Bioinformatické centrum HK

Základy programování v jazyce Python pro biology

Kateřina Rodrová
Dominik Matoulek
Radka Symonová

Tento materiál vznikl v rámci realizace projektu Erasmus+ s názvem Bioinformatické centrum HK č. KA203-0611433, kategorie Strategická partnerství ve vysokém školství

Spolufinancováno
z programu Evropské unie
Erasmus+

1. Úvod ... 3
1.1. O této příručce ... 3
1.2. Co je programování a k čemu je dobré? ... 4
1.3. Proč Python? ... 5
1.4. Instalace Pythonu a interaktivní prostředí ... 5
1.5. Jupyter notebook ... 7
1.6. Google Colab ... 11
2. Práce s textem ... 11
2.1. Funkce print ... 12
2.2. Chybová hlášení ... 13
2.3. Proměnné ... 14
2.4. Manipulace s řetězci ... 15
2.5. Klíčová slova ... 18
3. Čtení a vytváření souborů ... 19
3.1. Načtení a otevření souboru ... 19
3.2. Psaní do souboru ... 21
3.3. Zavření souboru ... 22
4. Seznamy a smyčky ... 22
4.1. Tvoření seznamů a extrakce prvků ... 23
4.2. Práce s prvky ... 24
4.3. Tvorba smyček ... 24
4.4. Řetězec jako seznam ... 26
4.5. Rozsahy ... 27
5. Funkce ... 28
5.1. Definice funkce ... 28
5.2. Volání a ladění funkce ... 30
5.3 Užitečné poznatky ... 32
5.3.1. Enkapsulace ... 32
5.3.2. Funkce nemusí mít argument ... 32
5.3.3. Funkce nemusí vždy vracet hodnotu ... 33
5.3.4. Funkci lze zavolat s argumentem ... 33
5.3.5. Argument jako výchozí hodnota funkce ... 34
5.3.6. Testování funkcí ... 35
6. Podmínky ... 36
6.1. Podmínky, True a False ... 36
6.2. If a Else ... 37
6.3. Elif ... 39

6.4. While 40
6.5. Komplexní kód s podmínkami 40
6.6. True a False 41
7. Slovníky 42
7.1. Tvorba slovníku 45
7.2. Iterace ve slovníku 48
8. Balíky pro grafické výstupy a statistiku 50
8.1. Matplotlib 50
plt.subplot() 52
plt.subplots() 52
plt.axes() 53
8.2. Pandas a praktické ukázky 55
8.3. Vizualizace dat 58
8.5.1. Řádkový graf 59
8.5.2. Rozptýlený graf 60
8.5.3. Histogram 61
8.5.4. Seaborn 61
9. RegEx 67
9.1. Nejběžnější výrazy 68
9.2. Stavební kameny regulérních výrazů a příklady 68
9.3. Opakování 70
10. Literární a webové zdroje v češtině 72
10.1. Knihy o jazyku Python 72
10.2. Webové zdroje a online tutoriály 73
11. Literární a webové zdroje v angličtině 74
12. Programátorské projekty 76
12.1. Gantt chart 76
12.2. A comprehensive exploratory data analysis with 3 lines of code in Python 76
12.3. Příklady využití genomových dat pro analýzy 76

1. Úvod

1.1. O této příručce

Na internetu je v angličtině k dispozici již nepřeberné množství knih, manuálů a učebnic k programování pro biology. V českém jazyce však podobné texty ještě chybí, což vedlo ke vzniku této příručky. Tato příručka si klade za cíl stát se přehledem základů bioinformatického

programování v jazyce Python zejména pro studenty na všech úrovních studia. Postupně v ní budeme shromažďovat jak základní příkazy a úkony obecného programování v jazyce Python, tak i užitečné funkce, které pomohou především s organizací bioinformatických dat, jejich vizualizací a následným užitím v praxi. Samozřejmostí jsou užitečné odkazy pro další rozvíjení nabytých schopností. Bez následného opakování a co nejintenzivnější praxe se programování dobře naučit nedá. Proto přidáváme současně i tipy, jak si udržovat kontakt s programováním a nadále se v něm zlepšovat. Příručka se zaměřuje především na začátečníky, ale i zkušenější programátoři by v ní mohli nalézt něco nového a užitečného právě z oblasti bioinformatiky.

1.2. Co je programování a k čemu je dobré?

Moderní biologie se bez výpočetní techniky již dlouho neobejde. Tím nejsou myšleny laboratorní přístroje za miliony korun, ale (výkonný) počítač, potažmo notebook. Téměř každý vědecký článek nebo diplomová práce už dnes obsahuje grafy, srovnávací tabulky a statistické vyhodnocení. Biolog ale potřebuje znát i trochu něco jiného než klasický programátor, např. upravovat text (tzn. DNA či proteinových sekvencí), data a generovat z nich tabulky (pro vytváření grafů) či regulární výrazy (pro vyhledání výrazu v textu resp. v sekvenci, počet opakování sekvence znaků atd.).

Se znalostí programování lze jednak ušetřit velké množství práce a hlavně lze výrazně rozšířit repertoár našich možností, tj. analýz, které můžeme provádět. Úkony, které bychom prováděli bez bioinformatiky i několik dní nebo vůbec provést nemohli, můžeme zvládnout třeba za hodinu.

Základními stavebními kameny pro všechny programy představují příkazy, pro představu:

„Udělej tohle a potom tohle.“

„Pokud je tato podmínka splněna, udělej tohle; jinak udělej tohle.“

„Tohle opakuj tolikrát, kolikrát je uvedené.“

„Dělej tohle, dokud se tato podmínka nestane platnou.“

Postupně však příkazy můžeme kombinovat dle našich specifických potřeb. Důležité ale je začít něčím jednoduchým a postupně se dostávat hlouběji a hlavně se nebát. Na rozdíl od minulých dob stačí několik minut a vhodným dotazem na Google zjistíme, že podobný problém už řešil někdo před námi. Soutěžení zde není na místě, od slušných programátorů se vám vždy dostane ochotné pomoci a podpory.

1.3. Proč Python?

Pokud se zeptáme programátora, který programovací jazyk je nejlepší, nejspíš nedostanete jednoznačnou a stručnou odpověď, protože správný programátor má přehled o více jazycích a každý z nich může používat na něco jiného. Pro účely bioinformatiky je ale Python jasnou volbou z toho důvodu, že je pro biologa (neprogramátora) snadno čitelný, přehledný a pro začátek velmi vhodný. Navíc, pokud potřebujeme použít i jiný jazyk, díky znalosti jazyka Python nám jeho osvojení již nezabere tolik času. Protože vývojáři vědí, že Python používá spousta vědců, má taktéž výtečné zázemí a podporu v podobě tutoriálů a zejm. diskusních fór.

Podobnými jazyky co do jejich uplatnění v biologii a bioinformatice jsou pak Perl a R; Perl je sice hůře čitelný, ale se znalostí Pythonu by neměl představovat velký problém. Jazyk R je další jasná volba bioinformatika - R disponuje rozsáhlými statistickými knihovnami, ale i nesčetnými balíčky pro evoluční biologii, fylogenetiku, ekologii a další oblasti biologie. Proto mu je věnována naše další příručka Základy programování v jazyce R pro biology.

1.4. Instalace Pythonu a interaktivní prostředí

Python lze spustit v příkazové řádce, kterou má každý operační systém (dále jen „OS“) – tudíž jak Windows, tak Linux a Mac. Nejjednodušší způsob, jak si ověřit, zda máme ve svém počítači nainstalovaný Python, je spustit příkazovou řádku:

- Windows: Start → napsat „cmd“
- Mac: Applications → Utilities → Terminal
- Linux (KDE): Hlavní menu → Konsole
- Linux (GNOME): Super → Terminál

Do okna terminálu napíšeme „python3“, stiskneme enter a uvidíme, co se stane.

Pokud pracujeme na Windows 10 a Python není nainstalován, měli bychom být přesměrováni do Microsoft Store, kde lze dále pokračovat s instalací dle instrukcí. Jakmile Python nainstalujeme, už by se nám mělo v příkazové řádce terminálu objevit něco podobného:

```
C:\Users\Vaše_jméno>python Python 3.7.7 (tags/v3.7.7:d7c567b08f, Mar 10 2020, 11:52:54) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Jelikož je Ubuntu jednou z nejpoužívanějších distribucí Linuxu, budou níže uváděny příkazy přímo pro ni. Na Linuxu už Python většinou nainstalovaný bývá. V této příručce však

potřebujeme verzi Python 3.10, takže pokud se ve výpisu z terminálu objeví např. „Python 3.4“, je třeba aktualizovat systém. Instalaci pak spustíme tímto příkazem:

```
$ sudo apt-get install python3
```

(Pozn.: Znak „$”(string) se nepíše, v terminálu se objevuje automaticky)

Pro zjištění naší současné nainstalované verze Pythonu stačí zadat:

```
python –version
```

„>>>“ znamená spuštěný Python, s nímž už můžeme komunikovat. Pro začátek si můžeme ověřit, že může fungovat jako rychlá kalkulačka:

```
>>> 5 + 2
7
```

Příklady zde jsou uvedeny s mezerami pro naši větší přehlednost, ale Python v tomto případě mezery nerozlišuje a vše pochopí i bez nich.

Tab.1 Výčet základních aritmetických funkcí a jejich operátorů

Operátor	**Funkce**	**Příklad**
+	Sčítání	5 + 2
-	Odečítání	5 – 2
*	Násobení	5 * 2
/	Dělení se zbytkem	5 / 2
%	Zbytek	5 % 2
**	Umocnění	5 ** 2
//	Dělení beze zbytku	5 // 2

Dále zkusíme první základní příkaz, a tím je *print* s klasickým vzkazem „Hello world!“.

```
>>> print("Hello world!")
Hello World!
```

Fungoval by tento příkaz i s příklady?

```
>>> print(3+2)
```

Odpověď zní: Ano! print je funkce, která „vytiskne“ požadovaný výstup, ať už jde o řetězec, nebo cokoli jiného, co je poté na řetězec převedeno. Proto za příkaz patří závorky ().

Pro naši práci bude však pohodlnější a hlavně přehlednější používat aplikaci Jupyter notebook, kterému se budeme věnovat v další kapitole. Pro ukončení Pythonu a navrácení do příkazové řádky stačí napsat quit() a místo kliknutí na křížek je možné zavřít i příkazovou řádku příkazem exit.

1.5. Jupyter notebook

Nejjednodušším způsobem, jak pracovat s Jupyterem, je pomocí balíku Anaconda Navigator. Anaconda je open-source projekt pro ty, kteří potřebují pracovat s velkým objemem dat a je volně ke stažení zde: https://www.anaconda.com/products/individual.

Instrukce k instalaci jsou na výše uvedené stránce k dispozici pro každý OS:

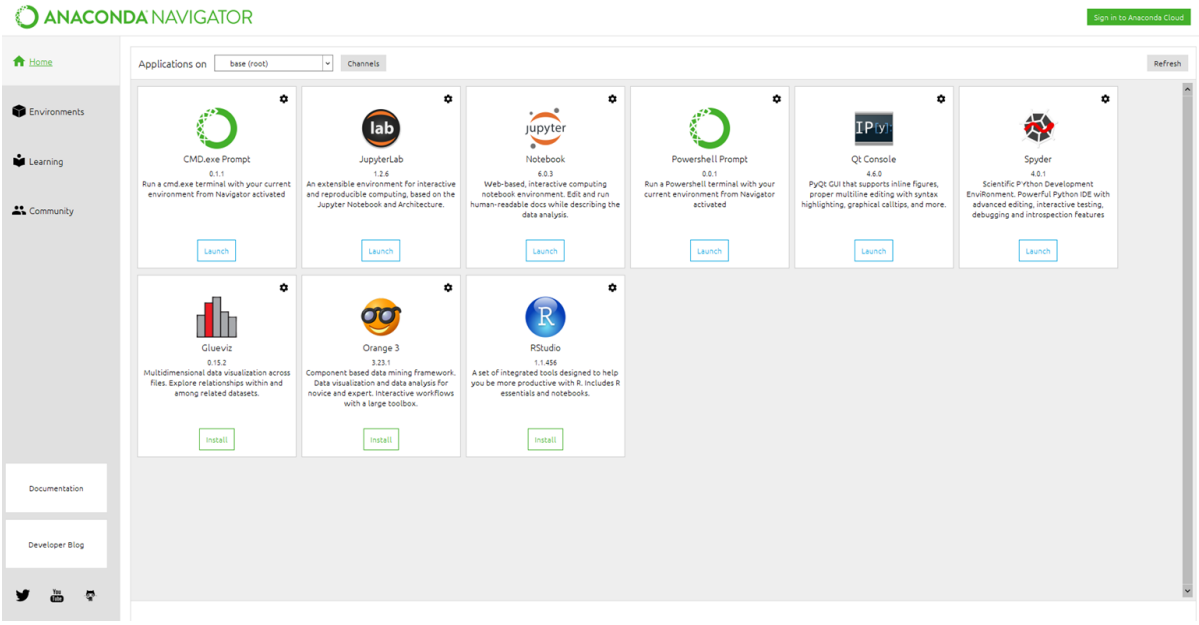
- Windows: např. https://medium.com/@GalarnykMichael/install-python-on-windows-anaconda-c63c7c3d1444
- Linux (Ubuntu 18.04): např. https://www.digitalocean.com/community/tutorials/how-to-install-the-anaconda-python-distribution-on-ubuntu-18-04

- MacOS: např. https://towardsdatascience.com/how-to-successfully-install-anaconda-on-a-mac-and-actually-get-it-to-work-53ce18025f97

Ukazatelem, že vše proběhlo správně a můžeme spustit Anacondu, je předpona (base), která se objeví v terminálu. Poté už stačí jen napsat do příkazové řádky anaconda navigator či přímo jupyter notebook a po chvíli se na obrazovce objeví požadované prostředí. Pro spuštění Jupyter notebooku z Anacondy (Obr. č. 1) stačí kliknout na „Launch“ a otevře se jako nové okno v internetovém prohlížeči, kde vidíme složky, které máme v počítači čili „Dashboard“. Pro otevření nového sešitu klikneme na „New“ a vybereme „Python 3“ (Obr. č. 2).

Nahoře v políčku názvu můžeme kliknout na „Untitled“ a sešit si přejmenovat dle potřeby, bude se ukládat ve formátu *.ipynb. Pro lepší přehlednost a práci je doporučeno dokumenty o více názvech ukládat s podtržítkem místo mezery, bez diakritiky a speciálních znaků, např. „jupyter_ntb_pokus1“.

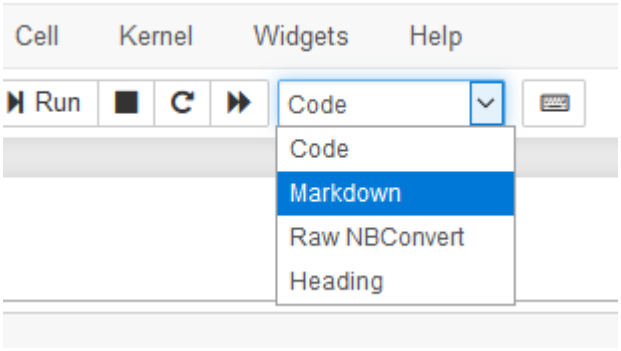
Řádek, neboli buňka, je vlevo označená zeleně, to znamená, že je aktivní a můžeme do ní psát kód. Pokud klikneme mimo buňku, tak zmodrá a bude neaktivní. Buňky jsou označené číslem v hranatých závorkách, což se nám bude hodit k odkazování se na jednotlivé buňky.



Obr. č. 1: Prostředí Anaconda Navigator

Obr. č. 2: Otevření nového sešitu v Jupyter Notebook

Prvním zde užitečným nástrojem je *Markdown*. Z buňky pro kód na něj přepneme jednoduše z voleb na liště (Obr. č. 3). Chová se vlastně podobně, jako kdybychom psali text ve Wordu a slouží k vložení poznámek nebo nadpisů a odrážkami a odlišnými styly písma.



Obr. č. 3: Změna formátu buňky z kódu na Markdown

Tab. 2 Přehled nejzákladnějších úprav textu:

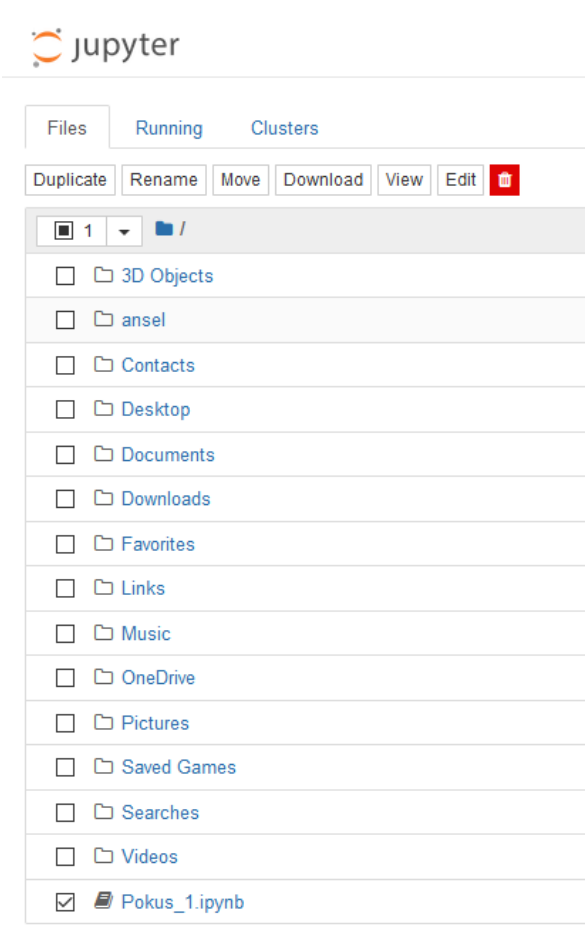
Prvek	**Markdown Syntax**
Nadpis	# Nadpis1 ## Nadpis2 ### Nadpis3 atd.
Tučné	**tučné**
Kurzíva	*kurzíva*
Citace	> citace
Číslovaný seznam	1. První položka 2. Druhá položka 3. Třetí položka atd.
Odrážkový seznam	- První položka - Druhá položka - Třetí položka atd.
Vložení kódu	'kód'
Oddělovač	---
Odkaz	[odkaz](https://www.google.com)
Obrázek	![popis](umístění\fotka.jpg)

Více např. zde: https://www.markdownguide.org/

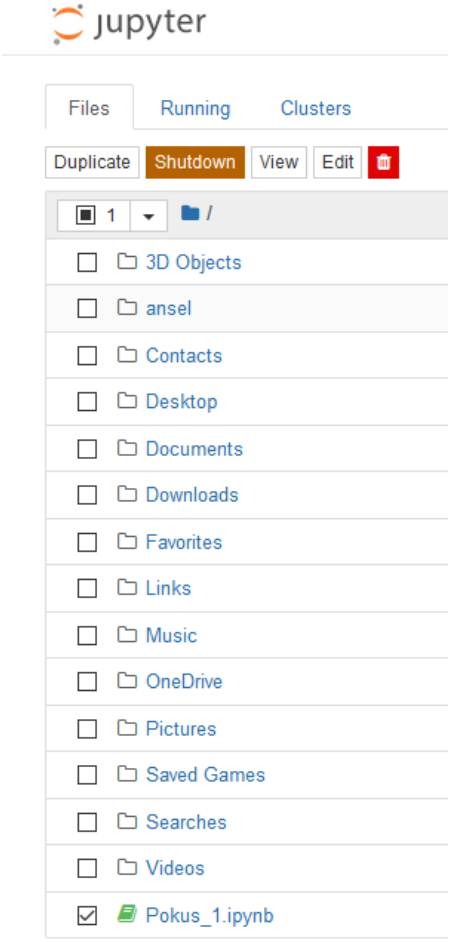
Po napsání kódu či markdownu buňku spustíme tlačítkem „Run“ nebo klávesovou zkratku „Shift+Enter“ a ukáže se nám výstup (output). Co se týče kódu, pracujeme stejně, jako jsme si již ukázali v prostředí terminálu, jen s rozdílem, že si jej nemusíme, např. v případě delšího kódu, psát někam vedle a pak ho teprve spustit v Pythonu, nýbrž rovnou. Současně se nám i barevně mění písmo, což je rychlejší pro orientaci a uživatelsky příjemnější zároveň. Můžeme tedy zkusit opět nějaké jednoduché výpočty a příkazy.

Soubor se ukládá automaticky do kořenové (root) složky. Pokud si najedeme na soubor v Dashboardu, můžeme jej vlevo zaškrtnout a ukáže se nám nabídka, co s ním udělat –

duplikovat, přejmenovat, přemístit, stáhnout, načíst, změnit či smazat (viz Obr. č. 4). Pozor, uvedené akce se sešitem nelze udělat, pokud ho nejdříve nezavřeme. To provedeme tlačítkem „Shutdown“ (Obr. č. 5).



Obr. č. 4 – Editace sešitu



Obr. č. 5 – Zavření sešitu

Sešit lze stáhnout i v jiném formátu, než výchozím *.ipynb, a to pomocí funkce File → Download as → a pro další možnosti nejlépe ve fomátu Python tj. *.py. Dále lze sešit editovat, pokud potřebujete např. buňky rozdělit, opětovně spojit či přidat další. K těmto úkonům existují i užitečné klávesové zkratky. Pokud klikneme vedle buňky a ta zmodrá, zřejmě nejčastěji budeme používat ty uvedené v Tab. 3.

Tab. 3 Klávesové zkratky pro práci s buňkami

Zkratka	**Funkce**
A	Přidá buňku nad aktuální
B	Přidá buňku pod aktuální

M	Změní aktuální buňku na Markdown
D + D (zmáčknout dvakrát za sebou)	Smaže aktuální buňku

Další klávesové zkratky najdeme pod Help → Keyboard Shortcuts.

1.6. Google Colab

Další uživatelsky příjemnou možností jak programovat v jazyce Python je služba týmu Google Research zvaná Colaboratory, zkráceně Colab. Služba Colab umožňuje napsat a spustit libovolný kód v Pythonu v internetovém prohlížeči, což je užitečné především pro účely strojového učení, analýzy dat a vzdělávání. Z technického hlediska je Colab hostovaná služba pro sešity nám již známého nástroje Jupyter, která nevyžaduje žádnou instalaci a nabízí bezplatný přístup k výpočetním zdrojům, včetně GPU. Vzhledem k tomu, že je Colab zdarma, jeho zdroje nejsou zaručeny a neomezené a limity využití tak někdy mohou kolísat. Pro uživatele, kteří chtějí mít spolehlivější přístup k lepším zdrojů, je určena služba Colab Pro nebo Colab Pro+.

Další detaily jsou k dispozici na následující webové adrese
https://research.google.com/colaboratory/intl/cs/faq.html
Vlastní pracovní prostředí v Colab lze vytvořit na webové adrese
https://colab.research.google.com/

2. Práce s textem

Při otevření jakékoli knihy o programování najdeme kódy, které pracují s čísly. Biologové zabývající se sekvencemi nukleových kyselin nebo proteinů potřebují pracovat spíše s textem, což je také primární záměr této příručky (bez čísel se samozřejmě také neobejdeme). Sekvence DNA/RNA či proteinů navíc představuje ideální „substrát“ pro aplikaci výpočetních metod. Programy, které budeme vytvářet, mohou dohromady vytvářet funkční celek a jako pomyslný potrubím (*pipeline*) jimi mohou postupně „protékat“ naše data. Pojem *pipeline* je v bioinformatice často používán a označuje kombinaci různých programů, tj. nejen našich vlastních kódů ale i převzatých z literatury nebo diskusních fór. V rámci *pipeline* je důležité, aby kód **rozuměl** výstupu jiného předchozího programu (čili *parsing*), a/nebo **provedl** výstup v takovém formátu, aby na něm jiný program mohl stavět. Obě varianty vyžadují práci s textem. Pro začátek si tedy představíme první důležitý pojem z programovacího žargonu – *string* neboli

řetězec. Řetězec je datový typ (druh hodnot), který obsahuje text, např. slovo, větu nebo různě dlouhou sekvenci nukleové kyseliny.

2.1. Funkce print

O funkci print jsme se zmínili již v první kapitole, teď se na ni však podíváme o něco zevrubněji:

```
print("Hello world!")
```

Celému řádku se říká *příkaz*. print je název funkce – ty říkají Pythonu, co má udělat, v tomto případě chceme něco zobrazit (doslova „vytisknout“). Za funkcí vždy následují závorky (existuje sice pár výjimek, ale těmi se teď nebudeme zatěžovat). Text v závorkách nazýváme *argument* (podobně jako argument funkce v matematice) a Pythonu specifikuje, co přesně po něm chceme.

Podobně jako je přímá řeč uvedena uvozovkami, stejně je tomu i u řetězce. Python je takto schopen rozlišit mezi instrukcemi (např. název funkce) a daty (to, co chceme vytisknout). Můžeme použít klasické „dvojité“ uvozovky tj. “ “ nebo „jednoduché“ anglické tj. ‘’, Python si poradí s oběma verzemi, kombinace různých uvozovek najednou už ale způsobí chybu. Jupyter notebook velmi chytře uvozovky doplňuje, aby se nic takového nestalo.

Zkusme si následující příkazy:

```
print("Hello world!")
print('Hello world!')
print('Hello world!")
```

Další užitečnou věcí jsou **komentáře**, které se hodí zejména po nás, pokud si vedle kódu chceme do stejné buňky poznamenat, co např. daný kód dělá.

```
# toto je komentář, Python jej bude ignorovat
print("Komentáře se vždycky hodí!")
```

Co když potřebujeme v textu přímou řeč?

```
print('Zeptala se: "Jak se máš?"')
```

Co když potřebujeme text rozdělit na několik řádků?

```
print("Hello
World!“)
```

Takto však Python nefunguje a vrátí nám chybu:

```
SyntaxError: EOL while scanning string literal
```

„EOL“ znamená „End of Line“ a *string literal* řetězec v uvozovkách, což přeloženo do češtiny znamená zhruba: „Začal jsem číst řetězec v uvozovkách a dostal jsem se na konec řádku, aniž bych narazil na druhé uvozovky.“ Python nevěděl, jestli byl nový řádek součást řetězce (což jsme chtěli), nebo část kódu (což si tak nakonec vyložil). Pokud chceme něco napsat na druhý řádek, použijeme zpětné lomítko s písmenem n, tj \n:

```
# Takhle dostaneme další řádek
print("Hello\nworld!")
```

Všimněme si, že mezi slovy a znakem \n není potřeba mezera.

2.2. Chybová hlášení

O jednom druhu chyby (error) jsme se již zmínili a i když se může zdát, že je na ně příliš brzy tak není. T**éměř žádný program nefunguje hned napoprvé**. Proto bychom se s chybovými hlášeními měli seznámit co nejdříve a naučit se s nimi žít a pracovat. Jednou z nejčastějších chyb je z**apomínání uvozovek** – může to znít banálně, ale i to se stává, navíc velmi snadno.

```
print(Hello world)
```

Python vrátí chybu:

```
  File "<ipython-input-1-50b4ae29d403>", line 1
    print(Hello world)
                ^
SyntaxError: invalid syntax
```

Z výstupu můžeme vyčíst, že chyba nastala na prvním řádku a Python její lokaci odhaduje před uzavřením závorky, na což není radno se přespříliš spoléhat, protože záleží na typu chyby a odhad tudíž nemusí být vždy správný. **SyntaxError** znamená, že Python kódu nerozumí, zkrátka jako kdyby na vás někdo spustil čínsky.

- **Chyba v pravopisu** – Co se stane, pokud nenapíšeme správně jméno funkce?

```
prin("Hello world")
```

Dostaneme trochu jinou chybu, a to **NameError** a zpráva vypadá o něco srozumitelněji:

```
NameError                 Traceback (most recent call last)
<ipython-input-2-919ef47b99be> in <module>
----> 1 prin("Hello world")

NameError: name 'prin' is not defined
```

Zde Python neodkazuje na řádek, kde chybu našel, ale rovnou ho celý ukáže společně se slovem, kterému nerozumí, takže náprava je celkem snadná.

2.3. Proměnné

Ukázali jsme si funkci print, nyní je načase představit další důležitou a užitečnou věc – ukládání řetězce do *proměnné*.

```
my_dna = "ATGCGTA"
```

Proměnná se zde jmenuje **my_dna** a obsahuje sekvenci DNA (řetězec) **"ATGCGTA".** Tento řetězec *přiřadili* do proměnné. Teď můžeme použít název proměnné místo toho, abychom stále dokola vypisovali celý řetězec. Výhodu přiřazení proměnné oceníme, jakmile budeme pracovat s delšími sekvencemi. Prozatím zkusme proměnnou vytisknout funkcí print:

```
my_dna = "ATGCGTA"
print(my_dna)
```

Všimněme si, že tady už nejsou potřeba uvozovky – ty jsou totiž už součástí proměnné. Navíc lze řetězec uložit do proměnné samostatně, pokud se po zmáčknutí Shift+Enter nic nestane, je to dobrá zpráva a proměnná se uložila v pořádku. Na dalším řádku můžeme dál provádět, co je třeba. Proměnnou můžeme uložit pod jakýmkoliv názvem chceme, ale měli bychom mít na paměti, že by název měl naznačovat, co proměnná obsahuje. Pokud bychom řetězec se sekvencí DNA uložili do proměnné s názvem mandarinka, asi by nám to nic neřeklo, zejména s odstupem několika týdnů či měsíců. Dále je důležité nezapomenout, že názvy proměnných jsou tzv. *case-sensitive*, což znamená, že např. názvy my_dna, MY_DNA, My_DNA a My_Dna jsou různé proměnné.

2.4. Manipulace s řetězci

Stejně jako pomocí znaménka + sčítáme čísla, můžeme spojovat i řetězce:

```
my_dna = "ATTA"+"GCGG"
print(my_dna)
```

Spojovat můžeme i řetězce s proměnnými:

```
vlakno = "AATA"
my_dna = vlakno + "ATGC"
print(my_dna)
```

Odborně se tomuto říká *concatenating* neboli *zřetězení (konkatenace)* a ještě lépe česky také *sloučení*.

Jako další užitečnou funkci si představíme **len** (zkráceně od *length*). Stejně jako u print jí stačí pouze jeden argument, což je řetězec, ačkoliv se chová trochu jinak. Používá se pro zjištění počtu znaků jak v číselných řadách, tak v řetězcích a výsledkem je tzv. *návratová hodnota* (*return value*), která se dá přiřadit do proměnné.

```
len("ATCGA")
```

```
delka_dna = len("ATCGA")
print(delka_dna)
```

Návratovou hodnotou není řetězec, nýbrž číslo přičemž **Python pracuje jinak s čísly a jinak s řetězci**. To si můžeme ukázat, když se pokusíme sloučit řetězec s nějakým číslem:

```
# uložení sekvence do proměnné
my_dna = "ATCGA"
# spočtení délky sekvence a uložení do proměnné
delka_dna = len(my_dna)
# chceme vidět výstup, který bude informovat o délce sekvence
print("Délka DNA sekvence je " + delka_dna)
```

Dostaneme chybovou hlášku:

```
TypeError                          Traceback (most recent call last)
<ipython-input-1-832453834a0c> in <module>
     4 delka_dna = len(my_dna)
     5 # chceme vidět výstup, který bude informovat o délce sekvence
----> 6 print("Délka DNA sekvence je " + delka_dna)
```

```
TypeError: can only concatenate str (not "int") to str
```

Hláška je krátká, ale výstižná – nelze sloučit řetězec (str) a číslo (int), což Python neumí, protože jde o rozdílné typy dat. Naštěstí má Python zabudovanou funkci str, která umí převést hodnotu na řetězec, aby byla následně ve výstupu.

```
my_dna = "ATCGA"
delka_dna = len(my_dna)
print("Délka DNA sekvence je " + str(delka_dna))
```

Tento zápis již už bude fungovat. Mezera před uvozovkou je pro větší přehlednost, aby nebyl text nahloučený na sobě. Všimněme si, že tu máme funkci uvnitř funkce a naše prohlášení je uzavřeno dvěma závorkami. Pro pořádek ještě shrneme, že str je funkce, která potřebuje jeden *argument* (číslo) a *vrací hodnotu* (řetězec) a zobrazuje tak číslo.

Dalším užitečným nástrojem jsou vestavěné **metody**. Chovají se jako funkce, ale **používají se pouze k typům dat, ke kterým patří**. Mluvit budeme o metodách lower a upper, které patří k řetězcům. Hodí se například pokud máme sekvenci DNA a chceme změnit všechna písmena na velká, či naopak na malá (obojí má v bioinformatice svůj specifický význam, např. u soft-maskování repetitivních sekvencí, viz níže):

```
my_dna = "atcga"
print(my_dna.upper())
```

Všimněme si rozdílu mezi použitím metody a funkce - zatímco u funkce je nejprve třeba napsat její název a poté závorky, u metody použijeme proměnnou, za ní tečku a až posléze její název a případně argument do závorek. V tomto případě argument nepotřebujeme, takže závorky zůstávají prázdné. Je důležité říct, že samotná metoda upper nemění vlastní proměnnou, ale její kopii. Tudíž je uložení do proměnné důležité, a protože tyto metody patří k řetězcům, s čísly nám nebudou fungovat.

Dále se velmi hodí metoda replace. Název už napovídá k čemu – vezme dva argumenty (řetězce) a vrátí kopii proměnné, kde bude výskyt prvního řetězce nahrazen druhým. Trochu kostrbatá definice, takže ukažme si příklad:

```
protein = "vlspadktnv"
# vyměníme valin za tyrosin
print(protein.replace("v", "y"))
# rozhodně lze nahradit i více, než jen jedno písmeno
```

```
print(protein.replace("vls", "ymt"))
# původní proměnná není zasažena
print(protein)
```

Co když máme dlouhý řetězec, ale chceme z něj vybrat jen část (tzv. *substring*)?

```
protein = "vlspadktnv"
# náhled aminokyselin na místě 3 až 5
print(protein[3:5])
# první pozice je nula, nikoliv jednička
print(protein[0:6])
# pokud použijeme pozici, která už je daleko za koncem, je to úplně jedno
print(protein[0:60])
```

Dvě důležité věci: **Python počítá od nuly**, takže zde je 3. místo pro nás na pozici 4. **Znak počáteční pozice se započítá, znak poslední pozice se vyloučí.** Příkaz `protein[3:5]` nám ukáže vše od čtvrtého znaku a končí před znakem pátým (takže pouze čtvrtý a pátý).

Běžně se v biologii setkáme s tím, že potřebujeme spočítat, kolikrát se opakuje sekvence DNA nebo proteinu. Vhodná metoda, kterou k tomu můžeme využít, se jmenuje `count`. Vezme jeden argument (string) a vrací (v čísle!), kolikrát ho v proměnné najde. Vynechání řádku je pro přehlednost, Python jej ignoruje:

```
protein = "vlspadktnv"
# proměnné pro výpočet aminokyselin
valin_count = protein.count('v')
lsp_count = protein.count('lsp')
tryptofan_count = protein.count('w')

# výstup pro výpočet
print("valin: " + str(valin_count))
print("lsp: " + str(lsp_count))
print("tryptofan: " + str(tryptofan_count))
```

Rádi bychom ještě věděli jejich umístění! K tomu nám pomůže metoda `find` a jejím výstupem bude číslo, na které pozici se prvně objeví (tzv. *index*). Nezapomeňme, že Python počítá (indexuje) od nuly:

```
protein = "vlspadktnv"
print(str(protein.find('p')))
print(str(protein.find('kt')))
print(str(protein.find('w')))
```

Ve výstupu u neexistující části řetězce dostaneme číslo -1. Obě metody count a find jsou limitované a umí hledat pouze přesné části. Všechny metody, které jsme zde probrali – replace, count a find – vyžadují ke správnému fungování alespoň dva řetězce, takže pozor na pořadí.

2.5. Klíčová slova

Klíčová slova jsou vyhrazená v rámci syntaxe Pythonu a programátor je tak již nemůže použít jako jména proměnných. Současná verze Python 3.10.2 si rezervuje 35 klíčových slov. Jejich platný seznam získáme zadáním následujících příkazů:

```
>>> import keyword
>>> keyword.kwlist
```

Výsledkem těchto příkazů je následující seznam:

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

Tab. 4. Seznam a základní charakteristika klíčových slov

Klíčová slova	**Skupina**
True, False, None	Reprezentaci logické a prázdné hodnoty
try, except, raise, finally, assert	Zpracování výjimek
async, await	Pro asynchronní programování
and, or, not, in, is	Operátory
for, while, break, continue	Pro práci s cykly
if, elif, else	Podmínky
return, yield	Pro navrácení hodnot
import, from	Pro Import
del, global, nonlocal	Pro manipulaci s proměnnými
def, class, with, as, pass, lambda	Pro tvorbu funkcí

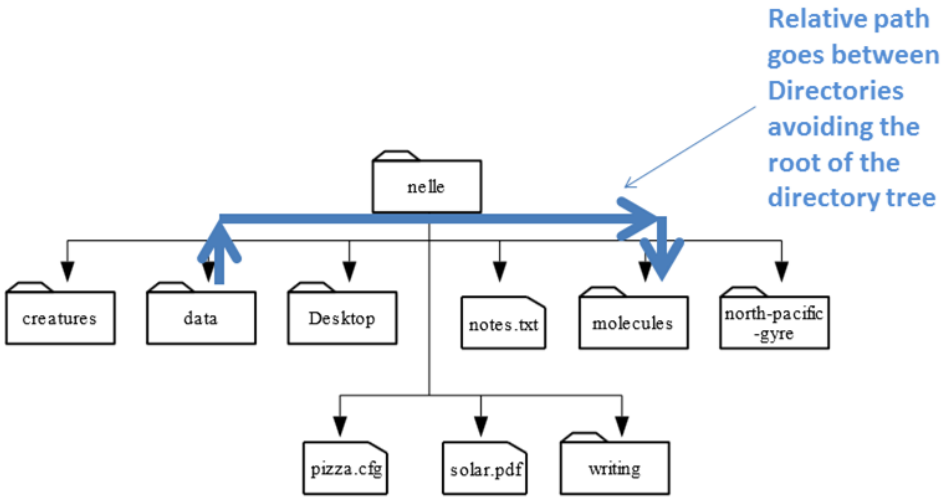
3. Čtení a vytváření souborů

Nejen biologové pracují s daty uloženými v souborech, proto se potřebujeme naučit, jak data extrahovat do našich programů a naopak. V předchozí kapitole jsme si do Pythonu vkládali krátké sekvence, ale v praxi budou tato data mnohem větší. Naštěstí jsou sekvence běžně dostupné v textových souborech (většinou FASTA formát), které ovšem nejsou pro člověka snadno čitelné.

Doposud jsme ukládali řetězce nebo čísla do proměnných a pro použití stačilo použít název proměnné. Při práci se soubory se ale často užívá *metod*, které nám umožní dělat spoustu užitečných věcí.

3.1. Načtení a otevření souboru

Zkusíme si vytvořit vlastní malý soubor. Do textového editoru vložíme náhodnou sekvenci DNA a uložíme. Poté potřebujeme soubor otevřít v Jupyteru, abychom s ním mohli pracovat. Prvně je ale nutné znát absolutní a relativní cestu k souboru. Absolutní cestou určíme polohu souboru na disku bez ohledu na aktuální adresář (složku). Relativní cesta je určení polohy souboru vzhledem k zadanému adresáři (složce) a vyskytují se v ní znaky „..\“, které ukazují na rodičovskou složku (v případě Linuxu).



Obr. č. 6 – Struktura souborů (zdroj: arc.leeds.ac.uk)

Předpokládejme, že máme v počítači takovéto složky a soubory. Budeme se chtít zaměřit na soubor „notes.txt“.

Pokud pracujeme ve Windows, absolutní cesta by vypadala takto: „C:\nelle\notes.txt“. Relativní cesta, např. ze složky „molecules“ takto: „..\notes.txt“. V Linuxu absolutní pak: „/nelle/notes.txt“ a relativní: „../notes.txt“. **Z praktického hlediska je však nejjednodušší a velmi žádoucí si vytvořit Jupyter notebook přímo ve složce, kde máme umístěné soubory, se kterými potřebujeme pracovat.**

Teď už víme, jak zadat soubor do Jupyteru, aby se nám správně načetl a my s ním mohli pracovat. Použijeme následující příkaz:

```
muj_soubor = open("cesta k souboru")
```

Soubor máme nyní uložen v proměnné muj_soubor. Prvně potřebujeme vědět, co v souboru vlastně je a k tomu použijeme metodu read. Obejde se bez argumentů a návratovou hodnotou je řetězec, který následně můžeme uložit do proměnné a poté se k ní chovat jako ke každému jinému řetězci, např.:

```
muj_soubor = open("dna.txt")
obsah_souboru = muj_soubor.read()
print(obsah_souboru)
```

Na začátku je velmi jednoduché zaměnit *objekt* souboru, *název* souboru a *obsah* souboru. Podívejme se na následující kód:

```
nazev_souboru = "dna.txt"
muj_soubor = open(nazev_souboru)
obsah_souboru = muj_soubor.read()
```

Na prvním řádku ukládáme řetězec *dna.txt* do proměnné nazev_souboru. Na druhém používáme proměnnou nazev_souboru jako argument pro funkci open a objekt souboru ukládáme do proměnné muj_soubor. Nakonec voláme metodu read na proměnnou muj_soubor a výsledný řetězec ukládáme do proměnné obsah_souboru.

Měli bychom být schopni rozpoznat, že máme tři různé proměnné, v nichž jsou uložené tři odlišné věci. nazev_souboru je řetězec obsahující název souboru na disku. muj_soubor je objekt a zobrazuje samotný soubor. obsah_souboru je řetězec s textem souboru. Řekli jsme si, že názvy proměnných jsou zcela libovolné, ale názvy souborů nikoliv – musí korespondovat s názvem souboru na disku.

Častou chybou je použití metody read na nesprávnou věc. Připomeňme si, že tato metoda se dá použít pouze na objekt souboru. Pokud se metodu pokusíme použít na název souboru:

```
nazev_souboru = "dna.txt"
obsah_souboru = nazev_souboru.read()
```

dostaneme AttributeError – Python si postěžuje, že k metodě read řetězce nepatří:

```
AttributeError: 'str' object has no attribute 'read'
```

Dále se může stát, že chceme otevřít obsah souboru, ale otevřeme jeho objekt:

```
nazev_souboru = "dna.txt"
muj_soubor = open(nazev_souboru)
print(muj_soubor)
```

Python ovšem nevrátí chybu, nýbrž podivně vypadající hlášku. Něco jako:

```
<open file 'dna.txt', mode 'r' at 0x7fc5ff7784b0>
```

Právě se nám povedlo zobrazit objekt souboru, nikoliv obsah.

3.2. Psaní do souboru

V dosud uváděných příkladech jsme si zobrazovali výstupy rovnou na dalším řádku. To se hodí pro úvod, když se učíme něco nového a zkoušíme, co vše je možné. U souboru s obsáhlejším textem by tento postup bylo poněkud těžkopádný a nepraktický. Zahltili bychom si disk nebo bychom zrovna potřebovali vyjmout z celého textu jen pár informací. Hlavně pokud ukončíme sezení, tak je celý výstup z programu, který mohl běžet třeba i několik hodin, jednoduše pryč.

V předchozí kapitole jsme si ukázali, jak otevřít soubor a přečíst si jeho obsah. Můžeme si také otevřít soubor a něco do něj zapsat, na což použijeme funkci open, ale v trochu jiném stylu. Funkce open bude jako první argument a poté přidáme druhý argument určující **režim**, ve kterém chceme soubor otevřít. Může to být „r“ („read“, čtení), „w“ („write“, zápis) nebo „a“ („appending“, přídavek). Pokud druhý argument nepoužijeme, Python má ve výchozím nastavení čtení („r“). Rozdíl mezi „w“ a „a“ není velký, ale je důležité si ho ujasnit. Pokud otevřeme *existující* soubor v režimu „w“, přepíšeme obsah, který již v souboru je, zatímco za použití argumentu „a“ do souboru připisujeme další data bez přemazání obsahu. Bez existujícího souboru se tyto dva argumenty budou chovat stejně – umožní zápis do nového souboru.

Funkce write pracuje trochu jako print, potřebuje jeden argument, ale místo výstupu se řetězec zapíše do souboru.

```
muj_soubor = open("pokus.txt", "w")
muj_soubor.write("Hello world")
```

Jelikož výstup zapisujeme do souboru, na dalším řádku se nám nic neukáže. Pro ověření úspěšnosti bychom si soubor museli otevřít.

Stejně jako u funkce print můžeme i s write použít jako argument jakýkoli řetězec i jakoukoli metodu nebo funkci, která řetězec vrací. Všechny následující argumenty lze použít:

```
#vypsat "abcdef"
muj_soubor.write("abc" + "def")
#vypsat "8"
muj_soubor.write(str(len("AGTCGTAG")))
#vypsat "TGTC"
muj_soubor.write("AGTC".replace("A", "T"))
#vypsat "agtc"
muj_soubor.write("AGTC".lower())
#vypsat obsah moje_promenna
muj_soubor.write(moje_promenna)
```

3.3. Zavření souboru

Opakem funkce open je metoda close. Po dokončení všech potřebných akcí bychom měli soubor zavřít, abychom se vyhnuli chybám a nežádoucí situaci s více otevřenými soubory najednou. Metoda close je jiná v tom, že nepotřebuje žádné argumenty a nic nevrací:

```
muj_soubor = open("pokus.txt", "w")
muj_soubor.write("Hello world")
muj_soubor.close()
```

4. Seznamy a smyčky

S tím, co jsme si zatím v Pythonu ukázali, bychom se v biologii příliš daleko nedostali. Často se stává, že se musíme probrat velkým množstvím dat, která je potřeba zpracovat obdobným způsobem. Psát kód pro všechno zvlášť je zbytečná práce. V této kapitole si tedy ukážeme základy, které lze implementovat do programů, abychom si práci ulehčili.

Víme, že existují různé typy dat (řetězce, čísla a soubory) obsahující nějakou informaci. Pro uložení více druhů informací (např. tři různé sekvence DNA), jsme jednoduše použili tři různé proměnné:

```
seq_1 = "ATCGATGCGCTATTGCTA"
seq_2 = "agcttgcatgaacgccta"
seq_3 = "ACTGAC-ACGT-ACGTA----CATGT"
```

Jenže tři sekvence jsou málo a navíc tady dopředu víme, kolik jich máme. Kdybychom to samé měli provést se třemi stovkami nebo dokonce třemi tisíci sekvencí a drtivou většinu kódu pořád ukládali do proměnných, za chvíli bychom se v nich nevyznali. Aby si náš program s tak velkým obsahem dat rozuměl a byl ho schopný zpracovat, budeme potřebovat nový způsob ukládání dat – *seznam (list)*.

Také jsme se setkali s programy, které řádek po řádku přesně udávají, co se bude dít, což je pro začátek nejlepší, protože přesně vidíme postup. Opakování řádků pro ten samý úkon je však zbytečně zdlouhavé, což Pythonu ubírá na eleganci. Opakující se sekvence s různými modifikacemi se vyskytují velice často a Python je umí řešit jednoduše – *smyčkami (loops)*.

4.1. Tvoření seznamů a extrakce prvků

Pro vytvoření seznamu potřebujeme do hranatých závorek vložit řetězce nebo čísla oddělená čárkou:

```
hadi = ["Coronella austriaca", "Vipera berus", "Natrix natrix"]
```

Cokoliv uvnitř seznamu se nazývá prvek. Pokud chceme ze seznamu nějaký prvek vytáhnout, použijeme název proměnné, pod kterou máme seznam uložený a do hranatých závorek ještě index, tzn. pořadí umístění prvku (Python počítá od nuly!):

```
hadi = ["Coronella austriaca", "Vipera berus", "Natrix natrix"]
print(hadi[0])
```

Totéž lze samozřejmě provést i obráceně – víme, který prvek chceme, ale nevíme, jaké má umístění, k tomu nám pomůže metoda index:

```
hadi = ["Coronella austriaca", "Vipera berus", "Natrix natrix"]
zmije_index = hadi.index("Vipera berus")
print(zmije_index)
```

Co kdybychom chtěli více než jeden prvek? Můžeme zadat tzv. „start“ a „stop“ pozici oddělenou dvojtečkou a uzavřenou v hranatých závorkách:

```
taxony = ["kmen", "trida", "rad", "celed", "rod", "druh"]
nizsi_taxony = taxony[3:6]
print(nizsi_taxony)
```

Podobné úkony jsme prováděli s řetězci ve druhé kapitole – první číslo prvek **zahrnuje**, druhé ho **vylučuje**. Pracovali jsme se řetězci prakticky jako se seznamy.

4.2. Práce s prvky

Pro přidání prvku do seznamu využijeme nám už známou metodu append:

```
hadi = ["Coronella austriaca", "Vipera berus", "Natrix natrix"]
hadi.append("Zamenis longissimus")
```

Tato metoda změní celou proměnnou a stejně jako u řetězců si můžeme počet prvků jednoduše ověřit funkcí len:

```
len(hadi)
```

Seznamy můžeme stejně tak i spojovat:

```
hadi = ["Coronella austriaca", "Vipera berus", "Natrix natrix"]
jesteri = ["Lacerta viridis", "Lacerta agilis"]
plazi = hadi + jesteri

print(str(len(hadi)) + " hadi")
print(str(len(jesteri)) + " jesteri")
print(str(len(plazi)) + " plazu")
```

Metody reverse a sort mění proměnnou, v tomto případě položky v seznamu:

```
taxony = ["kmen", "trida", "rad", "celed", "rod", "druh"]
print("puvodne: " + str(taxony))
taxony.reverse()
print("po prevraceni: " + str(taxony))
taxony.sort()
print("po serazeni: " + str(taxony))
```

Ve výchozím nastavení Pythonu seřadí metoda sort řetězce v abecedním pořadí a čísla vzestupně.

4.3. Tvorba smyček

Představme si, že bychom chtěli vzít náš seznam hadů a následně vidět výstup pro každý prvek zvlášť, např. takto:

```
Coronella austriaca je had
Vipera berus je had
Natrix natrix je had
```

Jedním způsobem, kterým lze provést, je tento:

```
print(hadi[0] + " je had")
```

```
print(hadi[1] + " je had")
print(hadi[2] + " je had")
```

Jenže tento způsob je zbytečně dlouhý, opakující se (tedy náchylný k chybám) a navíc musíme vědět, kolik položek v seznamu je. Python však zná způsob, kterým řekneme: “Vypiš prvky v seznamu jak jdou po sobě a za každý napiš ‚je had‘.“:

```
For had in hadi:
   print(had + " je had")
```

Tato smyčka má na prvním řádku `for x in y`, kde `y` je název seznamu, se kterým chceme pracovat a `x` název proměnné pro prvek, který bude opakovat. Můžeme si jej pojmenovat jak chceme, ale opět musíme mít na paměti, co má název vyjadřovat. V tomto případě se proměnná chová trochu jinak. Za normálních okolností bychom do ní něco uložili a hodnota by se nezměnila, dokud bychom ji nezměnili sami. Tady hodnotu nenastavujeme, automaticky se přiřadí ke každému prvku ze seznamu a mění se s opakováním smyčky. Dalším důležitým poznatkem je, že proměnná `x` zde existuje pouze **uvnitř smyčky**. To znamená, že se vytvoří na počátku smyčky a po ukončení zmizí. Navždy.

První řádek končí dvojtečkou a všechny řádky (v tomto případě jeden) pod ním se automaticky odsadí. Poté můžeme psát dál, ale vše musí mít stejné odsazení. Soubor odsazených řádků v kódu se často nazývá *blok*. Vše ve stejném bloku patří k jednomu seznamu a můžeme do něj zařadit veškeré funkce a metody, které už známe s jednou výjimkou: **uvnitř smyčky nelze seznam měnit**.

Zkusme něco obsáhlejšího:

```
hadi = ["Coronella austriaca", "Vipera berus", "Natrix natrix"]
for had in hadi:
   delka_nazvu = len(had)
   prvni_pismeno = had[0]
   print(had + " je had. Jméno začíná písmenem " + prvni_pismeno)
   print("Jméno se skládá z " + str(delka_nazvu) + " pismen.")
```

Tělo smyčky se skládá ze čtyř řádků, z čehož ve dvou je funkce `print`, tudíž s každou smyčkou dostaneme dva řádky výstupu.

S novými možnostmi přicházejí ale i nové problémy. V tomto případě se často můžeme potýkat s chybou v odsazení – `IndentationError`.

```
IndentationError: unexpected indent
```

Řešení je jednoduché – pozorně si projdeme svůj kód a zkontrolujeme, že všechny řádky v bloku mají stejné odsazení. Také dáme pozor, abychom odsazovali buď pomocí tabelátoru (klávesa Tab) nebo mezerníku, nikdy ne kombinací obojího.

4.4. Řetězec jako seznam

Ve druhé kapitole jsme si ověřili, že řetězec může fungovat jako seznam – získávali jsme z něj znaky či soubor znaků (v podstatě řetězce v řetězci). Můžeme k tomu přistoupit i naopak a pracovat s řetězcem jako se seznamem, tudíž použít smyčky i na něj.

```
jmeno = "barbora"
for pismeno in jmeno:
   print("Mame pismenko " + pismeno)
```

```
Mame pismenko b
Mame pismenko a
Mame pismenko r
Mame pismenko b
Mame pismenko o
Mame pismenko r
Mame pismenko a
```

Výstupem jsou jednotlivá písmena. Proces, kdy se opakují instrukce pro každý prvek v seznamu (či znak v řetězci), se nazývá *iterace*.

Doposud jsme si všechny seznamy vytvářeli sami. Python však disponuje funkcemi a metodami, jejichž výstupem je právě seznam. Jedna z takových se jmenuje split a používá se na řetězce. Potřebuje jeden argument, kterému se říká *delimiter* (česky „oddělovač“), a ten rozdělí řetězec vždy v daném místě a vytvoří seznam. Např.:

```
ropuchy = "bufo,viridis,calamita,alvarius"
druhy = ropuchy.split(",")
print(str(druhy))
```

Z výstupu vidíme, že se řetězec rozdělil vždy, když narazil na čárku tedy oddělovač:

```
['bufo', 'viridis', 'calamita', 'alvarius']
```

Po vytvoření seznamu můžeme iterovat jako obvykle. Stejně jako řetězec „předstírá“, že je seznam, podobně se chová i soubor. Zatímco u řetězce smyčka bere každý znak jako prvek, v souboru se tak děje s řádky.

```
Soubor = open ("libovolny_text.txt")
for line in soubor:
   # cokoliv, co potřebujeme
```

Pokud píšeme program, který čte data ze souboru, je lepší použít buď **pouze** metodu read (pro uložení obsahu do proměnné), **nebo** smyčku (pro zpracování každého řádku). Pokud použijeme obojí v jednom programu, Python nemusí pochopit, co chceme udělat.
U každého souboru má Python přehled o své pozici, takže pokud přečteme obsah metodou read a poté se pokusíme zpracovat každý řádek pomocí smyčky, nic se nestane. Python si myslí, že už je na konci souboru. Pokud potřebujeme tyto dva příkazy nevyhnutelně použít dohromady, pomůže, když soubor zavřeme a poté opět otevřeme.

4.5. Rozsahy

Někdy se nám může hodit iterace přes seznam čísel. Např. budeme mít sekvenci aminokyselin nějakého proteinu:

```
Protein = "vlspadktnv"
```

a potřebujeme jako výstup první tři aminokyseliny, pak čtyři, pak pět atd.:

```
vls
vlsp
vlspa
vlspad
...
```

Logicky bychom použili smyčku na extrakci řetězce z řetězce (substring) a jediné, co potřebujeme, je nastavení konečné (stop) pozice. Nemůžeme iterovat přes řetězec jen tak, protože bychom dostali jednotlivé zbytky, což nechceme. Můžeme si ale ručně nastavit seznam se stop pozicemi a iterovat pomocí nich:

```
stop_pozice = [3,4,5,6,7,8,9,10]
for stop in stop_pozice:
   substring = protein[0:stop]
   print(substring)
```

Tento způsob je poněkud těžkopádný a funguje pouze v případě, kdy známe délku proteinu. Lepším řešením je v Pythonu zabudovaná funkce range, která generuje seznamy čísel, přes které

můžeme iterovat. Chová se v závislosti na tom, kolik má argumentů. S jedním argumentem bude počítat od nuly do uvedeného čísla, které ale vyloučí:

```
for cislo in range (6):
   print(number)
```

Se dvěma čísly (argumenty) bude brát první číslo a poslední vyloučí:

```
For number in range(3, 8):
   print(number)
```

Se třemi bude počítat od prvního po druhé a přičítat k němu třetí:

```
For number in range(2, 14, 4):
   print(number)
```

5. Funkce

Řekněme, že bychom chtěli s použitím dosavadních znalostí napsat program, který nám vypočítá obsah nukleotidů adeninu a tyminu, tedy AT (A+T), z nějaké DNA sekvence, např.:

```
moje_dna = "ACTGATCGATTACGTATAGTATTTGCTATCATACATATATATCGATGCGTTCAT"
delka = len(moje_dna)
a_count = moje_dna.count('A')
t_count = moje_dna.count('T')
obsah_at = (a_count + t_count) / delka
print("Obsah AT je " + str(obsah_at))
```

Když vypustíme první a poslední řádek (uložení do proměnné a print výsledku), vidíme, že kód pro výpočet AT zabere čtyři řádky. To znamená, že pokaždé, když budeme tento kód pro výpočet AT potřebovat, musíme ty čtyři řádky přesně znovu překopírovat. Bylo by mnohem snadnější, kdyby Python měl takovou funkci zabudovanou, jenže nemá. To však neznamená, že si takovou funkci nemůžeme vytvořit sami! Vytváření vlastních funkcí má mnoho výhod. Dovoluje nám v programu použít stejný kód, aniž bychom ho museli stále kopírovat a také když potřebujeme kód pozměnit, stačí ho změnit pouze na jednom místě a samozřejmě ho můžeme použít i v jiném programu.

5.1. Definice funkce

Pojďme si tedy vytvořit funkci na výpočet obsahu AT. Prvně si potřebujeme ujasnit, co bude ve vstupu – typy a počty argumentů, a co ve výstupu – typ návratové hodnoty. Náš příklad

je celkem jasný – ve vstupu bude sekvence DNA a výstup nějaké desetinné číslo. Čili funkce vezme řetězec (string) a navrátí číslo (float).

```
def vypocet_obsahu_at(dna):
    delka = len(dna)
    a_count = dna.count('A')
    t_count = dna.count('T')
    obsah_at = (a_count + t_count) / delka
    return obsah_at
```

Nyní si postupně rozebereme jednotlivé kroky. První řádek začíná slovem `def`, což je zkratka pro *define* (definice funkce). Následuje libovolný název funkce, název proměnné, pro kterou chceme funkci použít a na konci dvojtečka, stejně jako u smyček. A rovněž stejně jako u smyček se další řádky automaticky odsadí – vytvoří se **tělo funkce**. Dokud se dodrží odsazení, může být velké jakkoliv budeme chtít. Uvnitř funkce se můžeme odkazovat na argumenty použitím proměnných, do kterých jsme je uložili, ve výše uvedeném případě `dna`. Na posledním řádku nám funkce vrací hodnotu AT, kterou vypočítala –za funkci `return` napíšeme hodnotu, kterou chceme vidět jako výstup.

Jako všude jinde, i u funkcí je potřeba si dát pozor na několik věcí. Je třeba rozlišovat mezi definováním funkce a voláním funkce. Kód, který bychom spustili v Pythonu nic neudělá, protože jsme mu neřekli, aby funkci rozběhl, pouze jsme ji definovali. Funkci můžeme zavolat např. takto jednoduše rovnou se sekvencí:

```
vypocet_obsahu_at("ATGACTGGACCA")
```

… ale jakmile nám funkce vypočte výsledek, hodnota AT se ztratí. Užitečnější by bylo uložit výsledek do proměnné:

```
obsah_at = vypocet_obsahu_at("ATGACTGGACCA")
```

Nebo ho rovnou použít:

```
print("Obsah AT je " + str(vypocet:obsahu_at("ATGACTGGACCA")))
```

Další věc, kterou je třeba si uvědomit, je, že do argumentu proměnné `dna` **není při definici funkce uložená žádná hodnota**. Jejím úkolem je pojmout jakoukoli hodnotu, která jí bude dána při volání funkce. Je to podobné jako s proměnnými u smyček – při každém opakování

také drží jinou hodnotu. Poslední důležitá věc – stejně jako u smyček platí i u funkcí, že jakákoliv proměnná vytvořená jako součást funkce existuje pouze **uvnitř funkce**, tudíž není zvenčí přístupná. Pokud zkusíme použít proměnnou uvnitř funkce:

```
def vypocet_obsahu_at(dna):
    delka = len(dna)
    a_count = dna.count('A')
    t_count = dna.count('T')
    obsah_at = (a_count + t_count) / delka
    return obsah_at

print(obsah_at)
```

Objeví se chybové hlášení:

```
NameError: name 'obsah_at' is not defined
```

5.2. Volání a ladění funkce

Napišme si malý program s použitím naší funkce, abychom si ukázali, jak s ním Python pracuje. Výsledek jednak uložíme do proměnné (červeně) nebo jej rovnou zobrazíme (zeleně):

```
def vypocet_obsahu_at(dna):
    delka = len(dna)
    a_count = dna.count('A')
    t_count = dna.count('T')
    obsah_at = (a_count + t_count) / delka
    return obsah_at

muj_obsah_at = vypocet_obsahu_at("ATGCGCGATCGATCGAATCG")
print(str(muj_obsah_at))
print(vypocet_obsahu_at("ATGCATGCAACTGTAGC"))
print(vypocet_obsahu_at("aactgtagctagctagcagcgta"))
```

```
0.45
0.5294117647058824
0.0
```

Výstup vypadá, že první způsob volání funkce funguje a vypočítal obsah AT jako 0,45 (tedy 45 % A+T resp. 55 % G+C). Výsledek je uložen do proměnné muj_obsah_at a poté vytištěn. Pohled na další dva výstupy už tak pěkný není. Jeden obsahuje příliš mnoho desetinných čísel a druhý nulový výsledek, což určitě není správně. Pro nápravu bude potřeba pár malých změn. Python má zabudovanou funkci round, která potřebuje dva argumenty – číslo, které chceme

zaokrouhlit a počet požadovaných desetinných míst. Funkci round můžeme volat ještě před návratovou hodnotou. Malá písmena sekvence změníme na velká již zmíněnou metodou upper:

```
def vypocet_obsahu_at(dna):
    delka = len(dna)
    a_count = dna.upper().count('A')
    t_count = dna.upper().count('T')
    obsah_at = (a_count + t_count) / delka
    return round (obsah_at, 2)

muj_obsah_at = vypocet_obsahu_at("ATGCGCGATCGATCGAATCG")
print(str(muj_obsah_at))
print(vypocet_obsahu_at("ATGCATGCAACTGTAGC"))
print(vypocet_obsahu_at("aactgtagctagctagcagcgta"))
```

Nyní dostaneme správný výsledek:

```
0.45
0.53
0.52
```

Funkci můžeme ještě vylepšit – třeba tím, že si počet požadovaných desetinných míst určíme s každým jejím zavoláním. V našem kódu jsme upřesnili, že chceme pouze 2 desetinná místa, ale třeba se rozhodneme, že jich chceme vidět víc. Takže jednoduše přidáme druhý argument; proměnnou k funkci round:

```
def vypocet_obsahu_at(dna, sig_figs):
    delka = len(dna)
    a_count = dna.upper().count('A')
    t_count = dna.upper().count('T')
    obsah_at = (a_count + t_count) / delka
    return round (obsah_at, sig_figs)
```

A vyzkouškoušíme:

```
test_dna = "ATGCATGCAACTGTAGC"
print(vypocet_obsahu_at(test_dna, 1))
print(vypocet_obsahu_at(test_dna, 2))
print(vypocet_obsahu_at(test_dna, 3))
```

Dle následujících výstupů vidíme, že zaokrouhlování funguje správně:

```
0.5
0.53
0.529
```

5.3 Užitečné poznatky

5.3.1. Enkapsulace

V předchozí kapitole jsme napsali funkci a kód pro její použití. Během psaní kódu jsme přišli na to, že v naší definici funkce je pár chyb. **Měli jsme však možnost se vrátit a funkci změnit, aniž bychom změnili smysl kódu, ve kterém byla funkce použita**. Toto je velice důležitý poznatek a nebude přehnané tvrdit, že pochopení jeho důsledků je klíčem ke schopnosti psát delší programy. Popisuje totiž programovací jev, kterému se říká *enkapsulace*. Jde pouze o rozdělení komplexního programu na menší části, na kterých lze pracovat nezávisle. Na výše uvedeném příkladu je kód rozdělen na části dvě – v jedné definujeme funkci a ve druhé ji používáme; můžeme tedy obě měnit nezávisle na sobě.

5.3.2. Funkce nemusí mít argument

Nikde v pravidlech Pythonu není řečeno, že funkce potřebuje argument, protože je možné ji definovat i bez něj:

```
import datetime

def print_dnesni_datum():
   dnesni_datum = datetime.datetime.now().date()
   print(dnesni_datum)

print_dnesni_datum()
```

Jenže takové funkce nebývají příliš užitečné. Samozřejmě můžeme vypocet_obsahu_at napsat ve verzi, kdy nebude potřebovat žádný argument a proměnnou dna nastavit uvnitř funkce:

```
def vypocet_obsahu_at():
   dna = "ACTGATGCTAGCTA"
   delka = len(dna)
   a_count = dna.upper().count('A')
   t_count = dna.upper().count('T')
   obsah_at = (a_count + t_count) / delka
   return round (obsah_at, 2)
```

Tento způsob je již na první pohled nešikovný ale možný:

```
def vypocet_obsahu_at():
   delka = len(dna)
   a_count = dna.upper().count('A')
   t_count = dna.upper().count('T')
   obsah_at = (a_count + t_count) / delka
   return round (obsah_at, 2)

dna = "ACTGATGCTAGCTA"
```

```
print(vypocet_obsahu_at())
```

Na první pohled se situace jeví jako dobrý nápad – vše funguje, protože funkce získá hodnotu *dna*, která je nastavena na předposledním řádku. Tím se však porušuje princip enkapsulace, kterého jsme se usilovně snažili dosáhnout. Funkce nyní funguje pouze v případě, že existuje proměnná s názvem *dna* a je nastavena v té části kódu, kde je funkce volána, takže tyto dvě části kódu na sobě již nejsou nezávislé. Pokud se přistihneme, že píšeme takovýto kód, je obvykle dobré určit, které proměnné z vnějšku funkce se používají uvnitř funkce, a přeměnit je na argumenty.

5.3.3. Funkce nemusí vždy vracet hodnotu

Podívejme se na tuto variaci naší funkce – místo *vracení* obsahu AT jej rovnou *tiskne*:

```
def vypocet_obsahu_at(dna, sig_figs):
    delka = len(dna)
    a_count = dna.upper().count('A')
    t_count = dna.upper().count('T')
    obsah_at = (a_count + t_count) / delka
    print(str(round(obsah_at, 2)))
```

Je velmi lákavé, zabít dvě mouchy jednou ranou. Nešikovné tady však je, že tento typ funkce už není flexibilní. Teď sice chceme vidět na obrazovce obsah AT, ale co když později zjistíme, že výslednou hodnotu potřebujeme do souboru nebo ji použít jako součást dalších výpočtů? Klíčem k napsání flexibilní funkce je si uvědomit, že *výpočet* a *výstup* obsahu AT jsou dva odlišné procesy. Vytvářejme funkce, které udělají pouze jeden krok, poté můžou následovat kroky další, jako kdybychom postupně, od základů, stavěli zeď z cihel.

5.3.4. Funkci lze zavolat s argumentem

Co potřebujeme o funkci vědět, abychom ji mohli použít? Musíme znát návratovou hodnotu, její typ a také typy a množství argumentů, nesmíme opomenout ani jejich **pořadí**. Např. při použití funkce `open` potřebujeme nejdřív název souboru s cestou k němu a pak režim, ve kterém ho chceme otevřít (`"r"`, `"w"`, `"a"`). Pro použití verze `obsah_at` se dvěma argumenty si musíme uvědomit, že na prvním místě bude sekvence DNA a následně množství desetinných míst. Python umí tzv. *argumenty s klíčovými slovy* (keyword arguments), které nám umožní volat funkce trochu jiným způsobem – místo seznamu argumentů v závorkách:

```
obsah_at("ATCGTGACTCG", 2)
```

můžeme vytvořit seznam s proměnnými včetně hodnot (funguje to i s metodami):

```
obsah_at(dna="ATCGTGACTCG", sig_figs=2)
```

Tento způsob volání funkcí má mnoho výhod. Na pořadí argumentů nezáleží, takže tyto dva kódy se budou chovat stejně:

```
obsah_at(dna="ATCGTGACTCG", sig_figs=2)
obsah_at(sig_figs=2, dna="ATCGTGACTCG")
```

Rovněž je na první pohled zřejmé, o co jde. Dokonce můžeme míchat oba styly dohromady, následující řádky fungují úplně stejně:

```
obsah_at("ATCGTGACTCG", 2)
obsah_at(dna="ATCGTGACTCG", sig_figs=2)
obsah_at("ATCGTGACTCG", sig_figs=2)
```

Nelze však začít s argumenty s klíčovými slovy a pak se vrátit k normálnímu – tento způsob vrátí chybu:

```
obsah_at(dna="ATCGTGACTCG", 2)
```

Můžou být tedy užitečné zejména pro funkce a metody s větším množstvím argumentů.

5.3.5. Argument jako výchozí hodnota funkce

S něčím podobným jsme se setkali v kapitole 3 při otevírání souborů. Funkce `open` vyžaduje dva argumenty – název souboru a režim. Ale pokud ji zavoláme pouze s názvem souboru, Python použije **výchozí hodnotu** (což je v tomto případě `"read"`). To můžeme využít i u funkcí jednoduše upřesněním výchozí hodnoty na prvním řádku, kdy definujeme funkci. Verze našeho `obsah_at` s výchozí hodnotou pro dvě desetinná místa by vypadala následovně:

```
def vypocet_obsahu_at(dna, sig_figs=2):
    length = len(dna)
    a_count = dna.upper().count('A')
    t_count = dna.upper().count('T')
    obsah_at = (a_count + t_count) / length
    return round(obsah_at, sig_figs)
```

Pokud tuto funkci zavoláme se dvěma argumenty, použije námi zvolený počet desetinných míst. Pokud ji zavoláme pouze s jedním argumentem, výchozí hodnota budou dvě desetinná místa:

```
vypocet_obsahu_at("ATCGTGACTCG")
vypocet_obsahu_at("ATCGTGACTCG", 3)
vypocet_obsahu_at("ATCGTGACTCG", sig_figs=4)
```

Funkce se o to v prvním případě postarala.

```
0.45
0.455
0.4545
```

Výchozí hodnoty argumentů funkcí nám umožňují psát velmi flexibilní funkce, které mohou mít různý počet argumentů. Má smysl je používat pouze pro argumenty, u kterých lze zvolit rozumnou výchozí hodnotu. Nemá smysl zadávat výchozí hodnotu pro argumenty typu *dna* v našem příkladu. Jsou užitečné zejména pro funkce, kde se některé z možností používají jen zřídka.

5.3.6. Testování funkcí

Při psaní kódu je dobré si pravidelně ověřovat, že jednotlivé části fungují správně. U všech kódů v předešlých kapitolách jsme si průběžně zkoušeli výstupy výsledků ještě předtím, než jsme je použili. Důvod, proč tak činit, je, že odhalíme rychleji chyby a současně si ověříme funkčnost kódu, u kterého známe správnou odpověď. V Pythonu lze využít funkci assert a za dvě rovnítka s výsledkem, který očekáváme. Např. pokud zkusíme zavolat funkci se sekvencí „ATCG“, dopředu víme, že by odpověď měla být 0,5. Toto tvrzení si otestujeme:

```
assert vypocet_obsahu_at("ATGC") == 0.5
```

Účel dvou rovnítek za sebou si vysvětlíme později. Funkce assert funguje velmi jednoduše – pokud je předpoklad nesprávný, tedy odpověď by nebyla 0,5, Python vrátí AssertionError.

Tvrzení neboli „assertions“ představují užitečný prostředek pro testování funkcí a hledání chyb. Když se ve výstupu objeví chyba programu, ve kterém je použitá určitá funkce, ale test funkce proběhl v pořádku, pak si můžeme být jisti, že chyba bude v kódu, který funkci volá. Rovněž nám umožňují funkci upravit a pak zkontrolovat, jestli jsme si ji nezkazili. Také můžeme otestovat, jak se funkce bude chovat při nějakém neobvyklém vstupu. Jak např.

otestujeme náš vypocet_obsahu_at, když dostane sekvenci DNA, která bude obsahovat neznámé báze (obvykle značené jako „N“)? Můžeme se s tím vypořádat tak, že je z výpočtu zkrátka odstraníme. Napíšeme následující tvrzení:

```
assert vypocet_obsahu_at("ATGCNNNNNNNNNN") == 0.5
```

Což samozřejmě neprojde, avšak funkci si přizpůsobíme:

```
def vypocet_obsahu_at(dna, sig_figs=2):
    dna = dna.replace('N, '')
    length = len(dna)
    a_count = dna.upper().count('A')
    t_count = dna.upper().count('T')
    obsah_at = (a_count + t_count) / length
    return round(obsah_at, sig_figs)
```

6. Podmínky

Od jednoduchých výpočtů jsme se posunuli ke komplikovanějším, ale všechny byly zatím prováděny víceméně stejně. Často budeme také potřebovat přimět program k tomu, aby se sám rozhodoval, což se naučíme v této kapitole, kde budeme používat *podmínky*.

6.1. Podmínky, True a False

Podmínka je v podstatě kousek kódu, jehož výstupem je odpověď True (pravda), nebo False (nepravda). Nejlépe tomu porozumíme přímo na příkladech. Otestujeme si několik matematických příkladů, řetězců a zda se daná hodnota vyskytuje v seznamu:

```
print(3 == 5)
print(3 > 5)
print(3 <=5)
print(len("ATGC") > 5)
print("GAATTC".count("T") > 1)
print("ATGCTT".startswith("ATG"))
print("ATGCTT".endswith("TTT"))
print("ATGCTT".isupper())
print("ATGCTT".islower())
print("V" in ["V", "W", "L"])
```

Ve výstupu vidíme odpověď na každý řádek:

```
False
False
True
```

```
False
True
True
False
True
False
True
```

Co nám vlastně říká? Na první pohled to vypadá, že tiskne „True“ a „False“, které jsme ale nikde v kódu neuvedli. Výstup představuje speciální zabudované hodnoty (psány s velkým počátečním písmenem), kterými Python vyhodnocuje pravdu a nepravdu. Do podmínek lze zahrnout velké množství prvků, zde si uvedeme pouze základní z nich:

- rovná se (==)
- větší a menší než (>, <)
- větší a menší než nebo rovná se (>=, <=)
- nerovná se (!=)
- je hodnotou v seznamu (in)

V uvedeném příkladu jsme se setkali např. s metodou startswith, která vrací True, pokud náš řetězec začíná řetězcem, který mu je dán v argumentu. Všimněme si, že pro test porovnání se používají **dvě** rovnítka, použití jednoho vrátí chybu.

6.2. If a Else

Nejjednodušší podmínkou je tvrzení if neboli *pokud*, protože má snadno pochopitelnou syntaxi:

```
A = 33
b = 200
if b > a:
  print("b je vetsi nez a")
```

Napíšeme tedy if, potom podmínku a řádek ukončíme dvojtečkou. Následuje blok s odsazenými řádky, tedy tělo (jako u funkcí), které proběhne pouze tehdy, pokud bude podmínka splněna. Tvrzení if můžeme použít např. pro testování vlastností nějaké neznámé, jejíž hodnotu během psaní programu nevíme. Pro příklad výše tedy příliš praktické využití nenajdeme. Vytvořme si seznam s identifikátory pro sekvence a použijme podmínku, aby se nám zobrazily pouze ty začínající na „a“:

```
sekvence = ['ab56', 'bh84', 'hv76', 'ay93', 'ap97', 'bd72']
for kody in sekvence:
   if sekvence.startswith('a'):
     print(sekvence)
```

Výstup nám umožní si zkontrolovat, že vše funguje, jak má:

```
ab56
ay93
ap97
```

Řádek s print je dvojitě odsazený – jednou pro smyčku a jednou pro podmínku. S tímto způsobem *indentace* se zde setkáváme poprvé, ale při vytváření obsáhlejších kódů jde o velmi běžný postup. Python odsazuje klidně donekonečna, což ale pro nás může ztratit na přehlednosti již po třetí úrovni. Pokud se tak stane, měli bychom zvážit zabalení nepřehledného kódu do funkce.

S podmínkou if úzce souvisí podmínka else. V uvedeném příkladu se vlastně program rozhoduje mezi „ano“ a „ne“. Často ale potřebujeme i rozhodnutí zahrnující „buď“ a „nebo“ , kdy se nám rozšíří možnosti výstupů. K tomu stačí přidat doložku else na stejnou úroveň s if:

```
a = 33
b = 200
if b > a:
  print("b je vetsi nez a")
else:
  print("b neni vetsi nez a")
```

Tvrzení else samo o sobě žádnou podmínku nemá, ani není bráno v potaz, pokud je podmínka, kterou splňuje tvrzení if, splněna.

Tento příklad užívá if a else pro rozdělení seznamu sekvencí do dvou souborů – ty, které začínají na „a“ se zapíšou do prvního seznamu, všechny ostatní pak do druhého:

```
soubor1 = open("jedna.txt", "w")
soubor2 = open("dva.txt", "w")
sekvence = ['ab56', 'bh84', 'hv76', 'ay93', 'ap97', 'bd72']
for kody in sekvence:
   if sekvence.startswith('a'):
     soubor1.write(sekvence + "\n")
   else:
     soubor2.write(sekvence + "\n")
```

Všimněte si opět několikanásobného odsazení, kdy if a else jsou na stejné úrovni.

6.3. Elif

Pokud bychom chtěli seznam rozdělit do více než dvou souborů, můžeme vměstnat další podmínku if pod doložku else:

```
soubor1 = open("jedna.txt", "w")
soubor2 = open("dva.txt", "w")
sekvence = ['ab56', 'bh84', 'hv76', 'ay93', 'ap97', 'bd72']
for kody in sekvence:
   if sekvence.startswith('a'):
     soubor1.write(sekvence + "\n")
   else:
     if sekvence.startswith('b'):
       soubor2.write(sekvence + "\n")
     else:
       soubor3.write(sekvence + "\n")
```

To sice funguje, ale špatně se čte. Navíc potřebujeme opět více odsazovat. Proto má Python ještě podmínku elif, která slučuje else a if a umožňuje nám zpřehlednění kódu:

```
soubor1 = open("jedna.txt", "w")
soubor2 = open("dva.txt", "w")
sekvence = ['ab56', 'bh84', 'hv76', 'ay93', 'ap97', 'bd72']
for kody in sekvence:
   if sekvence.startswith('a'):
     soubor1.write(sekvence + "\n")
   elif sekvence.startswith('b'):
      soubor2.write(sekvence + "\n")
   else:
      soubor3.write(sekvence + "\n")
```

Najednou potřebujeme pouze dvojité odsazení. Navíc můžeme podmínek elif v kódu použít, kolik chceme, a stále bude stačit pouze jedna úroveň odsazení:

```
for kody in sekvence:
   if sekvence.startswith('a'):
     soubor1.write(sekvence + "\n")
   elif sekvence.startswith('b'):
      soubor2.write(sekvence + "\n")
   elif sekvence.startswith('c'):
      soubor3.write(sekvence + "\n")
   elif sekvence.startswith('d'):
      soubor4.write(sekvence + "\n")
   else:
      soubor5.write(sekvence + "\n")
```

6.4. While

Poslední věc, kterou můžeme s podmínkami udělat, je určit jimi konec smyčky. Ve čtvrté kapitole jsme mluvili o smyčkách, které iterují nad daným souborem dat (seznam, řetězec nebo soubor). Python má kromě for ještě jeden typ smyčky, a sice while. Smyčka while na rozdíl od for, kde *předem známe počet opakování*, se používá, když cyklus závisí na nějaké podmínce, a tělo cyklu se opakuje, dokud je podmínka splněna.

```
cislo = 0
while cislo<10:
   print(cislo)
   count = cislo + 1
```

Výsledku z tohoto příkladu bychom lépe dosáhli s metodou range. Jelikož jsou „obyčejné“ smyčky dostačující, nesetkáme se s nimi v kódech Pythonu tolik, jako u jiných programovacích jazyků. Proto se s nimi nebudeme více zabývat.

6.5. Komplexní kód s podmínkami

Co když potřebujeme podmínku sestavenou z více částí? Např. extrahovat z našeho seznamu sekvencí jen ty, které začínají na „a“ a končí číslem 3. Můžeme použít dvakrát if:

```
sekvence = ['ab56', 'bh84', 'hv76', 'ay93', 'ap97', 'bd72']
for kody in sekvence:
   if sekvence.startswith('a'):
      if sekvence.endswith('3'):
         print(sekvence)
```

Ale tolik odsazení je zbytečné a nepotřebné. Zvládneme to lépe spojením dvou podmínek pomocí **and** a vytvoříme tak komplexní výraz:

```
sekvence = ['ab56', 'bh84', 'hv76', 'ay93', 'ap97', 'bd72']
for kody in sekvence:
   if sekvence.startswith('a') and sekvence.endswith('3'):
      print(sekvence)
```

Tato verze se navíc i dobře čte. Rovněž můžeme použít i **or**, čímž se vytvoří podmínky, které budou považovány za splněné, pokud alespoň jedna z nich bude potvrzena:

```
sekvence = ['ab56', 'bh84', 'hv76', 'ay93', 'ap97', 'bd72']
for kody in sekvence:
```

```
  if sekvence.startswith('a') or sekvence.endswith('3'):
    print(sekvence)
```

Vše můžeme i nakombinovat, pokud chceme sekvence začínající buď na „a“ nebo „b“ a končící na číslo 4:

```
sekvence = ['ab56', 'bh84', 'hv76', 'ay93', 'ap97', 'bd72']
for kody in sekvence:
  if (sekvence.startswith('a') or sekvence.startswith('a')) and sekvence.endswith('4'):
    print(sekvence)
```

Přidali jsme, stejně jako v matematice, závorky, abychom se vyhnuli nejasnostem. Nakonec můžeme jakoukoliv podmínku negovat prefixem **not**. V následujícím příkladu chceme sekvence začínající na „a“, ale nekončící na 6:

```
sekvence = ['ab56', 'bh84', 'hv76', 'ay93', 'ap97', 'bd72']
for kody in sekvence:
  if sekvence.startswith('a') and not sekvence.endswith('6'):
    print(sekvence)
```

Tyto tři operátory (**and, or, not**) jsou známé jako „booleovské operátory“ (boolean operators) a figurují snad úplně všude. Pokud bychom si chtěli na internetu vyhledat informace o použití Pythonu v biologii, ale nezajímala by nás sekce o hadech, zadali bychom do vyhledávače např. „Python biology - snake“. Tak jsme vytvořili podmínku, stejně jako u příkladů výše – Google si mezi klíčová slova automaticky dosadí *and* a pomlčce rozumí jako *not*, takže vyhledáváme stránky zmiňující Python a biologii, ale bez hadů.

6.6. True a False

Občas můžeme potřebovat funkci, kterou budeme moci použít v podmínce, což uděláme velmi snadno – jen zařídíme, aby se pokaždé vracelo buď *True* nebo *False*. Tato dvě tvrzení jsou v Pythonu zabudovaná, tudíž mohou být pominuta, uložena v proměnných a vracena, stejně jako čísla nebo řetězce.

Zde máme funkci, která určuje, jestli je sekvence DNA bohatá na obsah AT (v tomto případě tomu tak bude, pokud bude obsah AT vyšší než 0,65, tedy 65 %):

```
def at_bohate(dna):
  delka = len(dna)
  pocet_a = dna.upper().count('A')
  pocet_t = dna.upper().count('T')
  obsah_at = (pocet_a + pocet_t) / delka
```

```
    if at_bohate > 0.65:
        return True
    else:
        return False
```

Funkci si otestujeme na následujících příkladech:

```
print(at_bohate("ATTATCTACTA"))
print(at_bohate("CGGCAGCGCT"))
```

```
True
False
```

Výstup vrací tvrzení True nebo False, stejně jako ostatní podmínky, takže bychom mohli funkci použít i v kombinaci s if:

```
If at_bohate(moje_dna):
    # a teď se sekvencí něco udělej
```

Poslední čtyři řádky jsou věnovány vyhodnocení podmínky a vracení True nebo False. Lze však napsat kompaktnější verzi, ve které se podmínka vyhodnotí a rovnou ukáže výsledek:

```
def at_bohate(dna):
    delka = len(dna)
    pocet_a = dna.upper().count('A')
    pocet_t = dna.upper().count('T')
    obsah_at = (pocet_a + pocet_t) / delka
    return obsah_at > 0.65
```

Kód je sice stručnější, ale lépe se čte, pokud jsme s idiomy už seznámeni.

7. Slovníky

Vycházejme z toho, že chceme opět spočítat, kolik máme „A“ v sekvenci DNA. Jde o jeden z prvních úkolů, které jsme zkoušeli:

```
dna = "ATCGATCGATCGTACGCTGA"
pocet_a = dna.count("A")
```

Jak se kód změní, kdybychom chtěli vygenerovat kompletní seznam počtu bází v sekvenci, tj. A, C, G a T? Přidáme tedy proměnné:

```
dna = "ATCGATCGATCGTACGCTGA"
pocet_a = dna.count("A")
pocet_t = dna.count("T")
pocet_g = dna.count("G")
pocet_c = dna.count("C")
```

Jenže kód se poněkud opakuje, což není vysloveně chyba, obzvlášť, pokud chceme počítat pouze samostatné báze. Ale co kdybychom chtěli třeba kód pro všech 16možných dinukleotidů:

```
dna = "ATCGATCGATCGTACGCTGA"
pocet_aa = dna.count("AA")
pocet_at = dna.count("AT")
pocet_ag = dna.count("AG")
...atd. atd.
```

Nebo všech 64 trinukleotidů:

```
dna = "ATCGATCGATCGTACGCTGA"
pocet_aaa = dna.count("AAA")
pocet_aat = dna.count("AAT")
pocet_aag = dna.count("AAG")
...atd. atd.
```

Tímto způsobem by bychom se upsali. Naše sekvence má 20 bází, tedy se v ní překrývá „pouze“ 18 trinukleotidů a tudíž bychom vytvořili 64 různých proměnných, z nichž by asi 46 vyšlo s nulovou hodnotou. Jedna z možností by byla uložit hodnoty do seznamu. Můžeme pak použít trojitou smyčku – pro generaci všech možných trinukleotidů, pro jejich spočtení a nakonec uložení do seznamu:

```
dna = "AATGATCGATCGTACGCTGA"
pocty = []
for baze1 in ['A', 'T', 'G', 'C']:
    for baze2 in ['A', 'T', 'G', 'C']:
        for baze3 in ['A', 'T', 'G', 'C']:
            trinukleotid = baze1 + baze2 + baze3
            pocet = dna.count(trinukleotid)
            print("pocet trinukleotidu " + trinukleotid + " je " + str(pocet))
            pocty.append(pocet)
print(pocty)
```

Byť je tento kód kompaktní a neobsahuje příliš proměnných, výstup není oku lahodící:

```
pocet trinukleotidu AAA je 0
pocet trinukleotidu AAT je 1
pocet trinukleotidu AAG je 0
pocet trinukleotidu AAC je 0
pocet trinukleotidu ATA je 0
pocet trinukleotidu ATT je 0
```

```
pocet trinukleotidu ATG je 1

…

[0, 1, 0, 0, 0, 0, 1, 2, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 2, 0, 0, 2, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0]
```

Data jsou rozptýlená a většinou je hodnota nula. Navíc kdybychom se chtěli podívat na počet určitého trinukleotidu, např. TGA, nejdřív bychom museli zdlouhavě hledat, že naše smyčky jej vygenerovaly na 25. místě. Jedině tak dostaneme prvek, který chceme, ze správného indexu:

```
print("pocet TGA je " + str(pocty[24]))
```

Tento problém můžeme zkusit různými způsoby vyřešit. Co kdybychom vygenerovali dva seznamy – jeden na výpočet a jeden na vlastní trinukleotidy?

```
dna = "AATGATCGATCGTACGCTGA"
trinukleotidy = []
pocty = []
for baze1 in ['A', 'T', 'G', 'C']:
   for baze2 in ['A', 'T', 'G', 'C']:
        for baze3 in ['A', 'T', 'G', 'C']:
            trinukleotid = baze1 + baze2 + baze3
            pocet = dna.count(trinukleotid)
            trinukleotidy.append(trinucleotid)
            pocty.append(pocet)
print(pocty)
print(all_trinucleotides)
```

Nyní máme dva seznamy o stejné délce a trochu jsme si zpříjemnili náhled:

```
[0, 1, 0, 0, 0, 0, 1, 2, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0,
2, 0, 0, 0, 0, 0, 2, 0, 0, 2, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0]

['AAA', 'AAT', 'AAG', 'AAC', 'ATA', 'ATT', 'ATG', 'ATC', 'AGA', 'AGT',
'AGG', 'AGC', 'ACA', 'ACT', 'ACG', 'ACC', 'TAA', 'TAT', 'TAG', 'TAC',
'TTA', 'TTT', 'TTG', 'TTC', 'TGA', 'TGT', 'TGG', 'TGC', 'TCA', 'TCT',
'TCG', 'TCC', 'GAA', 'GAT', 'GAG', 'GAC', 'GTA', 'GTT', 'GTG', 'GTC',
'GGA', 'GGT', 'GGG', 'GGC', 'GCA', 'GCT', 'GCG', 'GCC', 'CAA', 'CAT',
'CAG', 'CAC', 'CTA', 'CTT', 'CTG', 'CTC', 'CGA', 'CGT', 'CGG', 'CGC',
'CCA', 'CCT', 'CCG', 'CCC']
```

Pro počet daného trinukleotidu se stačí podívat do seznamu s `trinukleotidy` a pak získat index ze seznamu `pocty`:

```
i = trinukleotidy.index('TGA')
```

```
c = pocty[i]
print('pocet pro TGA je ' + str(c))
```

Tento postup a jeho výsledek již vypadá lépe, ale stále má spoustu nevýhod. Ukládáme nulové hodnoty a musíme se starat o dva seznamy, což přináší potíže v přehlednosti a hlavně ve flexibilitě – uděláme změnu v jednom beze změny v druhém a indexy už nám nebudou souhlasit. Dále je tento postup pomalý, Python musí každý prvek opakovaně procházet do té doby, než najde ten, který hledáme. S obsáhlostí seznamu by se tedy zvyšoval i čas pro vyhodnocování. Když se ohlédneme zpět a zamyslíme se nad naším problémem obecněji, zjistíme, že potřebujeme systém, který by nám uložil data (trinukleotidy a jejich počty) tak, abychom mohli jednoduše zjistit počet pro jakýkoliv trinukleotid. Takový systém ukládání dat je v programování velice častý. V bioinformatice pravděpodobně budeme potřebovat ukládat:

- názvy sekvencí proteinů a jejich sekvence
- kodony s jejich aminokyselinovými zbytky
- jména kolegů s jejich e-mailovými adresami
- cizojazyčná slovíčka a jejich význam
- …

Všechny tyto příklady utváří dvojice **klíče** a **hodnoty**. Poslední příklad je zajímavý tím, že pro něj máme už vytvořený známý systém – slovník (dictionary). Python je touto pomůckou vybaven a nazývá se úplně stejně. V této kapitole se dozvíme, jak si takový slovník vytvořit a jak s ním pracovat.

7.1. Tvorba slovníku

Syntaxe je podobná jako u tvorby seznamu, pouze zde využíváme složených závorek. Každý pár dat skládající se z hodnoty a klíče je jednotlivou položkou uzavřenou uvozovkami a oddělují se čárkami. Podíváme se na kousek kódu s restrikčními enzymy ve slovníku o třech položkách:

```
enzymy = { 'EcoRI':r'GAATTC', 'AvaII':r'GG(A|T)CC', 'BisI':'GC[ATGC]GC' }
```

V tomto případě jsou klíč i hodnota řetězce (dokonce tzv. „raw string“). Rozdělení slovníku na několik řádků umožňuje lepší čtení a nemá vliv na kód:

```
enzymy = {
   'EcoRI' : r'GAATTC',
   'AvaII' : r'GG(A|T)CC',
   'BisI'  : r'GC[ATGC]GC'
}
```

K získání požadovaných dat stačí napsat název slovníku a klíče v hranatých závorkách:

```
print(enzymy['BisI'])
```

Podobně bychom to napsali i při práci se seznamem, ale místo indexu pro požadovaný prvek vkládáme klíč pro hodnotu, kterou chceme. Klíče *musí* být unikátní – nelze pod ně ukládat více hodnot. V reálu bychom se s tvorbou slovníku, jaká je uvedená výše, nesetkali. Spíše nejdříve vytvoříme prázdný slovník a poté do něho přidáváme klíče s hodnotami (jako při tvorbě prázdného seznamu s pozdějším přidáním prvků). Jednoduše přiřadíme složené závorky k názvu slovníku. Pak můžeme postupně přidávat položky:

```
enzymy = {}
enzymy['EcoRI'] = r'GAATTC'
enzymy['AvaII] = r'GG(A|T)CC'
enzymy['BisI'] = r'GC[ATGC]GC'
```

Klíč lze mazat pomocí metody **pop**, která vrátí hodnotu a zároveň klíč smaže:

```
# smaže ze slovníku enzym EcoRI
enzymy.pop('EcoRI')
```

Vraťme se k počtům trinukleotidů. Takto bychom je ukládali do slovníku:

```
dna = "AATGATCGATCGTACGCTGA"
pocty = {}
for baze1 in ['A', 'T', 'G', 'C']:
   for baze2 in ['A', 'T', 'G', 'C']:
        for baze3 in ['A', 'T', 'G', 'C']:
            trinukleotid = baze1 + baze2 + baze3
            pocet = dna.count(trinukleotid)
            pocty[trinukleotid] = pocet
print(pocty)
```

Z výstupu vidíme, že trinukleotidy s jejich počty jsou uloženy společně do jedné proměnné:

```
{'ACC': 0, 'ATG': 1, 'AAG': 0, 'AAA': 0, 'ATC': 2, 'AAC': 0, 'ATA': 0,
'AGG': 0, 'CCT': 0, 'CTC': 0, 'AGC': 0, 'ACA': 0, 'AGA': 0, 'CAT': 0,
'AAT': 1, 'ATT': 0, 'CTG': 1, 'CTA': 0, 'ACT': 0, 'CAC': 0, 'ACG': 1,
'CAA': 0, 'AGT': 0, 'CAG': 0, 'CCG': 0, 'CCC': 0, 'CTT': 0, 'TAT': 0,
'GGT': 0, 'TGT': 0, 'CGA': 1, 'CCA': 0, 'TCT': 0, 'GAT': 2, 'CGG': 0,
'TTT': 0, 'TGC': 0, 'GGG': 0, 'TAG': 0, 'GGA': 0, 'TAA': 0, 'GGC': 0,
```

```
'TAC': 1, 'TTC': 0, 'TCG': 2, 'TTA': 0, 'TTG': 0, 'TCC': 0, 'GAA': 0,
'TGG': 0, 'GCA': 0, 'GTA': 1, 'GCC': 0, 'GTC': 0, 'GCG': 0, 'GTG': 0,
'GAG': 0, 'GTT': 0, 'GCT': 1, 'TGA': 2, 'GAC': 0, 'CGT': 1, 'TCA': 0,
'CGC': 1}
```

Stále zůstávají nulové hodnoty, avšak hledání počtu pro určitý trinukleotid je mnohem snadnější:

```
print(counts['TGA'])
```

Už si nemusíme pamatovat pořadí nebo si utvářet dva seznamy. Pojďme zkusit najít způsob, jak se vyhnout ukládání nul. Můžeme přidat podmínku **if**, která nám pomůže ukládat pouze počty větší než nula:

```
dna = "AATGATCGATCGTACGCTGA"
pocty = {}
for baze1 in ['A', 'T', 'G', 'C']:
   for baze2 in ['A', 'T', 'G', 'C']:
        for baze3 in ['A', 'T', 'G', 'C']:
            trinukleotid = baze1 + baze2 + baze3
            pocet = dna.count(trinukleotid)
            pocty[trinukleotid] = pocet
            if pocet > 0:
               pocty[trinukleotid] = pocet
print(pocty)
```

Data z výstupu se zmenšila – ukázaly se nám pouze počty trinukleotidů, které se vyskytují alespoň jednou:

```
{'ATG': 1, 'ACG': 1, 'ATC': 2, 'GTA': 1, 'CTG': 1, 'CGC': 1, 'GAT': 2,
'CGA': 1, 'AAT': 1, 'TGA': 2, 'GCT': 1, 'TAC': 1, 'TCG': 2, 'CGT': 1}
```

A to by nebylo programování, kdybychom nemuseli řešit další problém. Kód pracuje skvěle, pokud je výsledek pozitivní:

```
print(pocty['TGA'])
```

Ale pokud se rovná nule, trinukleotid se neobjeví jako klíč,

```
print(counts['AAA'])
```

takže nám Python vrátí chybu:

```
KeyError: 'AAA'
```

Máme dvě možnosti, jak tento problém řešit. Můžeme si nejdřív ověřit existenci klíče (stejně jako existenci prvku v seznamu) a pokusit se teprve poté získat hodnotu:

```
if 'AAA' in pocty:
    print(pocty('AAA'))
```

Dále lze použít slovníkovou metodu get, která funguje obdobně jako hranaté závorky – následující dva řádky stejný výsledek:

```
print(pocty['TGA'])
print(pocty.get('TGA'))
```

Užitečnost metody get spočívá v tom, že pracuje s výchozím argumentem, který navrátí, pokud se klíč ve slovníku nevyskytuje. V našem případě, pokud daný trinukleotid není ve slovníku, jeho počet se rovná nule, tudíž můžeme nastavit nulu jako výchozí hodnotu a použít get pro výstup počtů jakéhokoliv trinukleotidu:

```
print("pocet TGA je " + str(pocty.get('TGA', 0)))
print("pocet AAA je " + str(pocty.get('AAA', 0)))
print("pocet GTA je " + str(pocty.get('GTA', 0)))
print("pocet TTT je " + str(pocty.get('TTT', 0)))
```

Jak vidíme z výstupu, už se nemusíme řešit, jestli je daný trinukleotid ve slovníku – **get** se o vše postará a příhodně vrátí nulu:

```
pocet pro TGA je 2
pocet pro TGA je 0
pocet pro TGA je 1
pocet pro TGA je 0
```

7.2. Iterace ve slovníku

Namísto hledání určitého prvku se tentokrát zaměříme na situaci, kdy chceme tu samou věc pro všechny položky, např. výstup všech trinukleotidů s počtem 2. Můžeme opět použít tři vnořené smyčky na vygenerování všech možných trinukleotidů, pak zjistit jejich počty a nakonec se rozhodnout, jestli je chceme vidět:

```
for baze1 in ['A', 'T', 'G', 'C']:
    for baze2 in ['A', 'T', 'G', 'C']:
        for baze3 in ['A', 'T', 'G', 'C']:
            trinukleotid = baze1 + baze2 + baze3
            if pocty.get(trinukleotid, 0) == 2:
                print(trinukleotid)
```

Kód funguje, jak očekáváme:

```
ATC
TGA
TCG
GAT
```

Je efektivní opět generovat všechny kombinace, když už máme veškeré informace ve slovníku? Lepším řešením je přečíst všechny klíče přímo ze slovníku – k tomu nám poslouží metoda keys. Pokud metodu keys použijeme v souvislosti se slovníkem, vrátí nám seznam všech klíčů:

```
print(pocty.keys())
```

Nyní kód na vrácení počtu všech trinukleotidů vyskytujících se v DNA sekvenci dvakrát vypadá stručněji a výsledek bude stejný:

```
for trinukleotid in pocty.keys():
    if pocty.get(trinukleotid) == 2:
        print(trinukleotid)
```

K uspořádání klíčů v abecedním pořadí existuje metoda sorted:

```
for trinukleotid in sorted (pocty.keys()):
    if pocty.get(trinukleotid) == 2:
        print(trinukleotid)
```

V našem kódu je potřeba ve smyčce jako první vždy vyhledat hodnotu pro aktuální klíč, což je běžný postup při iteraci slovníkem. Python pro něj má svoji zkratku. Namísto tohoto:

```
for klic in muj_slovnik.keys():
    hodnota = muj_slovnik.get(klic)
    # udělej něco s hodnotou a klíčem
```

můžeme použít metodu items pro iteraci páry dat:

```
for klic, hodnota in muj_slovnik.items():
    # udělej něco s hodnotou a klíčem
```

Metoda items se ode všech ostatních trochu liší; nenavrací jednu hodnotu, seznam hodnot, ale seznam párů hodnot. Proto umístíme na začátek smyčky dvě proměnné. Takto bychom items použili pro náš slovník s počty trinukleotidů:

```
for trinukleotid, pocet in pocty.items():
    if pocet == 2:
        print(trinukleotid)
```

Tato metoda je při iteracích nad slovníky obecně preferována, jelikož ušetří spoustu řádků kódu a současně jde o velmi elegantní řešení.

8. Balíky pro grafické výstupy a statistiku

8.1. Matplotlib

Nyní se přesuneme do vizualizace našich dat. Jednou z nejdůležitějších knihoven je *matplotlib*, který umožňuje rychle zobrazit data přímo ve výstupovém řádku. Hodně prvků je převzato z programového prostředí MATLAB, tudíž práce s knihovnou *matplotlib* není tak intuitivní. Strukturu *matplotlibu* bychom si mohli rozdělit na tři vrstvy:

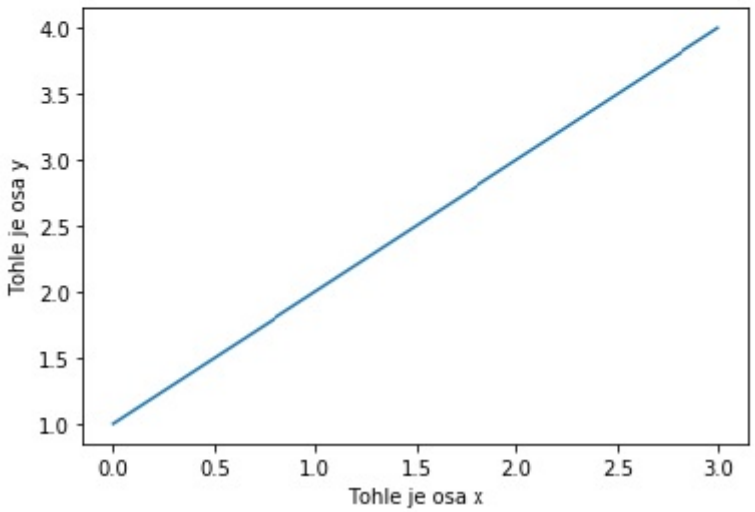
1. **Backendová vrstva** – Obecně se označuje jako backend nějaká „zadní část“ a často ji ani nemusíme vidět a věnovat ji pozornost. Komplexně se o všechno stará díky komunikaci s vykreslující sadou nástrojů.
2. **Umělecká vrstva** – Umožňuje řízení a vyladění celého obrazce.
3. **Skriptová vrstva** – Té se budeme věnovat a do jejího prostředí psát naše požadavky.

Pro zjednodušení si strukturu *matplotlibu* můžeme představit jako v cukrárně – zatímco o první dvě vrstvy se starají v kuchyni, my potřebujeme komunikovat pouze s třetí vrstvou (číšníkem), která se jmenuje **pyplot**. Rovnou si můžeme na zkoušku vytvořit v Jupyteru náš první graf. Nejprve je třeba si naimportovat knihovnu matplotlib (zavolat číšníka) a poté již psát požadavky:

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

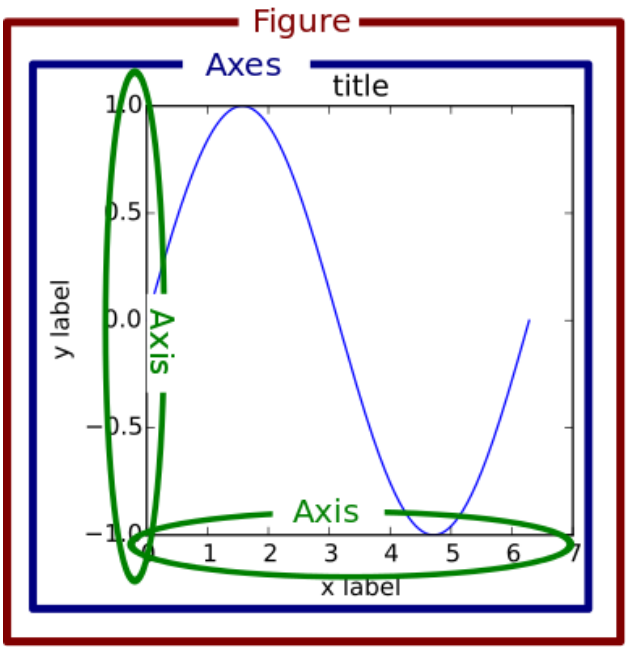
```
plt.figure()
plt.plot([1, 2, 3, 4])
plt.ylabel("Tohle je osa y")
plt.xlabel("Tohle je osa x")
```

```
Text(0.5, 0, 'Tohle je osa x')
```



Obr. č. 7 – Tvorba jednoduchého grafu

Jednotlivé kroky vypadají snadno, ale pojďme si je raději rozebrat. plt.figure() vytvoří samotný obrazec grafu (v podstatě jako prázdné plátno pro umělce), plt.plot potřebuje argument pro velikost os a nakonec obě osy pomocí plt.ylabel (xlabel) pojmenujeme a máme vytvořený rychlý graf jen tak pro představu. Pro potřeby obsáhlejších grafů je můžeme postupně budovat řádek po řádku a navíc zabřednout do umělecké vrstvy pro přizpůsobení a vytvoření sofistikovanějšího grafu, toho se však v této příručce dotkneme pouze okrajově.



Obr. č. 8 – Struktura grafu (zdroj: realpython.com)

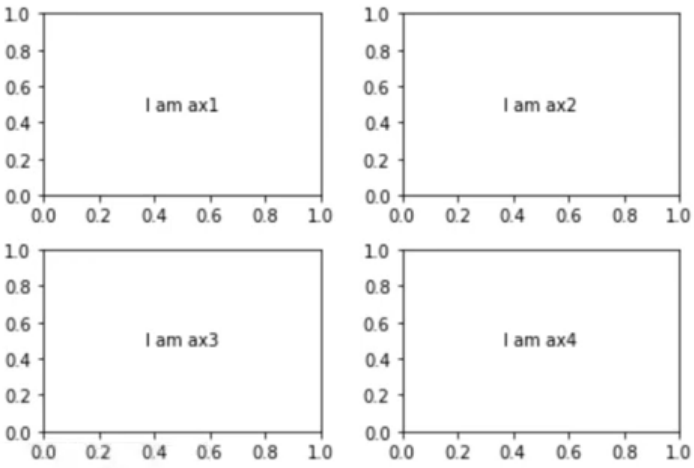
Pro začátek bychom si mohli do „pythonovského“ slovníčku zařadit pár nových pojmů, abychom měli na čem stavět. K tomu nám pomůže Obr. č. 8, kde červeně vyznačené *figure* zahrnuje celý graf (celé malířské plátno), v podstatě základna. Další vrstvu představují modře vyznačené *axes*, neboli osy, kterých může být i několik a určitě je možné mít v jednom poli více grafů najednou (ty se pak nazývají *subplots*, představme si několik obrazů na jednom velkém plátně). Zelené *axis* už jsou jednotlivé osy y a x.

Zmínili jsme tzv. subplots, se kterými to ale není tak jednoduché. Máme tři možnosti, jak přidávat osy – **plt.subplot()**, **plt.subplots()** a **plt.axes()**. Každá se hodí pro jinou příležitost, postupně si tedy všechny rozebereme.

plt.subplot()

Při vytváření jakéhokoliv grafu potřebujeme tedy prvně vytvořit „plátno“ zavoláním **fig = plt.figure()** a nahodit osy pomocí **ax = fig.add_subplot()**, ukládáme do proměnných, zde např. fig a ax. Pokud do závorek nespecifikujeme velikost grafu, výchozí nastavení je **(1, 1, 1)** (nebo **(111)**, je to v podstatě jedno), což znamená jednu osu na jednom řádku a v jednom sloupci. plt.subplot() vrací pouze jedny osy a graf vytváří automaticky, můžeme pomocí něj přidat i více os, ale je to komplikovanější a musí se přidávat postupně (Obr. č. 9), proto se pro tento účel hodí následující funkce.

```
fig = plt.figure()
ax1 = fig.add_subplot(221)
ax2 = fig.add_subplot(222)
ax3 = fig.add_subplot(223)
ax4 = fig.add_subplot(224)

ax1.annotate('I am ax1', (0.5, 0.5),
             xycoords='axes fraction', va='center', ha='center')
ax2.annotate('I am ax2', (0.5, 0.5),
             xycoords='axes fraction', va='center', ha='center')
ax3.annotate('I am ax3', (0.5, 0.5),
             xycoords='axes fraction', va='center', ha='center')
ax4.annotate('I am ax4', (0.5, 0.5),
             xycoords='axes fraction', va='center', ha='center')

plt.tight_layout()
```



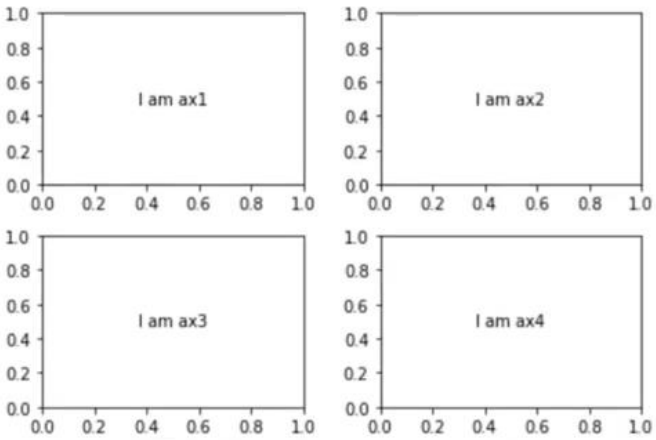
Obr. č. 9 – Funkce plt.subplot()

plt.subplots()

Lze jej přidat pomocí **ax = fig.subplots()** a hodí se, pokud chceme mít v jednom grafu více os, což je jeho hlavní síla – ostatně tomu napovídá už název. Potřebuje poziční argumenty, to

znamená množství řádků a sloupců, výchozí nastavení je opět **(1, 1)**. Pro přidání os nám stačí napsat vše na jeden řádek (Obr. č. 10).

```
fig = plt.figure()
axs = fig.subplots(nrows=2, ncols=2)

axs.flat[0].annotate('I am ax1', (0.5, 0.5),
                     xycoords='axes fraction', va='center', ha='center')
axs.flat[1].annotate('I am ax2', (0.5, 0.5),
                     xycoords='axes fraction', va='center', ha='center')
axs.flat[2].annotate('I am ax3', (0.5, 0.5),
                     xycoords='axes fraction', va='center', ha='center')
axs.flat[3].annotate('I am ax4', (0.5, 0.5),
                     xycoords='axes fraction', va='center', ha='center')

plt.tight_layout()
```

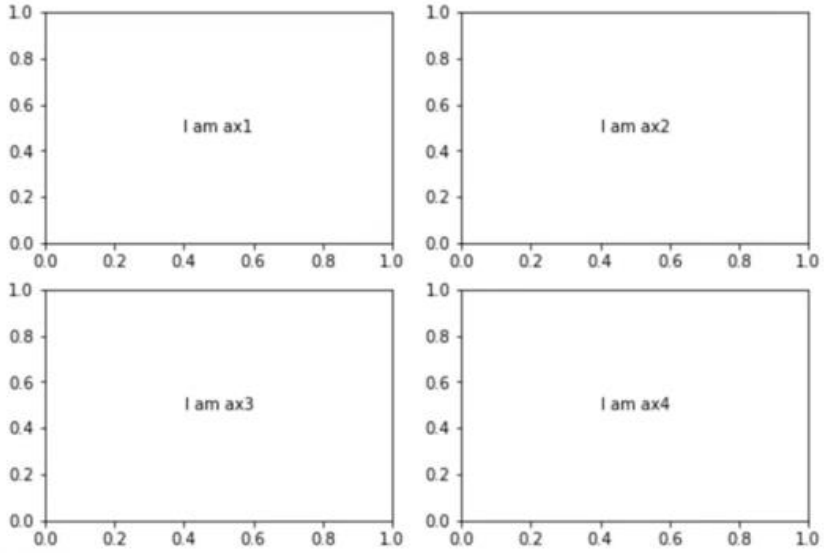


Obr. č. 10 – Funkce plt.subplots()

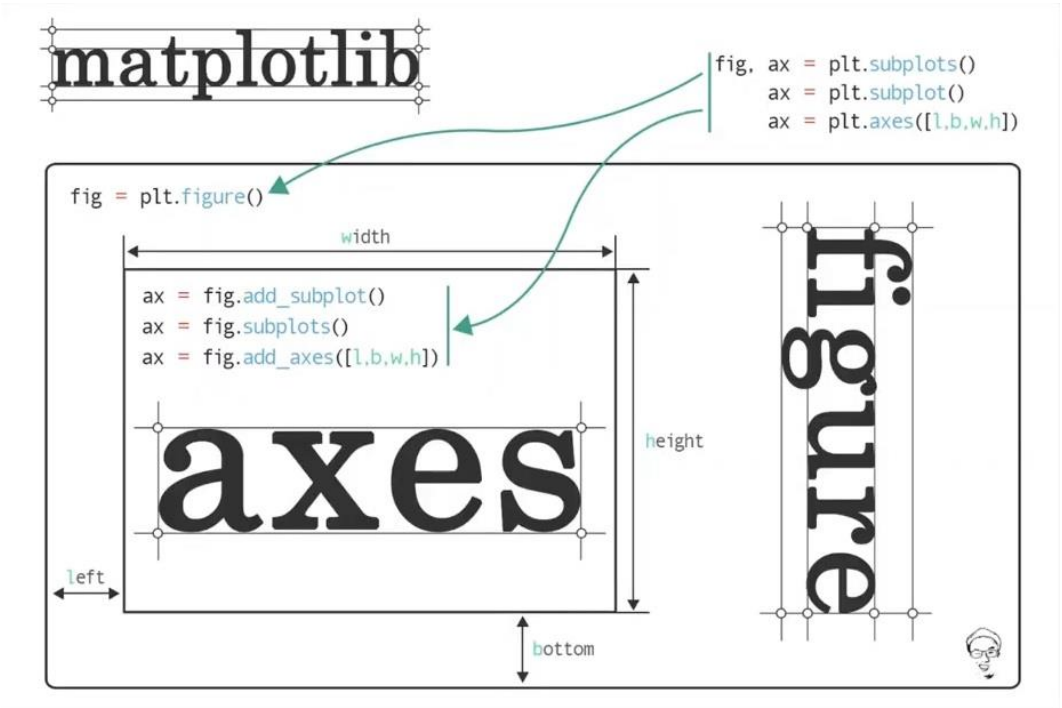
plt.axes()

Tato funkce se jen trochu liší od předchozí a umožňuje větší flexibilitu, pokud nastane potřeba nestandardních rozměrů grafu, to s sebou však přináší i komplikovanost. Pro přidání os do grafu použijeme ax = fig.add_axes([levá, spodní, šířka, výška]), přičemž všechna čtyři čísla musí být zlomky šířky a výšky; takto budeme mít pod kontrolou pozici a velikost os. Výchozí hodnoty jsou ([0, 0, 0.78, 0.78]). plt.axes() vrací jedny osy pomocí fig_add_subplot(1, 1, 1) a graf vytvoří automaticky, užití této funkce se najde např. při potřebě „grafech v grafu“ nebo grafech, které se překrývají. Takhle na úvod to všechno zní docela složitě, ale jakmile se seznámíme s praktickým využitím, tak se s funkcemi sžijeme a věci se budou zdát drobet snadnější a srozumitelnější. Pro přehlednost přikládáme jednu grafiku (Obr. č. 12).

```
fig = plt.figure()
ax1 = fig.add_axes([0, 0.6, 0.5, 0.5])
ax2 = fig.add_axes([0.6, 0.6, 0.5, 0.5])
ax3 = fig.add_axes([0, 0, 0.5, 0.5])
ax4 = fig.add_axes([0.6, 0, 0.5, 0.5])

ax1.annotate('I am ax1', (0.5, 0.5),
                 xycoords='axes fraction', va='center', ha='center')
ax2.annotate('I am ax2', (0.5, 0.5),
                 xycoords='axes fraction', va='center', ha='center')
ax3.annotate('I am ax3', (0.5, 0.5),
                 xycoords='axes fraction', va='center', ha='center')
ax4.annotate('I am ax4', (0.5, 0.5),
                 xycoords='axes fraction', va='center', ha='center')

plt.show()
```



Obr. č. 11 – Funkce plt.axes()



Obr. č. 12 – Souhrnná grafika k tvorbě grafů (zdroj: laptrinhx.com)

8.2. Pandas a praktické ukázky

Pandas je užitečná softwarová knihovna pro analýzu dat, název je ze zkratky Panel Data System, zabudovaná v Pythonu a funguje v něm zhruba jako Excel či R. Jedná se o svobodný software a používá se pro manipulaci a analýzu dat, lze je tak filtrovat, extrahovat a dále upravovat. Nejdříve si knihovnu Pandas standardně naimportujeme:

```
import pandas as pd
```

Opět, knihovnu lze mít naimportovanou pod jakýmkoliv názvem, ale pokud to po nás někdo má číst, tak by se třeba nemusel dopátrat. Pandas umí načíst mnoho typů souborů, ale nejběžněji se používá formát *.csv (comma separated values). Na internetu lze snadno dohledat a stáhnout datový soubor titanic.csv, který obsahuje jména všech pasažérů Titanicu, jejich věk, jestli přežili atd., k dispozici je zde: https://gist.github.com/michhar/2dfd2de0d4f8727f873422c5d959fff5. Naimportujeme si jej do Jupyteru a uložíme do proměnné s názvem titanic:

```
titanic = pd.read_csv("titanic.csv")
#do závorky patří vaše umístění souboru
```

Nyní jsme vytvořili objekt Pandas, kterému se říká **DataFrame**. Je to v podstatě jakási „nádoba“ či „obal“ a umí toho opravdu hodně – má velké množství zabudovaných funkcí pro zkoumání a manipulaci s daty. K počátečnímu průzkumu dat použijeme následující příkaz:

```
titanic.head()
```

Tím, že jsme do závorky nezadali žádný argument, příkaz nám ukáže prvních pět řádků v souboru. Pro opačný směr použijeme:

```
titanic.tail()
```

Takto si rovněž rychle ověříme, že se nám soubor naimportoval správně. Metoda .shape nám odhalí, kolik máme v souboru sloupců a řádků.

```
titanic.shape
```

Dále se můžeme podívat i na názvy jednotlivých sloupečků:

```
titanic.columns
```

Pokud chceme zobrazit pouze jednu sadu dat, je to stejné jako se slovníkem (jen pozor na citlivost velikosti písmen):

```
titanic["Age"]
```

Příkazy můžeme samozřejmě kombinovat:

```
titanic.Name.head()
```

Filtrací dat pro věk jsme se dostali z **DataFrame** do **Series** – **DataFrame** je celý soubor, kdežto **Series** pouze vybraný sloupec. Většinou pracují stejně, ale může se stát, že narazíte na metody, které budou fungovat pouze pro jeden typ dat.

Pro nejrychlejší získání informací o souboru použijeme .info() a Pandas nám je předloží jak nejlépe umí – zjistíme velikost souboru, počet sloupců a jejich názvy, typ dat atd. Další velmi užitečnou metodou je .describe(). Zkusme tedy např.:

```
titanic.Age.describe()
```

Získáme základní statistiku pro celý sloupec. To stejné můžeme udělat s celým souborem:

```
titanic.describe()
```

Také můžeme rychle zjistit průměr všech sloupců souboru najednou:

```
titanic.mean()
```

U PassengerId nám tento příkaz nic moc nedá, ale o přeživších už můžeme prohlásit, že jich bylo 38 %. Ne vždy se k nám však dostane krásný a přehledný *.csv soubor. Jak si s tím poradit? Budeme potřebovat soubor playgolf.csv. Nahrajeme si jej do Jupyteru jako proměnnou golf a zobrazíme si prvních 10 řádků tak, že do závorky metody head napíšeme hodnotu 10:

```
golf = pd.read_csv("PlayGolf.csv")
golf.head(10)
```

	07-01-2014|sunny|85|85|false|Don't Play
0	07-02-2014|sunny|80|90|true|Don't Play
1	07-03-2014|overcast|83|78|false|Play
2	07-04-2014|rain|70|96|false|Play
3	07-05-2014|rain|68|80|false|Play
4	07-06-2014|rain|65|70|true|Don't Play

Obr. č. 12 – Výstup golf.head()

Výsledek vypadá zvláštně a především nepřehledně. Vrátíme se s kurzorem na soubor na řádek, kde jsme si načetli soubor a stiskneme Shift+Tab. Objeví se nám informace o načteném souboru. Co nás momentálně zajímá, je sep=','. sep je zkratkou pro „separator“ neboli oddělovač. Vidíme, že ve výchozím nastavení souboru je oddělovačem čárka, my však budeme chtít pro přehlednost svislou čáru. Přidáme tedy k načtení souboru argument sep = "|" a ověříme si prvních pět řádků:

```
golf=pd.read_csv("PlayGolf.csv", sep = "|")
```

```
golf.head()
```

	07-01-2014	sunny	85	85.1	false	Don't Play
0	07-02-2014	sunny	80	90	True	Don't Play
1	07-03-2014	overcast	83	78	False	Play
2	07-04-2014	rain	70	96	False	Play
3	07-05-2014	rain	68	80	False	Play
4	07-06-2014	rain	65	70	True	Don't Play

Obr. č. 13 – Výstup golf.head() po úpravě

Povedlo se nám vyřešit jeden problém, zbývá ještě upravit hlavičky. Vytvoříme si seznam pravděpodobných názvů:

```
golf_hlavicky = ["Date", "Outlook", "Temperature", "Humidity", "Windy", "Result"]
```

Nahlédneme opět se Shift+Tab a Tab stiskneme ještě jednou (lze i víckrát, ale pro naše potřeby stačí dvakrát) a vidíme, že Pandas se snažila vytvořit hlavičky po svém („infer“), což se jí ale nepodařilo. Řekneme tedy, že hlavičky tvořit nemá a zadáme je ručně namísto names=None. Celý kód pro úpravu souboru bude nakonec vypadat následovně:

```
golf = pd.read_csv("playgolf.csv", sep = "|", header = None, names = golf_hlavicky)
```

Opět nahlédneme pomocí golf.head().

	Date	Outlook	Temperature	Humidity	Windy	Result
0	07-01-2014	sunny	85	85	False	Don't Play
1	07-02-2014	sunny	80	90	True	Don't Play
2	07-03-2014	overcast	83	78	False	Play
3	07-04-2014	rain	70	96	False	Play
4	07-05-2014	rain	68	80	False	Play

Obr. č. 14 – Výstup golf.head() po finální úpravě

A máme hotovo! V další části se budeme konečně věnovat grafům.

8.3. Vizualizace dat

Cílem této (pod)kapitoly není ze čtenáře udělat mága a předat mu zázračné schopnosti nádherné a okamžité vizualizace dat, nýbrž předložit dostatek porozumění a podnětů pro vlastní prozkoumávání nepřeberného množství možností. O knihovně matplotlib jsme mluvili již v úvodu a je skvělá především pro tvorbu jednoduchých grafů, klade důraz na rychlost a relativní přizpůsobivost. Nově zmíníme a zahrneme knihovnu seaborn, která na matplotlibu staví, ale disponuje větším množstvím tzv. funkcí out-of-the-box. To v tomhle případě znamená, že může fungovat ihned bez dalších úprav nebo složitějších nastavení (prakticky úplný opak toho, než jak se tento idiom používá v ne IT oborech).

Pojďme si tedy naimportovat vše potřebné:

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib notebook
```

Za procentem se schovávají tzv. *„magic commands“*. Jsou to v podstatě vylepšení dodávaná ke klasickému python kódu poskytovaná IPython kernelem a přidáváme je pro řešení jednoduchých problémů a představují také jakési zkratky a úspory času – **%matplotlib notebook** nám kupříkladu umožní interaktivní prostředí grafu.

Základní vzorec pro graf by mohl vypadat nějak takto:

- plt.plot(x, y) vytvoří základní podobu, .plot() však může být nahrazen i něčím jiným, např. .bar()
- plt.title("Název grafu") přidá libovolné pojmenování

- plt.xlabel("Rok") přidá název „Rok“ pro osu x.
- plt.ylabel("Populace") pojmenuje osu y „Populace“.
- plt.xticks([1, 2, 3, 4, 5]) udělá body 1, 2, 3, 4 a 5 na osu x.

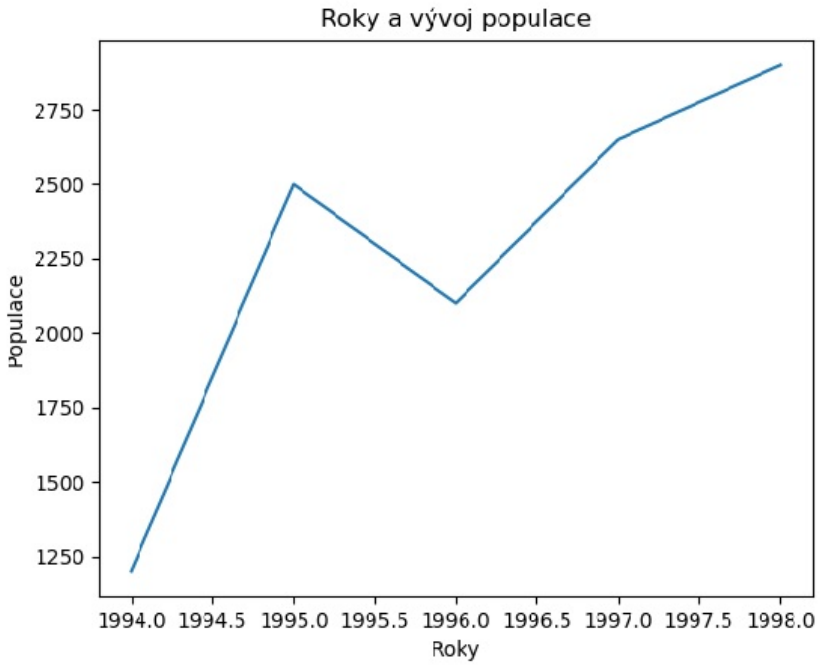
8.5.1. Řádkový graf

Nejčastější graf potřebný pro vyjádření nějakého trendu, který se mění v čase a je vyobrazen jako čára spojená jednotlivými body („markers“). Jako první argument se dávají data pro horizontální osu a druhý pro data na vertikální. Budeme potřebovat zavolat funkci plt.plot(), která graf vygeneruje, ale není vidět takže pro zobrazení musíme zavolat ještě funkci plt.show(), což se hodí pro následné úpravy. Zkusíme si vytvořit dva seznamy náhodných dat:

```
roky = [1994, 1995, 1996, 1997, 1998]
populace = [1200, 2500, 3000, 3300, 2900]
```

A pokračujeme rovnou na graf:

```
plt.plot(roky, populace)
plt.title("Roky a vývoj populace")
plt.xlabel("Roky")
plt.ylabel("Populace")
plt.show()
```



Obr. č. 15 – Řádkový graf

8.5.2. Rozptýlený graf

Tento typ grafu také vyobrazuje jednotlivé body, ale už nejsou spojeny. Užívá se především pro zobrazení korelace a porovnání dvou proměnných. Budeme potřebovat funkci plt.scatter(), pravidla pro osy zůstávají stejná. Stáhneme si soubor *iris.csv*, např. odtud: https://gist.github.com/curran/a08a1080b88344b0c8a7#file-iris-csv a naimportujeme si ho do Jupyter notebooku dle umístěni:

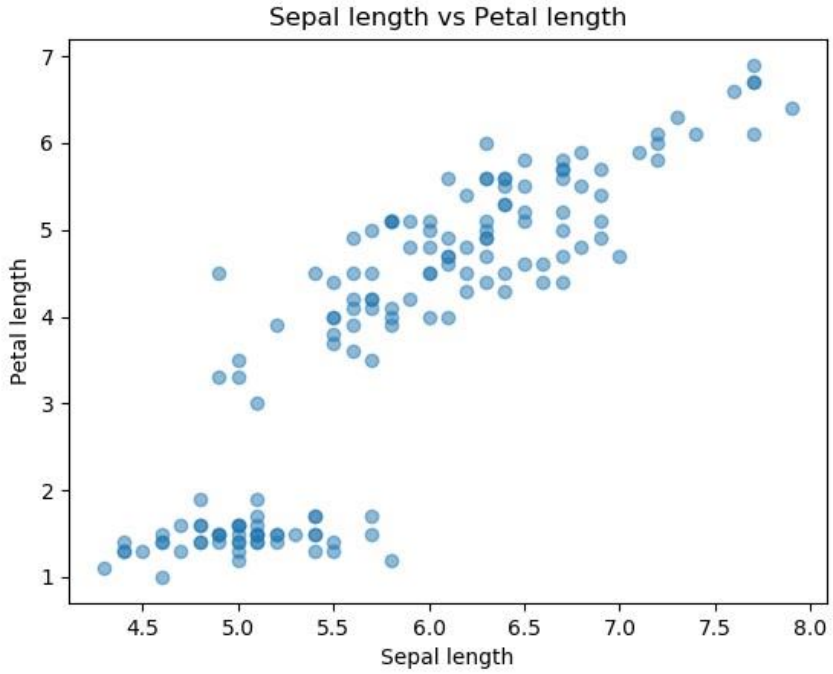
```
iris = pd.read_csv("iris.csv")
```

Můžeme do souboru samozřejmě nahlédnout, už víme, jak se to dělá…

```
iris.head()
```

Tentokrát budeme tedy graf vytvářet z DataFrame, takže si ze souboru vybereme např. délky okvětních lístků:

```
plt.scatter(iris.sepal_length, iris.petal_length)
plt.title("Sepal length vs Petal length")
plt.xlabel("Sepal length")
plt.ylabel("Petal length")
plt.show()
```



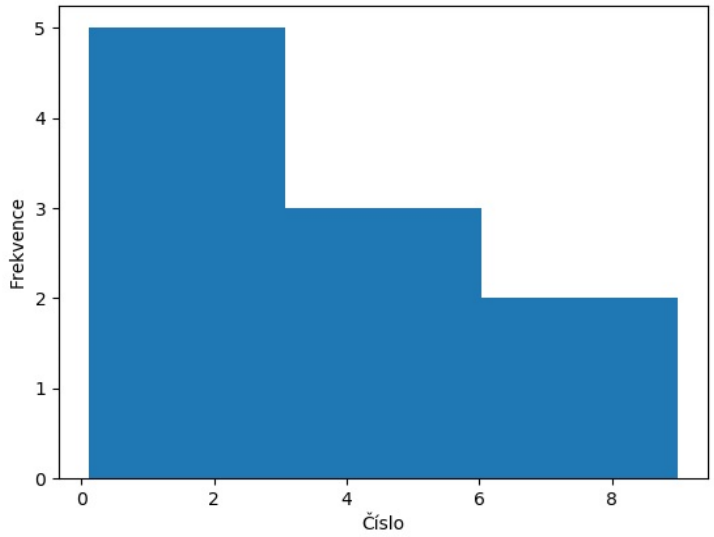
Obr. č. 16 – Rozptýlený graf

Pro přehlednost změti teček jsme přidali na první řádek ještě argument alpha=0.5. Komu by nevyhovovala výchozí modrá barva, lze si ji přenastavit argumentem color="barva" (pochopitelně v angličtině, takže např. purple, green atd.) A kdo by chtěl pozměnit i tvar teček, stačí přidat argument marker="tvar"(např. „>“, „+“ atd.)

8.5.3. Histogram

Histogram reprezentuje distribuci číselných dat, rozděluje rozsah hodnot do sérií intervalů (sloupečků, *bins*) a spočte, kolik hodnot do určitého rozsahu spadá. Použijeme tedy funkci plt.hist(), kde prvním argumentem budou numerická data a druhým počet sloupečků. Výchozí hodnota pro druhý argument je 10. Kolik sloupečků by v histogramu vlastně mělo být? Odpověď není tak jednoduchá a záleží na situaci. Kdo by však chtěl více hloubat, radu nalezne např. zde: http://users.stat.umn.edu/~gmeeden/papers/hist.pdf. Nyní už ale k tvorbě grafu:

```
cisla = [0.1, 0.5, 1, 1.5, 2, 4, 5.5, 6, 8, 9]
plt.hist(cisla, bins = 3)
plt.xlabel("Číslo")
plt.ylabel("Frekvence")
plt.show()
```



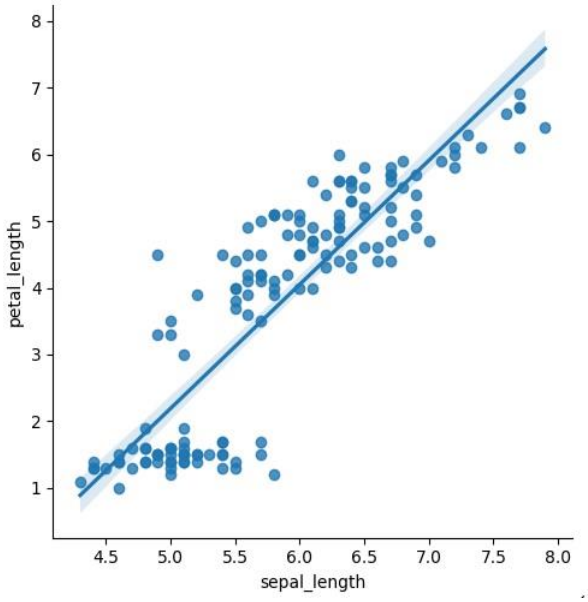
Obr. č. 17 – Histogram

8.5.4. Seaborn

Knihovna *seaborn* umí stejné věci jako *matplotlib*, svým způsobem ho rozšiřuje a velmi se hodí pro explorační analýzu dat. Zatímco síla *matplotlibu* tkví ve složitějších operacích, *seaborn* se snaží tyto postupy jednodušeji zpřístupnit, disponuje lepším výchozím nastavením

(ať už jde o barvy, zatržítka atd.) a práce s DataFrame je v něm o něco jednodušší. Zkusíme si znovu udělat rozptýlený graf s *iris.csv*:

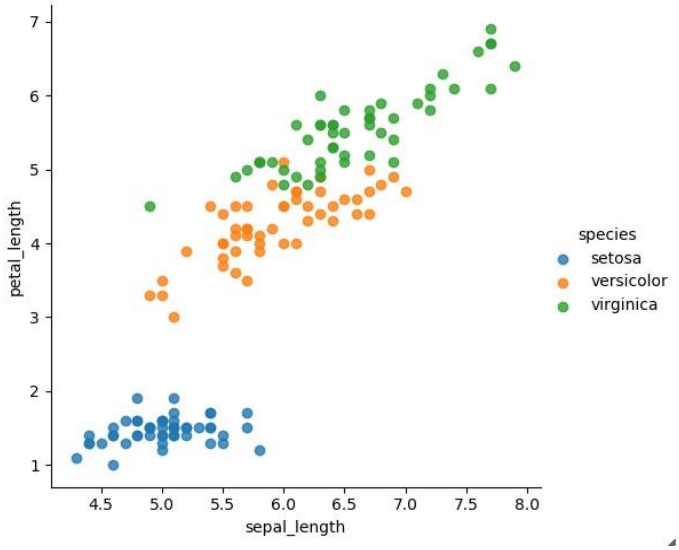
```
sns.lmplot(data = iris, x="sepal_length", y="petal_length")
plt.show()
```



Obr. č. 18 – Rozptýlený graf v *Seaborn*

Seaborn nám automaticky do grafu vložil i lineární regresi, což je skvělé, ale úpravy ještě budou potřeba. Pro přehlednost lze jedním příkazem přidat barvy zobrazených bodů:

```
sns.lmplot(data = iris, x="sepal_length", y="petal_length", fit_reg=False, hue="species")
plt.show()
```

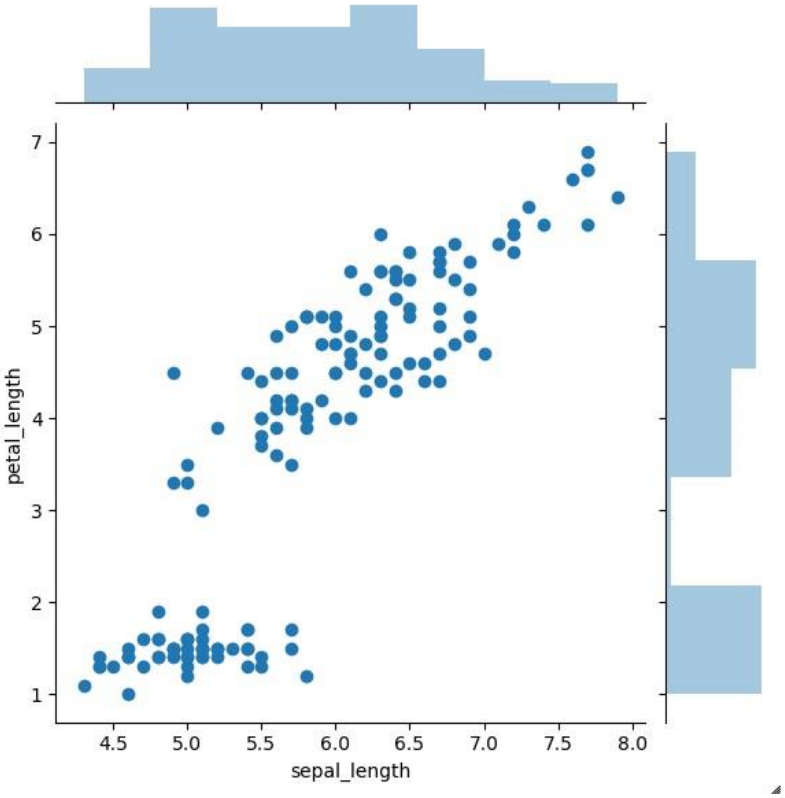


Obr. č. 19 – Rozptýlený graf v *Seaborn* po úpravě

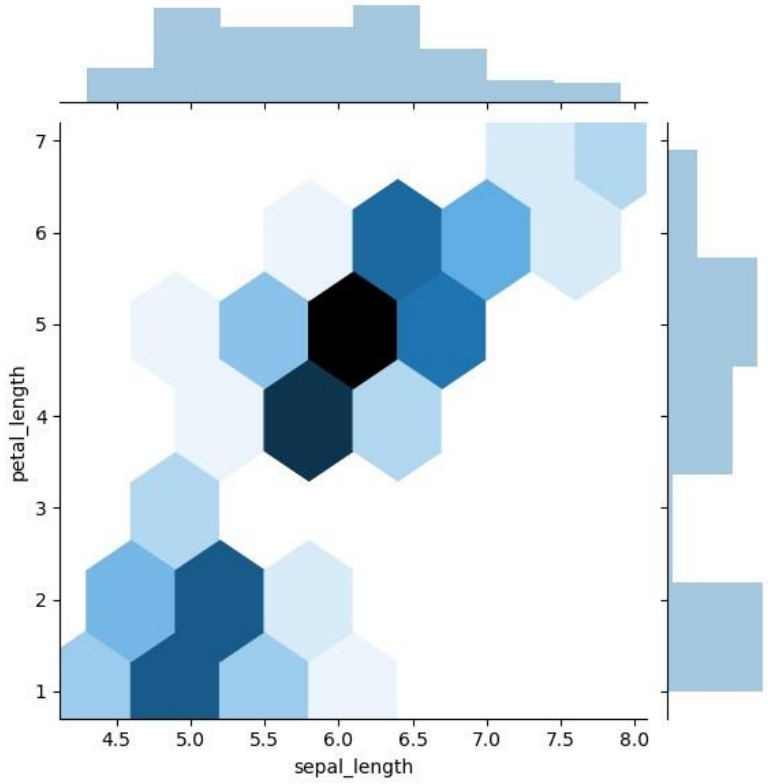
Nyní máme barevně roztříděné délky okvětních lístků podle druhů kosatců. Je pochopitelně nemožné si zapamatovat veškeré kódy pro přizpůsobení grafu, nic vám nebrání si je dohledat a graf si tak doladit zcela po svém. Podíváme se ještě na další užitečné grafy, které *Seaborn* umí, např. speciální druh grafu – jointplot, který je kombinací rozptýleného grafu a histogramu (Obr. č. 20):

```
sns.jointplot(data = iris, x="sepal_length", y="petal_length")
plt.show()
```

Změny tvaru bodů jsme se o několik řádků výše již dotkli, nyní můžeme přidat argument `kind="hex"` a získáme graf s hexagonálními body (Obr. č. 21). Změnit styl, lze pomocí příkazu `sns.set_style("zvolenystyl")` a nápověda Shift+Tab nám ukáže nabídku.



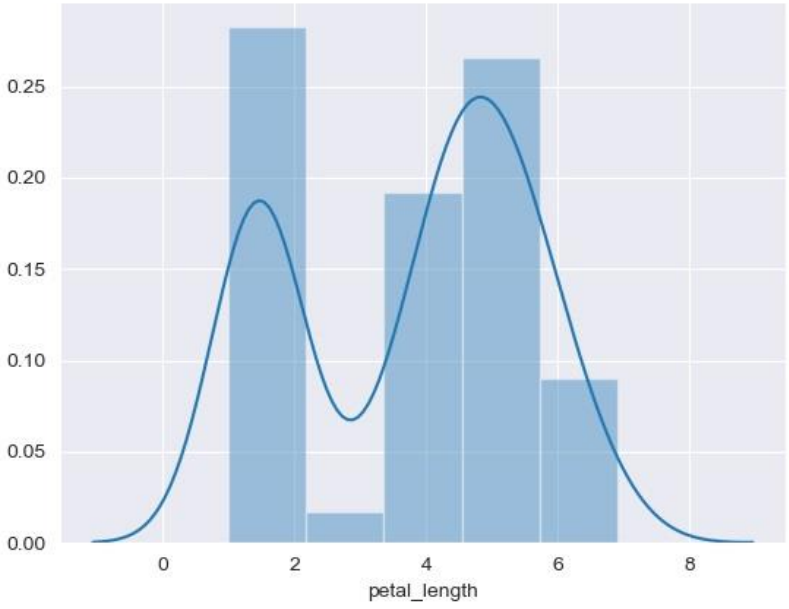
Obr. č. 20 – Jointplot



Obr. č. 21 – Jointplot s hexagonálními body

Zkusíme se *Seaborn* udělat ještě histogram:

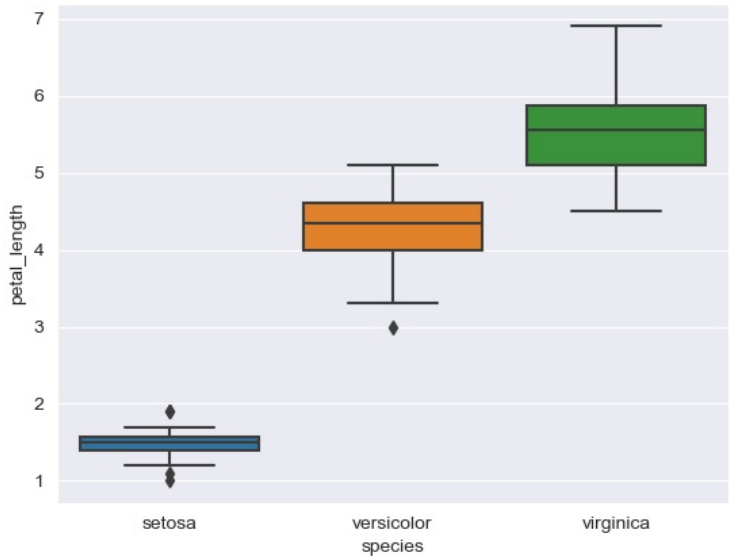
```
sns.distplot(iris.petal_length)
plt.show()
```



Obr. č. 22 – Histogram v *Seaborn*

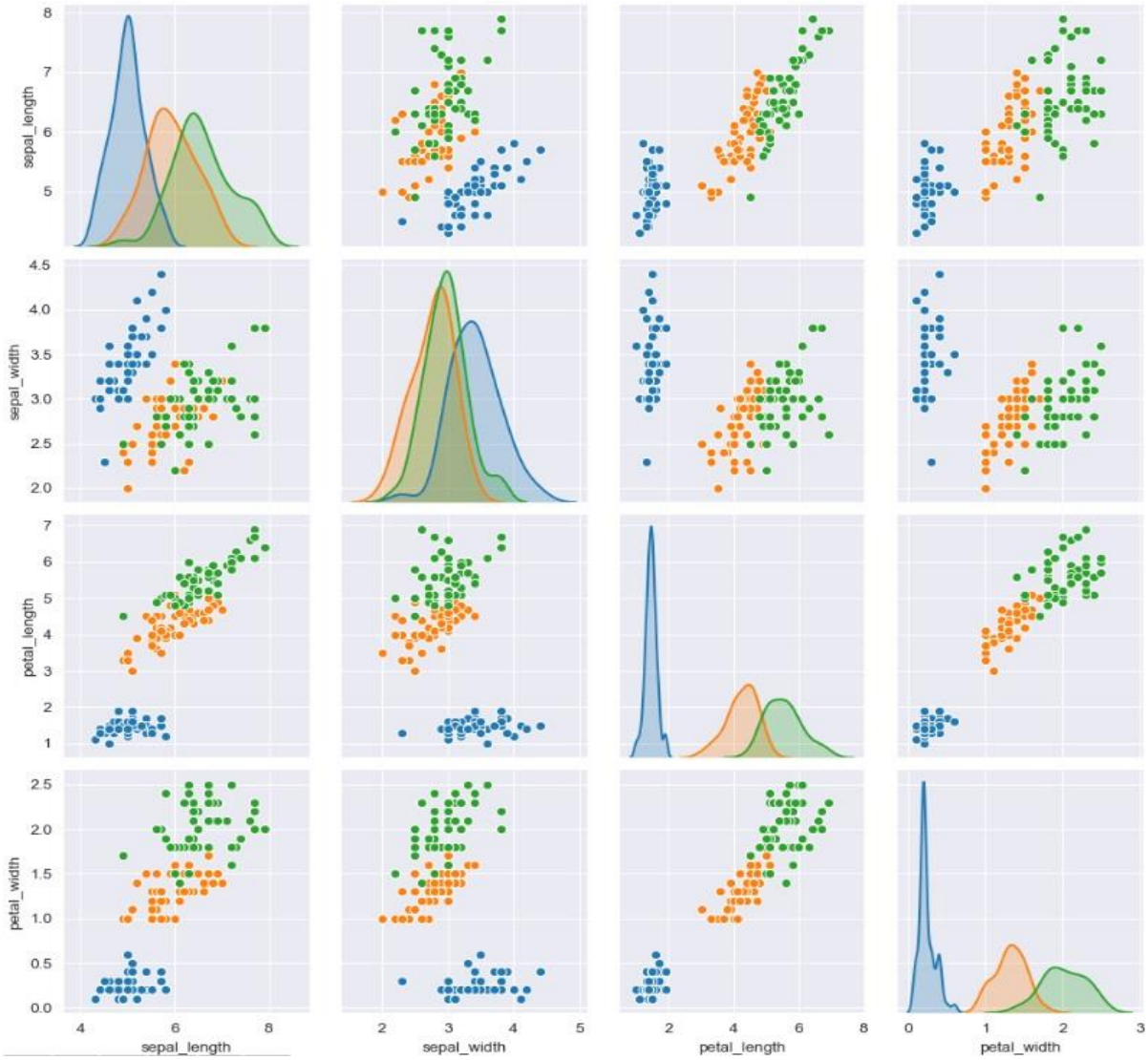
Velice oblíbeným a častým grafem pro deskriptivní statistiku je boxplot neboli krabicový graf, který zobrazuje numerická data pomocí kvartilů. „Krabicová“ část je shora ohraničena 3. kvartilem, zespodu 1. kvartilem („vousky“) a uprostřed se nachází linie mediánu.

```
sns.boxplot(data=iris, x="species", y="petal_length")
plt.show()
```



Obr. č. 23 – Boxplot

Dále existuje ještě např. swarmplot (kombinace histogramu s rozptýleným grafem), violinplot (houslový graf, variace boxplotu), pairplot (ukazuje vztahy mezi všemi proměnnými, což může být velmi užitečné! (Obr. č. 24) nebo heatmap (teplotní mapa).



Obr. č. 24 – Pairplot

To by bylo pro základní seznámení s knihovnou *Matplotlib* vše. Na další praktické využití se podíváme v jedné ze závěrečných kapitol, kde budeme pracovat s dostupnými genomovými daty, na čemž se dají stavět vědecké práce.

9. RegEx

Regular expression neboli regulérní výraz je sekvence symbolů a znaků, pomocí které lze v textu vyhledat specifický řetězec nebo jakýkoliv vzorec. Jde o podobný koncept, jako když si ve Wordu zapneme funkci „Najít a nahradit“ a potřebovali bychom nahradit např. každé „by jsme“ za správné „bychom“. Znalost regulérních výrazů může přijít velmi vhod. Jen jejich zápis je poněkud komplikovanější. Nic nám však nebrání si stáhnout nějaký tahák (cheat sheet), nebát se ho používat a hlavně obrnit se trpělivostí. Na rozdíl od všeho, co jsme zatím v Pythonu používali, nejsou totiž regexy tak dobře čitelné.

Vrhneme se tedy rovnou na znaky, které budeme používat:

\w+

\ – představuje tzv. escape character neboli únikový znak a označuje zápis znaků se speciálním významem, jinými slovy ohraničení výrazu, stejně jako uvozovky pro řetězce

w – „word“, hledáme jakékoliv slovo

+ – kvantifikátor, hledáme jeden nebo více (v tomto případě slovo)

Rovnou si to můžeme vyzkoušet zde: **https://regexr.com/**, kde do horního řádku vložíme náš výraz **„\w+“**, hledáme tedy všechna slova a vyznačil se nám téměř celý text.

Přímo v Pythonu (respektive v Jupyteru) si můžeme výrazy také vyzkoušet, stačí si naimportovat modul **re**:

```
import re

slovo_regex = "\w+"
re.match(slovo_regex,"Bioinformatické centrum HK")
```

Metoda **match()** hledá vzorec v zadaném řetězci, ale vrací objekt, který se nečte dobře. V podstatě tato metoda zkouší vyhledat vzorec pouze na začátku řetězce, zatímco metoda **search()** najde vše, pokud však nic nenajdou, obě vrací *None*. Metoda **re.findall()** vrací seznam, kde se vše požadované v textu objevuje. Metoda **re.split()** vrací seznam, ve kterém byl řetězec při každé shodě rozdělen. Tuto metodu jsme zkoušeli již v podkapitole 4.4., dělá tedy to samé, jen navíc ještě pojme regulérní výrazy. Metoda **re.sub()** nahradí jednu nebo více shod jiným řetězcem (jinými slovy princip „najít a nahradit“).

9.1. Nejběžnější výrazy

Všechny tyto výrazy potřebují mít před sebou zpětné lomítko „\“ pro označení speciálních znaků.

w	Shoda, pokud řetězec obsahuje libovolné slovní znaky (znaky od a do Z, číslice od 0 do 9 a znak podtržítka _).
d	Shoda, pokud řetězec obsahuje číslice (čísla od 0 do 9).
D	Shoda, pokud řetězec NEobsahuje číslice.
s	Shoda, pokud řetězec obsahuje bílé mezery (typicky mezerníkem či tabulátorem).
S	Shoda, pokud řetězec NEobsahuje bílé mezery.
.	Shoda na jakýkoliv znak (tak trochu RegExový žolík)
^	Shoda se začátkem řetězce, v [] však negace (např. [^\w] bude hledat vše kromě slov)
$	Shoda s koncem řetězce
*	Shoda s žádným nebo více opakováním
+	Shoda s jedním nebo více opakováním
|	Shoda včetně alternativ, např. „tramvaj|šalina“
[a-z]	Shoda pro libovolný znak v abecedním pořadí od a do z, malá NEBO velká písmena.
()	Definice skupiny
[]	Definice rozsahu znaků

9.2. Stavební kameny regulérních výrazů a příklady

Tzv. literály jsou v podstatě doslovné shody, kdy se řetězec shoduje. Např.:

```
sekvence = "python"
vzorec = r"python"
if re.match(vzorec, sekvence):
   print("Shoda!")
else:
   print("Žádná shoda!")
```

Pro další řádek se používá klasicky \n. V OS Windows se však můžeme setkat s verzí \r\n. Regulérní výrazy mají tolik speciálních znaků, že se často překrývají. Tudíž psaní jednoduchého úkonu na jeden řádek může ze začátku zabrat spoustu času.

Vyzkoušíme další metodu a zkusíme zahrnout i nějaké znaky:

```
re.search(r"p.thon", "python").group()
```

Hledáme jakékoliv slovo, za nímž po „p“ následuje jakýkoliv znak a pokračuje „thon“ a aby nebyl ve výstupu objekt, ale výsledek, použijeme metodu .group().

```
re.search(r"p\wthon", "python").group()
```

\w nám zde najde shodu s jakýmkoliv písmenem, číslicí nebo podtržítkem. V případě, že použijeme variantu \W, budeme hledat cokoliv, jen ne to, co s malým \w.

S \s zase vyhledáváme bílé mezery:

```
re.search(r"Bioinformatické\scentrum", "Bioinformatické centrum HK").group()
```

Pojďme zkusit čísla:

```
re.search(r"\d\d\d\d", "něco něco něco 3584").group()
```

A přidat další speciální znaky:

```
re.search(r"^Být", "Být či nebýt").group()
```

```
re.search(r"nebýt$", "Být či nebýt").group()
```

Toto se může hodit například pro hledání sekvence, která začíná nebo končí specifickým pořadím (typicky třeba start nebo stop kodony).

Hranaté závorky mohou fungovat podobně jako „|“:

```
re.search(r"[GC]", "G").group()
```

```
re.search(r"Číslo: [0-9]", "Číslo: 8").group()
```

Můžeme zkusit hledat počáteční velké písmeno:

```
re.search(r"\b[AEIOU]", "Abeceda").group()
```

Co když ale chceme najít přímo speciální znak? Tady pomalu začíná zábava:

```
re.search(r"\s", "\s").group()
```

Problém je jednoduše řešitelný:

```
re.search(r"\\s", "\s").group()
```

Avšak výstup nás upozorní, že hledáme speciální znak.

9.3. Opakování

Pokud hledáme více znaků, můžeme použít znaky pro vyznačení, že nějaký znak se může vyskytovat několikrát, místo opakování stejného speciálního znaku. Pro tyto účely slouží tedy + a *. V minulé podkapitole jsme např. hledali 4 čísla a 4x napsali "\d". Určitě to jde udělat elegantněji:

```
re.search(r"\d+", "3584").group()
```

Zmíněné speciální znaky jsou tzv. „greedy“, v překladu chamtivé, protože budou odpovídat co největší části hledaného řetězce. Např.:

```
re.search(r"la*", "laaa laa la la la laaaa l").group()
```

Chamtivost ale není pěkná vlastnost a můžeme si nastavit shodu pouze na počáteční část vzorce. Pomocí "*?" najdeme shodu s co nejmenším množstvím textu, jak jen to je možné. Samotný otazník už je opět chamtivější a hledá shodu přesně pro žádný nebo pouze jeden znak.

```
re.search(r"la*?", "laaa laa la la la laaaa l").group()
```

Případy, kdy chceme zjistit přesný počet, kolikrát se vzorec opakuje, dokáží regulérní výrazy rovněž pokrýt. Zkusíme opět hledat čísla a určit, že chceme přesně čtyři:

```
re.search(r"\d{4}", "3584").group()
```

Co když přesně nevíme, kolikrát se vzorec opakuje, ale víme, jaký má rozsah (1 až 4 písmena)? K flexibilnějšímu kódu stačí malá úprava:

```
re.search(r"\d{1,4}", "3584").group()
```

Kdyby bylo v hledaném textu pět čísel, zkrátka najde jen první čtyři.

Pokud nevíme, kolik číslic bude, druhou část závorky necháme prázdnou (tedy v překladu „jedna a více“):

```
re.search(r"\d{1,}", "3584").group()
```

Jak bylo již zmíněno výše, metodu .split() jsme již použili na rozdělení řetězce pomocí různých oddělovačů. Tuto metodu lze chytře zkombinovat s regulérními výrazy. Vyzkoušíme si to např. na různých zápisech datumu:

```
d = "2.2.1993, 3-13-1987, 4/23/1979"
dates = re.split(r"[./-]", d)
print(dates)
```

```
['2', '2', '1993, 3', '13', '1987, 4', '23', '1979']
```

Dokonce jde hledat shody i po skupinách, což jsme sice dělali pomocí metody .group(), ale bez čísla v závorkách, takže se nám vracel celý text. Abychom zahrnuli různé části textu, rozdělíme si je do skupin, kdy první bude hledat jeden vzorec, druhá druhý atd. Hledáme např. e-mailovou adresu:

```
email = "Muzete me kontaktovat na adresu ivan.skocdopole@seznam.cz"
match = re.search(r"([\w\.]+)@(\w+\.\w{2})", email)
```

Aby byl kód co nejvíce flexibilní, zahrnuli jsme do první skupiny slova a tečky jako případné oddělovače, poté pevně daný zavináč, a druhá skupina obsahuje další slovo, poté tečku, a nakonec doménu prvního řádu, v případě České republiky to tedy budou dvě slova. Na správný výstup musíme jít trochu sofistikovaněji:

```
if match:
    print(match.group()) #celková shoda
    print(match. group(1)) #uživatelské jméno
    print(match.group(2)) #doména
```

```
ivan.skocdopole@seznam.cz
ivan.skocdopole
seznam.cz
```

Pokud budeme chtít, lze si najít i zobrazit pozice shod pomocí metod .start() a .end(). Hledáme všechna čísla (*digits*), nesmíme však zapomínat, že Python počítá od nuly:

```
kod = "ddd9fgh855d24assnd12ff"
shoda = re.finditer(r"\d", kod)

for m in shoda:
   text = m.group() #text shody
   pos = m.start() #index shody
   print(text + " nalezeno na pozici " + str(pos))
```

```
9 nalezeno na pozici 3
8 nalezeno na pozici 7
5 nalezeno na pozici 8
5 nalezeno na pozici 9
2 nalezeno na pozici 11
4 nalezeno na pozici 12
1 nalezeno na pozici 18
2 nalezeno na pozici 19
```

Poslední a velmi užitečnou metodou je .findall(), jejíž výstup se uloží rovnou do seznamu:

```
vysledek = re.findall(r"\d", kod)
print(vysledek)
```

```
['9', '8', '5', '5', '2', '4', '1', '2']
```

10. Literární a webové zdroje v češtině

Jak bylo řečeno v úvodu této příručky, v češtině dosud neexistuje publikace, která by nabízela seznámení s bioinformatikou, ačkoliv se dají v brzké době očekávat. Při prvních krocích s bioinformatikou je však možnost studijního materiálu v rodném jazyce nanejvýš vhodná. Následující řádky tak nabízejí alespoň obecné programátorské zdroje dostupné v češtině. Těch pomalu ale jistě přibývá.

10.1. Knihy o jazyku Python

Knih o Pythonu v češtině přibývá, tudíž zájemce o bioinformatiku má již široké možnosti, vybudovat si robustní základy programování jejich samostudiem. Zde vybíráme dvě

„klasické“ knihy z nakladatelství Grada (grada.cz), přičemž obě lze zakoupit jako tištěnou knihu stejně jako E-knihu, která je po zaplacení ihned ke stažení:

Začínáme programovat v jazyku PYTHON. Pecinovský R. 2020. Grada

Python Kompletní příručka jazyka pro verzi 3.8. Pecinovský R. 2020. Grada

Další knihu lze jednak zdarma a tedy ihned stáhnout zde https://knihy.nic.cz/ současně je k dispozici u knihkupců v tištěné podobě

Ponořme se do Python(u) 3. Pilgrim M. http://diveintopython3.py.cz/index.html

Za zmínku dále stojí rozsáhlé publikace nakladatelství Computer Press vydaná již v několikrát avšak vždy pouze v tištěné podobě:

Python 3 Výukový kurz. Summerfield M. 2012, 2021. Computer Press

Stejné nakladatelství vydalo již v roce 2008 publikaci

Začínáme programovat v jazyce Python. Harms DD. 2008. Computer Press

10.2. Webové zdroje a online tutoriály

Webových zdrojů k výuce jazyka Python je v češtině již také celá řada.

Začněme stránkou nazvanou „Učíme se programovat v jazyce Python 3“, která je adaptací textu „How to think like a computer scientist – Learning with Python“ přeložená, upravená a doplněná Jaroslavem Kubiasem:

http://howto.py.cz/index.htm

Jde o kurz o 13 kapitolách, s rozsáhlým seznamem literatury v angličtině hned v úvodu a hlavně s úctyhodným množstvím ukázek kódu v Pythonu.

Dalším zajímavým zdrojem je česká stránka věnovaná Pythonu

www.python.cz

kde na stránce python.cz/zacatecnici lze najít velké množství odkazů na další zdroje. Mj. na velmi pěkné materiály k obsáhlému online (zdarma) začátečnickému kurzu od PyLadies

https://naucse.python.cz/course/pyladies/

Na stránkách tohoto kurzu lze také u většiny lekcí stáhnout taháky, např.

https://pyvec.github.io/cheatsheets/basic-functions/basic-functions-cs.pdf

Taháky slouží jako užitečný a rychlý referenční materiál naprosto nepostradatelný v začátcích, kdy člověk ještě syntaxi, metody, funkce a další náležitosti nezná úplně zpaměti a neovládá automaticky. Taháky, tzv. cheat sheets jsou velmi oblíbeným a již etablovaným zdrojem přehledných informací v angličtině, kde jich existuje celá řada k Pythonu obecně, různým specializovaným balíkům, atd. (viz níže).

Další online „cvičebnicí“ Pythonu je např.:

https://www.umimeprogramovat.cz/programovani-v-pythonu

11. Literární a webové zdroje v angličtině

V angličtině existují jak obecné programátorské publikace o jazyku Python, tak specializované na bioinformatiku a práci s velkými bioinformatickými (většinou sekvenačními) daty. Těch obecných programátorských publikací je tolik, že je zde nemá smysl zmiňovat, zejm. díky srovnatelným knihám v češtině (předchozí odstavec). Zde se zaměříme právě na ty specializované, kde je výhodou znát již základy programování – to se snažíme vyložit v této příručce. Co však nemůžeme opomenout a je naprosto základní, neodmyslitelnou a doslova nevyčerpatelnou literaturou, je oficiální dokumentace jazyka Python:

https://docs.python.org/3/ a https://www.python.org/

Pro začínající ale i pokročilé bioinformatiky využívající Python je základní webová „centrála“ stránka Martina Jonese, autora nyní již proslulé publikace PYTHON FOR BIOLOGISTS zde https://pythonforbiologists.com/ . Na těchto stránkách jsou tutoriály a cvičení k nyní již čtyřem knihám od tohoto autora (všechny publikace lze zakoupit i jako E-knihu na výše uvedeném webu):

1) Python for Biologists: A complete programming course for beginners. Jones M. 2013, Createspace Independent Publishing Platform

2) Effective Python Development for Biologists: Tools and techniques for building biological programs. Jones M. 2016. Createspace Independent Publishing Platform

3) Advanced Python for Biologists. Jones M. 2014. Createspace Independent Publishing Platform

4) Biological Data Exploration with Python, Pandas and Seaborn. Clean, filter, reshape and visualize complex biological datasets using the scientific Python stack. Jones M., 2020. Createspace Independent Publishing Platform

Se základními znalostmi jak obecného programování tak bioinformatiky se nám otevírá možnost studovat oficiální dokumentaci specializované bioinformaticky zaměřené distribuce Pythonu – BioPython:

https://biopython.org/

Další vhodné bioinformatické knihy v angličtině zaměřené na Python jsou např.:
Reproducible Bioinformatics with Python - How to Write Flexible, Documented, Tested Python Code for Research Computing. Youens-Clark K. 2021. O´Reilly, Inc. USA

Bioinformatics with Python Cookbook, Second Edition. Tiago A. 2018. Packt Pub

Python Programming for Biology. Bioinformatics and Beyond. Stevens TJ. 2015. Cambridge University Press

Bioinformatics Programming in Python. Flaig R-M. 2008. Wiley-VCH Verlag GmbH

Nakonec musíme zmínit oblíbené taháky v angličtině. Jejich množství stoupá stejně jako kvalita jejich grafického provedení. Taháky zde nebude kopírovat, nýbrž uvedeme linky na stránky, kde jsou k nalezení. Jinak stačí zadat do Googlu „Python Cheat Sheet“, event. název balíku či problematiky, kterou chceme pomocí Pythonu řešit (např. statistika, grafy, atd.) a už můžeme vybírat:
https://perso.limsi.fr/pointal/_media/python:cours:mementopython3-english.pdf
https://websitesetup.org/python-cheat-sheet/
https://www.utc.fr/~jlaforet/Suppl/python-cheatsheets.pdf

12. Programátorské projekty

Na závěr si ukážeme několik reálných příkladů využití Pythonu v praxi, tedy už ne pouze cvičný kód.

12.1. Gantt chart

Ganttův diagram je nezbytnou součástí jakéhokoliv vědeckého (ale i technického) projektového návrhu nebo třeba harmonogramu PhD studia. Proto existuje celá řada možností, jak tento diagram mj. i v Pythonu vytvořit.

https://medium.com/geekculture/create-an-advanced-gantt-chart-in-python-f2608a1fd6cc
Podrobný návod lze najít i jako videotutoriál na youtube:
https://www.youtube.com/watch?v=D5GUpd8DKjg&ab_channel=1Mviews
Python má svou vlastní třídu k tvorbě Ganttova diagramu https://pypi.org/project/python-gantt/
Užitečný tutoriál je např. zde:
https://plotly.com/python/gantt/#:~:text=A%20Gantt%20chart%20is%20a,the%20duration%20of%20each%20activity.

12.2. A comprehensive exploratory data analysis with 3 lines of code in Python

```
pip install pandas-profiling
```

```
df = pd.read_csv('my_data_file.csv',index_col=0)
```

```
from pandas_profiling import ProfileReport
report = ProfileReport(df)
report
```

12.3. Příklady využití genomových dat pro analýzy

Moderní doba nabízí online genomové databáze, které jsou zdarma přístupné pro kohokoliv, ať už studenty, vědecké pracovníky, či širokou veřejnost. Obsahují genomická data nejrůznějších organismů od bakterií po živočichy včetně člověka. Sekvence DNA jsou do těchto databází poskytovány různými autory a projekty z celého světa. Analýza takovýchto dat pomocí

Pythonu je nejvhodnější a nejjednodušší cestou, proto se velmi hodí jej umět, pokud bychom si zkusili otevřít jediný takový soubor obsahující pouze kódující část DNA sekvence ve Wordu, zabralo by to více než 10 tisíc stran.

Pro analýzu většího množství genomových dat byl vytvořen Ing. Dominikem Matoulkem nástroj dna_puller volně dostupný zde: https://github.com/bioinfohk/evangelist/tree/master/dna_puller. Z databáze *Ensembl.org* dokáže stáhnout požadované sekvence, např. DNA, cDNA a cds, přes protokol FTP (file transfer protocol), což je starší a jednoduchá technologie s nezašifrovaným přenosem dat mezi koncovými stanicemi pomocí sítě. Zabalený soubor se poté rozbalí a ideálně konvertuje do formátu *.csv (comma-separated values, hodnoty oddělené čárkami). S takto připraveným souborem dokáže dobře pracovat nám již známá knihovna Pandas. Ensembl se několikrát ročně aktualizuje a je záhodno vědět, z jaké verze pochází zpracovaná data a současně stahovat aktuální verze (release). Stahovací nástroj si je třeba stáhnout k sobě do zvolené složky a přímo v ní si v Jupyter notebooku otevřít nový, příslušně pojmenovaný notebook.

Nejprve si ukážeme analýzu kódující sekvence (cds = „coding sequence“) pro jeden druh, poté hromadnou analýzu za pomoci dna_pulleru. Naimportujeme si tedy potřebné balíčky:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import gzip
from ftplib import FTP
from Bio import SeqIO
from collections import Counter
```

Modul gzip poskytuje jednoduché rozhraní pro kompresi a dekompresi souborů s koncovkou *.gz, které se na rozdíl od souborů typu *.rar apod. vyznačují značnou kompaktností. Pomocí modulu ftplib se lze napojit na úložiště serveru a poté stáhnout požadované soubory. SeqIO z BioPythonu představuje rozhraní Sequence Input/Output pro manipulaci se sekvencemi a umožňuje čtení, zápis a indexování se soubory ve formátu *.fa apod. Modul Counter vytváří prostředí pro výpočet jednotlivých prvků ve slovníku. Přihlásíme se prvně na FTP server Ensemblu:

```
ftp = FTP('ftp.ensembl.org')
ftp.login()
```

Pak potřebujeme upřesnit serveru, ze které složky a který soubor chceme stáhnout:

```
ftp.cwd('/pub/current_fasta/myotis_lucifugus/cds')
```

Soubor stahujeme jako binární kód a zapisujeme do *.fa:

```
with open('file.fa.gz', 'wb') as file:
    ftp.retrbinary('RETRMyotis_lucifugus.Myoluc2.0.cds.all.fa.gz',file.write)

handle = gzip.open('file.fa.gz')
with open('myo_luc _cds.fa', 'wb') as out:
    for line in handle:
        out.write(line)
```

BioPython má metodu to_dict, která z iterátoru nebo seznamu vytvoří rovnou slovník. Z něj pak můžeme celou sekvenci uložit do formátu DataFrame:

```
record_dict = SeqIO.to_dict(SeqIO.parse("myo_luc_cds.fa", "fasta"))
record_dict = {record_id: Counter(record_seq) forrecord_id, record_seq in record_dict.items()}
df = pd.DataFrame.from_dict(record_dict, orient='index')
```

V rámci průběžné kontroly je záhodno se podívat, jestli se nám vše naimportovalo a zapsalo správně:

```
df.head()
```

Tab. 5 – Prvních pět řádků DataFrame

	A	T	G	C	N
ESMLUT00000027040.1	191	190	296	277	NaN
ESMLUT00000030196.1	258	242	290	428	NaN
ESMLUT00000001598.2	376	317	491	613	NaN
ESMLUT00000007188.2	134	126	92	137	NaN
ESMLUT00000012029.2	405	454	719	681	NaN

Místa, kde se objevuje NaN, nahradíme nulou, protože pandas potřebuje pracovat s numerickými daty:

```
df = df.fillna(0)
```

Pro vypočtení GC%:

```
df['cds GC%'] = (df['G'] + df['C']) / (df['A'] + df['G'] + df['C'] + df['T'] - df['N'])
```

Tab. 6 – Prvních pět řádků DataFrame s výpočtem GC%

	A	T	G	C	N	cds GC%
ESMLUT00000027040.1	191	190	296	277.0	0.0	0.60
ESMLUT00000030196.1	258	242	290	428.0	0.0	0.58
ESMLUT00000001598.2	376	317	491	613.0	0.0	0.61
ESMLUT00000007188.2	134	126	92	137.0	0.0	0.46
ESMLUT00000012029.2	405	454	719	681.0	0.0	0.61

Pro základní statistické vyhodnocení:

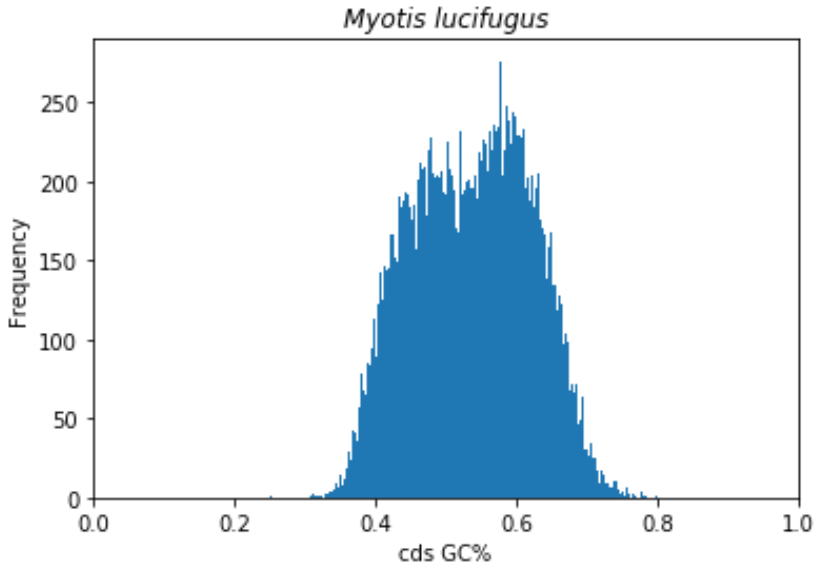
```
df.describe()
```

Tab. 7 – Základní statistika GC% DataFrame

	A	G	C	T	N	cds GC%
count	44934	44934	44934	44934	44934	44934
mean	514.31	521.43	511.78	427.63	0.07	0.52
std	615.83	529.79	512.06	477.03	5.78	0.1
min	9	4	7	7	0	0.25
25%	205	226	225	186	0	0.46
50%	354	388	380	311	0	0.52
75%	637	644	632.75	516	0	0.59
max	33038	24087	21760	24051	1000	12

S takto připraveným DataFrame si již můžeme zobrazit histogram:

```
df['cds GC percentage'].plot.hist(bins=200)
plt.xlim([0, 1])
plt.xlabel("cds GC%")
plt.title("Myotis lucifugus", style="italic")
```



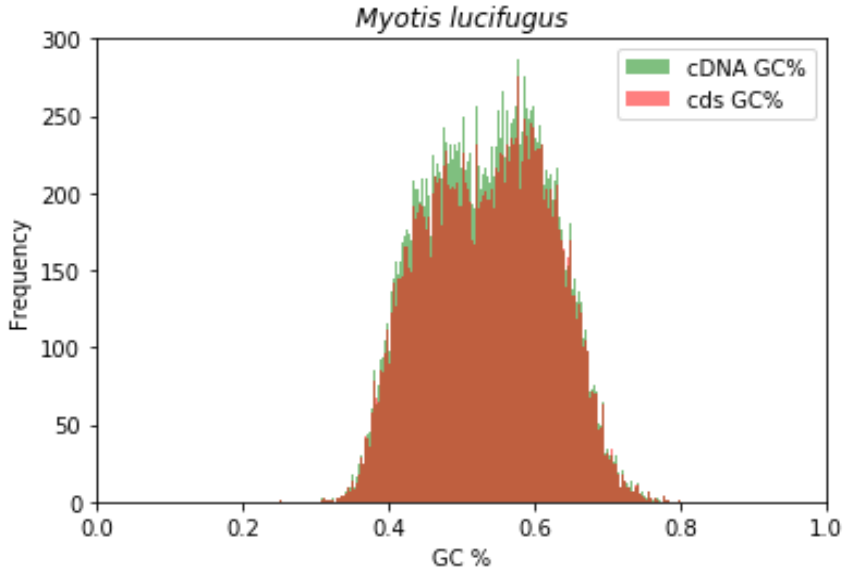
3

Obr. č. 25 - *Histogram frekvence s GC% cds* Myotis lucifugus

Od dusíkatých bází odečítáme ještě „N“, které značí neznámé báze. U některých druhů se jich vyskytuje méně, u některých více, záleží na kvalitě dostupných dat, která se mohou v těchto poměrech i výrazně lišit. Stejný postup uplatníme i při psaní kódu pro cDNA (i celkové DNA). Pro lepší představu a praktičnost lze proložit hodnoty cds i cDNA do jednoho grafu, je však potřeba importovat další knihovnu – NumPy (https://numpy.org/, verze 1.18.1). Tato knihovna disponuje funkcemi pro práci s vícerozměrnými poli, maticemi, vektory apod. Kód by mohl vypadat následovně:

```
import numpy as np


x = df1['cds GC%].plot.hist(bins=200, alpha=0.5, color="red")
y = df2['cdna GC%'].plot.hist(bins=200, alpha=0.5, color="green")
bins = np.linspace(0, 1)
plt.xlim([0, 1])
plt.legend(loc='upper right')
plt.ylabel('Frequency')
plt.xlabel('GC %')
plt.title("Myotis lucifugus", style="italic")
plt.show()
```

Pokud se nám mají v grafu nějaká data prolínat, je vhodné nastavit větší průhlednost (alpha) a především barvy – ideálně barvy ne vedle sebe stojící v barevném spektru, v našem případě červená a zelená.



Obr. č. 26 - Histogram s překrývajícími hodnotami cds a cDNA

Pro větší objem dat využijeme nástroj dna_puller. Nutno říci, že tento nástroj nemusí fungovat v JN na OS Windows, výstupy zde zobrazené probíhaly zejména na OS Linux Ubuntu 20.04 LTS. Před spuštěním kódu je nutné si vedle složky s nástrojem dna_puller také vytvořit ještě složku „jsons“. Nástroj funguje tím způsobem, že po zadání požadovaných druhů organismů a typů sekvencí se stáhnou zabalené soubory FASTA jeden po druhém a ukládají do slovníku ve formátu *.json. JSON je zkratkou pro „JavaScript Object Notation“ a reprezentuje univerzální způsob zápisu dat, se kterým je následně možné pracovat v jakémkoli programovacím jazyku. Při stahování dalšího balíku se souborem FASTA se předchozí balík smaže, aby zbytečně nezabíral místo na disku, jelikož nabývají velikostí i několik Gb.

Např. potřebujeme analyzovat zástupce nadřádu Afrotheria a jejich cds a cDNA:

```
import dna_puller.dna_puller as puller


species = ['Loxodonta_africana', "Procavia_capensis", "Echinops_telfairi"]
puller = puller.DnaPuller(species, True, True, ['cds', 'cdna'])
puller.download_and_parse_data()
```

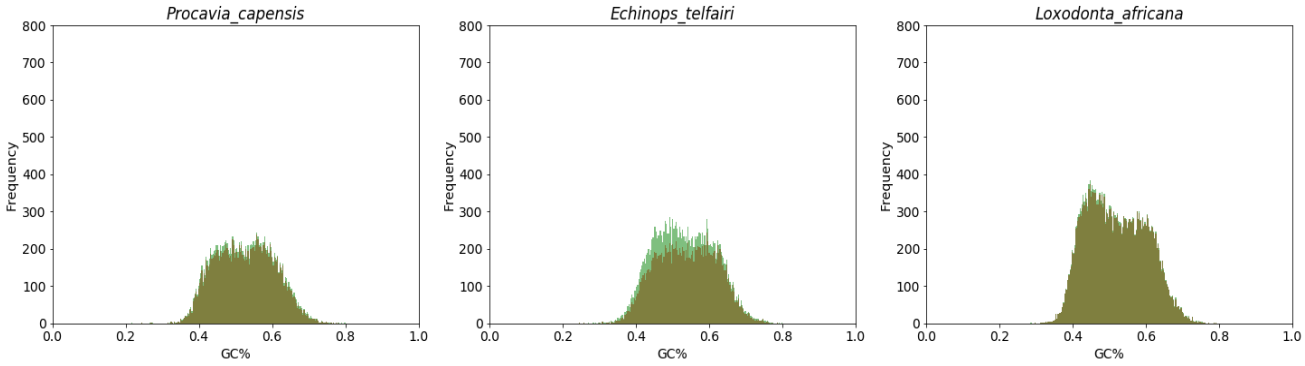
Takto máme připraveny soubory pro hromadnou analýzu. Všechna data potřebujeme sloučit, abychom nad nimi mohli iterovat (opakovat proces v měnícím se kontextu), v našem případě GC% u cds a cDNA:

```
import os, json

species_files = os.listdir('jsons')

dna_types = ['cdna', 'cds']

files = []

for species_file in species_files:
    if species_file[-5:] == '.json':
            files.append(species_file)

species_data = {}

for species_file in files:
        with open('jsons/' + species_file) as file:
                species_name = species_file[0:-5]
                species_data[species_name] = {}
                for type in dna_types:
                   species_data[species_name][type] = {}
                data = json.load(file)
                for type in dna_types:
                   for gene_key, gene_data in data[type].items():
                species_data[species_name][type][gene_key] = gene_data

aggregated_data = {}
for type in dna_types:
   aggregated_data[type] = {}
   for species_key, datas in species_data.items():
            aggregated_data[type][species_key] = []
   for key, data in datas[type].items():
            all_count = float(data['all']) - float(data['N'])
           aggregated_data[type][species_key].append(float(data['G'] + data['C'])
```

```
        /all_count)
```

Nyní už jen potřebujeme umístit vše do jednoho souboru. Rozměry a rozmístění grafů upravíme dle obsahu a potřeb a uložíme ve formátu *.png:

```
import matplotlib.pyplot as plt

fig, axs = plt.subplots(2, 3, figsize=(30,50))

for index in range(0, len(aggregated_data['cdna'])):
    species_key = list(aggregated_data['cdna'].keys())[index]
    y = index % 3
    x = int(index / 3)
    axs[x, y].set_title(species_key[0].upper() + species_key[1:],
    style='italic')
    axs[x, y].set_ylim(0, 500)
    axs[x, y].set_xlim(0, 1)

cdna_data = aggregated_data['cdna'][species_key]
cds_data = aggregated_data['cds'][species_key]

axs[x, y].hist(cds_data, bins=200, range=(0.2, 0.8), color='red', alpha=0.5)
axs[x, y].hist(cdna_data, bins=200, range=(0.2, 0.8), color='green', alpha=0.5)

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



Obr. č. 27 - Histogramy GC% cds a cDNA skupiny Afrotheria