **union1d()** - na vstup bere dvě pole a vrací jejich sjednocení (dále můžeme specifikovat parametr ”assume_unique = True”, což zajistí, že každá hodnota ve výstupu bude ve výstupním poli zastoupena právě jednou)

3. **intersect1d()** - na vstup bere dvě pole a vrací jejich průnik(dále můžeme specifikovat parametr ”assume_unique = True”, což zajistí, že každá hodnota ve výstupu bude ve výstupním poli zastoupena právě jednou)

4. **setdiff1d()** - na vstup bere dvě pole a vrací jejich množinový rozdíl (dále můžeme specifikovat parametr ”assume_unique = True”, což zajistí, že každá hodnota ve výstupu bude ve výstupním poli zastoupena právě jednou)

Práci s poli z knihovny NumPy jako s množinami pomocí výše zmíněných *ufunc* ilustruje následující příklad:

```
>>> a = np.array([1, 2, 5, 4])
>>> b = np.array([4, 5, 7, 8])
>>> c = np.array([8,4, 1, 1])
>>> np.unique(c)
array([1, 4, 8])
>>> np.union1d(a, b)
array([1, 2, 4, 5, 7, 8])
>>> np.intersect1d(a, b)
array([4, 5])
>>> np.setdiff1d(a, b)
array([1, 2])
```

Zdá-li se snad čtenáři, že tato nabídka funkcí typu *ufunc*, které jsou v NumPy již předprogramované, je nedostatečná, je možné si naprogramovat vlastní univerzální funkce. Postup vytvoření vlastní *ufunc* je velmi jednoduchý. Stačí naprogramovat vlastní funkci, která popíše, jak bude nová *ufunc* nakládat s jednotlivými prvky vektorů a následně na tuto funkci zavolat funkci **frompyfunc()** z knihovny NumPy. Dalšími parametry **frompyfunc()**, které je nutno specifikovat jsou *nin*, ve kterém specifikujeme kolik polí bude brát nová *ufunc* na vstupu, a *nout*, ve kterém specifikujeme počet polí, která nová univerzální funkce vrátí na výstupu. Funkce **frompyfunc()** následně zapíše novou univerzální funkci do knihovny NumPy stažené na vašem počítači a je ji možno používat jako předprogramované univerzální funkce. Vytvoření nové univerzální funkce demonstruje následující příklad, v němž bude vytvořena a volána funkce, které na vstupu předáváme pole přilehlých stran pravoúhlých trojúhelníků a pole protilehlých stran a vrátí pole přepon těchto trojúhelníků:

```

import numpy as np

def hypotenuse(a, b):
    return np.sqrt(a*a + b*b)

hypotenuses = np.frompyfunc(hypotenuse, nin = 2, nout = 1)

a = np.array([3, 4, 5, 4, 12])
b = np.array([4, 3, 5, 14, 1])

print(hypotenuses(a, b))
```

Kód vypíše pole přepon trojúhelníků s přeponami a protilehlými přeponami z b, které vypadá takto:

```
[5.0 5.0 7.0710678118654755 14.560219778561036 12.041594578792296]
```

Tímto způsobem můžeme vektorizovat mnoho funkcí, které bychom chtěli opakovaně používat v rámci nějakého cyklu a vektorizací pomocí NumPy běh programu výrazně zrychlit. I vzhledem k časové výhodnosti tohoto postupu a menší náchylnosti k chybám je ovšem, je-li to možné, lepší používat univerzální funkce, které jsou v NumPy již předprogramované. Dále je třeba dbát na to, abychom při vytváření nových NumPy nepřepsali již naprogramované funkce, které pak již nebude možno používat. V případě, že si uživatel přesto některou z předprogramovaných funkcí přepíše, nejlepším řešením je celou knihovnu NumPy přeinstalovat.

2.5 Broadcasting

Operace mezi poli z knihovny Numpy obvykle provádíme na stejně velkých polích tak, že spolu interagují prvky se stejným indexem. Specifikem NumPy oproti jiným programovacím jazykům je to, že pokus o aplikaci funkce na dvě pole o různých rozměrech neskončí vždy chybovou hláškou, ale Python pole pouze upraví tak, aby bylo možné výpočet provést. Tento systém může být zdrojem nepříjemných chyb, ale také poskytuje možnost pro úsporný zápis prováděných výpočtů. Slovem "broadcasting"rozumíme systém pravidel pro úpravu nestejně velkých polí, podle kterých Python výpočty mezi takovými poli v rámci knihovny NumPy provádí.

Provádíme-li vektorizovanou operaci mezi libovolným polem a skalárem, rozšíří se skalár na pole o stejném počtu prvků a následně se daná operace provede. Aplikujeme tedy operaci, kterou bychom normálně aplikovali na dva prvky vektoru se stejným indexem, na každý prvek pole společně se zadaným skalárem. Aplikujeme-li funkci na pole s různým počtem dimenzí, zkopíruje se pole s menším počtem dimenzí několikrát pod sebe tak, aby mělo výsledné pole stejný počet prvků jako větší pole. Např. násobíme-li matici 3x5 vektorem délky 3, zkopíruje se tento vektor 5-krát pod sebe a následně se sobě odpovídající prvky obou matic vynásobí a funkce vrátí výslednou matici rozměrů 3x5. Pokud by byl vektor, kterým matici 3x5 násobíme, čtyřprvkový, tento postup by nefungoval a Python ohlásí chybu. Broadcasting tedy umožní provést operaci mezi poli s různým počtem prvků pouze tehdy, pokud ve větším poli existují dimenze s velikostí odpovídající dimenzím menšího pole. Pokud má menší pole alespoň jednu dimenzi takové velikosti, jakou žádná dimenze většího pole nemá, pak bude ohlášena chyba. Vyjímkou je případ, kdy mají pole různé velikosti jednotlivých dimenzí, ale je možné zopakovat v některém směru jedno pole a v jiném směru druhé pole tak, aby bylo možno operaci provést. Například násobíme-li 3-prvkový sloupcový vektor 4-prvkovým řádkovým vektorem, je možné přidat k prvnímu vektoru napravo čtyři stejné napravo a výslednou matici násobit maticí získanou jako druhý vektor zopakovaný třikrát pod sebe.

Broadcasting u různých velikostí polí demonstruje následující příklad:

```
>>> import numpy as np
>>> a = np.array([1,3,4])
>>> b = np.array([[1],[4],[5],[2]])
>>> a * b
array([[ 1,  3,  4],
       [ 4, 12, 16],
       [ 5, 15, 20],
       [ 2,  6,  8]])
>>> c = np.array([[1,4,5], [5,6,8], [2,2,2]])
>>> a * c
array([[ 1, 12, 20],
       [ 5, 18, 32],
       [ 2,  6,  8]])
>>> d = np.array([[[1,4,5], [5,6,8], [2,2,2]],
            [[1,7,5], [1,6,8], [4,2,9]]])
>>> c * d
array([[[ 1, 16, 25],
```

```
         [25, 36, 64],
         [ 4,  4,  4]],

        [[ 1, 28, 25],
         [ 5, 36, 64],
         [ 8,  4, 18]]])
>>> d * a
array([[[ 1, 12, 20],
         [ 5, 18, 32],
         [ 2,  6,  8]],

        [[ 1, 21, 20],
         [ 1, 18, 32],
         [ 4,  6, 36]]])
```

2.6 Výpočty linearní algebry pomocí NumPy

V předchozích kapitolách jsme často psali o NumPy polích jako o maticích nebo vektorech. Běžné násobení apod. u nich ovšem neodpovídá násobení maticovému a Python k nim ani takto nepřistupuje. V pravém smyslu slova se o matice nejedná. S jistým stupněm abstrakce o nich ovšem jako o těchto objektech známých z lineární algebry je možno uvažovat. Knihovna NumPy umožňuje aplikovat na pole většinu funkcí a operací, které pro výpočty lineární algebry potřebujeme. Můžeme tak v Pythonu provádět maticové výpočty. Většina těchto operací je uložena v modul *linalg* náležícím pod knihovnu NumPy. Namísto běžné syntaxe **np.název_funkce** tak budeme často využívat syntaxi **np.linalg.název_funkce**.

Nejprve si vytvoříme dvě matice, na nichž budeme výpočty lineární algebry prováděné pomocí knihovny NumPy demonstrovat:

```
A = np.array([[1,2,3], [2,3,5], [5,4,7]])
B = np.array([[4,8,3], [1,8,5], [7,4,1]])
```

Sčítání a odčítání matic i násobení po prvcích můžeme provádět jednoduše pomocí již dříve užívaných univerzálních funkcí:

```
C = A + B
CC = A - B
CCC = A * B
```

Skalární součin je implementován pomocí funkce **vdot()** a vnější součin pomocí funkce **inner()**:

```
D = np.inner(A, B)
DD = np.vdot(A,B)
```

Funkce **dot()** provádí skalární součin mezi dvěma 1D poli a maticové násobení mezi dvěma 2D poli:

```
E = np.dot(A, B)
```

Maticové násobení je implementováno pomocí funkce **matmul()**, dá se ale zapsat i pomocí znaku @ mezi dvěma maticemi:

```
E = A @ B
E = np.matmul(A, B)
```

Matice můžeme transponovat pomocí příkazu **transpose()**:

```
EE = np.transpose(E)
```

Determinant počítáme pomocí funkce **det()** z balíku *linalg*:

```
determinant_A = np.linalg.det(A)
```

V balíku *linalg* jsou také obsaženy funkce **inv()** pro výpočet multiplikativní inverze matice, **pinv()** pro výpočet pseudoinverze matice:

```
AN = np.linalg.inv(A)

ANP = np.linalg.pinv(A)

print(AN @ A){\bf }
```

V tomto příkladu je možno se přesvědčit, že násobení matice její inverzí nevrátí jednotkovou matici, nýbrž matici, která má prvky mimo diagonálu velmi malé a je tak "skoro jednotková". Při výpočtech vznikají numerické chyby. Počítač počítá diskrétně nikoli spojitě a tak musí zaokrouhlovat. Tato numerická chyba se pak v průběhu výpočtu může dále zvětšovat. S numerickými chybami je tedy třeba počítat a snažit se je minimalizovat.

Příkaz **trace()** používáme pro výpočet stopy matice:

```
stopa_A = np.trace(A)
```

Příkaz **matrix_rank()** slouží pro výpočet hodnosti matice:

```
hodnost_A = np.linalg.matrix_rank(A)
```

Řešení systému rovnic tvaru $Ax = b$ získáme pomocí funkce **solve(A, b)** z balíku *linalg*. Vždy musíme definovat i vektor b:

```
koreny_A = np.linalg.solve(A, b = np.array([0,0,0]))
```

Chceme-li vypočítat normu matice, použijeme k tomu funkci **norm()** z balíku *linalg*. Funkce **norm()** umožňuje počítat osm různých maticových norem. Konkrétní normu, kterou chceme u daného pole spočítat, definujeme v parametru *ord* (hodnoty náležící různým maticovým normám možno nalézt v dokumentaci, např. spektrální normu matice značí *ord = 'nuc'*). Nespecifikujeme-li parametr *ord*, funkce **norm()** automaticky počítá s Frobeniovou normou:

```
np.linalg.norm(A)

np.linalg.norm(A, ord = 'nuc')
```

Dále je možné pomocí balíku *linalg* vypočítat vlastní čísla a vlastní vektory matice. Chceme-li znát pouze vlastní čísla použijeme příkaz **eigvals()**, který vrátí pole vlastních čísel matice, chceme-li znát i vlastní vektory, použijeme příkaz **eig()**. Funkce **eig()** vrátí vektor vlastních čísel a 2D pole vlastních vektorů, kde každý sloupec tohoto pole je jeden z vlastních vektorů. Vlastní vektory jsou normované. Výpočet vlastních čísel a vektorů demonstruje následující příklad:

```
vlastni_cisla = np.linalg.eigvals(A)

vlastni_cisla, vlastni_vektory = np.linalg.eig(A)
```

2.7 Vyhledávání v polích a řazení polí

V existujícím poli v rámci programování často chceme vyhledávat prvky s nějakou konkrétní vlastností. Ne nadarmo jsou efektivní algoritmy pro vyhledávání jednou z nejstarších úloh teoretické informatiky. Ve velmi rozměrných polích to chceme dělat efektivně, což pole knihovny NumPy umožňují. Vyhledávání v klasických seznamech ze základního Pythonu je oproti vyhledávání v NumPy polích velmi pomalé.

Základní metodou pro vyhledávání v polích v knihovně NumPy je metoda **where()**. Tato funkce bere na vstup logický výraz v němž vystupuje název pole a vrátí indexy prvků pole, které tomuto logickému výrazu odpovídají. Pro vícerozměrná pole tato metoda nefunguje. Chceme-li vyhledávat přes n-rozměrné pole, vrátí funkce **search()** n jednorozměrných polí, přičemž n-tice prvků z každého pole se stejným indexem odpovídá vícerozměrnému indexu prvku, který vyhovuje logickému výrazu. Opět je třeba dbát na to, že Python indexuje od nuly. Syntaxe pro vyhledávání v NumPy polích pomocí metody **where()** vypadá takto:

```
>>> a = np.array([2,7,1,5,8,7,3])
>>> A = np.array([[7, 4, 5], [5,7,1]])
>>> np.where(a > 4)
(array([1, 3, 4, 5], dtype=int64),)
>>> np.where(A == 7)
(array([0, 1], dtype=int64), array([0, 1], dtype=int64))
```

S indexy, které vrátí funkce **search()** můžeme dále pracovat a pole v němž jsme vyhledávali podle něj filtrovat. K filtrování polí podle nějakého logického výrazu ale existuje v NumPy i efektivnější cesta. Můžeme snadno vyhodnotit nějaký logický výraz nad polem a podle výsledku indexovat původní pole. Tím vytvoříme nové pole pouze z prvků původního pole, které vyhovují námi specifikovanému logickému výrazu. Můžeme také přímo indexovat pole logickým výrazem. Pole z předchozího příkladu můžeme filtrovat například takto:

```
>>> filtered_ind = a > 3
>>> aa = a[filtered_ind]
>>> aa
array([7, 5, 8, 7])
>>> a[a > 3]
array([7, 5, 8, 7])
```

Další užitečnou funkcí je funkce **searchsorted()**, která bere na vstupu seřazené pole a prvky, které chceme do tohoto pole vložit. Funkce vrátí indexy prvků, za které můžeme prvky vložit tak, aby pole zůstalo seřazené. V následujícím příkladu nám funkce **searchsorted()** vrátí pole [1, 4, 3], což znamená, že když vložíme prvek 2 na druhé místo, prvek 5 na páté místo a prvek 4 na čtvrté místo pole [1, 2, 3, 4], toto pole zůstane seřazené:

```
>>> b = np.array([1, 2, 3, 4])
```

```
>>> np.searchsorted(b, [2, 5, 4])
array([1, 4, 3], dtype=int64)
```

Abychom mohli vyhledávat v seřazeném poli, což má své výhody z hlediska efektivity vyhledávacího algoritmu, je třeba pole nejdříve seřadit. Knihovna NumPy nabízí pro seřazování polí užitečnou funkci **sort()**. Předáme-li této funkci na vstup jednorozměrné pole, seřadí ho, předáme-li jí na vstup vícerozměrné pole, seřadí všechna jednorozměrná pole, která toto pole obsahuje. Syntaxe funkce **sort()** vypadá následovně:

```
>>> a = np.array([2,7,1,5,8,7,3])
>>> A = np.array([[7, 4, 5], [5,7,1]])
>>> np.sort(a)
array([1, 2, 3, 5, 7, 7, 8])
>>> np.sort(A)
array([[4, 5, 7],
       [1, 5, 7]])
```

2.8 Doplňující příklady ke kapitole

2.8.1 Příklady k procvičení

1. příklad:
   Vygenerujte pomocí funkcí z knihovny NumPy pole X s deseti prvky vygenerovanými z normálního rozdělení se střední hodnotou 1 a rozptylem 5. Následně vygenerujte pole Y s deseti prvky rovnými číslu 11, pole Z s deseti prvky od 1 do 10, pole U, které má deset řádků a deset sloupců, přičemž všechny prvky mimo hlavní diagonálu budou nulové a prvky na diagonále budou celá čísla generovaná z intervalu od 2 do 14.

2. příklad:
   S využitím předprogramovaných univerzálních funkcí z knihovny NumPy spočtěte hodnotu výrazu $log_2(Z)+Y/Z+UX-U^2$ pro pole vytvořená v předchozím příkladu. Výsledek uložte do matice A. Jaké rozměry bude mít matice A vzhledem k pravidlům Broadcastingu?

3. příklad:
   Naprogramujte novou univerzální funkci *My_new_great_ufunc()*, která vyhodnotí na sobě odpovídajících prvcích dvou polí a,b funkci $4*a+b^2-4b$. Následně vyhodnoťte tuto funkci na prvcích polí X, Y a výsledek vypište na standardní výstup.

4. příklad:
   Spočtěte determinant, hodnost, inverzi a vlastní čísla i vektory matice A vytvořené v druhém příkladu. Následně maticově vynásobte inverzi matice A a matici U a výsledek uložte do proměnné B.

5. příklad:
   Vypište z matice B všechny záporné prvky. Následně hodnotu všech těchto prvků v matici B změňte na hodnotu 2. Následně seřaďte prvky všech řádků matice B od nejmenšího po největší.

2.8.2 Řešení

1. příklad:

```
import numpy as np

X = np.random.normal(1, 5, 10)
Y = np.full(3, 11)
Z = np.arange(1,11)
U = np.diag(np.random.randint(2, 15, 3))

```

2. příklad:

```
A = np.log2(Z) + np.divide(Y, Z) + U * X  - np.power(U, 2)
print(A.shape)

```

Jak se můžeme přesvědčit z výsledku, který vypíše kód výše, výsledné pole bude mít rozměry 10x10.

3. příklad:

```
import numpy as np

def my_new_great_func(a,b):
return 4*a + b ** 2 - 4 * b

My_new_great_ufunc = np.frompyfunc(my_new_great_func, nin = 2,
    nout = 1)

print(My_new_great_ufunc(X, Y))

```

4. příklad:

```
determinant_A = np.linalg.det(A)

inverze_A = np.linalg.inv(A)

hodnost_A = np.linalg.matrix_rank(A)

vlastni_cisla, vlastni_vektory = np.linalg.eig(A)

B = np.matmul(inverze_A, U)

```

5. příklad:

```
print(B[B < 0])

B[B < 0] = 2

print(np.sort(B))

```

Kapitola 3

Knihovna SciPy

3.1 Úvod do SciPy

SciPy je další z užitečných Pythonovských knihoven pro potřeby matematických a statistických výpočtů v Pythonu. Jedná se o především o nástavbu nad knihovnou NumPy a často jsou obě knihovny využívány současně. Je ovšem možné používat knihovnu SciPy i samostatně.

Knihovna SciPy umožňuje využívat celou řadu vyšších matematických nástrojů, které nejsou obsaženy v základním Pythonu. Umožňuje například derivovat, integrovat, provádět optimalizační výpočty nebo interpolovat a extrapolovat z dat. Obsahuje také většinu základních funkcí pro statistické testování hypotéz. Dále umožňuje provádět efektivně operace s řídkými maticemi nebo různé výpočty teorie grafů. Jsou v ní také obsaženy všechny základní matematické konstanty a fyzikální jednotky i základní metody umožňující s těmito jednotkami efektivně pracovat. Oproti knihovně NumPy, která umožňuje efektivně řešit pouze lineární rovnice a jejich soustavy za použití nástrojů lineární algebry, které má v sobě zapracované, umožňuje knihovna SciPy řešit numericky i nelineární rovnice a jejich soustavy. NumPy a SciPy společně tedy umožňují provádět za použití počítače prakticky všechny myslitelné numerické výpočty lineární algebry, teorie grafů i matematické analýzy a statistiky.

SciPy obsahuje některé balíčky, které plní obdobné funkce jako již dříve popsané balíčky z knihovny NumPy. Například balíček *linalg* existuje v knihovně SciPy i v knihovně NumPy a má velmi podobné funkce i vlastnosti. V určitých případech může být efektivnější použít funkci implementovanou ve SciPy a v jiných funkci implementovanou v NumPy. Těmto balíčkům se ale nebudeme věnovat, neb je to pro potřeby této práce nadbytečné. Raději se zaměříme na specifické funkce knihovny SciPy, pro které se vyplatí používat právě tuto knihovnu. Tyto dodatečné informace ale i konkrétní a podrobnější informace o vlastnostech a parametrech zde popsaných funkcí je možno nalézt v dokumentaci knihovny SciPy. [10]

3.2 Konstanty a převody jednotek

S využitím modulu *constants* z knihovny SciPy můžeme v Pythonu volat hodnoty mnoha důležitých fyzikálních a matematických konstant. Syntaxe je velmi jednoduchá. Konkrétní konstantu voláme pomocí syntaxe *constants.název_konstanty* . Následující příklad demonstruje, jak můžeme v Pythonu volat konstanty základních předpon užívaných pro fyzikální jednotky (např. kilo pro 1000 základních jednotek): [11]

```
1
2 from scipy import constants
3
4 print(constants.yotta)    #1e+24
5 print(constants.zetta)    #1e+21
6 print(constants.exa)      #1e+18
7 print(constants.peta)     #1000000000000000.0
8 print(constants.tera)     #1000000000000.0
9 print(constants.giga)     #1000000000.0
10 print(constants.mega)    #1000000.0
11 print(constants.kilo)    #1000.0
12 print(constants.hecto)   #100.0
13 print(constants.deka)    #10.0
14 print(constants.deci)    #0.1
15 print(constants.centi)   #0.01
16 print(constants.milli)   #0.001
17 print(constants.micro)   #1e-06
18 print(constants.nano)    #1e-09
19 print(constants.pico)    #1e-12
20 print(constants.femto)   #1e-15
21 print(constants.atto)    #1e-18
22 print(constants.zepto)   #1e-21
```

Další možností modulu *constants* je volání převodních konstant mezi metrickými a imperiálními jednotkami, jak můžeme vidět zde:

```
1
2 from scipy import constants
3
4 print(constants.inch)              #0.0254
5 print(constants.foot)              #0.30479999999999996
6 print(constants.yard)              #0.9143999999999999
7 print(constants.mile)              #1609.3439999999998
8 print(constants.mil)               #2.5399999999999997e-05
9 print(constants.pt)                #0.00035277777777777776
10 print(constants.point)            #0.00035277777777777776
11 print(constants.survey_foot)      #0.3048006096012192
12 print(constants.survey_mile)      #1609.3472186944373
13 print(constants.nautical_mile)    #1852.0
```

Můžeme také volat hodnoty některých méně známých fyzikálních jednotek vyjádřené v základních jednotkách dané veličiny:

```
1
2 print(constants.fermi)             #1e-15
3 print(constants.angstrom)          #1e-10
```

```
print(constants.micron)             #1e-06
print(constants.au)                 #149597870691.0
print(constants.astronomical_unit) #149597870691.0
print(constants.light_year)         #9460730472580800.0
print(constants.parsec)             #3.0856775813057292e+16
```

Nabídka konstant v knihovně SciPy je skutečně široká. V dokumentaci je možné najít metody pro volání mnoha dalších konstant i fyzikálních jednotek.

3.3 Numerické řešení rovnic a optimalizace

S pomocí knihovny NumPy je možné v Pythonu řešit lineární rovnice a jejich soustavy, pro numerické řešení nelineárních rovnic složí modul *optimize* v knihovně SciPy. Konkrétně používáme především funkci **root()** pro nalezení kořenů vektorové nelinerární funkce (soustavy nelineárních funkcí). Na vstup bere buď běžnou pythonovskou funkci, kterou metodě předáme jejím názvem bez závorky a parametrů nebo *lambda funkci*. Danou funkci položí rovnou nule (resp. nulovému vektoru) a následně hledá kořeny vzniklé rovnice. Druhý parametr je *x0*, tedy počáteční odhad kořene, který Python použije jako počáteční iteraci numerické metody. Nechceme-li zvolit defaultní numerickou metodu, je možné specifikovat, kterou numerickou metodu chceme použít v parametru *method*. Následující příklad demonstruje řešení skalární rovnice pomocí funkce **root()** a to nejdříve s použitím běžné Pythonovské funkce:

```

import numpy as np
from scipy.optimize import root

def eqn(x):
    return x ** 2 - x - 1

solution = root(eqn, 1)

print(solution)
```

Python vypíše následující informace:

```
fjac: array([[-1.]])
     fun: array([-2.22044605e-16])
 message: 'The solution converged.'
    nfev: 10
     qtf: array([3.78164167e-12])
       r: array([-2.23606804])
  status: 1
 success: True
       x: array([1.61803399])
```

Povšimněme si, že funkce **root()** vrátí výrazně více informací než jen řešení rovnice. Chceme-li zjistit pouze řešení zavoláme pouze parametr *x* z výsledného řešení:

```
print(solution.x)
```

Celý předchozí kód můžeme zapsat výrazně úsporněji s využitím *lambda funkce*:

```
print(root(lambda x: x ** 2 - x - 1, 0, method = 'broyden1').x)
```

Čtenář si mohl povšimnout, že v obou metodách výpočtu jsme zvolili jinou počáteční hodnotu a v důsledku toho získali jiné řešení rovnice. To je vlastnost většiny numerických metod a nemá nic společného s implementací těchto metod v Pythonu. Ve druhém případě jsme použili Broydenovu metodu, neboli vícerozměrnou quasi-newtonovu metodu, zatímco v prvním jsme použili defaultní metodu, což je Powellova metoda. Nakonec si povšimněme, že jsme zde vlastně vypočetli hodnotu zlatého řezu.

Soustavy rovnic můžeme řešit opět pomocí funkce **root**, které ale tentokrát předáme vektorovou funkci a počáteční podmínku jako vektor. V tomto případě používáme NumPy pole, jež byla podrobně popsána v předchozí kapitole. Numerické řešení soustavy nelineárních rovnic a navíc i zkoušku správnosti výpočtu demonstruje následující příklad:

```
def system(x):
    return np.array([x[0]**2 - 4*x[1] - 4,
                     x[1] + x[0]**3,
                     x[2] - x[0]* x[1]])

sol = root(system, np.array([0,0,0]))

#reseni
print(sol.x)

#zkouska
print(system(sol.x))
```

Nalezeným řešením je vektor [0.92322773 -0.78691264 -0.72649957]. Po jeho dosazení do původní soustavy rovnic sice nezískáme nulový vektor, ale získáme vektor [0.00000000e+00 1.11022302e-16 -1.44328993e-15], jehož prvky jsou velmi blízké nule. Je to důsledek numerických chyb, kterým se při numerickém řešení nelineárních rovnic na počítači bohužel nemůžeme vyhnout. Při algoritmickém vyhodnocování výsledků je dobré stanovit předem přesnost s jakou chceme počítat a výsledky porovnávat vzhledem k této předem zvolené přesnosti.

Součástí modulu *optimize* je také funkce **minimize()**, která umožňuje numericky hledat minimum zadané funkce. Funkci předáváme název funkce, kterou chceme minimalizovat, odhad souřadnic minima funkce (počáteční iteraci) a případně také metodu numerické minimalizace, kterou chceme použít. Funkce předaná funkci **minimize()** může mít jednu nebo i více proměnných tak jako u metody **root()**. Nedefinujeme-li metodu, kterou chceme použít, bude použita jedna z defaultních metod (v závislosti na zadání), což obvykle funguje dobře. Chceme-li funkci maximalizovat, použijeme opět funkci **minimize()**, pouze před název funkce napíšeme znaménko mínus. Minimalizace záporně vzaté funkce je totiž vlastně její maximalizací. Funkci **minimize** můžeme použít následujícím způsobem:

```

from scipy.optimize import minimize
import numpy as np

#funkce, kterou budeme minimalizovat
def func(x):
```

```
    return x[0] ** 2 + x[1] * x[0]  + x[1] ** 2

#vypocet minima
min_func = minimize(func, np.array([0,0]))
min_func2 = minimize(func, np.array([1,1]))
```

Stejně jako u funkce **root()** i zde vrací funkce více informací než jen souřadnice výsledku. Pokud chceme znát pouze výsledek použijeme argument x tak jako u funkce **root()**. Funkci jsme minimalizovali dvakrát s různými počátečními iteracemi a z výsledku je patrné, že na volbě počáteční iterace záleží. Funkce $f(x) = x^2 + xy + y^2$ má minimum v bodě [0,0], který jsme v prvním případě zvolili jako počáteční iteraci. Ve druhém případě bylo nalezeno minimum v bodě [-7.27305111e-09 -4.66495398e-09], tedy došlo k očekávatelné numerické chybě. Tento bod je ovšem hledanému minimu dostatečně blízko. Náš kód je také možno napsat kompaktněji za využití lambda funkce, jak je možno vidět zde:

```
from scipy.optimize import minimize
import numpy as np

min_func = minimize(lambda x: x[0] ** 2 + x[1] * x[0]  + x[1] ** 2,
    np.array([1,1]))
```

Budeme-li chtít stejnou funkci maximalizovat, je třeba použít stejnou funkci **minimize()**, pouze funkci vynásobíme číslem (-1), což zajistí, že minimum této nové funkce bude ve stejném bodě jako maximum původní funkce. Můžeme tuto operaci provést např. přes lambda funkci takto:

```
from scipy.optimize import minimize
import numpy as np

max_func = minimize(lambda x: -func(x), np.array([1,1]))
```

V tomto případě funkce vrátí souřadnice [2.5561421e+08 2.5561421e+08]. To zjevně není naše hledané maximum. Maximum naší funkce se nachází v nekonečnu. To ovšem numericky není možné dopočítat. Proto je zde chyba výpočtu nekonečně velká. V tomto případě by bylo pravděpodobně vhodnější použít symbolický výpočet pomocí knihovny SymPy.

3.4 Numerická integrace

Pro numerickou integraci v Pythonu můžeme použít modul *integrate* z knihovny SciPy. Základní funkcí pro jednoduchý integrál je funkce **quad()**. Ta bere na vstup *lambda* funkci nebo název dříve definované funkce a dva parametry určující dolní a horní mez intervalu, přes který má funkce integrovat. Například integrál funkce $f(x) = x^3 + x^2 + 4$ na intervalu [2, 4] vypočteme pomocí funkce **quad()** následujícím způsobem:

```

from scipy.integrate import quad
import numpy as np

print(quad(lambda x: x ** 3 + x ** 2 + 4, 2, 4))
```

Jak se můžete přesvědčit, funkce vypíše dvojici čísel (86.66666666666666, 9.62193288008469e-13). První číslo je odhadem hledaného integrálu a druhé číslo je horní závorou pro chybu výpočtu. V tomto případě tedy můžeme říci, že je metoda poměrně přesná.

Jednoduchý integrál ale často není dostatečný pro naplnění všech našich potřeb. Naštěstí modul *integrate* z knihovny SciPy nám umožňuje počítat i dvojné integrály pomocí funkce **dblquad()**, trojné integrály pomocí funkce **tplquad()** a další vícenásobné určité integrály pomocí funkce **nquad()**. Těmto funkcím předáváme jako prví argument funkci, kterou chceme integrovat tak jako u jednoduché funkce **quad()**. Vícenásobné integrály ovšem mohou integrovat i přes oblast omezenou funkcí. Python se s tímto problémem vyrovnává velmi intuitivně. Třetí a čtvrtý parametr u **dblquad()**, resp. i pátý a šestý parametr u **tplquad()**, které udávají omezení vnitřních integrálů, mohou být názvy funkcí, které jim předáváme stejně jako jednoduché funkci **quad()**. Tedy do parametru uvedeme buď název funkce bez závorky a parametrů nebo lambda funkci. U funkce **nquad()** předáváme meze každého integrálu jako seznam o dvou prvcích, které jsou mezemi integrace. Všechny tyto dvouprvkové seznamy musí být předány najednou v jedné proměnné, která je seznamem těchto seznamů (**nquad()** má tedy jedinou proměnnou určující meze všech integrálů, která je vlastně polem polí) Nesmíme zapomínat na obecně platná pravidla integrace. Meze alespoň jednoho z integrálů musí být čísla a počet proměnných funkcí, které ohraničují jednotlivé integrály se musí snižovat a být u úplně prvního integrálu alespoň o jedna menší než celkový počet integrálů. V případě, že má funkce další parametry, podle kterých nechceme integrovat, můžeme jejich hodnotu předat integračním estimátorům jako n-tici v parametru *args*. Využití vícenásobných integrálů demonstrujeme na následujícím příkladu:

```
from scipy.integrate import dblquad, tplquad, nquad
import numpy as np

#funkce dvou promenych
def multifunc(x, y, a, b):
    return x ** 2 + y ** b + a

#dvojny integral funkce dvou promenych s parametry
a, b = 5, 2
int2 = dblquad(multifunc, 0, 1, 0, lambda x: 1 - x, args = (a, b))

#trojny integral funkce tri promenych
int3 = tplquad(lambda x, y, z : x ** 2 + y + z, 0, 1, 0, 1, 0, 3)

#funkce ctyr promenych
def multifunc2(k, l, m , n):
    return k ** 2 + l + m + n ** 3

#vypocet vicenasobneho integralu
intn = nquad(multifunc2, [[0, 16],[7, 14],[2, 6], [7, 12]])
```

Opět je třeba pamatovat na numerické chyby, které při výpočtu vznikají. Jak se ovšem můžeme přesvědčit na výsledcích předchozího příkladu, v nichž je obsažen i horní odhad chyby, nejsou numerické chyby všech funkcí založených na **quad()** velké. Největší horní odhad chyby v předchozím příkladu nalézáme u **nquad()** a i tam se případná chyba objeví

až v osmém desetinném místě.

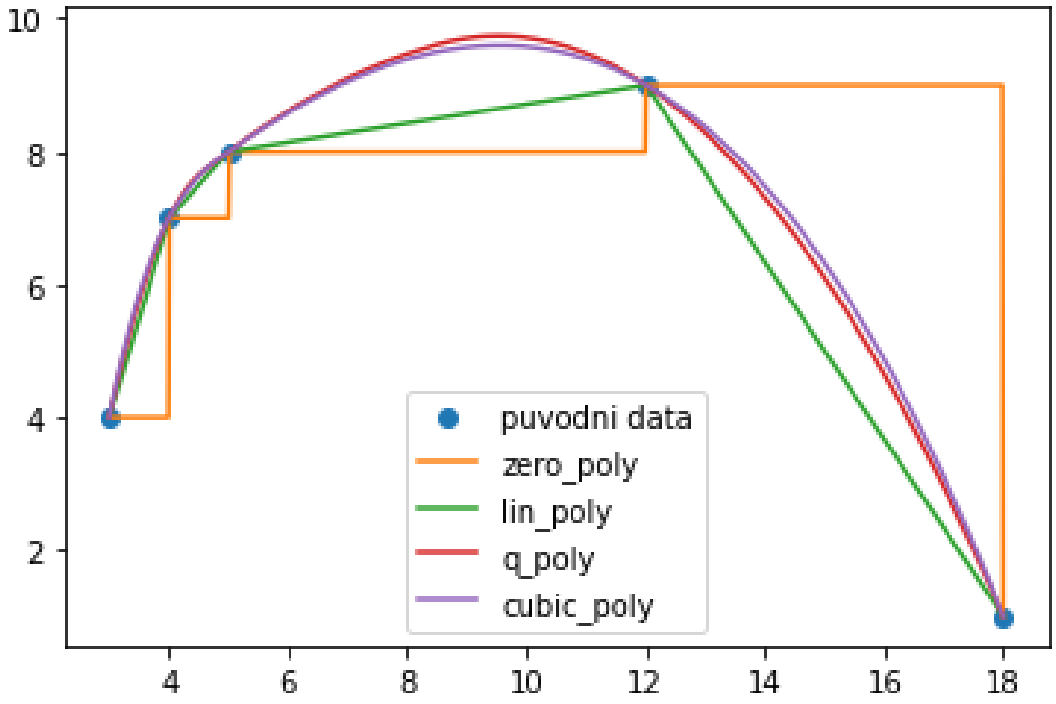
3.5 Interpolace a extrapolace

Máme-li nasbíraná data často z nich chceme interpolovat funkci, která těmto datům v nějakém smyslu nejpřesněji odpovídá a umožní nám predikovat hodnoty funkce při jiných hodnotách vstupních parametrů. Základní interpolační nástroje v Pythonu jsou obsaženy v modulu *interpolate* z knihovny SciPy. Základní funkcí z této knihovny je funkce **interp1d()**. Tato funkce využívá tzv. splajnovou interpolaci. Funkci můžeme interpolovat nulovými, lineárními, kvadratickými a nebo kubickými polynomy, což určíme v parametru *kind*, který může nabývat hodnot ”zero”, ”linear”, ”quadratic”nebo ”cubic”, ale i dalších, které lze dohledat v dokumentaci. Dále funkci **interp1d()** předáváme jako první argument *x* a druhý argument *y* souřadnice bodů, kterými budeme interpolovat. Funkce **interp1d()** následně vrátí funkci, která pro zadanou hodnotu zevnitř intervalu, který ohraničují zadané hodnoty v parametru *x*, vrátí hodnotu interpolační funkce. Pokud se pokusíme vyhodnotit vrácenou funkci na hodnotě nenáležící do tohoto intervalu, Python vrátí chybovou hlášku. Chceme-li mít možnost tuto funkci vyhodnotit i na jiných hodnotách, musíme nastavit parametr *fill*_value="extrapolate". Následující kód demonstruje použití funkce **interp1d()** s výše zmíněnými čtyřmi hodnotami parametru *kind*:

```
1
2  import numpy as np
3  from scipy.interpolate import interp1d
4  import matplotlib.pyplot as plt
5
6  #vytvoreni dat, ktera budeme interpolovat
7  x = np.array([3, 4, 5, 12, 18])
8  y = np.array([4, 7, 8, 9, 1])
9
10 #takto interpolujeme konstantnimi splajny
11 zero_poly = interp1d(x, y, kind = "zero")
12
13 #takto interpolujeme linearnimi splajny
14 lin_poly = interp1d(x, y, kind = "linear")
15
16 #takto interpolujeme kvadratickymi splajny
17 q_poly = interp1d(x, y, kind = "quadratic")
18
19 #takto interpolujeme kubickymi splajny
20 cubic_poly = interp1d(x, y, kind = "cubic")
21
22 #interpolace hodnoty funkce v dalsich bodech
23 xx = np.linspace(3, 18, 1001)
24 y2 = zero_poly(xx)
25 y3 = lin_poly(xx)
26 y4 = q_poly(xx)
27 y5 = cubic_poly(xx)
28
29 #zobrazeni vysledku interpolace
30 plt.plot(x, y, 'o', label = "puvodni data")
31 plt.plot(xx, y2, label = "zero_poly")
```

```
plt.plot(xx, y3, label = "lin_poly")
plt.plot(xx, y4, label = "q_poly")
plt.plot(xx, y5, label = "cubic_poly")
plt.legend()
plt.show()
```

V tomto příkladu nejdříve vytvoříme hodnoty, kterými chceme proložit interpolační funkci. Následně interpolujeme tyto body všemi čtyřmi možnými metodami a vrácené funkce vyhodnotíme na dalších hodnotách z intervalu, na kterém interpolujeme. Poslední části kódu uvedené pod komentářem "zobrazení výsledku interpolace"nemusí vážený čtenář zatím věnovat pozornost. Jedná se o kód, který vypíše graf, který ilustruje rozdíl mezi jednotlivými interpolačními funkcemi. Tento graf si může laskavý čtenář prohlédnout zde:



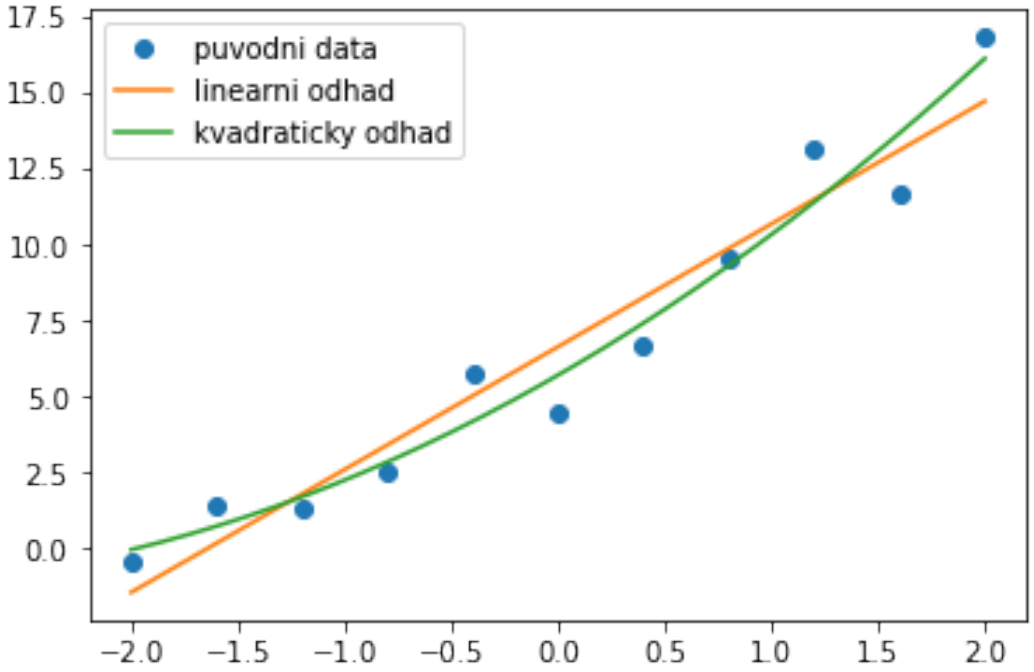
Knihovna SciPy samozřejmě nabízí více interpolačních funkcí. Například pro interpolaci funkcí dvou proměnných můžeme použít funkci **interp2d()**, jejíž užití je velmi podobné použití funkce **interp1d()**. Pro běžné použití je ovšem poměrně dostačující funkce **interp1d()** a případně je možné další vhodné funkce dohledat v dokumentaci knihovny SciPy.

Interpolace sama umožňuje doplňovat chybějící hodnoty pouze uvnitř intervalu interpolace. Interpoluje-li pomocí polynomiálních splajnů, jak je obvyklé, začínají být interpolační funkce mimo interval, na němž interpolujeme, velmi rychle velmi nepřesné (nejsou-li data, která se snažíme interpolovat produktem daného polynomu). Pro extrapolaci hodnot, která umožní odhadnout průběh funkce i mimo námi zvolený interval nabízí SciPy vhodnější nástroje. Nejvhodnější je použít metodu nejmenších čtverců. K tomu slouží funkce **curve_fit()** z modulu *optimize*, který jsme si představili již v dřívějších kapitolách. Funguje velmi intuitivně. Nejprve si definujeme funkci o několika parametrech. První z těchto parametrů musí být naše vysvětlující proměnná a další parametry funkce **curve_fit()** od-

hadne a vrátí pole polí, v němž první pole obsahuje optimální hodnoty našich parametrů v takovém pořadí, v jakém jsme je uvedli v definici naší funkce a druhé pole obsahuje odhad jejich rozptylu ve stejném pořadí. Použije k tomu nelineární metodu nejmenších čtverců (nebo lineární metodu nejmenších čtverců pokud je naše funkce lineární funkcí). Aby mohla provézt odhad parametrů, musíme funkci **curve_fit()** předat také data, jimiž má proložit funkci. K tomu slouží parametry *xdata*, pro hodnoty nezávisle proměnné a *ydata* pro naměřené hodnoty závisle proměnné. Jednoduchý příklad využití funkce **curve_fit()** demonstruje následující příklad:

```
1
2  import numpy as np
3  from scipy.optimize import curve_fit
4  import matplotlib.pyplot as plt
5
6  #generovani dat s normalni chybou
7  x = np.linspace(-2, 2, 11)
8  y = x ** 2 + 4 * x + 4 + np.random.normal(size = len(x), scale = 2)
9
10
11 #definice funkci, ktere budeme odhadovat
12 def func1(x, a, b):
13     return a * x + b
14
15 def func2(x, a, b, c):
16     return a * (x ** 2) + b * x + c
17
18 #odhad parametru funkci metodou nejmensich ctvercu
19 beta1, cov1 = curve_fit(func1, xdata = x, ydata = y)
20 beta2, cov2 = curve_fit(func2, xdata = x, ydata = y)
21
22 print(beta1, "\n", beta2)
23
24 #vykresleni odhadnutych funkci do grafu
25 xx = np.linspace(-2, 2, 1001)
26
27 plt.plot(x, y, 'o', label = 'puvodni data')
28 plt.plot(xx, func1(xx, beta1[0], beta1[1]),
29          label = 'linearni odhad')
30 plt.plot(xx, func2(xx, beta2[0], beta2[1], beta2[2]),
31          label = 'kvadraticky odhad')
32 plt.legend()
33 plt.show()
```

Data v příkladu jsou generována pomocí kvadratické funkce, k níž jsme přidali šum generovaný z normálního rozdělení. Na taková data lze dobře aplikovat metodu nejmenších čtverců. Poslední část kódu pod komentářem "vykresleni odhadnutych funkci do grafu"může čtenář opět ignorovat neb využívá objektu knihovny *Matplotlib*, která bude popsána později. Tato část kódu vykresluje následující obrázek, který graficky demonstruje námi odhadnuté funkce a data z nichž jsme je odhadli:



Metoda nejmenších čtverců je jedním z hlavních nástrojů regresní analýzy. Funkce **curve_fit()** slouží především k využití metody nejmenších čtverců v numerické matematice. Pro regresní analýzu nabízí Python vhodnější funkce. O některých z nich bude pojednáno později v této práci.

3.6 Statistické výpočty pomocí knihovny SciPy

SciPy nabízí v modulu *stats* nejrůznější užitečné funkce pro provádění statistických výpočtů. Tak jako dříve představené knihovny *random* nebo *NumPy* například umožňuje generovat prvky z různých pravděpodobnostních rozdělení. Metody pro generování statistických rozdělení jsou ovšem v knihovně SciPy komplexnější a jejich nabídka je významně rozsáhlejší než v jiných knihovnách. Není ambicí této práce zde popsat všechna rozdělení pravděpodobnosti, která knihovna SciPy nabízí. Vážený čtenář si může příslušné metody snadno dohledat v dokumentaci knihovny Scipy. [10] Vlastnosti objektů reprezentujících pravděpodobnostní rozdělení zde demonstrujeme na funkci **norm()** z modulu *stats*, která reprezentuje asi nejvýznamnější pravděpodobnostní rozdělení - normální rozdělení pravděpodobnosti.

Funkce **norm()** vytváří objekt třídy *norm*, který dědí od třídy *rv_continuous*, tedy od třídy všech spojitých rozdělení, která definuje jejich společné vlastnosti. Obdobně existují uvnitř *Scipy.stats* třídy diskrétních rozdělení a vícerozměrných statistických rozdělení. Při vytváření objektu třídy *norm* je třeba definovat jeho parametry. Základními parametry jsou parametr *loc*, který definuje střední hodnotu normálního rozdělení a parametr *scale*, který definuje jeho směrodatnou odchylku. Nezvolíme-li tyto parametry, Python implicitně zvolí hodnoty $loc = 0$ *a* $scale = 1$, tedy střední hodnotu a směrodatnou odchylku standardizova-

ného normálního rozdělení. Po vytvoření objektu třídy *norm* na něj můžeme volat metody vlastní jemu nebo jeho "parent class". Hlavními metodami, které používáme u spojitých rozdělení třídy *rv_continuous* jsou metody **rvs()** pro generování pole náhodných výběrů z tohoto rozdělení (velikost pole definujeme v parametru *size*), metoda **pdf()**, která pro zadané hodnoty náhodné proměnné s normálním rozdělením vrátí jim odpovídající hodnoty distribuční funkce tohoto rozdělení a metoda **ppf**, která pro danou hodnotu pravděpodobnosti vrátí odpovídající kvantil daného rozdělení. Takto vytvoříme objekt pro standardizované normální rozdělení a pro normální rozdělení se střední hodnotou 10 a rozptylem 2:

```

from scipy import stats
import numpy as np

normal = stats.norm()
normal2 = stats.norm(loc = 10, scale = 2)
```

A takto můžeme z těchto rozdělení generovat výběry (v tomto případě délky 10 a 8), kvantily (zde `95% a 97,5% kvantil`) a hodnoty jejich hustoty pravděpodobnosti v konkrétních bodech:

```
>>> normal.rvs(size = 10)
array([ 1.01332455, -0.26653891, -0.45484709,  0.43936478,
    -0.23801113,
        -0.08325556, -1.18969746,  1.50091472, -0.03775948,
    -1.6244031 ])
>>> normal2.rvs(size = 8)
array([ 8.96073151, 14.25264928, 10.32919889,  6.8042171 ,
    10.9758553 ,
         7.69599036, 13.63851871, 12.35169637])
>>> normal.pdf(1)
0.24197072451914337
>>> normal2.pdf([12, 9, 23])
array([1.20985362e-01, 1.76032663e-01, 1.33477831e-10])
>>> normal.ppf(0.95)
1.6448536269514722
>>> normal.ppf(0.975)
1.959963984540054
```

Pokud již máme nějaká data a chceme je analyzovat, prvním krokem obvykle bývá zpracování popisných statistik. K tomu lze použít funkci **describe()** z modulu *stats* následujícím způsobem (ke generování dat jsme použili objekt standardizovaného normálního rozdělení z předchozího příkladu):

```
x = np.array(normal.rvs(size = 1000))

x_popisne = stats.describe(x)

print(x_popisne)
```

Kód vypíše nejvýznamnější popisné statistiky následujícím způsobem:

```
DescribeResult(nobs=1000,
minmax=(-3.4757011942363922, 2.9722194006927145),
mean=-0.012877545512116787, variance=1.0355460361210402, skewness
    =-0.039967891552400416, kurtosis=0.009335163705869842)
```

Vždy ovšem nepotřebujeme vypsat všechny popisné statistiky, které funkce **describe()** spočítá, můžeme tedy vypsat jen některé z nich např. takto:

```
print("minimum: ", x_popisne.minmax[0], "\n",
      "prumer: ", x_popisne.mean, "\n",
      "rozptyl: ", x_popisne.variance)
```

Je dobré si povšimnout, že popisné statistiky můžeme vypsat buď všechny najednou nebo pouze některé z nich pomocí vhodné funk. Nechceme-li ztrácet čas výpočtem všech popisných statistik, budeme-li používat pouze některé z nich, je možné použít k tomu vhodnou funkci. Funkce **min()** a **max()** pro minimum a maximum z daného souboru dat jsou známi už základnímu Pythonu. Funkce **mean()** pro výpočet průměru, **var()** pro výpočet výběrového rozptylu, a **std()** pro výpočet výběrové směrodatné odchylky můžeme najít v knihovně NumPy. Méně často používané statistické ukazatele jako koeficient špičatosti pak můžeme najít v modulu *stats* knihovny SciPy:

```

#maximum a minimum
print("maximum: ", max(x), ", mimimum: ", min(x))

#vypocet prumeru, rozptylu a smerodatne odchylky
print(np.mean(x), np.var(x), np.std(x))

#vypis koeficientu spicatosti
print(stats.kurtosis(x))
```

Máme-li více než jeden soubor dat, můžeme také chtít zjistit, jestli mezi nimi existuje nějaký vztah. Pro zjištění míry lineární závislosti mezi dvěma náhodnými veličinami, můžeme použít Pearsonův korelační koeficient pomocí funkce **pearsonr()**. Dále nám knihovna SciPy umožňuje odhadnout parametry jednoduchého lineárního regresního modelu s jednou vysvětlující proměnnou. K tomu slouží funkce **linregress()**. Metoda bere na vstup dva datové vektory stejné délky a vrátí odhadnutý objekt třídy *LinregressResult* představující regresní přímku. Tento objekt v sobě obsahuje informace o sklonu regresní přímky v parametru *slope*, odhadnutém absolutním členu regresní přímky v parametru *intercept*, p-value pro nulovou hypotézu, že je sklon regresní přímky roven nule v parametru *pvalue*, pearsonův korelační koeficient v parametru *rvalue* a směrodatné odchylky obou parametrů regresní přímky v parametrech objektu *stderr a intercept_stderr*. Můžeme si opět nechat vypsat všechny informace o tomto modelu, nebo pouze některé. Odhadnuté parametry modelu můžeme také použít k predikci pro další možné hodnoty nezávisle proměnné:

```
from scipy import stats
import numpy as np

#generovani dat jejichz korealaci budeme zkoumat
a = np.linspace(-1, 1, 1000)
b = 2 * a + 5 + stats.norm().rvs(1000)

#korelacni koeficient dvou vyberu
print(stats.pearsonr(a, b))

#odhad linearniho regresniho modelu
model = stats.linregress(a, b)
```

```
13
14 #vypis vsech parametru modelu
15 print(model)
16
17 #vypis nekterych parametru modelu
18 print("sklon regresni primky: ", model.slope, "\n",
19       "absolutni clen regresni primky: ", model.intercept, "\n",
20       "p-value: ", model.pvalue)
21
22 #odhad hodnot funkce pro nove hodnoty nezavisle promene
23 xx = np.array([-2, 1.5, 4])
24 yy = map(lambda x: model.slope * x + model.intercept, xx)
25 print("odhadnute hodnoty: ", list(yy))
```

V Pythonu je samozřejmě možné odhadovat parametry i složitějších regresních modelů. Tím se ovšem zatím nebudeme zabývat. Vhodná je k tomu například knihovna Scikit-learn nebo knihovna Statmodels.

Poslední užitečnou skupinou funkcí modulu *stats* knihovny SciPy, kterou si zde představíme, jsou metody pro provádění statistických testů. Knihovna Scipy nabízí mnoho nejrůznějších funkcí pro provádění nejrůznějších statistických testů. Popsat všechny možné testy, které knihovna SciPy nabízí, by bylo opět nad rámec rozsahu této práce. Vážený čtenář si ovšem může metody pro provádění nejrůznějších testů snadno dohledat v dokumentaci knihovny SciPy. [10] Základní syntaxi pro provádění statistických testů předvedeme na metodách pro provádění t-testu. Chceme-li provést párový t-test testující nulovou hypotézu, že se střední hodnoty dvou náhodných proměnných rovnají, použijeme k tomu funkci **ttest**_ind_from_stats(). Tato funkce bere na vstup počet pozorování obou výběrů (parametry *nobs1, nobs2*), výběrové směrodatné odchylky obou pozorování (parametry *std1, std2*) a jejich výběrové průměry (parametry *mean1, mean2*). Metoda vrací testovou statistiku v parametru *statistic* a p-hodnotu pro nulovou hypotézu, že se střední hodnoty rovnají, v parametru *pvalue*. Máme-li k dispozici data na nichž t-test provádíme, můžeme si práci ulehčit a nemusíme počítat hodnoty výše zmíněných parmetrů. Stačí použít funkci **ttest**_ind() a předat jí na vstup oba vektory dat. Výstup bude stejný jako u funkce **ttest**_ind_from_stats(). Chceme-li provádět jednovýběrový t-test o nulové hypotéze předpokládající, že se střední hodnota rozdělení, z něhož generujeme náhodný výběr, rovná nějaké konkrétní hodnotě, použijeme funkci **ttest**_1samp(), které předáme na vstup naše data v parametru *a* a hodnotu střední hodnoty rozdělení, kterou chceme testovat, v parametru *popmean*. Metoda opět vrátí testovou statistiku a p-hodnotu. Syntaxi t-testů v Pythonu demonstruje následující jednoduchý příklad:

```
1 from scipy import stats
2 import numpy as np
3
4 #generovani dat na nichz budeme testovat hypotezy
5 x = stats.norm(loc = 5, scale = 2).rvs(size = 100)
6 y = stats.norm(loc = 4.5, scale = 1).rvs(size = 50)
7
8 #testovani hypotezy, ze se stredni hodnoty rovnaji
9 print(stats.ttest_ind_from_stats(mean1 = np.mean(x), mean2 = np.mean
     (y),
10                               nobs1 = 100, nobs2 = 50,
```

```
                              std1 = np.std(x), std2 = np.std(y)))

#testovani hypotezy, ze se stredni hodnoty rovnaji pomoci ttest_ind
    ()
print(stats.ttest_ind(x, y))

#testovani hypotezy, ze x ma stredni hodnotu 5
print(stats.ttest_1samp(a = x, popmean = 5))
```

Testy vypíší následující informace (i když je samozřejmě možné přistupovat k jednotlivým proměnným výstupu i jednotlivě):

```
Ttest_indResult(statistic=0.6504960906337116, pvalue
    =0.5163803759869349)
Ttest_indResult(statistic=0.6468260057611686, pvalue
    =0.518746450772954)
Ttest_1sampResult(statistic=-1.5503527114955207, pvalue
    =0.12424788289521069)
```

3.7 Výpočty teorie grafů

V poslední části této kapitoly již jen krátce popíšeme základní metody knihovny SciPy pro výpočty několika základních grafových algoritmů. K práci s grafy v Pythonu slouží modul *scipy.sparse.csgraph*. Některé algoritmy teorie grafů jsou také implementovány v modulu *scipy.spatial* pro práci s body v prostoru.

Grafy v modulu *scipy.sparse.csgraph* jsou reprezentovány incidenční maticí. Jedná se o matici, jejíž řádky i sloupce reprezentují vrcholy grafu, které jsou nějak očíslované. Nachází-li se mezi vrcholem m a vrcholem n hrana, je prvek v matici s indexy m, n roven jedné. Ostatní prvky matice jsou nulové. Jsou-li hrany v grafu, který chceme reprezentovat, ohodnocené nějakým číslem, reprezentujeme ohodnocenou hranu v matici incidence tímto číslem namísto jedničky. Matice může být ohodnocena v různých směrech různým číslem a dokonce může být v jednom směru hrana reprezentována nulou, zatímco v druhém směru nějakým číslem, což implikuje, že daná hrana je orientovaná a lze ji použít jen v jednom směru. Na diagonále matice incidence jsou vždy nuly (vrchol nemůže být spojen hranou sám se sebou. Navíc je zřejmé, že matice incidence neorientovaného grafu je vždy symetrická. Matici incidence (”Compressed Sparse Row matrix”) vytvoříme z Numpy pole pomocí příkazu **crs_matrix()**. Například matici incidence pro graf o pěti vrcholech, který obsahuje hrany mezi 1. a 2 vrcholem, 1. a 5. vrcholem a mezi 4. a 3. vrcholem, vytvoříme takto:

```

import numpy as np
from scipy.sparse import csgraph
from scipy.sparse import csr_matrix

arr = np.array([[0,1,0,0,1], [1,0,0,0,0],
                [0,0,0,1,0], [0,0,1,0,0],
                [1,0,0,0,0]])
```

```
9
10 newarr = csr_matrix(arr)
11
12 print(newarr)
```

Vypíšeme-li pole na standardní výstup pomocí příkazu **print()** tak, jako v příkladu výše, vypíše sloupec dvojic reprezentujících hrany a vedle nich váhu každé takové hrany takto:

```
1   (0, 1)  1
2   (0, 4)  1
3   (1, 0)  1
4   (2, 3)  1
5   (3, 2)  1
6   (4, 0)  1
```

I na tomto příkladu je vidět zdroj častých chyb, totiž Pythonovské indexování. Nezapomínejme, že Python indexuje od nuly. Nyní zkusíme mírně pozměnit náš kód a přiřadit různým stranám váhy jiné než jedna:

```
1
2 import numpy as np
3 from scipy.sparse import csgraph
4 from scipy.sparse import csr_matrix
5
6 arr = np.array([[0,2,0,0,1], [4,0,0,0,0],
7                 [0,0,0,1,0], [0,0,1,0,0],
8                 [-1,0,0,0,0]])
9
10 newarr = csr_matrix(arr)
11
12 print(newarr)
```

Výsledná reprezentace grafu, kterou Python vypíše, vypadá takto:

```
1   (0, 1)  2
2   (0, 4)  1
3   (1, 0)  4
4   (2, 3)  1
5   (3, 2)  1
6   (4, 0)  -1
```

Nyní když máme graf, můžeme na něj začít aplikovat populární grafové algoritmy z Teorie grafů. Můžeme například hledat souvislé komponenty našeho grafu pomocí funkce **connected_components()** modulu *scipy.sparse.csgraph* takto:

```
1
2 arr = np.array([[0,2,0,0,0], [4,0,0,0,0],
3                 [0,0,0,1,0], [0,0,1,0,0],
4                 [0,0,0,0,0]])
5
6 newarr = csr_matrix(arr)
7
8 print(csgraph.connected_components(newarr))
```

Pro graf v tomto případě vrátí příkaz dvojici (3, array([0, 0, 1, 1, 2])). První prvek dvojice je číslo reprezentující počet souvislých komponent v grafu. Druhým prvkem dvojice je

pole, které má stejný počet prvků jako zkoumaný graf a kde jsou stejným číslem označeny vrcholy náležící do stejné souvislé komponenty.

Dalším užitečným grafovým algoritmem je Dijkstrům algoritmus pro hledání nejkratší cesty grafem, který můžeme používat pomocí metody **dijkstra()**. Ta bere na vstup matici incidence v parametru *csgraph*. Vrátí matici nxn, kde hodnota uvedená v prvku s indexem ij reprezentuje délku nejkratší cesty z vrcholu i do vrcholu j. Pokud nastavíme parametr *return_predecessors* na hodnotu TRUE, vrátí také matici, podle které je možné danou nejkratší cestu grafem zreplikovat. Řádek i představuje všechny informace o cestě z vrcholu i do ostatních vrcholů. Prvek [i, j] daného pole představuje index předchozího vrcholu na cestě z vrcholu i do vrcholu j. Dijkstrův algoritmus přiřazuje na začátku všem vrcholům hodnotu ∞ a tato hodnota zůstane na konci běhu algoritmu přiřazena těm vrcholům, do nichž se z vrcholu, z něhož hledáme nejkratší cesty, nedá dostat. Python při běhu algoritmu přirozeně nepracuje s nekonečnem, neb by to působilo značné výpočetní obtíže. Místo toho bude těmto vrcholům přiřazena hodnota -9999, což je pro většinu grafů číslo dostatečně blízké nekonečnu, aby v průběhu běhu algoritmu nepůsobilo potíže. Chceme-li hledat nejkratší cesty pouze z několika vrcholů, nebo jen z jednoho vrcholu, a nechceme ztrácet čas hledáním ostatních cest, můžeme specifikovat vrcholy, z nichž hledáme nejkratší cestu pomocí parametru *indices*, jemuž předáme jejich indexy.

Pro demonstraci běhu Dijkstrova algoritmu si vytvoříme nový graf, jehož všechny vrcholy budou propojeny nějakou ohodnocenou hranou. Syntaxe hledání nejkratších cest v tomto grafu pak může vypadat například následovně:

```

arr = np.array([[0,2,0,0,1], [2,0,4,0,0],
                [0,4,0,1,0], [0,0,1,0,5],
                [1,0,0,5,0]])

newarr = csr_matrix(arr)

print(csgraph.dijkstra(newarr, return_predecessors=True, indices =
    [0, 1, 3]))
```

Výsledná matice délek nejkratších cest a *matice předchůdců* bude vypadat následovně:

```

(array([[0., 2., 6., 6., 1.],
       [2., 0., 4., 5., 3.],
       [6., 5., 1., 0., 5.]]),

array([[-9999,     0,     1,     4,     0],
       [    1, -9999,     1,     2,     0],
       [    4,     2,     3, -9999,     3]]))
```

Domníváme se, že výše zmíněný příklad dostatečně demonstruje možnosti a způsob používání modulu *scipy.sparse.csgraph*. Vážený čtenář si může další užitečné funkce pro provádění grafových algoritmů dohledat v dokumentaci. Tyto funkce obvykle fungují velmi podobně jako výše představená funkce **dijkstra()**.

Posuneme se nyní k modulu *scipy.spatial*, který obsahuje několik dalších užitečných

funkcí představujících známé algoritmy teorie grafů. Může být ovšem využita i pro úlohy z jiných oblastí matematiky než jen teorie grafů. Základním prvkem, s nímž tento modul pracuje, je nějaká diskrétní množina osamocených bodů v prostoru. Tyto body reprezentujeme jako pole polí (nebo pole n-tic). Každé jednorozměrné pole (nebo n-tice) v takové množině reprezentuje souřadnice nějakého bodu v Eukleidovském prostoru. Body mohou mít souřadnice rovné pouze booleovským nulám a jedničkám (nebo i True/False), což může být vhodné při výpočtech diskrétní matematiky. Knihovna tedy není primárně určena pro výpočty teorie grafů, ale pro práci s geometrickými obrazci, které z takových bodů můžeme poskládat. Umožňuje např. počítat jejich vzdálenost pomocí různých norem, provádět Delauneyovu triangulaci nebo hledat nejkratší cestu z nějakého vrcholu KD stromu.

Nejdříve si ukážeme funkce pro výpočet vzdáleností z modulu *scipy.spatial.distance*. K tomu můžeme použít funkci **euclidean()** pro výpočet eukleidovské vzdálenosti, **cityblock()** pro výpočet vzdálenosti pomocí L1 normy (Manhattan nebo též cityblock norma) a **hamming()** pro výpočet hammingovské vzdálenosti mezi binárními výrazy. Syntaxe pro výpočet vzdálenosti bodů těmito způsoby vypadá takto:

```

import numpy as np
from scipy.spatial import distance as dist

#body v Eukleidovskem prostoru
p1 = (1, 4, 5)
p2 = (4, 8, 0)

#body v prostoru binarnich vyrazu
p3 = (True, False, True, False)
p4 = (False, False, False, True)

#vypocet eukleidovske vzdalenosti bodu p1 a p2
print("Eukleidovska vzdalenost: ", dist.euclidean(p1, p2))

#vypocet cityblock vzdalenosti bodu p1 a p2
print('Manhattonska vzdalenost: ', dist.cityblock(p1, p2))

#vypocet hammingovske vzdalenosti bodu p1 a p2
print('Hammingovska vzdalenost: ', dist.hamming(p3, p4))
```

S body dále můžeme provádět celou řadu operací, pro něž nám *scipy.spatial* nabízí vhodné metody. Všechny tyto metody je možné dohledat v dokumentaci. Ukážeme si ještě slavný grafový algoritmus pro hledání konvexního obalu množiny bodů. Konvexní obal množiny bodů hledáme pomocí funkce **ConvexHull()** z modulu *scipy.spatial*. Na vstup předáváme množinu bodů, jejíž konvexní obal chceme najít a na výstupu získáme objekt reprezentující konvexní obal této množiny. Pomocí parametru *simplices* tohoto objektu získáme souřadnice hraničních bodů konvexního obalu, jejichž spojením rovnými čarami ve správném pořadí získáme nejmenší konvexní ohraničení celé množiny bodů. Syntaxe pro použití této funkce vypadá následovně:

```
from scipy import spatial
import numpy as np
```

```
3
4 #vytvoreni mnoziny bodu
5 myset = np.array([[1, 4], [5, 4],[7, 9], [9, 8],
6     [11, 1],[1, 2],[-4, 2],[-10, -2], [4, 5], [5, 12]])
7
8 #vypis hranicnich bodu konvexniho obalu teto mnozinz
9 print(spatial.ConvexHull(myset).simplices)
```

Tento kód vypíše 2D pole [[4 7],[9 7],[3 4],[3 9]], které představuje 4 body ohraničující konvexní obal námi definované množiny bodů.

Vzhledem k zaměření této práce již nebudeme dále rozebírat další funkce sloužící pro práci s grafy. Vážený čtenář si může všechny další užitečné funkce dohledat v dokumentaci. Jejich syntaxe i princip fungování jsou velmi podobné výše představeným funkcím.

3.8 Doplňující příklady ke kapitole

3.8.1 Příklady k procvičení

1. příklad:
   Spočtěte numericky jedno z řešení rovnice $x^4 + x - 12 = 0$. Dále spočtěte numericky minimum funkce $f(x) = x^4 + x - 12$ a její určitý integrál na intervalu [-1, 1].

2. příklad:
   Řešte numericky soustavu rovnic:

   $x^2 + yx + 1 = 0$
   $x + y + z = 0$
   $x + y + z^2 = 0$

   Až získáte výsledek, proveďte zkoušku, tedy tento výsledek ověřte dosazením do původních rovnic.

3. příklad:
   Spočtěte pomocí funkcí z knihovny SciPy dvojný integrál:
   $\int_0^1 \int_{x^2}^{x} x^2 + y - xydydx$

4. příklad:
   Představme si, že jsme v rámci nějakého experimentu měřili závislost vysvětlované proměnné *Y* na vysvětlující proměnné *X*. Pro hodnoty proměnné X = [1, 2, 4, 5] jsme naměřili funkční hodnoty Y = [14, 3, 4, 12]. Interpolujte funkci Y = f(X) pomocí kvadratických splajnů a následně odhadněte hodnotu funkce f(x) v bodě X = 3 a tuto odhadnutou hodnotu vypište na standardní výstup.

5. příklad:
   Generujte pomocí modulu *scipy.stats* 1000 výběrů z normálního rozdělení se střední hodnotou 2 a rozptylem 4. Následně z těchto dat spočtěte výběrový průměr a výběrový rozptyl. Nakonec náhodně vyberte 10 dříve generovaných hodnot a spočtěte v těchto

hodnotu funkce hustoty pravděpodobnosti normálního rozdělení se střední hodnotou 2 a rozptylem 4.

3.8.2 Řešení

1. příklad:

```
import numpy as np
from scipy.optimize import root, minimize
from scipy.integrate import quad

def func1(x):
    return x ** 4 + x - 12

solution_func1 = root(func1, 1)

min_func1 = minimize(func1, 0)

integral_func1 = quad(func1, -1, 1)

```

2. příklad:

```
def system(xyz):
    return np.array([
        xyz[0] ** 2 + xyz[0] * xyz[1] + 1,
        xyz[0] + xyz[1] + xyz[2],
        xyz[0] + xyz[1] + xyz[2] ** 2
    ])

solution_system = root(system, np.array([0, 0, 0]))

print(solution_system)
print(system(solution_system.x))

```

3. příklad:

```
from scipy.integrate import dblquad

integral = dblquad(lambda x,y: x**y + y - x*y,
                   0, 1, lambda x: x ** 2, lambda x: x)

```

4. příklad:

```
import numpy as np
from scipy.interpolate import interp1d

```

```
X = np.array([1, 2, 4, 5])
Y = np.array([14, 3, 4, 12])

f = interp1d(X, Y, kind = 'quadratic')
print(f(3))

```

5. příklad:

```
from scipy import stats
import numpy as np

distribution = stats.norm(loc = 2, scale = 4)

vybery = distribution.rvs(1000)

prumer = np.mean(vybery)

rozptyl = np.var(vybery)

vyber2 = np.random.choice(vybery, 10)

hustota = distribution.pdf(vyber2)

```

Kapitola 4

SymPy

4.1 Úvod do SymPy

SymPy je základní knihovnou v Pythonu pro provádění symbolických výpočtů. Matematické problémy jako výpočet integrálu, řešení rovnic a jejich soustav nebo výpočet vlastních čísel matice jsme v předchozích kapitolách řešili především numericky. Numerické metody ovšem řeší matematické problémy pouze aproximativně a dopouští se přitom numerických chyb. Občas je výhodné řešit matematické úlohy tzv. symbolicky, tedy podobně jak by k tomuto úkolu přistoupil např. student na hodině matematiky při řešení úlohy na papíře. Python ovšem umožňuje provádět všechny operace a výpočty násobně rychleji. Samozřejmě existují i jiné programy a specializované systémy počítačové algebry, které jsou schopny symbolicky řešit matematické úlohy. Využití knihovny Sympy ovšem umožňuje získané výsledky dále použít v Pythonovských programech a předat funkcím i z jiných knihoven. SymPy je tedy vhodným substitutem populárních programů pro symbolické výpočty, jakými jsou např. Maple nebo Sage. Všechny potřebné informace o knihovně SymPy uváděné v této kapitole a informace i nad rámec této kapitoly je možné dohledat v dokumentaci knihovny SymPy. [12]

4.2 Základní symbolické operace

SymPy pracuje s čísly odlišně od ostatních knihoven zmiňovaných v této práci. V případě, že pracuje s racionálními čísly, vypíše číslo samé. U iracionálních čísel ovšem (nestanovíme-li jinak) vrací symbolický výraz tak, aby nevznikla nepřesnost při zaokrouhlení. Tuto vlastnost můžeme vidět na následujícím příkladu, v němž počítáme druhou odmocninu pomocí knihovny SymPy (v této chvíli už čtenář nemusí být překvapen, že základní matematické operace jsou napříč pythonovskými knihovnami nazývány podobně, tedy funkce **sqrt()** představuje stejnou matematickou funkci v *SymPy* i v knihovně *math*):

```
>>> import sympy
>>> import math
>>> math.sqrt(3)
1.7320508075688772
>>> sympy.sqrt(3)
2*sqrt(2)
```

Funkce **sqrt()** z nám známé knihovny math vypsala aproximativní hodnotu třetí odmocniny ze tří. Stejná funkce z knihovny SymPy ví, že třetí odmocnina ze tří je iracionální číslo a proto vrátila jen její symbolické vyjádření, aby se nedopustila numerické chyby.

Chceme-li takový výraz převést na aproximativní vyjádření pomocí desetinného čísla, můžeme k tomu použít funkci **N()** nebo metodu **evalf()**. Pro výše zmíněný výraz bychom to mohli provést takto:

```
print(sympy.sqrt(3).evalf(), sympy.N(sympy.sqrt(3)))
```

Samozřejmě bylo by poměrně málo užitečné používat SymPy jako kalkulačku. Budeme chtít, aby se v našich symbolických výrazech začaly objevovat proměnné. K tomu používáme "symbolické proměnné", které definujeme takto:

```
from sympy import symbols
x, y = symbols('x y')
```

Nyní máme definované naše první dvě symbolické proměnné *x, y*. Nyní si vytvoříme první sympolický výraz s těmito proměnnými. Používáme k tomu známé operace +, -, *, / a ** v obvyklém významu základních matematických operací. Začneme jednoduchým polynomem $p(x,y) = xy + x^2 - y^2$, který v Pythonu definujeme takto:

```
p = x*y + x**2 - y**2
```

Tento výraz můžeme následně upravovat velmi intuitivně. Například zde k polynomu nejprve přičteme výraz x^3, následně přičteme výraz $y^2 - xy + x$ a nakonec celý výraz násobíme $2x$, abychom získali výsledný polynom $p(x) = 2x(x^3 + x^2 + x)$:

```
>>> p = p + x**3
>>> p = p + (y**2 - x*y + x)
>>> p = 2*x*p
>>> p
2*x*(x**3 + x**2 + x)
```

Následně můžeme chtít roznásobit závorky, což zajistí příkaz **expand()**. Budeme-li naopak chtít výraz rozložit na součin ireducibilních členů, můžeme to udělat pomocí příkazu **factor()**. Roznásobení a následný rozklad polynomů tedy provádíme takto:

```
>>> sympy.expand(p)
2*x**4 + 2*x**3 + 2*x**2
>>> sympy.factor(p)
2*x**2*(x**2 + x + 1)
```

Chceme-li funkci vyhodnotit v nějakém bodě, použijeme k tomu metodu **evalf()**, které v parametru *subs* předáme hodnotu, kterou má dosadit za naši proměnnou. Hodnotu našeho polynomu p(x) v bodě x=3 (p(3) = 234) můžeme vypsat takto:

```
>>> p.evalf(subs = {x : 3})
234.000000000000
```

Metoda **evalf()** umožňuje také vyhodnotit výraz tak, že za některou z proměnných dosadíme jiný symbolický výraz. Takto můžeme například dosadit za x v našem polynomu výraz $y - 4$:

```
>>> p.evalf(subs = {x : y -4})
2.0*(y - 4.0)*(y + 64.0*(0.25*y - 1)**3 + 16.0*(0.25*y - 1)**2 -
    4.0)
```

Vypsaný výraz je poměrně složitý a pracovalo by se s ním obtížně. Pokud bychom počítali danou úlohu na papír, výsledek bychom pravděpodobně chtěli zjednodušit. Pro zjednodušování symbolických výrazů poskytuje knihovna SymPy funkci **simplify()**, kterou můžeme s výhodnou použít v našem předchozím případě a kód bude vypadat takto:

```
>>> sympy.simplify(p.evalf(subs = {x : y -4}))
2.0*y**4 - 30.0*y**3 + 170.0*y**2 - 432.0*y + 416.0
```

Dále bychom mohli chtít najít kořeny našeho polynomu, což lze udělat snadno pomocí příkazu **solve()**, kterému předáme na vstup levou stranu rovnice (pravou stranu automaticky položí rovnou nule) a proměnnou, jejíž kořeny má funkce najít. Kořeny našeho polynomu p(x) nalezneme následujícím způsobem:

```
>>> sympy.solve(p, x)
[0, -1/2 - sqrt(3)*I/2, -1/2 + sqrt(3)*I/2]
```

Na vypsaném výsledku si můžeme povšimnout, že SymPy značí imaginární jednotku poměrně nestandardně velkým písmenem I.

Samozřejmě s pomocí stejného příkazu je možno snadno spočítat i řešení složitějších výrazů. Musíme si ovšem dát pozor, aby funkce, které se pokoušíme v rámci SymPy řešit měly analytické řešení, jinak Python vrátí chybovou hlášku. Symbolicky je možno spočítat jen to, co bychom byli schopni vyřešit na papíře, i když často za výrazně delší čas, než jaký bude potřebovat SymPy. Rovnici $sqrt(3) - 3 * exp(x) = 0$ například vyřešíme takto:

```
>>> q = sympy.sqrt(3) - 3 * sympy.exp(x)
>>> sympy.solve(q, x)
[-log(3)/2]
```

4.3 Výpočty matematické analýzy

Knihovna SymPy dále umožňuje snadno provádět symbolické výpočty matematické analýzy. Je opět třeba pamatovat na to, že ne všechny tyto problémy mají analytické řešení a ty, které nemají nám ani funkce knihovny SymPy nebude schopna vyřešit. Začneme výpočtem derivací, které provádíme jednoduše pomocí příkazu **diff()**. Naše v předchozí podkapitole definované funkce p(x) a q(x) můžeme derivovat následujícím způsobem:

```
>>> b = sympy.expand(sympy.diff(p, x))
>>> a = sympy.diff(q, x)
>>> a
8*x**3 + 6*x**2 + 4*x
>>> b
-3*exp(x)
```

Derivovat můžeme i vícenásobně a podle různých proměnných. Používáme k tomu velmi intuitivní syntaxi v rámci které předáváme funkci **diff()** jako další parametry postupně proměnné podle, kterých chceme derivovat. Takto můžeme například zderivovat polynom $g(x,y) = xy + x * y^2 - y^4 + y^3 * x^5$ nejdříve podle x, následně podle y a nakonec znovu podle x:

```
g = x*y + x * y**2 - y**4 + y**3 * x**5
ggg = sympy.diff(g, x, y, x)
```

Chceme-li funkce a, b zpětně integrovat pomocí neurčitého integrálu, použijeme k tomu funkci **integrate()** z knihovny SymPy takto:

```
>>> sympy.integrate(a, x)
-3*exp(x)
>>> sympy.integrate(b, x)
2*x**4 + 2*x**3 + 2*x**2
```

Můžeme také snadno používat vícenásobné integrály pomocí syntaxe velmi podobné vícenásobnému derivování. Takto integrujeme polynom b třikrát podle proměnné x a polynom g podle x a pak podle y:

```
>>> sympy.integrate(b, x, x, x)
x**6/15 + x**5/10 + x**4/6
>>> sympy.integrate(g, x, y)
x**6*y**4/24 + x**2*y**3/6 + x**2*y**2/4 - x*y**5/5
```

Knihovna SymPy umožňuje dokonce symbolicky řešit (integrovat) obyčejné diferenciální rovnice a jejich systémy. K tomu je nejprve potřeba nejprve definovat krom našich symbolických proměnných i symbolickou abstraktní funkci, jejíž předpis budeme hledat. Takovou abstraktní funkci definujeme pomocí funkce **Function()** takto:

```
from sympy import Function

f = Function('f')
```

Na takovou funkci pak můžeme použít funkci **Derivative()**, která symbolizuje, že danou abstraktní funkci derivujeme. Pomocí námi definovaných symbolů nyní napíšeme pravou stranu naší diferenciální rovnice (levou Python automaticky položí rovnou nule) a předáme jí na vstup spolu se symbolem funkce, kterou chceme zjistit, funkci **dsolve()** pro řešení obyčejných diferenciálních rovnic. Celé řešení diferenciální rovnice si ukážeme na rovnici $\frac{d^2f}{dx^2} + 4\frac{df}{dx} + 9f = 0$, kterou pomocí *Sympy* řešíme takto:

```
from sympy import Function, dsolve, Eq, Derivative, sin, cos,
    symbols
x = symbols('x')
f = Function('f')
dsolve(Derivative(f(x), x, x) + 4* Derivative(f(x), x) + 9*f(x), f(x
    ))
```

Funkce vrátí krásně napsané řešení naší diferenciální rovnice, jako bychom pro jeho sázení využili LaTeX:

$$f(x) = \left(C_1 \sin\left(\sqrt{5}x\right) + C_2 \cos\left(\sqrt{5}x\right)\right) e^{-2x}$$

LaTeXový kód příslušející výrazům napsaným pomocí funkcí knihovny SymPy ostatně můžeme nechat Python i vypsat pomocí příkazu **latex()** z knihovny SymPy. SymPy má v sobě LaTeXzabudovaný. LaTeXový kód příslušící této funkci bychom mohli získat například použitím následujícího kódu:

```
from sympy import Function, dsolve, Eq, Derivative, sin, cos,
    symbols, latex
x = symbols('x')
f = Function('f')
```

```
print(latex(dsolve(Derivative(f(x), x, x) + 4* Derivative(f(x), x) +
    9*f(x), f(x))))
```

Tento program by vypsal LaTeXkód:

```
f{\left(x \right)} = \left(C_{1} \sin{\left(\sqrt{5} x \right)} + C_
    {2} \cos{\left(\sqrt{5} x \right)}\right) e^{- 2 x}
```

Knihovna SymPy umí pracovat i se systémy diferenciálních rovnic a nabízí celou řadu dalších funkcí pro symbolické výpočty matematické analýzy. Vzhledem k relativně menšímu významu knihovny SymPy, oproti ostatním knihovnám zmíněným v této práci a celkovému zaměření práce se jim zde již nebudeme věnovat. Zvídavý čtenář může všechny potřebné funkce snadno dohledat v dokumentaci.[12]

4.4 Výpočty lineární algebry

Nakonec této kapitoly se ještě krátce zmíníme o možnostech výpočtů lineární algebry pomocí knihovny SymPy. Konkrétně k tomu používáme modul *sympy.matrices*. Abychom mohli takové výpočty provádět, je třeba si nejprve vytvořit naše první matice a vektory. Matice i vektory vytváříme pomocí příkazu **Matrix()**. Této funkci jednoduše předáme seznam reprezentující naši matici nebo vektor. Matice můžeme vytvářet také pomocí syntaxe podobné jazyku R, kdy příkazu **Matrix** předáme v tomto pořadí počet řádků, počet sloupců matice a nakonec jednorozměrný seznam, který bude přetransformován do nové matice. Pozor je třeba dávat jen na to, že na rozdíl od knihovny NumPy, knihovna SymPy automaticky předpokládá, že vektory jsou sloupcové. Naši první matici a vektor tedy můžeme vytvořit například takto:

```
>>> from sympy.matrices import Matrix
>>> M = Matrix([[1,0,0], [0,0,0]])
>>> M
Matrix([
[1, 0, 0],
[0, 0, 0]])
>>> v = Matrix([1, 4, 4])
>>> v
Matrix([
[1],
[4],
[4]])
>>> MM = Matrix(2, 3, [1, 2, 3, 4, 5, 6])
>>> MM
Matrix([
[1, 2, 3],
[4, 5, 6]])
```

Speciální typy matic můžeme vytvářet pomocí příkazů **eye()**, **zeros()**, **ones()** a **diag()**, které fungují podobně jako jejich ekvivalenty v knihovně NumPy. Zajímavější způsob vytváření matic je jejich deklarace pomocí lambda funkce. Příkazu **Matrix()** jednoduše předáme rozměry matice a dále mu předáme *lambda funkci* v proměnných *i, j*, která pro každý řádkový index *i* a sloupcový index *j* vrátí hodnotu prvku matice s těmito souřadnicemi, takto:

```
>>> Matrix(4, 4, lambda i, j: i + j + 1)
Matrix([
[1, 2, 3, 4],
[2, 3, 4, 5],
[3, 4, 5, 6],
[4, 5, 6, 7]])
```

Pro volání konkrétních prvků matic i jejich řádků, sloupců a podmatic používáme indexování pomocí hranatých závorek stejně jako u polí knihovny NumPy. Podobně jako v Numpy fungují také metody pro základní výpočty lineární algebry **det**(), **inv**(), **norm**(), **rank**(), **dot**() atd.

Prvky matice mohou být i symbolické výrazy a jejich funkce. I pro takové matice, jejichž prvky jsou symbolické proměnné, je možné počítat determinanty a inverze (ačkoli takové obecné inverze budou značně velké) a následně je vyhodnocovat pro různé hodnoty proměnných, takto:

```
>>>import sympy
>>> x,y = sympy.symbols('x, y')
>>> A = Matrix([[x, y, x + y], [x, 2, 3], [3+x, y*x, x-1]])
>>> A
Matrix([
[    x,   y, x + y],
[    x,   2,     3],
[x + 3, x*y, x - 1]])
>>> A.det()
x**3*y + x**2*y**2 - 4*x**2*y + 2*x*y - 8*x + 3*y
>>>A.inv()
Matrix([
[(-3*x*y + 2*x - 2)/(x**3*y + x**2*y**2 - 4*x**2*y + 2*x*y - 8*x +
   3*y), (x**2*y + x*y**2 - x*y + y)/(x**3*y + x**2*y**2 - 4*x**2*y
   + 2*x*y - 8*x + 3*y),         (-2*x + y)/(x**3*y + x**2*y**2 - 4*
   x**2*y + 2*x*y - 8*x + 3*y)],
[ (-x**2 + 4*x + 9)/(x**3*y + x**2*y**2 - 4*x**2*y + 2*x*y - 8*x +
   3*y),           (-x*y - 4*x - 3*y)/(x**3*y + x**2*y**2 - 4*x**2*y
   + 2*x*y - 8*x + 3*y), (x**2 + x*y - 3*x)/(x**3*y + x**2*y**2 - 4*
   x**2*y + 2*x*y - 8*x + 3*y)],
[(x**2*y - 2*x - 6)/(x**3*y + x**2*y**2 - 4*x**2*y + 2*x*y - 8*x +
   3*y),       (-x**2*y + x*y + 3*y)/(x**3*y + x**2*y**2 - 4*x**2*y
   + 2*x*y - 8*x + 3*y),        (-x*y + 2*x)/(x**3*y + x**2*y**2 - 4*
   x**2*y + 2*x*y - 8*x + 3*y)]])
>>> A.inv().evalf(subs = {x : 1, y : 2})
Matrix([
[-5.19875567647653e+128,  5.19875567647653e+128, 0.e-11],
[ 1.03975113529531e+129, -1.03975113529531e+129, 0.e-11],
[-5.19875567647653e+128,  5.19875567647653e+128, 0.e-11]])
```

Dále je možné využít speciální metody a funkce pro symbolickou lineární algebru, které jsou vlastní pro matice knihovny SymPy jako **QRdecomposition**() pro výpočet QR rozkladu matice, **charpoly**(), která vrací *lambda funkci* představující charakteristický polynom matice nebo **eigenvals**() pro výpočet vlastních hodnot matice:

```
>>> B = Matrix([[1, 2, 4], [1, 4, 7], [1, 1, 2]])
>>> B.QRdecomposition()
```

```
(Matrix([
[sqrt(3)/3,   -sqrt(42)/42, 3*sqrt(14)/14],
[sqrt(3)/3,  5*sqrt(42)/42,  -sqrt(14)/14],
[sqrt(3)/3, -2*sqrt(42)/21,   -sqrt(14)/7]]), Matrix([
[sqrt(3), 7*sqrt(3)/3,   13*sqrt(3)/3],
[      0,  sqrt(42)/3, 23*sqrt(42)/42],
[      0,           0,    sqrt(14)/14]]))
>>> B.charpoly()
PurePoly(lambda**3 - 7*lambda**2 + lambda + 1, lambda, domain='ZZ')
>>> B.eigenvals()
{7/3 + 46/(9*(298/27 + 2*sqrt(237)*I/9)**(1/3)) + (298/27 + 2*sqrt
   (237)*I/9)**(1/3): 1, 7/3 + 46/(9*(-1/2 + sqrt(3)*I/2)*(298/27 +
   2*sqrt(237)*I/9)**(1/3)) + (-1/2 + sqrt(3)*I/2)*(298/27 + 2*sqrt
   (237)*I/9)**(1/3): 1, 7/3 + (-1/2 - sqrt(3)*I/2)*(298/27 + 2*sqrt
   (237)*I/9)**(1/3) + 46/(9*(-1/2 - sqrt(3)*I/2)*(298/27 + 2*sqrt
   (237)*I/9)**(1/3)): 1}
```

Knihovna SymPy nabízí celou řadu dalších funkcí a metod pro řešení různých i velmi komplexních výpočtů a algoritmů lineární algebry. Fungují obvykle velmi podobně jako výše představené funkce, proto se jimi v této práci již nebudeme zabývat. Všechny funkce knihovny SymPy je možné snadno dohledat v dokumentaci. [12]

4.5 Doplňující příklady ke kapitole

4.5.1 Příklady k procvičení

1. příklad:
   Definujte pomocí knihovny SymPy symbolickou proměnnou x a s její pomocí následně definujte symbolický objekt představující výraz $p(x) = x^4 + x^2 + x + 4$. Vyhodnoťte tento výraz v bodě x = 3.

2. příklad:
   Najděte pomocí funkcí z knihovny SymPy nulové body výrazu z předchozího příkladu.

3. příklad:
   Nalezněte pomocí funkcí z knihovny SymPy neurčitý integrál funkce $p(x) = x^4 + x^2 + x + 4$.

4. příklad:
   Vytvořte pomocí funkcí knihovny SymPy matici A dimenze 3x3, která má všechny prvky na diagonále rovné jedné a všechny prvky mimo diagonálu rovné dvěma. Následně spočtěte symbolicky determinant, inverzi a vlastní čísla této matice.

5. příklad:
   Řešte symbolicky pomocí funkcí z knihovny SymPy obyčejnou diferenciální rovnici $2\frac{d^2f}{dx^2} + 3\frac{df}{dx} + f = 0$. Vypište latexový kód představující výsledek této úlohy.

4.5.2 Řešení

1. příklad:

```
from sympy import symbols
x, y = symbols('x, y')
p = x ** 4 + x ** 2 + x + 4
print(p.evalf(subs = {x : 3}))

```

2. příklad:

```
reseni = sympy.solve(p, x)

```

3. příklad:

```
integral = sympy.integrate(p, x)

```

4. příklad:

```
import sympy
A = sympy.ones(3) + sympy.eye(3)
determinat_A = A.det()
inverze_A = A.inv()
vlastni_cisla_A = A.eigenvals()

```

5. příklad:

```
from sympy import Function, dsolve, Eq, Derivative, sin, cos,
    symbols, latex
x = symbols('x')
f = Function('f')
print(latex(dsolve(2 * Derivative(f(x), x, x) + 3* Derivative(f
    (x), x) + f(x), f(x))))

```

Řešením diferenciální rovnice je funkce: $f(x) = C_1 e^{-x} + C_2 e^{-\frac{x}{2}}$

Kapitola 5

Knihovna Pandas

5.1 Úvod do Pandas

Pandas je hlavní Pythonovská knihovna používaná pro načítání, čištění a úpravu dat. Název knihovny Pandas odkazuje na sousloví ”Panel data”a ”Python Data Analysis”, což napovídá, k čemu knihovna slouží. Jedná se tedy o možná nejdůležitější Pythonovskou knihovnu používanou v datové vědě. Tato knihovna umožňuje pracovat s datovými tabulkami (*Data frames*), v nichž jsou data obvykle uložena. Tabulky můžeme načítat z csv souborů, JSON souborů a dalších obvyklých typů souborů pro ukládání dat. Můžeme také vytvářet zcela nové datové tabulky. Následně můžeme z tabulek odstraňovat chybějící hodnoty, měnit datový typ sloupců tabulky, měnit vzhled tabulky apod. Na datech v tabulkách můžeme spouštět mnohé agregační funkce a zjišťovat popisné statistiky. Nakonec můžeme tabulku zpětně uložit do některého z populárních typů souborů. Práce s datovými tabulkami bude připadat velmi povědomá těm čtenářům, kteří už mají zkušenosti s jazykem R. *Data frames* knihovny Pandas se chovají velmi podobně jako objekty *Data frames* nebo *tibble* v jazyce R. Při psaní této kapitoly jsme vycházeli především z dokumentace knihovny Pandas. [13] V dokumentaci knihovny Pandas je také možné dohledat veškeré důležité informace o využití knihovny Pandas, které by snad čtenáře mohly zajímat a v této práci je nenalezne.

Pokaždé, když chceme pracovat s objekty knihovny Pandas, musíme ji nejprve načíst, což uděláme pomocí kódu:

```
1 import pandas as pd
```

Importování knihovny Pandas pod zkráceným názvem *pd* je běžnou praxí a proto se ho budeme držet i v této práci. Řádkem uvedeným výše by měl začínat každý kód využívající objekty knihovny Pandas. V této práci budeme při některých příkladech z důvodu úspory místa tento řádek vynechávat. Čtenář by ale měl mít na paměti, že je nutný.

5.2 Data Frame a Series

Data Frame je vlastně tabulka, v níž každý řádek reprezentuje jedno pozorování a každý sloupec jednu pozorovanou proměnnou. Chceme li takovou tabulku vytvořit, využijeme k

tomu příkaz **DataFrame()** z knihovny Pandas, kterému předáme na vstup slovník, v němž řetězce, které použijeme jako klíče, reprezentují názvy našich pozorovaných proměnných a seznamy, které použijeme jako hodnoty uložené pod těmito klíči, reprezentují postupně hodnoty těchto proměnných v různých řádcích tabulky. Například takto si můžeme vytvořit tabulku v níž budeme zaznamenávat zaběhnuté časy dvou kamarádů v nějakém závodě v pěti po sobě následujících dnech:

```
import pandas as pd

sportovni_vykony = pd.DataFrame(
{
    'Den' : [1, 2, 3, 4, 5],
    'sportovec 1' : [5.2, 4.2, 3.3, 4.2, 5.8],
    'sportovec 2' : [1.2, 2.5, 4.3, 5.1, 12.5]
}
)
```

Tabulku, kterou jsme právě vytvořili, můžeme snadno vypsat příkazem **print()**. Python následně vypíše tabulku, která vypadá následovně:

	Den	sportovec 1	sportovec 2
0	1	5.2	1.2
1	2	4.2	2.5
2	3	3.3	4.3
3	4	4.2	5.1
4	5	5.8	12.5

Ke generování LaTeXového kódu pro *DataFrame* používáme v této práci užitečnou metodu objektů třídy *DataFrame* **to_latex()**. Metoda vypíše LaTeXový kód, který v LaTeXovém souboru vytvoří stejnou tabulku, jakou by vrátil Python po aplikaci příkazu **print()** na konkrétní *DataFrame*. Kód pro generování tabulky výše jsme získali pomocí příkazu:

```
print(sportovni_vykony.to_latex())
```

Povšimněme si, že Python automaticky přiřadil tabulce indexy řádků. Dále uspořádal naše data do sloupců tabulky, které jsou pojmenované podle našich proměnných. Chceme-li přiřadit řádkům tabulky jiné indexy, můžeme to snadno udělat pomocí parametru *index* takto:

```
import pandas as pd

sportovni_vykony = pd.DataFrame(
{
    'Den' : [1, 2, 3, 4, 5],
    'sportovec 1' : [5.2, 4.2, 3.3, 4.2, 5.8],
    'sportovec 2' : [1.2, 2.5, 4.3, 5.1, 12.5]
},
    index = ['a', 'b', 'c', 'd', 'e']
)
```

Naše tabulka bude nově indexována pomocí písmen a nikoli pomocí čísel, takto:

	Den	sportovec 1	sportovec 2
a	1	5.2	1.2
b	2	4.2	2.5
c	3	3.3	4.3
d	4	4.2	5.1
e	5	5.8	12.5

Jednotlivé řádky naší tabulky jsou objekty třídy *Series*. Jsou-li objekty třídy *Data Frame* jakýmsi ekvivalentem *Data Frame* vlastních jazyku R, pak objekty třídy *Series* jsou ekvivalentem vektorů jazyka R. Jedná se o řady prvků, které odpovídají jednotlivým hodnotám naší proměnné pro různá pozorování, kterým jsou přiřazené indexy. Jednotlivé sloupce naší tabulky můžeme volat podle jejich názvu v hranatých závorkách. Takto můžeme například zavolat první sloupec naší dříve vytvořené tabulky, který představuje zaběhnuté časy prvního z našich sportovců:

```
>>> sportovni_vykony['sportovec 1']
a    5.2
b    4.2
c    3.3
d    4.2
e    5.8
Name: sportovec 1, dtype: float64
```

Můžeme také vytvořit úplně nový objekt třídy *Series* pomocí příkazu **Series()**. Například vektor zaběhnutých časů prvního sportovce, který jsme výše vypsali z tabulky, můžeme také vytvořit přímo takto:

```
sportovec1 = pd.Series([5.2, 4.2, 3.3, 4.2, 5.8], name="sportovec1")
```

Můžeme také vybírat různé podmnožiny tabulky. Z tabulky můžeme vybrat několik sloupců a uložit je do nové tabulky. Takto např. můžeme vybrat z naší tabulky pouze sloupce 'sportovec 1' a 'sportovec 2':

```
sportovni_vykony[['sportovec 1', 'sportovec 2']]
```

Dále můžeme vybrat z tabulky jen některé řádky. Obvykle k tomu používáme nějakou logickou podmínku. Takto vybereme z naší tabulky jen ty řádky, kde má první sportovec čas lepší než 4 sekundy:

```
sportovni_vykony[sportovni_vykony['sportovec 1'] > 4]
```

Chceme-li vybrat řádky obsahující nějaké konkrétní hodnoty můžeme použít metodu **isin()** takto :

```
sportovni_vykony[sportovni_vykony['sportovec 1'].isin([4.2, 5.8])]
```

Chceme-li vybrat jen řádky tabulky, ve kterých se nevyskytují chybějící hodnoty, můžeme k tomu použít metodu **notna()** takto:

```
sportovni_vykony[sportovni_vykony['sportovec 1'].notna()]
```

Můžeme také kombinovat výběr specifických řádků a sloupců pomocí metody **loc[]**. Následující výraz vrátí tabulku pouze se sloupce 'sportovec 2', ale vynechá všechny řádky, kde má proměnná 'sportovec 1' hodnoty menší než 4:

```
sportovni_vykony.loc[sportovni_vykony['sportovec 1'] > 4, 'sportovec
    1']
```

Pomocí metody **iloc[]** je možné vybírat z tabulky řádky a sloupce pomocí jejich číselných indexů podobně, jako bychom vybírali podmatici z dvourozměrné matice v knihovně NumPy. Nezapomínejme ale, že Python indexuje od nuly. Takto bychom z naší tabulky vybrali první dva sloupce a první tři řádky:

```
sportovni_vykony.iloc[0:3, 0:2]
```

Pomocí podobné syntaxe můžeme prvky tabulky i měnit. Zjistíme-li, že výsledky prvního sportovce z prvních třech závodů musí být anulovány kvůli použití dopingu, můžeme to do tabulky zapsat takto:

```
sportovni_vykony.iloc[0:3, 1] = 'doping'
```

Tabulka nyní vypadá následovně:

	Den	sportovec 1	sportovec 2
a	1	doping	1.2
b	2	doping	2.5
c	3	doping	4.3
d	4	4.2	5.1
e	5	5.8	12.5

Stejně jako skoro na vše v Pythonu můžeme i na Pythonovské *Series* a *Data Frame* volat funkci **type()**, která nám v případě, že si nejsme jistí, potvrdí, že se skutečně jedná o daný typ objektu. Dále můžeme zjišťovat rozměry tabulky pomocí argumentu **shape**, v němž je u každé tabulky uložena dvojice obsahující počet řádků a sloupců tabulky. O naší tabulce zjistíme tyto informace takto:

```
>>> type(sportovni_vykony)
<class 'pandas.core.frame.DataFrame'>
>>> type(sportovni_vykony['sportovec 1'])
<class 'pandas.core.series.Series'>
>>> sportovni_vykony.shape
(5, 3)
```

5.3 Agregační funkce nad sloupci tabulky

Na jednotlivých sloupcích tabulky můžeme spouštět agregační funkce. Základní popisné statistiky vypíšeme pomocí metody **describe()**. Jednotlivé popisné statistiky vypsané příkazem **describe()** můžeme také vypsat samostatně pomocí jim příslušných metod. Například průměr dat ve sloupci tabulky získáme aplikací metody **mean()** na tento sloupec

a výběrovou směrodatnou odchylku získáme pomocí metody **std()**. Všechny metody pro významné popisné statistiky jsou nazvány obvyklým způsobem. Pro našeho prvního sportovce můžeme popisné statistiky získat takto:

```
>>> sportovni_vykony['sportovec 1'].describe()
count    5.000000
mean     4.540000
std      0.973653
min      3.300000
25%      4.200000
50%      4.200000
75%      5.200000
max      5.800000
Name: sportovec 1, dtype: float64
>>> sportovni_vykony['sportovec 1'].mean()
4.54
>>> sportovni_vykony['sportovec 1'].std()
0.9736529155710468
```

Povšimněme si, že metoda **describe()** vrací opět objekt třídy *Series*, který v sobě tentokrát obsahuje informace o počtu pozorování, průměru, výběrové směrodatné odchylce, minimu, maximu a významných kvantilech naší proměnné.

Popisné statistiky můžeme vypočíst také pro několik sloupců tabulky současně. Stačí použít zdvojené hranaté závorky, přičemž do vnitřních hranatých závorek vepíšeme názvy sloupců, přes které chceme agregovat. Takto například spočítáme medián času pro oba naše sportovce zároveň:

```
>>> sportovni_vykony[['sportovec 1', 'sportovec 2']].median()
sportovec 1    4.2
sportovec 2    4.3
dtype: float64
```

Můžeme také nechat vypsat několik různých popisných statistik pro více sloupců v tabulce pomocí metody **agg()**. Učiníme pomocí následující syntaxe, která vrátí tabulku námi vybraných popisných statistik:

```
sportovni_vykony.agg(
    {
        "sportovec 1": ["min", "max", "median", "skew"],
        "sportovec 2": ["min", "max", "median", "mean"],
    }
)
```

Tento kód vrátí tabulku:

	sportovec 1	sportovec 2
min	3.300000	1.20
max	5.800000	12.50
median	4.200000	4.30
skew	0.136616	NaN
mean	NaN	5.12

Kovarianční matici sloupců tabulky můžeme vypsat pomocí metody **cov()**. U naší tabulky sportovních výsledků bychom to mohli udělat takto:

```
sportovni_vykony.cov()
```

Hledaná kovarianční matice vypadá následovně:

	Den	sportovec 1	sportovec 2
Den	2.5	0.3000	6.3000
sportovec 1	0.3	0.9480	2.1565
sportovec 2	6.3	2.1565	19.3420

Korelační matici sloupců tabulky můžeme vypsat pomocí metody **corr()**. U naší tabulky sportovních výsledků bychom to mohli udělat takto:

```
sportovni_vykony.corr()
```

Hledaná korelační matice vypadá následovně:

	Den	sportovec 1	sportovec 2
Den	1.000000	0.194871	0.905983
sportovec 1	0.194871	1.000000	0.503610
sportovec 2	0.905983	0.503610	1.000000

5.4 Načítání dat ze souborů

Datové tabulky obvykle nevytváříme přímo v Pythonu tak jako zde. Data načítáme spíše z externích zdrojů. Načtená data uložíme do *Data Frame* a následně s nimi pracujeme. Datové tabulky, které takto upravíme a využijeme je pro tvorbu statistických modelů, následně můžeme zpětně uložit do některého z populárních typů souborů pro ukládání dat. Nejčastěji data načítáme (a poté do stejných souborů ukládáme) ze souborů csv, tsv, JSON, xlsx a html. Ukážeme si tedy v této kapitole, jak načítat data z těchto typů souborů. Python umožňuje načítat data i z jiných typů souborů s využitím specifických funkcí a knihoven. Vzhledem k relativní nedůležitosti ostatních způsobů ukládání dat se jimi zde nebudeme zabývat.

Načítání dat budeme v této kapitole demonstrovat na datech o nájemní ceně půdy pro pěstování vojtěšky v různých okresech amerického státu Minesota v roce 1977. [14] Data byla stažena z internetových stránek StatSci.org. [15] Tento datový soubor jsme zvolili pro jeho relativní jednoduchost a zároveň vhodné vlastnosti pro demonstraci všech zde demonstrovaných možností Pythonu. Datová tabulka obsažená v tomto souboru má pět sloupců:

1. Rent - průměrný nájem za akr půdy využívané pro pěstování vojtěšky v okrese
2. AllRent - průměrný nájem za akr orné půdy v okrese
3. Cows - počet krav na čtvereční míli v okrese

4. Pasture - podíl zemědělských pozemků využívaných jako pastviny
5. Liming - indikátor nutnosti vápnění pro pěstování vojtěšky v daném okrese ('Yes' pokud je vápnění nutné a 'No' pokud ne)

Tuto tabulku by si vážený čtenář měl dobře zapamatovat, neb nás bude na následujících stranách provázet ještě v mnoha příkladech. V tabulce je uvedeno 67 pozorování (Minesota má zjevně 67 okresů). To už je poměrně hodně pozorování na to, abychom si je vždy nechali všechny vypsat na obrazovku. Pokud se tedy pokusíme vypsat tabulku na standardní výstup, Python defaultně vypíše pouze prvních pět a posledních pět pozorování a mezi ně v každém sloupci umístí tři tečky. Další možností jak se u větších datových souborů `"podívat na data"` je použít metodu třídy *Data Frame* **head()** pro výběr pouze prvních pěti pozorování nebo metodu **tail()** pro výběr posledních pěti pozorování. Pokud naopak z nějakého důvodu toužíme po výpisu celé tabulky, nejjednodušším způsobem, jak to udělat je použití metody **to_string()**, která převede datovou tabulku na řetězec a ten již Python vypíše celý. Prvních pět ze 66 pozorování v naší datové tabulce vypadá následovně:

	Rent	AllRent	Cows	Pasture	Liming
0	18.38	15.50	17.25	0.24	No
1	20.00	22.29	18.51	0.20	Yes
2	11.50	12.36	11.13	0.12	No
3	25.00	31.84	5.54	0.12	Yes
4	52.50	83.90	5.44	0.04	No

Nakonec si, než začneme načítat data ze souborů, dovolíme krátkou poznámku o knihovně *os*. Jedná se o Pythonovskou knihovnu sloužící pro získávání dat o nastavení operačního systému, na kterém pracujeme a pro změnu některých těchto nastavení (ale nezapomeňme, že ne všechny změny nám operační systém vždy povolí, zvláště nejedná-li se o nějakou distribuci Linuxu). Tato knihovna má celou řadu funkcí, ale není cílem této práce je zde popisovat. Pro naše účely je potřeba především zjistit, nad kterým adresářem pracujeme a případně jak pracovní adresář změnit. Nastavení vhodného pracovního adresáře je vhodné, abychom nemuseli při načítání a ukládání souborů vždy psát celou absolutní cestu k souboru. Po nastavení pracovního adresáře stačí používat cestu relativní. Pro čtenáře, kteří se již dříve seznámili s jazykem R, jsou jistě známé příkazy **getwd()**, pro získání pracovního adresáře nad nímž náš program v R pracuje, a **setwd()** pro změnu tohoto pracovního adresáře. Obdobou těchto příkazů v Pythonu jsou příkazy z knihovny *os* **getcwd()** pro získání cesty k současnému pracovnímu adresáři a **chdir()** pro změnu pracovního adresáře. Před načítáním výše zmíněné tabulky ze složky 'bakalarka' jsme si vypsali současný pracovní adresář a nastavili nový takto:

```
>>> import os
>>> os.getcwd()
'C:\\Users\\salav\\AppData\\Local\\Programs\\Python\\Python39'
>>> os.chdir(r'C:\Users\salav\OneDrive\Dokumenty\bakalarka')
```

Data obvykle načítáme pomocí funkcí **read_[typ souboru]** a ukládáme pomocí metod, které jsou nazývány způsobem **to_[typ souboru]**. Naši tabulku s daty o ceně půdy pro pěstování vojtěšky můžeme z csv souboru načíst pomocí následující syntaxe:

```
landrent = pd.read_csv('landrent.csv')
```

Obdobně můžeme načíst stejnou datovou tabulku uloženou v jiných typech souborů. Využijeme k tomu následující syntaxe:

```
#html
landrent1 = pd.read_html('landrent.html')

#JSON
landrent2 = pd.read_json('landrent.json')

#Excel
landrent3 = pd.read_excel('landrent.xlsx')
```

Chceme-li načíst datovou tabulku ze souboru typu tsv, je možné použít funkci **read_csv**(), pouze stačí změnit separátor pomocí parametru *sep*. Naši datovou tabulku uloženou v tsv souboru můžeme načíst následujícím způsobem:

```
landrent4 = pd.read_csv('landrent.txt', sep='\t')
```

Zpětné uložení našich dat do různých typů souborů můžeme provézt následujícím způsobem:

```
#csv
landrent.to_csv('landrent.csv')

#html
landrent.to_html('landrent.html')

#JSON
landrent.to_json('landrent.json')

#Excel
landrent.to_excel('landrent.xlsx')
```

5.5 Úprava tvaru tabulky a transformace dat

Ve statistice často chceme naše data nějak transformovat nebo vytvořit novou proměnnou pomocí nějaké kombinace ostatních proměnných. Sloupce dat v datových tabulkách z knihovny Pandas můžeme transformovat velmi snadno. Chceme li nějaký sloupec transformovat, jednoduše ho z naší tabulky vybereme pomocí hranatých závorek tak, jak jsme to již demonstrovali dříve a přiřadíme mu novou hodnotu, která je nějakou transformací tohoto sloupce a nebo i sloupců ostatních. Chceme-li v naší tabulce vytvořit nový sloupec transformovaných hodnot, jednoduše vybereme pomocí hranatých závorek název sloupce, který v tabulce ještě není a do něj vložíme nové hodnoty.

Chceme-li logaritmovat hodnoty proměnné *Cows* v naší tabulce *landrent*, můžeme to udělat takto:

```
landrent['Cows'] = np.log(landrent['Cows'])
```

Chceme-li do tabulky přidat sloupec zachycující podíl proměnných *Rent* a *AllRent*, učiníme tak následujícím způsobem:

```
landrent['RentToAllRent'] = landrent['Rent'] / landrent['AllRent']
```

Pro změnu názvu sloupce tabulky používáme metodu **rename()**, které předáváme v proměnné *columns* slovník, v němž budou jako klíče uloženy názvy sloupců, které chceme měnit, a pod každým klíčem bude uložen nový název proměnné. Pokud například chceme změnit název proměnné *Cows* na *logCows*, učiníme tak takto:

```
landrent = landrent.rename(columns = {'Cows' : 'logCows'})
```

V tabulce také můžeme přeuspořádat řádky a podle nějakého klíče je seřadit. Seřazení řádků tabulky provádíme pomocí metody **sort_values()**, které v parametru *by* předáme název (názvy) sloupců, podle kterých chceme řádky tabulky seřadit. Předáme-li metodě v tomto parametru seznam více sloupců, seřadí řádky nejprve podle prvního zapsaného sloupce a pak seřadí řádky se stejnou hodnotou v tomto sloupci podle dalších sloupců. Chceme-li, aby byly řádky seřazeny sestupně, nastavíme hodnotu parametru *ascending* na *False*. Můžeme také řadit sestupně podle jednoho sloupce tabulky a vzestupně podle ostatních. Pythonu takové zadání sdělíme jednoduše tak, že mu v parametru *ascending* nepředáme jednu logickou hodnotu, ale seznam hodnot *False* a *True*.Naši tabulku *landrent* můžeme například seřadit sestupně podle proměnných *Rent* a *AllRent* takto:

```
landrent = landrent.sort_values(by = ['Rent', 'AllRent'], ascending
    = False)
```

Prvních pět řádků v tabulce *landrent* by po takových změnách vypadalo následovně:

	Rent	AllRent	logCows	Pasture	Liming	RentToAllRent
56	99.17	75.73	3.567559	0.05	No	1.309521
35	96.67	71.41	3.061988	0.05	No	1.353732
63	90.00	82.00	2.065596	0.03	Yes	1.097561
48	85.00	69.05	3.832547	0.22	Yes	1.230992
5	82.50	72.25	3.014063	0.05	Yes	1.141869

Dále můžeme chtít z nějakého důvodu použít nějaké opakující se hodnoty proměnné z tabulky jako nový sloupec. Převádíme tedy tzv. `"long format"` tabulky na `"wide format"`. Pro demonstraci tohoto postupu si naši tabulku nejprve trochu ”ořežeme”, aby byly změny lépe vidět. Vybereme z naší tabulky pouze proměnné *Liming* a *Rent* a uložíme je do nové tabulky takto:

```
Rent = landrent[['Liming', 'Rent']]
```

Prvních pět řádků tabulky nyní vypadá následovně:

	Liming	Rent
56	No	99.17
35	No	96.67
63	Yes	90.00
48	Yes	85.00
5	Yes	82.50

Následně budeme chtít zobrazit hodnoty proměnné *Rent* do dvou sloupců podle hodnoty proměnné *Liming*. Použijeme k tomu metodu **pivot()**, které v parametru *columns* předáme sloupec, jehož hodnoty budou použity jako nové sloupce, a v parametru *values* sloupec, jehož hodnoty budou rozděleny do nových sloupců. Sloupce si ještě přejmenujeme ”Liming”a ”No Liming”:

```
Rent = Rent.pivot(columns = 'Liming', values = 'Rent').rename(
    columns = {'No': 'No Liming', 'Yes': 'Liming'})
```

Nová tabulka *Rent* vypadá následovně:

Liming	No Liming	Liming
0	18.38	NaN
1	NaN	20.0
2	11.50	NaN
3	NaN	25.0
4	52.50	NaN

Můžeme si povšimnout, že tímto postupem vznikne v tabulce několik hodnot ’NaN’, tedy chybějících hodnot. To nám může a nemusí vadit. Je ovšem obvykle vhodnější převádět tabulku z `"long format"` na `"wide format"` v případě, že známe data pro všechny možnosti. Naši tabulku by tedy bylo vhodné takto upravit především v případě, kdy by byla v tabulce uvedena data o nájemní ceně půdy vyžadující vápnění i o půdě vápnění nevyžadující.

Obrácený postup, kde názvy sloupců tabulky využijeme jako hodnoty nové proměnné a hodnoty v těchto sloupcích jako hodnoty další proměnné, můžeme realizovat pomocí metody **melt()**:

```
Rent = Rent.melt()
```

U složitějších tabulek můžeme stanovit proměnnou *id_vars*, kde specifikujeme, které sloupce mají být použity jako hodnoty nového sloupce. Jména nových sloupců specifikujeme v proměnných *var_name* a *value_name*. Specifikovat všechny tyto proměnné se vyplatí zejména u složitějších tabulek, ale pro ilustraci bychom to mohli v našem případě udělat takto:

```
Rent = Rent.melt(var_name = 'Liming', value_name = 'Rent')
```

Metoda tabulku přeskládá tak, že v první sloupci bude informace o tom, z jakého původního sloupce tabulka pochází a v druhém sloupci bude konkrétní hodnota. Zde ovšem nastane problém. Změnou formátu tabulky z ”long”na ”wide”a zpět vznikne v naší tabulce 67 nových pozorování s hodnotou druhé proměnné *NaN*. Prvních pět řádků tabulky tedy

nově vypadá následovně:

	Liming	Rent
0	No Liming	18.38
1	No Liming	NaN
2	No Liming	11.50
3	No Liming	NaN
4	No Liming	52.50

Řádky, které obsahují hodnotu *NaN* tedy musíme odebrat pomocí dříve demonstrovaného postupu. Abychom po odebrání řádků zachovali indexování od 0 do 66, použijeme metodu **reset_index()** (pozor na to, že tato metoda vytvoří v tabulce nový sloupec původních indexů, což nám ovšem v tomto případě nevadí):

```
Rent = Rent.melt(var_name = 'Liming', value_name = 'Rent')
Rent = Rent[Rent['Rent'].notna()].reset_index()
```

Tabulka tedy vypadá podobně jako před použitím metody **pivot()**:

	index	Liming	Rent
0	0	No Liming	18.38
1	2	No Liming	11.50
2	4	No Liming	52.50
3	6	No Liming	25.00
4	8	No Liming	12.00

5.6 Kombinace dat z více tabulek

Na závěr kapitoly o knihovně Pandas si ukážeme, jak je možné kombinovat data z více datových tabulek do jedné tabulky. Tabulky můžeme skládat vedle sebe nebo nad sebe pomocí funkce **concat()** a nebo pomocí společného sloupce podobně jako v jazyce SQL pomocí funkce **merge()**. Pro demonstraci těchto možností si k naší datové tabulce *landrent* přidáme sloupec indexů pomocí *reset_index()* a následně si tabulku rozdělíme na tabulky *landrentA* obsahující sloupce *index* a *Rent*, *landrentB* obsahující sloupce *index* a *AllRent*, *Pasture* a *landrentC* obsahující sloupce *Liming* a *Cows*. Dále si vytvoříme tabulku *landrentC1* obsahující řádky z tabulky *landrentC*, kde má proměnná *Liming* hodnotu 'Yes' a *landrentC2* obsahující ostatní řádky tabulky *landrentC*. Tabulky, se kterými budeme dále pracovat jsme vytvořili následujícím způsobem:

```
import pandas as pd
import numpy as np
import os

os.chdir(r'C:\Users\salav\OneDrive\Dokumenty\bakalarka')

landrent = pd.read_csv('landrent.txt', sep='\t').reset_index()


```

```
landrentA = landrent[['index', 'Rent']]

landrentB = landrent[['index', 'AllRent', 'Pasture']]

landrentC = landrent[['Liming', 'Cows']]

landrentC1 = landrentC[landrentC['Liming'] == 'Yes']
landrentC2 = landrentC[landrentC['Liming'] == 'No']
```

Abychom získali tabulku *landrentC* zpět z tabulek *landrentC1* a *landrentC2*, musíme tyto tabulky složit pod sebe, což můžeme snadno udělat pomocí funkce **concat()**, které předáme seznam tabulek, které má poskládat vedle sebe a v parametru *axis* definujeme, zda-li má tabulky složit pod sebe nebo nad sebe (hodnota 0 znamená skládání pod sebou a hodnota 1 znamená skládání vedle sebe). Abychom navíc získali tabulku, která bude mít řádky ve stejném pořadí jako původní tabulka *landrentC*, můžeme použít metodu **sort_index()**, která seřadí řádky podle původních řádkových indexů:

```
landrentC = pd.concat([landrentC1, landrentC2], axis = 0).sort_index
    ()
```

Dále vytvoříme tabulku *landrentBB*, která vznikne z tabulek *landrentC* a *landrentB* složením vedle sebe. To provedeme snadno pomocí funkce **concat()**, pouze tentokrát použijeme *axis = 1*:

```
landrentBB = pd.concat([landrentB, landrentC], axis = 1)
```

Nakonec, abychom získali zpět původní tabulku *landrent* (až na permutaci sloupců), musíme složit vedle sebe tabulku *landrentBB* a *landrentA*. Obě tabulky ale mají společný sloupec *index*. Použijeme-li tedy funkci **concat()**, bude ve výsledné tabulce tento sloupec obsažen dvakrát. Lepší je proto použít funkci **merge()**. Tato funkce funguje podobně jako příkaz **join** v jazyce SQL a je ekvivalentem funkce **merge()** z jazyka R. Ke složení použije společný sloupec, který musí jednoznačně identifikovat každé pozorování. Společný sloupec definujeme v proměnné *on*. Dále můžeme definovat proměnnou *how*, ve které uvedeme zda-li provádíme ”left join”, ”right join”, ”outer join”, ”cross join”a nebo ”inner join”uvedením hodnoty ”left”, ”right”, ”outer”, ”cross”a nebo ”inner”. V případě ”left join”říkáme Pythonu, že má vzít všechny řádky z první tabulky a přiřadit jim hodnoty v odpovídajících sloupcích druhé tabulky. ”Right join”dělá pravý opak. ”Outer join”zachytí ve výsledné tabulce řádky z obou tabulek a v řádcích, které v jedné z obou tabulek chybí, budou hodnoty NaN. Naopak ”Inner join”vloží do výsledné tabulky pouze řádky vyskytující se v obou tabulkách. ”Cross join”vrátí karteziánský součin obou tabulek. Tabulky *landrentBB* a *landrentA* složíme pomocí funkce **merge()** takto:

```
landrent = pd.merge(landrentBB, landrentA, how = 'left', on = 'index
    ')
```

Získáme zpět tabulku *landrent* (až na permutaci sloupců) jejíchž prvních pět řádků vypadá následovně:

	index	AllRent	Pasture	Liming	Cows	Rent
0	0	15.50	0.24	No	17.25	18.38
1	1	22.29	0.20	Yes	18.51	20.00
2	2	12.36	0.12	No	11.13	11.50
3	3	31.84	0.12	Yes	5.54	25.00
4	4	83.90	0.04	No	5.44	52.50

Tabulku dat o nájemních cenách půdy pro pěstování vojtěšky tedy již umíme načíst, rozložit a znovu složit a její sloupce nejrůznějším způsobem transformovat. To je dobře, neb se s ní budeme na následujících stránkách často setkávat. V příští kapitole se naučíme jak tato data zobrazit do grafu.

5.7 Doplňující příklady ke kapitole

5.7.1 Příklady k procvičení

1. příklad:
   Vytvořte tabulku o 3 sloupcích, ve které bude pro všechny země V4 uveden jejich oficiální název, oficiální jazyk, příjmení prezidenta a počet obyvatel v milionech zaokrouhlený na celé miliony.

2. příklad:
   Vypište z tabulky název a příjmení prezidenta státu s nejvyšším počtem obyvatel.

3. příklad:
   Vypište průměrný počet obyvatel států v této tabulce.

4. příklad:
   Přidejte do tabulky sloupec udávající rozlohu státu v tisících km^2 zaokrouhlenou na celé km^2. Následně přidejte do tabulky sloupec udávající pro každý stát průměrný počet obyvatel na 1000 km^2 (tento sloupec vytvořte transformací sloupců, které již v tabulce jsou).

5. příklad:
   Vytvořte novou tabulku o třech sloupcích, kde bude pro všechny státy V4 uveden jejich název, křestní jméno prezidenta a HDP na obyvatele. Následně tyto dvě tabulky spojte pomocí funkce **merge()** podle sloupce s názvem státu.

5.7.2 Řešení

1. příklad:

```
import pandas as pd

V4 = pd.DataFrame({
    'stat' : ['Cesko', 'Slovensko', 'Polsko', 'Ma arsko'],
    'jazyk' : ['cestina', 'slovenstina', 'polstina', '
   madarstina'],
```

```
    'prezident' : ['Zeman', 'Caputova', 'Duda', 'Novakova'],
    'pocet_obyvatel' : [11, 5, 38, 10]})

```

2. příklad:

```
print(V4[['stat', 'pocet_obyvatel']][V4['pocet_obyvatel'] ==
   max(V4['pocet_obyvatel'])])

```

3. příklad:

```
print(V4['pocet_obyvatel'].mean())

```

4. příklad:

```
V4['rozloha'] = [79, 49, 323, 93]

V4['obyvatele_na_plochu'] = V4['pocet_obyvatel']/V4['rozloha']

```

5. příklad:

```
V4_new = pd.DataFrame({
    'stat' : ['Cesko', 'Slovensko', 'Polsko', 'Ma arsko'],
    'jmeno_prezidenta' : ['Milos', 'Zuzana', 'Andrzej', '
   Katalin'],
    'HDP' : [40027, 34654, 35561, 28375]
})

V4 = pd.merge(V4, V4_new, how = 'left', on = 'stat')

```

Kapitola 6

Knihovna Matplotlib

6.1 Úvod do Matplotlib

Knihovna Matplotlib je nejdůležitější Pythonovskou knihovnou pro vizualizaci dat. Umožňuje snadno vykreslit všechny běžně používané grafy a výsledky našich výpočtů. Tato kapitola přirozeně navazuje na kapitoly předchozí. V grafech knihovny Matplotlib totiž obvykle zobrazujeme především data z datových tabulek knihovny Pandas a polí knihovny NumPy. Je ovšem možné vykreslit data uložená i v seznamech a jiných typech polí. Knihovnu Matplotlib můžeme do našeho kódu importovat jednoduše pomocí kódu:

```
1 import matplotlib
```

Tento postup ovšem obvykle nepoužíváme. Většina běžně používaných funkcí knihovny Matplotlib je uložena v modulu *matplotlib.pyplot* z této knihovny. Obvykle tedy importujeme přímo tento modul. Modul *matplotlib.pyplot* je obvyklé importovat pod pseudonymem *plt*, takto:

```
1 import matplotlib.pyplot as plt
```

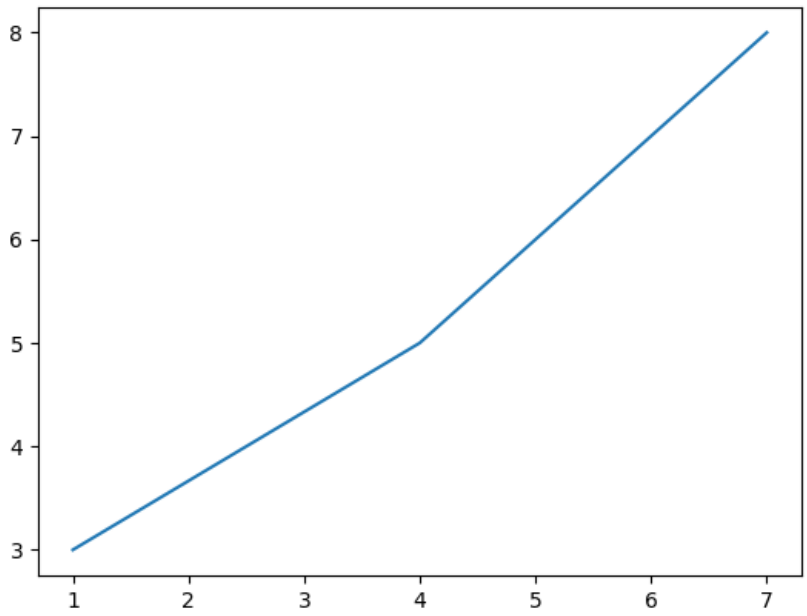
Při psaní této kapitoly jsme vycházeli především z dokumentace knihovny Matplotlib, kde může vážený čtenář také dohledat všechny dodatečné zde neuváděné informace, které by ho mohly zajímat. [16]

6.2 Spojnicový graf a obecné vlastnosti grafů

Základní funkcí knihovny Matplotlib je funkce **plot()**, která slouží k navržení spojnicového (a případně i bodového) grafu. Na vstupu bere funkce **plot()** pole x-ových souřadnic bodů v grafu a pole y-nových souřadnic těchto bodů. Graf vykreslíme pomocí funkce **show()**. Jednoduchý spojnicový graf s body [1, 3], [4, 5] a [7, 8] vykreslíme takto:

```
1 import matplotlib.pyplot as plt
2 plt.plot([1, 4, 7], [3, 5, 8])
3 plt.show()
```

Graf, který Python vypíše, vypadá následovně:

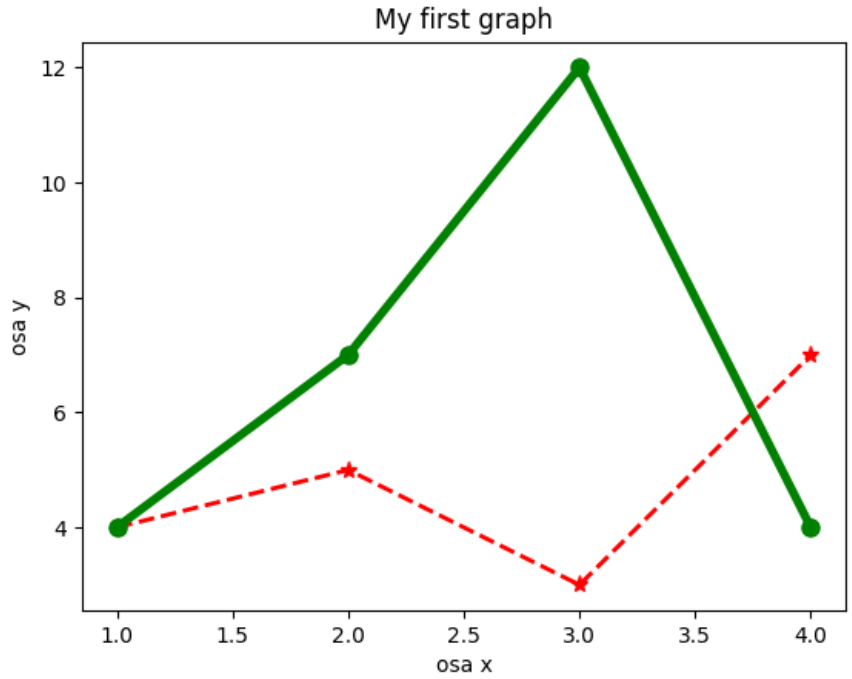


Takový graf samozřejmě esteticky neodpovídá požadavkům, které na grafy obvykle máme. Je tedy nutné nastavit i další parametry, které specifikují estetické parametry našeho grafu. Barvu spojnic v grafu nastavujeme v parametru *color*, jejich šířku v parametru *linewidth* a styl v parametru *linestyle*. Velikost vykreslených bodů stanovujeme v parametru *markersize* a styl jejich vykreslení v parametru *marker*. Všechny možné hodnoty těchto parametrů může čtenář snadno dohledat v dokumentaci knihovny Matplotlib. [16] Pro přidání dalších čar do grafu jednoduše využijeme funkci **plot** několikrát předtím, než graf vypíšeme pomocí funkce **show()**. Je-li to pro vykreslení nových čar do grafu třeba, Python automaticky upraví rozsah os grafu. Nadpis grafu stanovujeme pomocí funkce **title()**. Osy grafu pojmenováváme pomocí funkcí **xlabel()** a **ylabel()**. Takto můžeme například upravit vzhled našeho prvního grafu, vykreslit do něj druhou čáru a upravit nadpis i názvy os.

```
1  import matplotlib.pyplot as plt
2
3  plt.plot([1, 2, 3, 4], [4, 5, 3, 7], color = 'red',
4           linewidth = 2, linestyle = 'dashed', markersize = 8, marker
       = '*')
5
6  plt.plot([1, 2, 3, 4], [4, 7, 12, 4], color = 'green',
7           linewidth = 4, linestyle = 'solid', markersize = 8, marker
       = 'o')
8
9  plt.title('My first graph')
10 plt.xlabel('osa x')
11 plt.ylabel('osa y')
12
```

```
plt.show()
```

Náš nový upravený graf nyní vypadá takto:



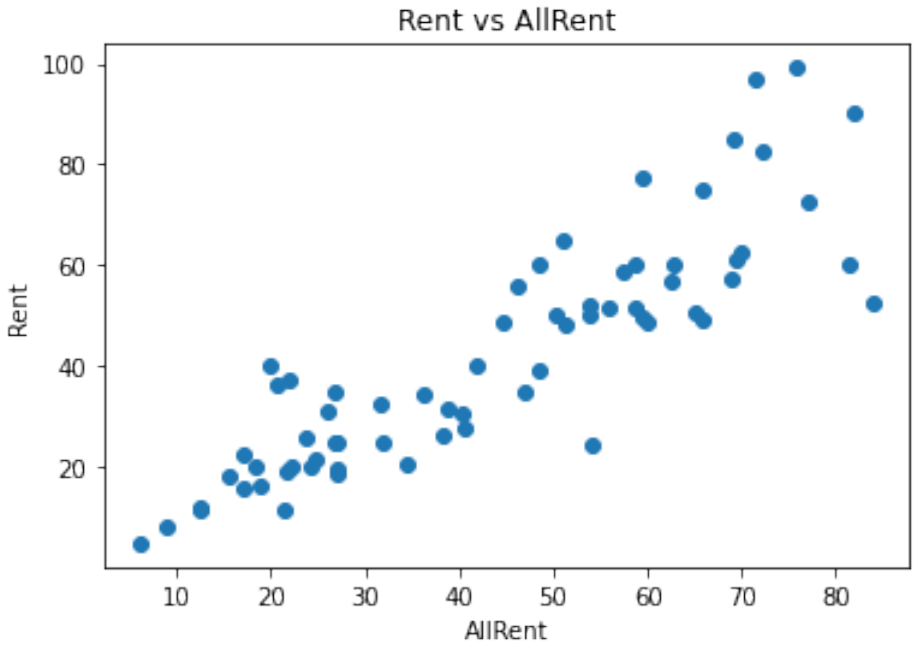
Chceme-li obrázek místo zobrazení uložit do složky, nad ktrou zrovna pracujeme, využijeme k tomu metodu **savefig()**, které předáme na vstupu název souboru, do kterého chceme náš obrázek uložit. Obrázek vykreslený v předchozím příkladu bychom mohli exportovat do formátu png např. takto:

```
plt.savefig('muj_obrazek.png')
```

Jako souřadnice bodů v grafech knihovny Matplotlib můžeme používat též Pandas *Series* (sloupce datových tabulek). Vraťme se nyní k naší tabulce *landrent* z předchozí kapitoly. Můžeme z ní vykreslit do grafu například hodnoty nájemní ceny půdy určené pro pěstování vojtěšky a nájemní ceny veškeré půdy v našich okresech, takto:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

landrent = pd.read_csv('landrent.txt', sep='\t')

plt.plot(landrent['AllRent'], landrent['Rent'], 'o')

plt.title('Rent vs AllRent')
plt.xlabel('AllRent')
plt.ylabel('Rent')

plt.show()
```

Symbol 'o' v tomto případě říká Pythonu, že má vykreslit pouze body grafu a ne spojnice. Graf, který tímto způsobem vykreslíme, vypadá následovně:

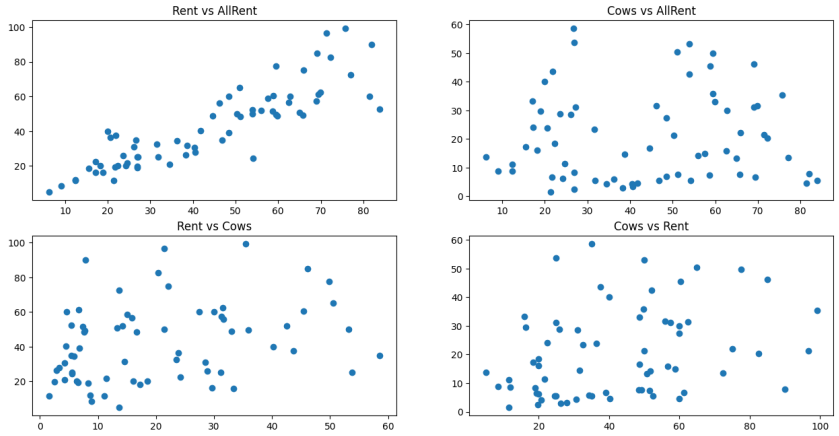


Můžeme také vykreslit více grafů najednou a to pomocí funkce **subplot()**. Vykreslené grafy budou uspořádány do řádků a sloupců. Funkce **subplot()** bere na vstup tři čísla. První dvě z nich budou pro každý graf, který chceme vykreslit, stejná a reprezentují počet řádků a počet sloupců našeho výsledného uspořádání grafů. Třetí číslo reprezentuje index grafu. Grafy budou uspořádány tak, že se podle indexů nejdříve vyplní řádek a následně jsou grafy přidávány na další řádky. Funkci **subplot()** s příslušným indexem musíme v našem kódu umístit nad každý graf, který chceme tímto způsobem vykreslit. Takto můžeme například vykreslit čtyři grafy, které zachytí vztah mezi proměnnými *Rent, AllRent* a *Cows* z naší tabulky *landrant*:

```
import maplotlib.pyplot as plt
import numpy as np
import pandas as pd

landrent = pd.read_csv('landrent.txt', sep='\t')

plt.subplot(2, 2, 1)
plt.plot(landrent['AllRent'], landrent['Rent'], 'o')
plt.title('Rent vs AllRent')

plt.subplot(2, 2, 2)
plt.plot(landrent['AllRent'], landrent['Cows'], 'o')
plt.title('Cows vs AllRent')

plt.subplot(2, 2, 3)
plt.plot(landrent['Cows'], landrent['Rent'], 'o')
plt.title('Rent vs Cows')

```

```
19 plt.subplot(2, 2, 4)
20 plt.plot(landrent['Rent'], landrent['Cows'], 'o')
21 plt.title('Cows vs Rent')
22 plt.show()
```

Výsledný vykreslený graf tedy vypadá následovně:

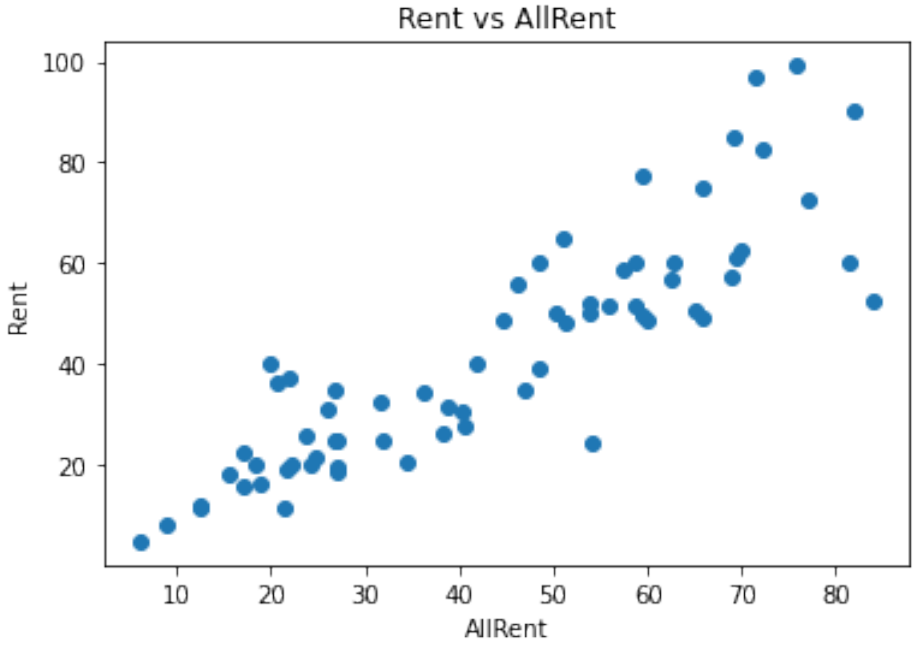


6.3 Bodový graf

Jak jsme ukázali na konci předchozí podkapitoly, k vykreslení bodových grafů lze využít funkci **plot()** z knihovny Matplotlib. Pro vykreslování bodových grafů knihovna Matplotlib ovšem nabízí také speciální funkci **scatter()**. Funguje velmi podobně jako funkce **plot()**, ale nabízí o něco větší volnost v nastavování parametrů našeho bodového grafu. Na vstup bere pole x-ových a pole y-ových souřadnic bodů, které následně vykreslí. Rovněž všechny estetické atributy této funkce jsou velmi podobné (pouze atributy upravující vzhled spojnic v grafu logicky chybí). Tak jako u všech grafů z modulu *pyplot* i zde nastavujeme nadpis a popisky os pomocí funkcí **title(), xlabel(), ylabel()** a graf zobrazujeme pomocí funkce **show()**. Stejně tak můžeme i grafy vytvořené pomocí funkce **scatter()** zobrazovat najednou a to i jinými typy grafů pomocí funkce **subplot()**. Graf zachycující vztah mezi proměnnými *Rent, AllRent*, který jsme v předchozí podkapitole vykreslili pomocí funkce **plot()** můžeme vykreslit pomocí funkce **scatter()** takto:

```
1 plt.scatter(landrent['AllRent'], landrent['Rent'])
2
3 plt.title('Rent vs AllRent')
4 plt.xlabel('AllRent')
5 plt.ylabel('Rent')
6
7 plt.show()
```

Výsledný graf vykreslený pomocí funkce **scatter()** bude vypadat stejně tak, jako graf vykreslený pomocí funkce **plot()**, tedy takto:

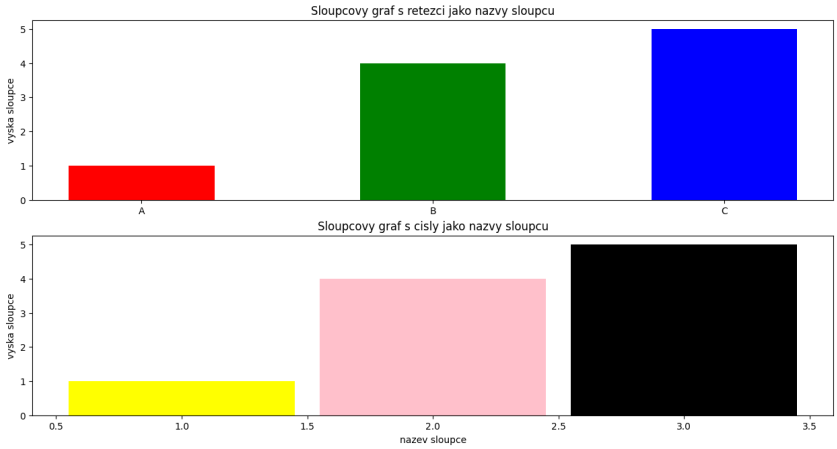


6.4 Sloupcové grafy a histogramy

Dalším důležitým typem grafů je graf sloupcový. V Pythonu můžeme sloupcový graf vykreslit pomocí funkce **bar()** z knihovny Matplotlib. Této funkci předáváme na vstup seznam názvů sloupců, které chceme zobrazit, a seznam jejich výšek. Šířku sloupců stanovujeme v proměnné *width* a barvu sloupců v proměnné *color*. Pokud proměnné *color* předáme název nebo hexadecimální kód pro jednu barvu, obarví touto barvou všechny sloupce. Předáme-li této proměnné seznam se stejným počtem prvků jako je počet sloupců, kde každý prvek je kódem pro nějakou barvou, můžeme každý prvek obarvit jinou barvou. Sloupcové grafy můžeme vykreslovat např. takto:

```
plt.subplot(2, 1, 1)
plt.bar(['A', 'B', 'C'], [1, 4, 5],
        color = ['red', 'green', 'blue'], width = 0.5)

plt.title('Sloupcovy graf s retezci jako nazvy sloupcu')
plt.xlabel('')
plt.ylabel('vyska sloupce')

plt.subplot(2, 1, 2)
plt.bar([1, 2, 3], [1, 4, 5],
        color = ['yellow', 'pink', 'black'], width = 0.9)

plt.title('Sloupcovy graf s cisly jako nazvy sloupcu')
plt.xlabel('nazev sloupce')
plt.ylabel('vyska sloupce')

plt.show()
```

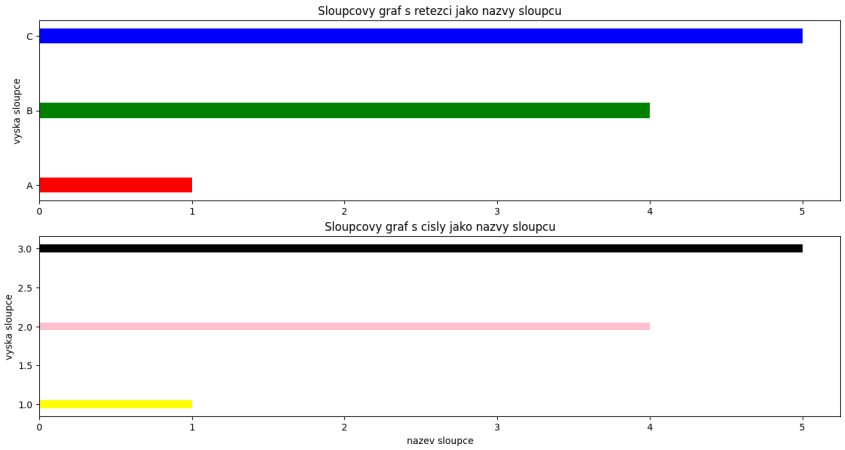
Sloupcové grafy, které tento kód vykreslí, vypadají následovně:



Chceme-li vykreslit sloupcový graf horizontálně, můžeme k tomu použít funkci **barh**() z knihovny Matplotlib. Princip fungování této funkce je prakticky stejný jako u funkce **bar**(), pouze místo argumentu *width* používáme argument *height*. Sloupcový graf výše můžeme překreslit otočený o 90° následujícím způsobem (z estetických důvodů jsme navíc zmenšili šířku sloupců):

```
plt.subplot(2, 1, 1)
plt.barh(['A', 'B', 'C'], [1, 4, 5],
        color = ['red', 'green', 'blue'], height = 0.2)

plt.title('Sloupcovy graf s retezci jako nazvy sloupcu')
plt.xlabel('')
plt.ylabel('vyska sloupce')

plt.subplot(2, 1, 2)
plt.barh([1, 2, 3], [1, 4, 5],
        color = ['yellow', 'pink', 'black'], height = 0.1)

plt.title('Sloupcovy graf s cisly jako nazvy sloupcu')
plt.xlabel('nazev sloupce')
plt.ylabel('vyska sloupce')

plt.show()
```

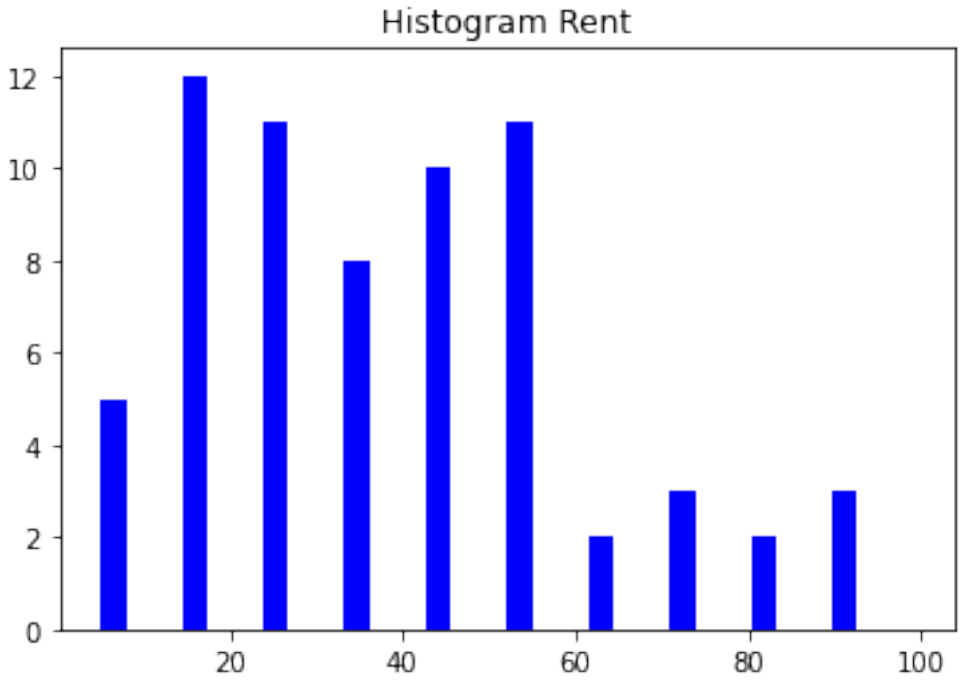
Výsledný otočený graf vypadá následovně:



Speciálním typem sloupcového grafu je histogram. Histogram zachycuje četnost různých numerických nebo kategoriálních hodnot v datech. V Pythonu můžeme histogram vykreslit pomocí funkce **hist()** z modulu *pyplot*. Estetické vlastnosti našeho histogramu stanovujeme pomocí stejných parametrů jako u sloupcového grafu. Například histogram proměnné *Rent* naší tabulky *landrent* můžeme vykreslit pomocí následujícího kódu:

```
plt.hist(landrent['Rent'], width = 2.9, color = 'blue')
plt.title('Histogram Rent')
plt.show()
```

Náš histogram vypadá následovně:

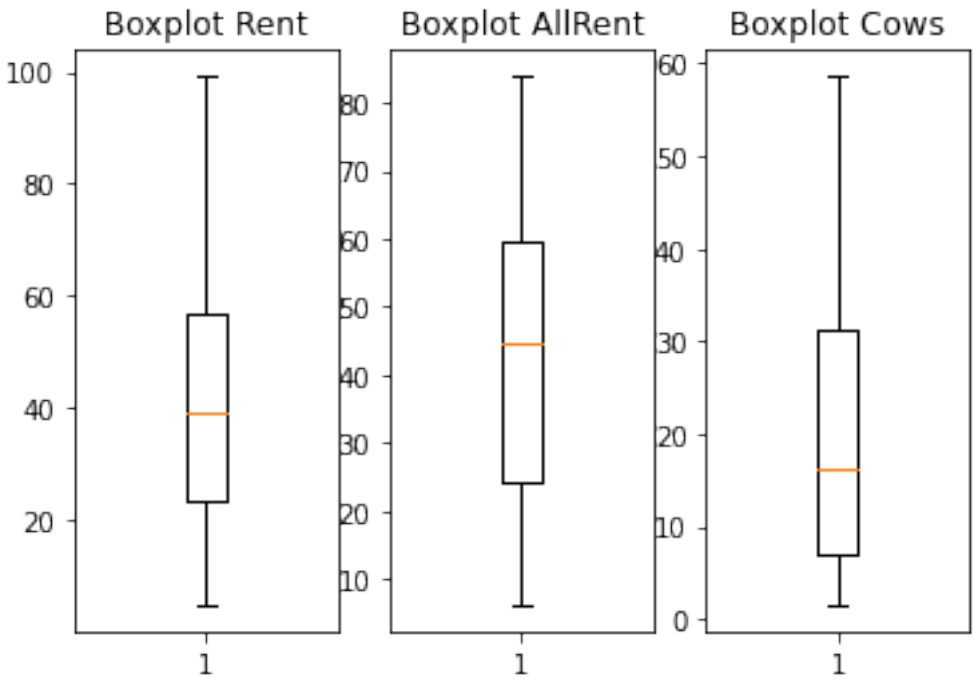


6.5 Další užitečné grafy

Obzvláště pro data s mnoha pozorováními je vhodné používat krabicový graf. Je velmi vhodný mimo jiné pro detekci odlehlých pozorování. Krabicový graf (neboli boxplot) v Pythonu vykreslujeme pomocí funkce **boxplot()** z knihovny Matplotlib. Krabicový graf některých sloupců z naší tabulky *landrent* můžeme zobrazit následovně:

```
plt.subplot(1, 3, 1)
plt.boxplot(landrent['Rent'])
plt.title('Boxplot Rent')

plt.subplot(1, 3, 2)
plt.boxplot(landrent['AllRent'])
plt.title('Boxplot AllRent')

plt.subplot(1, 3, 3)
plt.boxplot(landrent['Cows'])
plt.title('Boxplot Cows')

plt.show()
```

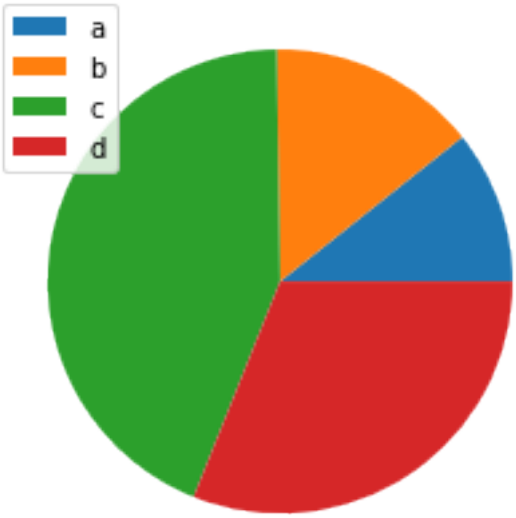
Tento kód vykreslí následující graf:



Dále můžeme chtít vykreslit koláčový graf, který je ideálním zobrazením podílu jednotlivých hodnot v datech. Pro vykreslení koláčového grafu používáme v Pythonu funkci **pie()** z knihovny Matplotlib. Tato funkce bere na vstup seznam četností jednotlivých hodnot a podle jejich poměru vykreslí koláčový graf. Pomocí metody legend můžeme navíc přidat legendu (vhodná pro koláčový graf, ale je možno ji přidat stejným způsobem i k ostatním dříve popsaným grafům) Koláčový graf můžeme vykreslit např. takto:

```
plt.pie([11, 15, 45, 32])
plt.legend(['a', 'b', 'c', 'd'])
plt.show()
```

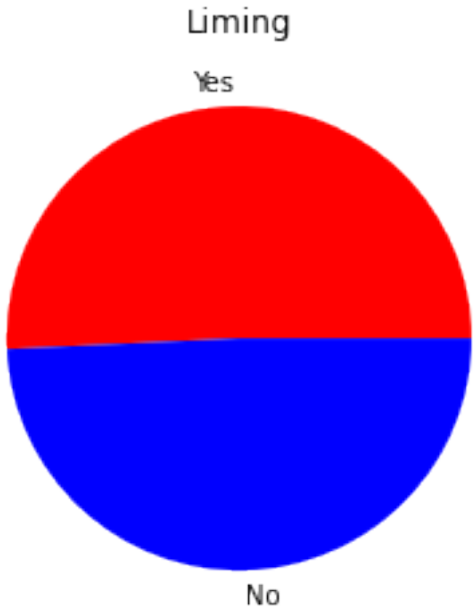
Takový koláčový graf by vypadal následovně:



Takový graf by samozřejmě k ničemu nebyl. Estetiku koláčového grafu můžeme nastavit pomocí podobných argumentů jako u ostatních grafů. Názvy jednotlivých kruhových výsečí z grafu nastavujeme v proměnné *labels* a jejich barvy v parametru *colors*. V grafu navíc je vhodné zobrazovat spíše četnosti reálných proměnných než vymyšlené četnosti jako v posledním příkladu. K tomu můžeme využít funkce z knihovny Pandas, se kterými jsme se seznámili v předchozí kapitole. Můžeme například v koláčovém grafu zobrazit relativní četnost okresů, kde je pro pěstování vojtěšky nutné vápnění, v našich datech, takto:

```
a = landrent.groupby('Liming').count()
plt.pie(a.iloc[:,1], labels = ['Yes', 'No'], colors = ['red', 'blue'
    ])
plt.title('Liming')
plt.show()
```

Tento nový lepší graf vypadá následovně:

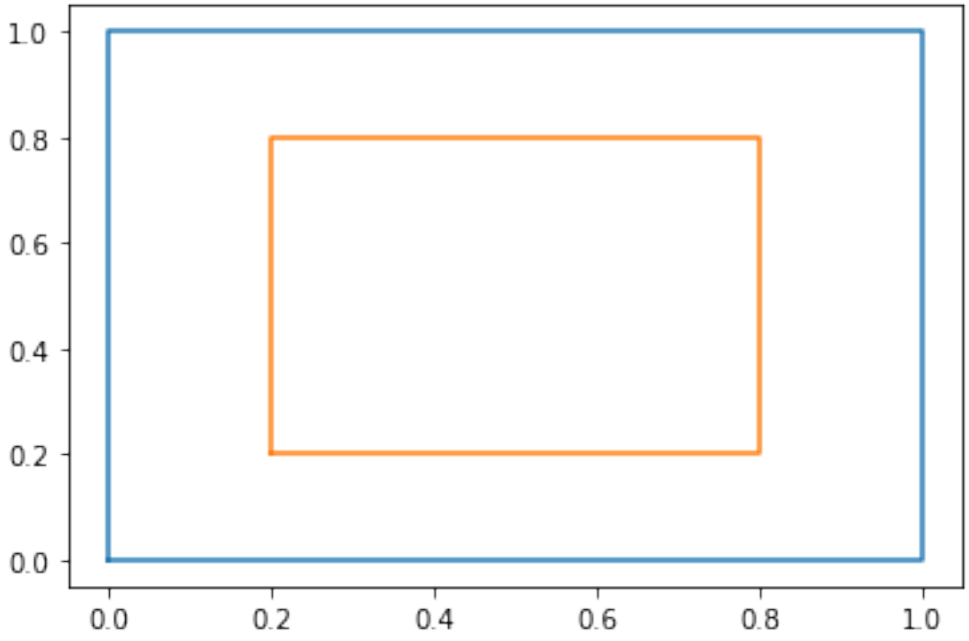


Dále je možné pomocí funkcí z knihovny Matplotlib vykreslit i další specifické druhy grafů. Po stažení příslušných doplňujících balíků je v Pythonu také možné kreslit 3D grafy apod. Vzhledem k relativně nižšímu významu dalších typů grafů a omezenému rozsahu této práce se jimi zde ale již nebudeme zabývat. Vážený čtenář si může funkce pro vykreslování dalších typů grafů, které by mohl potřebovat, snadno dohledat v dokumentaci knihovny Matplotlib. [16]

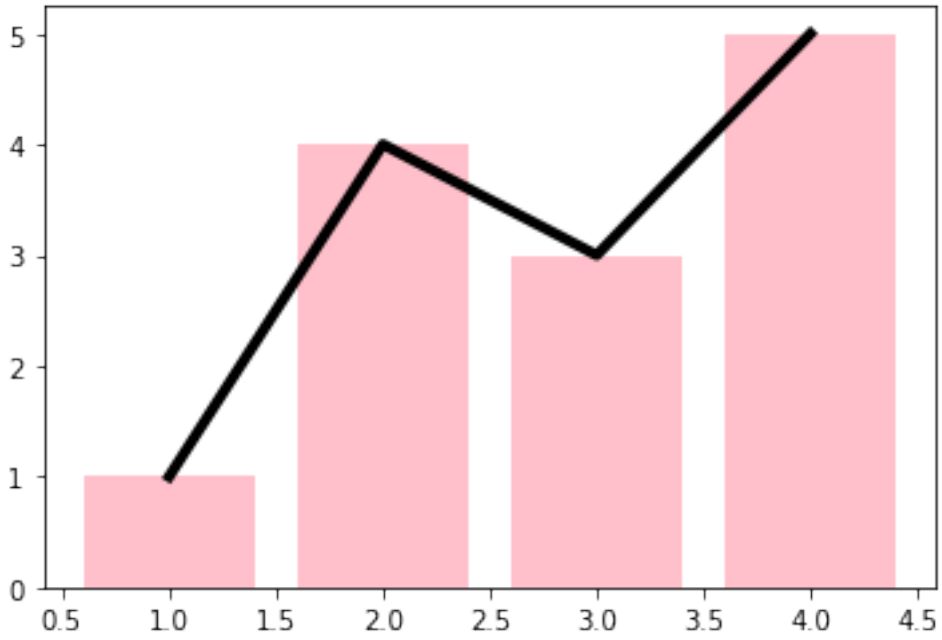
6.6 Doplňující příklady ke kapitole

6.6.1 Příklady k procvičení

1. příklad:
   Vykreslete pomocí funkcí z knihovny Matplotlib následující obrázek:



2. příklad:
   Vykreslete pomocí knihovny Matplotlib následující graf:

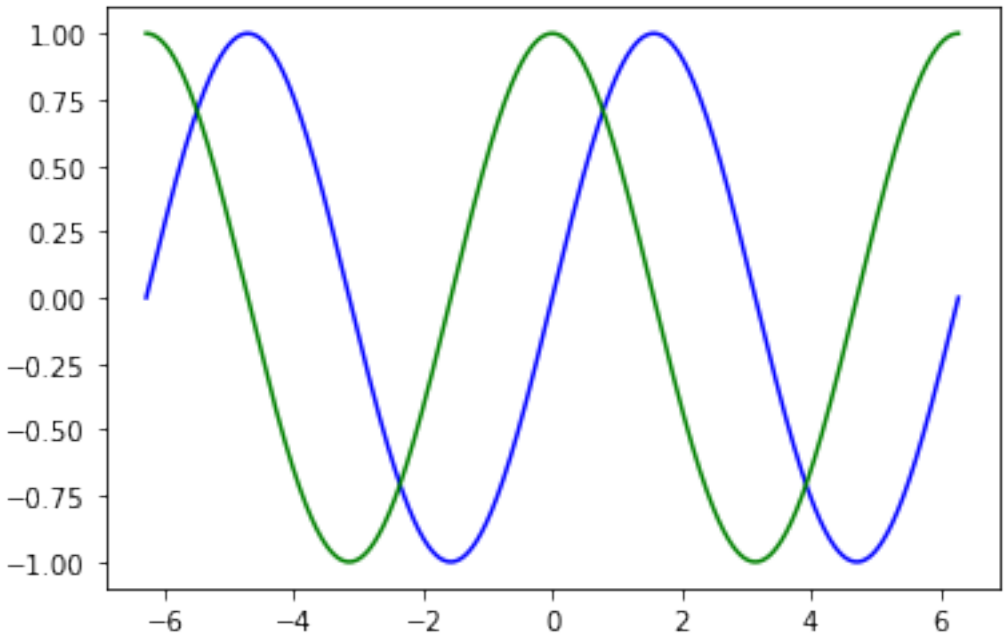


3. příklad:
   Honza, Zdenda a Tomáš pracovali na splečném projektu. Zdenda odvedl 50 % práce, Tomáš odvedl čtvrtinu práce a čtvrtinu práce odvedl Honza. Zdenda byl na kamarády naštvaný a oni slíbili, že se příště polepší. Když dělali příští společný projekt, Tomáš se polepšil a odvedl 60 % práce, Zdenda odvedl 30 % a Honza se naopak zhoršil a odvedl pouze 10 % práce. Vykreslete vedle sebe dva koláčové grafy (jeden pro každý projekt) zachycující podíl všech tří kamarádů na obou projektech.

Nezapomeňte přidat legendu zachycující, která barva značí kterého ze tří kamarádů a nadpisy obou grafů.

4. příklad:
   Načtěte tabulku *landrent*, se kterou jsme se seznámili v kapitole o knihovně Pandas. Vykreslete do stejného grafu bodový graf průměrného nájmu za akr půdy pro pěstování vojtěšky a průměrný nájem za akr ostatní orné půdy v závislosti na počtu krav na čtvereční míry. Body zachycující tyto dvě různé nájemní ceny půdy barevně odlište.

5. příklad:
   Vykreslete do stejného grafu křivku funkce sinus a cosinus a barevně je odlište, takto:



6.6.2 Řešení

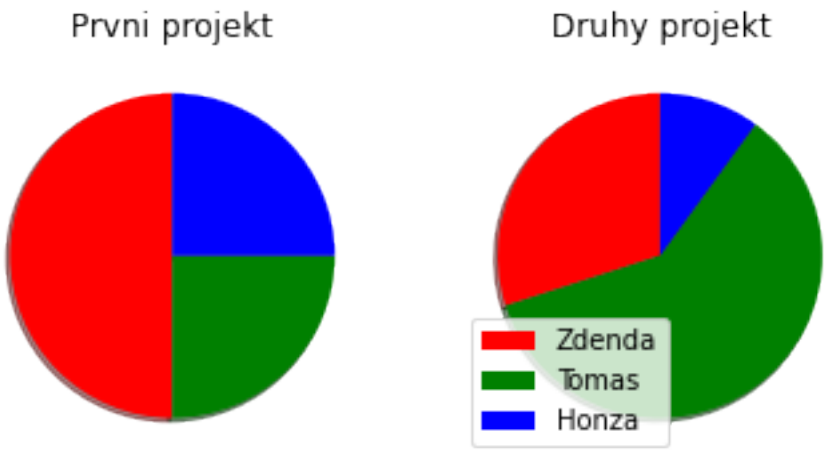
1. příklad:

```
import matplotlib.pyplot as plt
plt.plot([0, 0, 1, 1, 0], [0, 1, 1, 0, 0])
plt.plot([0.2, 0.2, 0.8, 0.8, 0.2], [0.2, 0.8, 0.8, 0.2, 0.2])

```

2. příklad:

```
import matplotlib.pyplot as plt
plt.bar([1, 2, 3, 4],[1, 4, 3, 5], color = 'pink')
plt.plot([1, 2, 3, 4],[1, 4, 3, 5], color = 'black', linewidth
    = 4)

```

3. příklad:
V tomto úkolu je čtenáři dána poněkud větší volnost, co se týče konkrétního zpracování. Výsledek by mohl vypadat například takto:



Kód pro takový graf vypadá následovně:

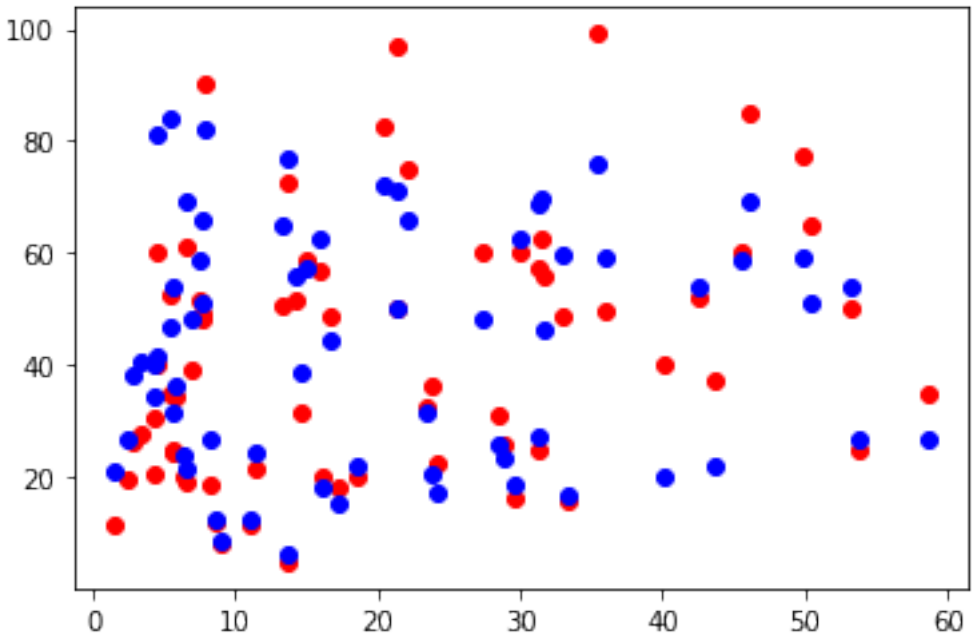
```
import matplotlib.pyplot as plt

colors = ['red', 'green', 'blue']
labels = ['Zdenda', 'Tomas', 'Honza']

plt.subplot(1, 2, 1)
plt.title('Prvni projekt')
plt.pie([50, 25, 25], colors=colors, shadow=True, startangle
    =90)

plt.subplot(1, 2, 2)
plt.title('Druhy projekt')
plt.pie([60, 30, 10], colors=colors, shadow=True, startangle
    =90)
plt.legend(labels, loc = 'best')

```

4. příklad:
Tuto úlohu můžete zpracovat např. takto:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import os

os.chdir(r'adresar_ve_kterem_je_ulozena_tabulka_landrent')

landrent = pd.read_csv('landrent.txt', sep='\t')

plt.plot(landrent['Cows'], landrent['Rent'], 'o', color = 'red'
    )
plt.plot(landrent['Cows'], landrent['AllRent'], 'o', color = '
    blue')
```

```
12
```

Výsledný graf by vypadal následovně:



5. příklad:

```
1  import pandas as pd
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  x = np.linspace(-2 * np.pi, + 2 * np.pi, 10001)
6
7  y = np.sin(x)
8
9  z = np.cos(x)
10
11 plt.plot(x, y, color = 'blue')
12 plt.plot(x,z, color = 'green')
13
```

Kapitola 7

Statistické modelování v Pythonu

7.1 Úvod do statistického modelování v Pythonu

Tato kapitola bude krátkým úvodem do statistického modelování v Pythonu. Není ambicí této práce popsat v celé šíři všechny funkce a metody, které je možno v Pythonu využít pro statistické modelování. Chceme však čtenáři poskytnout krátký přehled, sloužící spíše jako motivace pro další studium problematiky. Konkrétně se zde zaměříme na dvě pro statistické modelování nejpoužívanější knihovny jazyka Python - Statsmodels a Scikit-learn.

Obě knihovny jsou hojně využívány v praxi. Statsmodels bude pravděpodobně blízká těm čtenářům, kteří jsou obeznámeni s tvorbou stochastických modelů prostřednictvím jazyka R. Je ideální především pro ekonometrii a klasické statistické modelování. Při psaní části této práce pojednávající o knihovně Statsmodels vycházíme z dokumentace knihovny Statsmodels. [17]

Knihovna Scikit-learn přistupuje ke stejnému problému z jiného úhlu. Využívána je především pro potřeby strojového učení a příbuzných oborů. Poskytuje také některé funkce a metody specifické pro strojové učení, které ve Statsmodels vůbec není možné využít. V části této práce pojednávající o knihovně Scikit-learn vycházíme především z informací uvedených na webových stránkách projektu. [18]

Na následujících stránkách si ukážeme, jak odhadnout lineární statistický model z našich dat o cenách půdy pro pěstování vojtěšky. [14] Jednou tak učiníme s využitím knihovny Statsmodels a jednou za využití knihovny Scikit-learn. Na závěr si dovolíme již jen pár poznámek o několika dalších užitečných knihovnách, které je možno využít pro potřeby statistického modelování v Pythonu. Tyto knihovny již ale nebudeme vzhledem k omezenému rozsahu této práce podrobně zkoumat. Další studium pokročilejších metod matematického modelování v Pythonu již ponecháme na čtenáři. Nepochybujeme však, že po přečtení této práce je čtenář vybaven dostatečnou znalostí Pythonu, aby byl schopen se samostatně dovzdělat.

7.2 Lineární regresní model v knihovně Statsmodels

Možnosti knihovny Statsmodels budeme demonstrovat na základním prvním modelu, který bývá obvykle probírán na hodinách statistického modelování, totiž na normálním lineárním regresním modelu. Vraťme se k naší tabulce s daty o cenách půdy určené pro pěstování vojtěšky. Chceme vysvětlit cenu této půdy (proměnná *Rent*) pomocí proměnné *AllRent* zachycující průměrnou cenu veškeré zemědělské půdy v daném okrese a proměnné *Cows* představující průměrný počet krav na čtvereční míli v těchto okresech. Regresní model, který budeme odhadovat má tedy následující regresní rovnici:

$$\mathbf{Rent} = \beta_0 + \beta_1 \cdot \mathbf{AllRent} + \beta_2 \cdot \mathbf{Cows} + \varepsilon$$

Pro odhad koeficientů tohoto lineárního modelu můžeme využít následující kód:

```
import pandas as pd
import statsmodels.formula.api as smf
import os

os.chdir(r'adresar_v_nemz_je_ulozena_tabulka_landrent')

# nacteni dat
landrent = pd.read_csv('landrent.txt', sep='\t')

# odhad modelu
model = smf.ols(formula='Rent ~ AllRent + Cows', data=landrent).fit
    ()

# vypis vysledku
print(model.summary())
```

Pro použití lineární regresních modelů tohoto typu nemusíme importovat celou knihovnu statsmodels, ale postačuje pouze api balíku *formula*. Pro odhad lineárního regresního modelu obyčejnou metodou nejmenších čtverců slouží funkce **ols()**, které předáme na vstup data a regresní rovnici modelu. Regresní rovnici specifikujeme pomocí formule, kterou mnozí čtenáři již znají z jazyka R. Formule začíná vysvětlovanou proměnnou, která je od ostatních proměnných oddělena znakem ~. Za tento znak píšeme vysvětlující proměnné, které od sebe oddělujeme znakem **+**. Úrovňovou konstantu Python do modelu přidá automaticky. Nechceme-li, aby náš model obsahoval úrovňovou konstantu, musíme do formule modelu přidat **+ 0**. Na objekt třídy *ols* můžeme aplikovat různé metody. Pro odhad modelu používáme metodu **.fit()**. Alternativně můžeme pro modely s obecnou kovarianční maticí použít estimátor zobecněné metody nejmenších čtverců pomocí funkce **GLS()**. Knihovna Statsmodels nabízí také celou řadu dalších estimátorů např. pro případ autokorelace náhodných složek či metodu instrumentálních proměnných. Použít můžeme ale i daleko sofistikovanější modely pro panelová data nebo časové řady. Pro výpis základních informací o odhadnutém modelu používáme metodu **.summary()**. V našem případě tato metoda vypíše následující výstup:

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                   Rent   R-squared:                       0.838
Model:                            OLS   Adj. R-squared:                  0.833
Method:                 Least Squares   F-statistic:                     165.3
Date:                Fri, 03 Feb 2023   Prob (F-statistic):           5.22e-26
Time:                        16:46:57   Log-Likelihood:                -242.48
No. Observations:                  67   AIC:                             491.0
Df Residuals:                      64   BIC:                             497.6
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -6.1143      2.961     -2.065      0.043     -12.030      -0.199
AllRent        0.9214      0.054     17.121      0.000       0.814       1.029
Cows           0.3925      0.074      5.289      0.000       0.244       0.541
==============================================================================
Omnibus:                        4.113   Durbin-Watson:                   2.273
Prob(Omnibus):                  0.128   Jarque-Bera (JB):                3.609
Skew:                           0.318   Prob(JB):                        0.165
Kurtosis:                       3.943   Cond. No.                         138.
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

Ve výpisu jsou obsažena všechna základní kritéria pro hodnocení kvality modelu, odhadnuté regresní koeficienty, testová statistika i p-hodnota F-testu pro nulovou hypotézu, že jsou všechny koeficienty rovny nule, věrohodnost modelu, testové statistiky i p-hodnoty pro testy významnosti koeficientů atd. Ti čtenáři, kteří jsou seznámeni s jazykem R, si jistě povšimnou, že výpis je nápadně podobný výsledku funkce **summary()** v jazyce R. Můžeme se také odkazovat na jednotlivá kritéria modelu, která získáme pomocí příslušného atributu objektu získaného pomocí metody **fit()**. Například Akaikeho informační kritérium výše odhadnutého modelu můžeme získat pomocí kódu:

```
model.aic
```

Na odhadnutém modelu můžeme následně provádět nejrůznější testy. Například pomocí metody **.f_test()** můžeme provádět F-test lineárních omezení kladených na koeficienty modelu. Lineární omezení specifikujeme pomocí odpovídající matice lineárních omezení definované jako pole z knihovny NumPy. Například chceme-li testovat v našem odhadnutém modelu hypotézu, že koeficienty koeficienty β_1 a β_2 jsou stejné a součet koeficientů β_0 a β_1 je nula, můžeme to udělat pomocí následujícího jednoduchého kódu:

```
print(model.f_test(np.array([[0, 1, -1], [1, 1, 0]])))
```

Kód vrátí testovou statistiku, p-hodnotu, počet stupňů volnosti a počet lineráních omezení.

Knihovna Statsmodels nabízí samozřejmě i další užitečné funkce pro statistické modelování. Vzhledem k zaměření práce se jim zde již nebudeme věnovat a ponecháme je čtenáři pro samostudium.

7.3 Lineární regresní model v knihovně Scikit-learn

Nyní si ukážeme jak odhadnout jednoduchý lineární regresní model na našich datech o cenách půdy pro pěstování vojtěšky pomocí knihovny Scikit-learn. Opět budeme pracovat se stejnou regresní rovnicí ve tvaru:

$$\mathbf{Rent} = \beta_0 + \beta_1 \cdot \mathbf{AllRent} + \beta_2 \cdot \mathbf{Cows} + \varepsilon$$

Jak již bylo napsáno výše, knihovna Scikit-learn je zaměřena spíše na statistické modelování v praxi strojového učení. Přirozenou metodou testování výsledků našeho modelu je ve filozofii strojového učení crossvalidace (tj. rozdělení dat na trénovací a testovací sadu a následné testování "out of sample"predikčních schopností modelu). Nejdůležitějším kritériem pro určení kvality modelu je koeficient determinace a jeho obdoba počítaná na testovacích datech. Knihovna Scikit-learn neobsahuje funkce pro výpočet F-testů, t-testů ani dalších oblíbených kritérií využívaných v klasické statistice a ekonometrii. Odhadnout náš jednoduchý regresní model a otestovat jeho "out of sample"predikční schopnosti můžeme pomocí knihovny Scikit-learn následujícím způsobem:

```
import pandas as pd
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
import os

os.chdir(r'adresar_v_nemz_je_ulozena_tabulka_landrent')

# nacteni dat
landrent = pd.read_csv('landrent.txt', sep='\t')

# rozdeleni dat na zavisle a nezavisle promene
X = landrent[["AllRent", "Cows"]]
y = landrent["Rent"]

# Rozdeleni dat na testovaci a trenovaci sadu
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.2, random_state=0)

# Odhad modelu
model = LinearRegression()
model.fit(X_train, y_train)

# Predikce na testovacich datech
y_pred = model.predict(X_test)


# Vypis vysledku
print("Coefficients :", model.coef_)
print("Intercept:", model.intercept_)
print("R2 (train data): ", model.score(X_train, y_train),
      "\nR2 (test data): ",  model.score(X_test, y_test))
```

Z knihovny Scikit-lern jsme si nejprve importovali funkci **LinearRegression()** z balíku *linear_model* a funkci **train_test_split()** z balíku *model_selection*. Následně je třeba rozdělit naše data do dvou tabulek. Jedna z nich je sloupcem hodnot vysvětlované proměnné a v druhé tabulce jsou uloženy hodnoty všech vysvětlujících proměnných (tato tabulka nápadně připomíná "matici plánu", ale není v ní uveden sloupec jedniček pro absolutní člen regresní rovnice). Tyto proměnné dále rozdělíme pomocí funkce **train_test_split()** na trénovací a testovací část. Funkce **LinearRegression()** slouží k vytvoření objektu reprezentujícího náš regresní model. Pomocí metody **.fit()**, které předáme na vstup naše trénovací data můžeme následně odhadnout model. Na základě našeho modelu můžeme provést predikci a spočítat vyrovnané hodnoty pro testovací data pomocí metody **.predict()**. Pro výpis výsledků můžeme použít atributy modelu **.coef_** a **.intercept_**. Pro výpočet koeficientu determinace používáme metodu **.score()**, které předáme na vstup trénovací data. Obdobný postup je možno provést i u dat testovacích. Vypsané výsledky, které indikují, že náš model je poměrně dobrý, vypadají následovně:

```
Coefficients : [0.81829281 0.39306338]
Intercept: -2.672756587195721
R2 (train data):  0.8414975489133656
R2 (test data):  0.7864818479593565
```

V této chvíli můžeme pojednání o funkcích knihovny Scikit-learn ukončit. Knihovna Scikit-learn obsahuje i celou řadu dalších užitečných objektů, které slouží pro konstrukci složitějších regresních modelů, klasifikaci, regresi, metodu redukci dimenze atd. Rozsah této práce však již neumožňuje se všemi možnostmi této základní knihovny strojového učení zabývat. Ponecháme proto všechny tyto další objekty čtenáři pro samostudium.

7.4 Závěrečné poznámky o dalších důležitých knihovnách

Na konec této kapitoly si dovolíme ještě několik poznámek o dalších užitečných knihovnách v Pythonu, které jsou využívány pro potřeby statistického modelování a stojí za zmínku i přesto, že na ně v této práci nezbylo místo. Váženému čtenáři si dovolujeme doporučit k dalšímu studiu především knihovnu Keras sloužící ke konstrukci neuronových sítí v Pythonu a knihovnu TensorFlow sloužící pro výpočetně efektivní trénování pokročilejších modelů strojového učení. Dále je vhodné se zaměřit na knihovnu PyTorch, která umožňuje efektivně využívat metody zpracování přirozeného jazyka a obrazu. Pro optimalizaci matematických výpočtů se dále hodí knihovna Theano. Pro pokročilejší vizualizace dat je vhodná knihovna Seaborn, která staví na knihovně Matplotlib a nabízí i pokročilejší vizualizační metody. Nakonec ještě zmíníme knihovnu BeutifulSoup, která umožňuje snadnou extrakci dat z HTML a XML souborů. Všem těmto knihovnám by bylo vhodné věnovat samostatnou kapitolu. Bohužel rozsah ani zaměření této práce to neumožňují. Nabádáme proto čtenáře, aby se studiu těchto knihoven věnoval alespoň samostatně z dostupných zdrojů na internetu nebo ve vzdělávacích institucích.

Závěr

V závěru této práce bychom rádi pobídli čtenáře k dalšímu studiu programování v jazyce Python. Python je výkonný a univerzální programovací jazyk, který lze využít k řešení širokého spektra matematických a statistických úloh. Díky rozsáhlým knihovnám, jako je NumPy, SciPy, SymPy, Pandas, Matplotlib, Scikit-learn a Statsmodels, je Python mocný nástroj pro matematické výpočty a analýzu dat. Jeho studium je tedy nejen obohacující a zajímavé, ale také může být velmi nápomocné v budoucím studiu i kariéře.

V této práci jsme se snažili poskytnout čtenáři detailní pohled na možnosti využití jazyka Python pro matematické a statistické výpočty a zpracování dat. Doufáme, že jsme v ní zodpověděli veškeré otázky a že bude sloužit jako užitečný studijní materiál pro studenty matematiky na Přírodovědecké fakultě Masarykovy univerzity v Brně a pro každého, kdo má zájem se naučit programovat v Pythonu.

V neposlední řadě bychom chtěli poděkovat všem, kteří nás během psaní této práce podporovali. Děkujeme také všem čtenářům, kteří dočetli až sem a doufáme, že jim práce poskytla dostatečný základ pro další studium matematiky a programování v Pythonu.

Seznam použité literatury

[1] Python.org [online]. Wilmington: Python Software Foundation, 2022 [cit. 2022-07-04]. Dostupné z: https://www.python.org/downloads/

[2] Visual studio code [online]. Albuquerque: Microsoft, 2022 [cit. 2022-07-04]. Dostupné z: https://code.visualstudio.com/

[3] RStudio [online]. San Francisco: R Consortium, 2022 [cit. 2022-10-14]. Dostupné z: https://www.rstudio.com/products/rstudio/download/

[4] Python 3.10.5 documentation. Python.org [online]. Wilmington: Python Software Foundation, 2022 [cit. 2022-07-29]. Dostupné z: https://docs.python.org/3/

[5] RAMALHO, Luciano. Fluent Python: clear, concise, and effective programming. Second edition. Sebastopol: O'Reilly, 2022. ISBN 9781492056355.

[6] Pip documentation v22.2.2 [online]. Delaware, 2022 [cit. 2022-08-07]. Dostupné z: https://pypi.org/project/pip/

[7] VANDERPLAS, Jacob T. Python data science handbook: essential tools for working with data. Beijing: O'Reilly, [2017]. ISBN 1491912057

[8] W3schools.com [online]. w3schools.com, 2022 [cit. 2022-09-14]. Dostupné z: https://www.w3schools.com/python/numpy/numpy_ufunc_differences.asp

[9] NumPy [online]. Austin: NumFocus Foundation, 2022 [cit. 2022-09-23]. Dostupné z: https://numpy.org/doc/

[10] SciPy [online]. Austin: NumFocus Foundation, 2022 [cit. 2022-09-24]. Dostupné z: https://docs.scipy.org/doc/scipy/tutorial/general.html

[11] W3schools.com [online]. w3schools.com, 2022 [cit. 2022-09-14]. Dostupné z: https://www.w3schools.com/python/scipy/scipy_constants.php

[12] SymPy 1.11 documentation [online]. SymPy Development Team, 2022 [cit. 2022-10-12]. Dostupné z: https://docs.sympy.org/latest/tutorials/intro-tutorial/intro.html

[13] Pandas [online]. The pandas development team, 2022 [cit. 2022-11-03]. Dostupné z: https://pandas.pydata.org/docs/index.html

[14] Weisberg, S. (1985). Applied Linear Regression. Wiley, New York. Problem 6.5

[15] StatSci.org [online]. StatSci.org, 2022 [cit. 2022-11-06]. Dostupné z: http://www.statsci.org/index.html

[16] Matplotlib [online]. John Hunter, Darren Dale, Eric Firing, Michael Droettboom and the Matplotlib development team, 2022 [cit. 2022-11-17]. Dostupné z: https://matplotlib.org/stable/index.html

[17] Seabold, Skipper, and Josef Perktold. "statsmodels: Econometric and statistical modeling with python." Proceedings of the 9th Python in Science Conference. 2010.

[18] Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.