
nofy056

T. L.

12.01.2024

Contents:

1	Úvod	3
1.1	Počítače pro vědecké výpočty	3
1.2	Počítačové jazyky, kompilátor a interpret	4
1.3	Jazyk Python	5
1.4	Euklidův algoritmus	6
1.5	Jak vypadá program v Pythonu	6
1.6	Úvodní rozjímání nad kódem v jazyce Python	7
1.7	Spouštíme kód v Pythonu	8
1.8	Kontrolní úlohy	11
2	Spouštění programů v Pythonu	13
2.1	Spouštění *.py na příkazové řádce	13
2.2	Spouštění *.py ve IDE	14
2.3	Spouštění *.py v Jupyteru	15
3	Proměnné, výrazy, základní matematické funkce	17
3.1	Vyčíslení vzorečku	17
3.2	Proměnné	18
3.3	Zápis čísel	19
3.4	Pojmenované konstanty	19
3.5	Aritmetické výrazy	20
3.6	Logické výrazy	20
3.7	Priorita aritmetických operací	21
3.8	Průběh vyhodnocování výrazů	22
3.9	Neúplné vyhodnocování logických výrazů	23
3.10	Operandy	23
3.11	Knihovna math	24
3.12	Zabudované funkce	26
3.13	Kontrolní úlohy	27
4	Příkazy	29
4.1	Příkazy a jejich posloupnosti	29
4.2	Přiřazovací příkaz	30
4.3	Příkaz import	32
4.4	Příkaz print a volání podprogramu	33
4.5	Větvení - podmíněný příkaz if	33
4.6	Cyklus while	34

4.7	Cyklus for	35
4.8	Příkaz skoku break	37
4.9	Indentace (odsazování) kódu	37
4.10	Pokračování řádků	38
4.11	Více příkazů na jeden řádek	38
4.12	Kontrolní úlohy	38
5	Proměnné a typy dat	41
5.1	Proměná: typ + hodnota	42
5.2	Typ výsledku operace je dán typem operandů	43
6	Příkaz print a něco o řetězcích	45
6.1	Příkaz print	45
6.2	Řetězce	46
7	Textové soubory	51
7.1	Výstup textových dat do souboru	51
7.2	Vstup textových dat ze souboru	53
7.3	Přesměrování vstupu a výstupu programu	55
8	Funkce	59
8.1	Definice funkce	60
8.2	Volání funkce	61
8.3	Nepovinné argumenty	61
8.4	Předávání argumentů	63
8.5	Proměnné lokální a globální	63
8.6	Funkce nebo procedura?	65
8.7	Lokální proměnné a procedurální programování	66
8.8	Lokální vs globální proměnné	67
8.9	Životní cyklus proměnných	67
8.10	Příkaz def	67
8.11	Funkce jako argument jiné funkce	68
8.12	Rekurze	70
9	Funkce, co něco počítají	75
9.1	Příklad jednoduché funkce dané vzorečkem	76
9.2	Složitější funkce	77
9.3	Polynomy a Hornerovo schéma	78
9.4	Testujeme funkce	78
9.5	Příklad demonstrující doktrínu strukturovaného programování	79
9.6	Funkce definované rekurentními vztahy	82
9.7	Hledání kořenů funkcí, metoda půlení intervalu	85
9.8	Newtonova metoda hledání kořene (metoda sečen)	87
9.9	Inverzní funkce	89
9.10	Reálná čísla v počítači	90
9.11	Lokalizace chyb	92
10	Pole a seznamy	95
10.1	Eratosthenovo síto	95
10.2	Pole – celek a část. Přístup k prvku pole (a[i])	99
10.3	První položka pole má index 0	99
10.4	Bereme na vědomí typy list a tuple	100
10.5	Unpacking	100
10.6	Pole nebo seznam?	100
10.7	Vytvoření pole (numpy.full a další)	101

10.8	Přiřazovací příkaz s poli	103
10.9	Typy prvků polí v <code>numpy</code>	104
11	Funkce a pole	107
11.1	Argumenty typu pole – vstupní argument (jen pro čtení)	107
11.2	Volání funkcí s argumentem typu pole	109
11.3	Malování obrázků (knihovna <code>matplotlib</code>)	109
11.4	Univerzální funkce a výrazy s poli	110
11.5	Argumenty typu pole – výstup	112
11.6	Globální proměnná v roli statické lokální proměnné	116
11.7	Globální proměnná v roli konstanty	118
11.8	Numerická kvadratura	119
12	Elementární algoritmy pracující s poli	123
12.1	Obsazování polí hodnotami	123
12.2	Vyhledávání v poli	125
12.3	Ověření vlastností hodnot uložených v poli	126
12.4	Hledání optimálního prvku seznamu	127
12.5	Hledání optimálního prvku seznamu s podmínkou	127
12.6	Jedno pole, dva cykly	128
12.7	Bubble-sort	131
12.8	Řezy polí	132
12.9	Stěhujeme prvky pole	134
12.10	Polynomy jako pole koeficientů	138
13	Matice	141
13.1	<code>numpy</code> - vytvoření matice	141
13.2	Vytvoření kopie	145
13.3	Řezy matic	146
13.4	Monte Carlo	146
13.5	Stručně o modulech	149
14	Řetězce II	151
14.1	Řetězce vs. pole znaků	152
14.2	Vybrané metody typu <code>str</code>	153
15	Pole a výpočty lineární algebry	159
15.1	2D a 3D vektory	160
15.2	Matice a vektory	162
15.3	Gaussova-Jordanova eliminace	164
16	Časová náročnost algoritmů	169
17	Vybrané numerické algoritmy	175
17.1	Přehled doposud probraných numerických témat	175
17.2	Polynomiální interpolace	176
17.3	Polynomiální regrese	180
17.4	Optimalizace (jako příklad použití knihovního algoritmu)	182
17.5	Numerické derivování	183
17.6	Řešení obyčejných diferenciálních rovnic	185
18	Další datové typy	197
18.1	Asociativní pole (<code>dict</code>)	197
18.2	Složené datové typy	199
18.3	Příklad na použití vlastní datové struktury	203

18.4	Rozbalování polí představujících struktury	207
18.5	Dynamické datové struktury	207
19	Jak zrychlit výpočet	213
19.1	Urychlení běhu programů Pythonu s použitím knihovny <code>numba</code>	214
19.2	Monte Carlo II (ukázka použití <code>numba.njit</code>)	215
19.3	QuickSort (ukázka chytrého/rychlého algoritmu třídění)	216

Programování pro fyziky (Podklady pro přednášku) =====

1.1 Počítače pro vědecké výpočty

Potřeba delších numerických výpočtů provází fyziku od jejího vzniku. Rozvoj techniky ve druhé polovině 20. století vedl k exponenciálnímu růstu možností počítačů a schopnosti i těch nejobyčejnějších dnes velmi překračují schopnosti člověka počítačům "rozumět". Je ale žádoucí, aby právě student fyziky dokázal počítač užívat ku svému prospěchu. Zdá se, že nejnovější výdobytky oboru počítačů shrnuté pod štítkem umělá inteligence (AI) tomu budou značně nápomocny. Ve své současné podobě představuje AI nástroj extrapolující dostupné znalosti a používá tedy jazyka v němž jsou tyto znalosti formulovány. V problému vědeckotechnických výpočtů tedy dnes používá ve svých odpovědích metody toho, čemu říkáme Programování. Proto bude i cílem tohoto kurzu seznámit studenty se základy programování elementárních výpočtů na počítačích tak, aby pak ve své kariéře dokázali využít počítače i za pomoci přicházejících revolučních prostředků práce s nimi.

Nejprve shrňme základní pojmy, se kterými se budeme potkávat a které jsou vám jistě známy.

Počítač je dnes zařízení vykonávající požadované úkoly na základě povelů. Automatické vykonávání zajišťuje **procesor** (řídící jednotka počítače, CPU), který řídí operace s jednotlivými částmi počítače. Povely jsou uloženy v elektronické formě v podobě tzv. **instrukcí** procesoru. Tyto instrukce jsou typicky elementární operace, které dokáže CPU vykonat v co nejkratším čase. Jde o číselné kódy přináležející různým operacím, konkrétní podoba závisí na druhu CPU. (Vývoj procesorů je velmi náročný a tak jsou architektury CPU většinou pojmenovány po komerčních entitách jako je Intel/AMD, ARM, ...).

Pro uložení povelů pro práci počítače stejně tak jako dat, se kterými počítač pracuje, slouží počítačová **paměť**. Vybrané části paměti mohou být určeny je ke čtení počítače (read-only memory - ROM), univerzálnost použití počítačů ale vyžaduje aby významná část paměti mohla sloužit jak k zapisování tak čtení (RAM). Vstup a výstup dat obstarávají vám známá zařízení, jako jsou klávesnice, displeje, tiskárny...

Z ekonomických důvodů, kvůli možnosti zachování dat při výpadku napájení počítače ale i např. transportu dat je část paměti realizována odlišnou technologií (disky magneto-mechanické a polovodičové, DVD-ROM, ...). Organizaci takto uložených dat složí souborové systémy, které daný umožňují danému souboru dat přiřadit jméno a uspořádat takové soubory v rámci hierarchické struktury (adresar/podadresar/podpodadresar/.../jméno.souboru.pripona). V OS Windows je na začátku uveden ještě logický disk označený jedním písmenem a dvojtečkou, tedy máme cestu např. C:\Users\student, navíc zde u jména souboru nezáleží na volbě velké/malé písmeno.

Operace se vykonávají s údaji, které mohou mít rozličnou podobu a většinou budeme považovat za interní záležitost, jak jsou data v počítači uložena. Přesto musíme tušit, že základem informací s nimiž dnešní počítače zacházejí je **bit** - hodnota 0 nebo 1. Z bitů jsou složeny větší celky určené k ukládání celých čísel, buňka paměti o velikosti n bitů může nabývat 2^n různých stavů. Praktickou jednotkou informace se stala osmice bitů, tzv **byte**, protože 256 možnými hodnotami lze pohodlně pokrýt základní sadu znaků anglické abecedy a nejdůležitějších symbolů (+* , :@ atd.). Aby počítače pracovaly rychleji, pracují najednou se slovem nikoli bytem, přičemž šířka slova v PC bývala 16, 32, dnes je to 64 bitů. Čísla uvnitř počítačů jsou kvůli rychlosti uložena v binárním tvaru, nikoli pomocí cifer dekadického zápisu.

Vysoký výkon elektronických komunikací umožňuje mít výkonný počítač (**server**) a jeho uživatele velmi daleko. To poskytuje různé výhody. Je k tomu nutné projít oba prostřednictvím **počítačové sítě**. Uživatelův (obvykle méně výkonný) počítač je vybaven SW (**client**) zprostředkujícím komunikaci v požadované formě. Příkladem klienta je webový prohlížeč, jde mimochodem o velmi komplikovaný program. Jednoduchým klientem je např. terminál, který se tváří, jako bychom seděli u elektrického psacího stroje napojeného na vzdálený počítač. V oblasti vědeckých výpočtů jde o častý způsob práce, přičemž přes terminál jsou spouštěny i ty nejdůležitější (a nejdražší) výpočty.

Servery mohou být doplněny o velkokapacitní úložiště data, provozovány ve velkém počtu a přidělovány v různém počtu podle aktuálních požadavků (služby typu **cloud**).

Naivní měřítko výkonu počítačů je počet aritmetických operací za sekundu a množství paměti. Aktuálně jde u jednoho počítače typicky o miliardy číselných operací za sekundu (giga-FLOPS) a desítky miliard bytů. Jsou-li potřeba vyšší hodnoty, řeší se to dnes tak, že

- v součástce stále označované jako CPU je obsaženo více (jednotky až stovky) **jader** (což je označení pro jednotlivé CPU),
- jádro dokáže najednou provádět 4-8 stejných aritmetických operací,
- součástí počítače může být přídatný HW obsahující i tisíce specializovanějších výpočetních jader (typicky tzv. grafická karta, GPU),
- více počítačů ($10 - 10^4$) řeší problém souběžně, data komunikují přes vysoce propustnou datovou síť.

Sekvence instrukcí CPU uložených v pořadí, v jakém se mají vykonávat, tvoří hlavní součást tzv. **spustitelného souboru**. Takové soubory jsou pak načteny do paměti a procesoru je přikázáno začít vykonávat instrukce od určeného místa. Toto zařizuje rozsáhlý balík spustitelných programů známý pod jménem **operační systém** (OS). Jinou částí operačního systému jsou programy převádějící přání uživatele počítače zadaná prostřednictvím **klávesnice** a **myši** do podoby akcí (spuštění programu, předání kódu klávesy běžícímu programu např. textovému editoru, ukončení běžícího programu, ...). Důležitou prací operačního systému je **sdílení** CPU a dalších omezených prostředků počítače mezi jednotlivými programy a uživateli. Z bezpečnostních důvodů musí OS uživateli některé činnosti zakázat a také izolovat navzájem programy spuštěné různými uživateli. Součástí OS jsou dnes grafická uživatelská rozhraní (GUI) poskytující efektivní představu programů jako okének na pracovní ploše ovládaných myší a klávesnicí.

Kromě spustitelných souborů pracujeme i s dalšími soubory. Zejména jde o data (záznamy měření, fotografie, audiozáznam), texty (kancelářské, vědecké, ...) a textový zápis kódů programů.

1.2 Počítačové jazyky, kompilátor a interpret

Řídit běh počítač přímo prostřednictvím instrukcí CPU je pracné a nepraktické, spustitelné programy tak dnes vytváří **kompilátor** překladem **zdrojového kódu programu**. Ten vyjadřuje návod, jakým má činnost počítače postupovat prostřednictvím člověku i stroji srozumitelného počítačového jazyka. Podoba jazyka bývá uzpůsobena řešenému problému. Příklady jazyků tohoto druhu jsou:

- **FORTRAN** vznikl v samých počátcích počítačů jako prostředek pro zápis vědeckotechnických výpočtů.
- **C** vznikl jako jazyk pro vývoj operačního systému UNIX, stal se ale i jazykem pro vědecké výpočty.
- **Pascal** měl podporovat "správné" postupy při psaní programů a ve výuce programování.
- **C++** rozšířil jazyk C o mnoho moderních prvků podporujících jak efektivní tak i bezpečnější psaní kódu.

Dosáhnutí výsledku v této variantě vyžaduje tři kroky

1. Napsání zdrojového kódu programu ve vybraném počítačovém jazyce.
2. Přeložení prostřednictvím kompilátoru (vytvoří spustitelný soubor)
3. Spuštění tohoto souboru

Například kód zapsaný v daném programovacím jazyce C pojmenovaný `hello.c`

```
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
}
```

Ten je kompilací převeden do podoby spustitelného programu, tedy např. `hello.exe`. Tento program je pak potřeba spustit. Pověšme si, že často používáme totéž slovo program pro kód i spustitelný soubor.

V terminálu linuxového počítače vypadají tyto operace takto

```
tomas@dronte:~/tmp/prog$ gcc -o hello.exe hello.c
tomas@dronte:~/tmp/prog$ ls -l hello*
-rw-rw-r-- 1 tomas tomas   69 may  1 12:18 hello.c
-rwxrwxr-x 1 tomas tomas 15960 may  1 12:19 hello.exe
tomas@dronte:~/tmp/prog$ ./hello.exe
hello, world
tomas@dronte:~/tmp/prog$
```

Příkaz `gcc` je kompilátor jazyka C, příkaz `ls` vypíše všechny soubory začínající na "hello" a ukáže, že zatímco zdrojový kód má pouhých 69 bytů (znaků), spustitelný soubor je mnohem větší. Spuštění programu má podobu napsání jeho jména z bezpečnostních důvodů musí být na začátku tečka a lomítko. Následně se výstup "hello, world" se také objeví na terminálu.

Místo kompilátoru můžeme použít program, tzv. **interpret**, který zdrojový kód přeloží i provede. V principu by mohlo jít jen o zkratku dvou kroků (kompilace+spuštění) do jednoho, ale nabízí se i možnost, že ono provádění probíhá postupně, jak jsou interpretu příkazy zadávány uživatelem. Interpret tedy umožňuje interaktivní vykonávání a i psaní kódu programu. Příklady jazyků tohoto druhu jsou mj. Python jako jazyk pro všeobecné použití a Wolfram Mathematica specializovaný na matematicko-fyzikální výpočty.

1.3 Jazyk Python

Jazyk **Python** verze 3, který budeme v kurzu používat pro psaní programů, spadá do této kategorie interpretovaných jazyků.

Je to jazyk, který je dnes na čele seznamu počítačových jazyků, které profesionální fyzikové používají jako každodenní nástroje k řešení problémů. Jeho použití ve vědeckých výpočtech usnadňuje to, že vzniklo mnoho knihoven, které dnes umožňují řešit jednoduché ale praktické problémy za pomoci několika málo řádek kódu.

Současné rychlé počítače umožňují pak tolerovat, že výpočty v Pythonu nemusejí probíhat nejrychleji. Jsou také připraveny cesty, jak toto omezení obejít.

Pro začátečníky však přináší Python ztíženou možnost automatické identifikace chyb, kterých se při psaní kódu dopustí. Většinou se o chybě vyskytující se na nějakém místě dozvíme až při provádění tohoto příkazu.

Představíme si tzv. procedurální přístup k řešení jednoduchých úloh inspirovaných matematikou a fyzikou prvních ročníků studia fyziky na MFF UK. Průběžně se seznámíme s potřebnými prvky jazyka Python, přičemž se nevyhýbáme

ani "modernějším" vlastnostem jazyka, pokud vedou ke kódu, který lépe reflektuje řešený problém. Je třeba vědět, že použitý jazyk nejen přímo určuje podobu tohoto zápisu nějakého výpočtu ale nepřímo ovlivňuje i použitý postup tím, jak daný jazyk některé způsoby řešení usnadňuje a jiné ztěžuje. Protože jde o úvodní kurz budeme při výkladu volit postupy, které v daném jazyce nevypadají nejlépe, ale je potřeba se s nimi seznámit.

Text předpokládá, že v případě nejasností použijete nějakou učebnici jazyka nebo jeho dokumentaci.

1.4 Euklidův algoritmus

Sekvence elementárních kroků, které zaručeně vedou k požadovanému výsledku se nazývá **algoritmus**. Tento pojem má v rámci matematiky svoji formální definici, která pak umožňuje formulovat matematické věty, mj. o tom, co počítač může a co nemůže "spočítat". Pro nás bude algoritmus ale jen slovní zkratkou nahrazující spojení správný postup vedoucí k vyřešení nějakého problému v konečném čase. Jako první si představíme Euklidův algoritmus pro výpočet největšího společného dělitele (NSD) dvou kladných celých čísel.

Euklidův návod říká, že *NSD dvou čísel kladných celých čísel spočteme tak, že opakovaně od většího odečítáme to menší, dokud nedostaneme dvojici stejných čísel*. Operace odečítání nemění NSD a protože takových odečítání může pro danou dvojici čísel nastat jen konečný počet, musí Euklidův návod vést v konečném počtu kroků k požadovanému výsledku.

1.5 Jak vypadá program v Pythonu

Ambicí programátora je takový výpočet nechat provádět počítač. Jak již víme, aby nám byla jeho síla zpřístupněna máme spolu s počítači k dispozici programovací jazyky, kterými svá přání počítačům sdělujeme právě v podobě návodů, co mají počítat. Nacházíme se na začátku času, kdy počítače dokáží porozumět zadání algoritmu tak, jak je uveden výše, a kdy jej pak můžeme požádat, aby napsal program, který Euklidův algoritmus aplikuje na dvojici čísel 4477 a 1073. Protože jde o známý algoritmus nemusíme jej ani popisovat:

Human: I need a program that computes greatest common divisor of two numbers using the Euclidean algorithm. As an example use numbers 4477 and 1073.

AI: Sure! Here's a Python program that computes the greatest common divisor (GCD) of two numbers using the Euclidean algorithm:

```
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

# Example usage
num1 = 4477
num2 = 1073

result = gcd(num1, num2)
print("The GCD of", num1, "and", num2, "is:", result)
```

K tomu AI dodá

When you run this program, it will output:
The GCD of 4477 and 1073 is: 97

In this example, the GCD of 4477 and 1073 is 97, which is the greatest common divisor of the two numbers computed using the Euclidean algorithm.

Zde jsme použili službu <https://chat.openai.com/> ve verzi z jara 2023 zjednodušeně označovanou termínem *umělá inteligence*. Ta automaticky zvolila jako jazyk právě Python. Zatímco kód programu je správně, strojem odhadnutý výsledek výpočtu se správnému výsledku, tedy číslu 37 jen podobá. Takový výpočet jsme ale po našem pomocníkovi nechtěli — spustit takto jednoduchý kód je oproti jeho vymyšlení snadné.

Na složitějších příkladech lze ukázat, že se umělá inteligence může splést i v kódu programu a lze tak předpokládat, že v nadcházející době bude uživatel počítače jako nástroje pro výpočty muset stále používat za tímto účelem programovací jazyk, ovšem kód v tomto programovacím jazyce bude vznikat za stále intenzivnějšího použití nových, doposud nevídaných nástrojů. V oblasti fyziky lze předpokládat, že pro řešení nových problémů v nejbližší době budeme muset stále psát programy.

Je tedy na čase si podrobněji popsat, co takový zápis programu vyjadřuje. Pro naše potřeby má výše uvedený program nevýhodu, že jednak používá rozdělení kód programu na funkci a hlavní program, a také představuje vylepšenou verzi Euklidova algoritmu založenou na operaci zbytku po dělení celých čísel. Proto pro náš další výklad použijeme následující podobu kódu pro výpočet největšího společného dělitele:

```
# program pro Vypocet NSD

a = 4477          # vezmi dvě čísla: 4477
b = 1073         # a 1073

print( 'NSD(' ,a ,',' ,b ,')=' , sep=' ' , end=' ' ) # vypíše:  NSD(4477,1073)=

while not b==a:  # dokud jsou obě čísla různá
    if a>b:      # zjistí, které je větší
        a = a-b # a odečti menší od většího
    else:
        b = b-a

print( a )
```

1.6 Úvodní rozjímání nad kódem v jazyce Python

- Čísla si na začátku asociujeme s písmeny a a b. Jde o tzv. identifikátory a vidíme, že v programu zastupují konkrétní hodnoty. Operace probíhají, jak jsme z algebry zvyklí, s těmito identifikátory, nikoli přímo s čísly.
- Identifikátory a a b označují tzv. *proměnné*.
- Zavedení proměnné má povahu přiřazovacího příkazu (např. ``a = 4477``).
- Na rozdíl od mnoha jiných jazyků v tomto kroku nevyjadřujeme přání, že proměnná je určena k ukládání celých čísel. To proto, že zatím je v Pythonu zvykem se neobtěžovat s uváděním typu u proměnných. Navíc, v tomto jazyce může proměnná za dobu běhu programu nabývat různých typů podle toho, co do ní "přiřadíme". Uvidíme, že podle situace může jít při psaní programu o výhodu i komplikaci. (Porovnejte alternativu v C++, kde i když při deklaraci neuvědeme explicitně typ proměnné, např. `auto x=1.2`, nemůžeme později přiřadit `x="text"`.)
- Kromě přiřazovacího příkazu `promenna = nova_hodnota` jsou v programu i další příkazy. Komunikaci směrem k uživateli programu představuje `print(neco, neco_dalsiho, ...)`. Opakování zařizuje příkaz `while podminka: toto_opakuj`. Rozhodnutí co učinit představuje příkaz `if podminka: co_pak else: co_jinak`.
- Python nemá "programové závorky" jako jsou např. složené závorky `{...}` v C.
- Rozdělení kódu programu na řádky má v kódu klíčový význam protože konce řádků ukončují příkazy (většinou).

- Mezery (tabelátor) na začátku řádků jsou velmi důležité. Určují totiž například, které příkazy jsou součástí těla cyklu `while` a které následují až po jeho ukončení.
- Zatímco přiřazovací příkaz používá jednoduché `=`, porovnání na rovnost používá zdvojené rovnítko `==`.
- Příkaz `print` automaticky vypisuje mezi jednotlivými argumenty mezery, abychom je potlačili použijeme nepovinný pojmenovaný argument `sep=' '`, kterým oznámíme, že mezi argumenty chceme vkládat nic (podobně jako v jiných jazycích `' ', ' ', ' '` představuje zápis žádné, jedné a dvou mezer).
- Příkaz `print` zakončí výpis automatickým přechodem na nový řádek, to potlačíme uvedením `end=' '` jako nepovinného argumentu funkce `print`.
- Komentáře vysvětlující záměr programátora následují po znaku `#`.

1.7 Spouštíme kód v Pythonu

Další velkou výhodou je velký výběr prostředí umožňujících programování v Pythonu. Pro nás to ale znamená, že se musíme seznámit hned s několika metodami, jakými budeme programy v tomto jazyce psát a spouštět.

1.7.1 Python na příkazové řádce

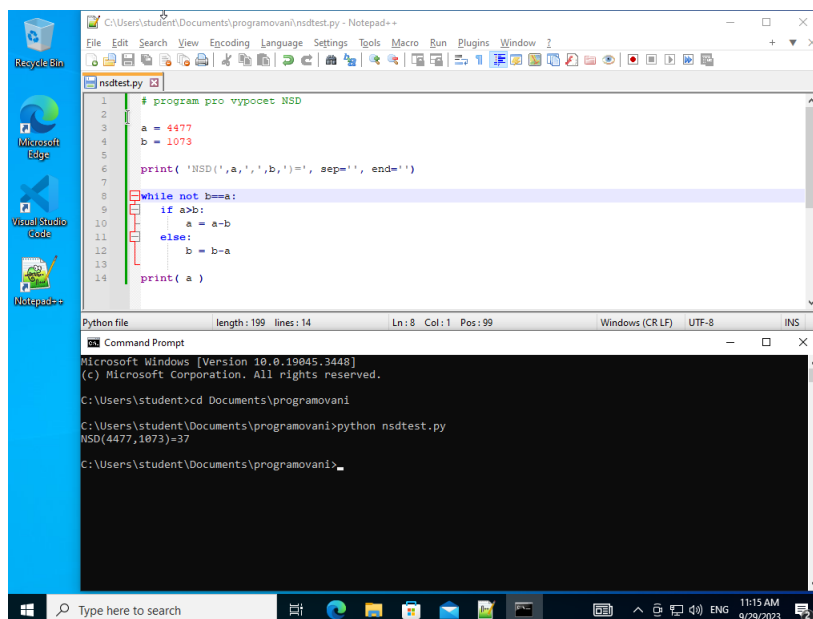
Prvním příkladem bude situace, kdy spouštíme program v prostředí OS Windows 10. Následující obrázek zachycuje pracovní plochu s dvěma spuštěnými programy. Textovým editorem Notepad++ nahoře a tzv. příkazovou řádkou emulující tzv. terminál počítače. Textový editor zobrazuje uvažovaný zdrojový kód zmíněný výše, na příkazové řádce jsou pak vidět dva příkazy. První,

```
cd Documents\programovani
```

říká systému, že chceme změnit běžný adresář (`cd=change directory`). To proto abychom v dalším řádku nemuseli psát podrobně cestu k souboru `C:\Users\student\Documents\programovani\nsdtest.py` se zdrojovým kódem našeho programu. Místo toho napíšeme příkaz, kde uvedeme jen název programu

```
python nsdtest.py
```

Tento příkaz říká - spusť interpret jazyka python a tomu řekni, že má provést příkazy v souboru `nsdtest.py`.



The screenshot shows a Windows desktop environment. A Notepad++ window is open, displaying a Python script named `nsdtest.py`. The code is as follows:

```

1 # program pro vypocet NSD
2
3 a = 4477
4 b = 1073
5
6 print( 'NSD(',a,',',b,')=', sep='', end='')
7
8 while not b==a:
9     if a>b:
10        a = a-b
11    else:
12        b = b-a
13
14 print( a )

```

Below the Notepad++ window, a Command Prompt window is open, showing the execution of the script:

```

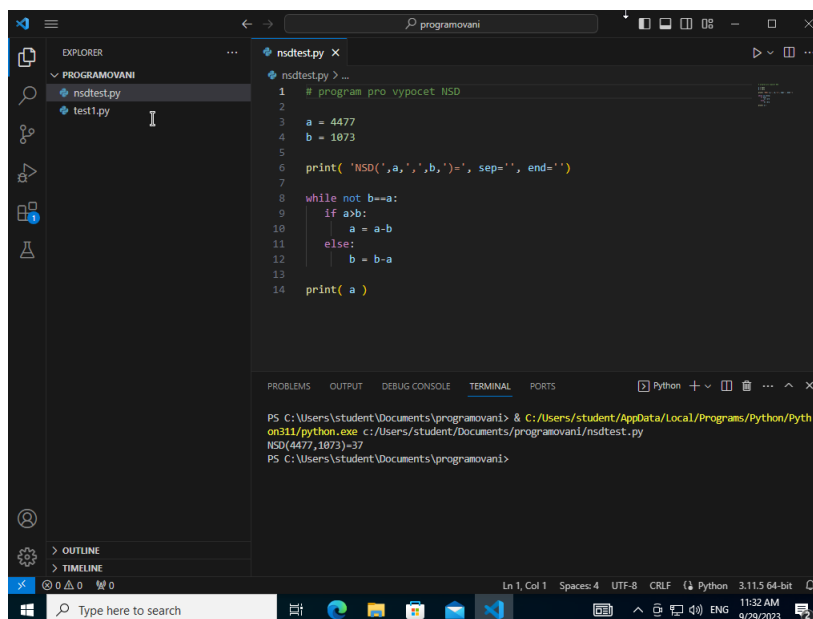
C:\Users\student>cd Documents\programovani
C:\Users\student\Documents\programovani>python nsdtest.py
NSD(4477,1073)=37
C:\Users\student\Documents\programovani>

```

Zde jsme použili samostatné programy na psaní kódu a pro terminál s příkazovou řádkou.

1.7.2 Python v IDE Visual Studio Code

Pro pohodlí programátora je časté tyto spojit do jediného *integrovaného vývojového prostředí* (IDE). Jeho příklad ukazuje následující obrázek. Je na něm v editoru IDE otevřen soubor `nsdtest.py` a tento je pak spuštěn znakem `>` vpravo nahoře. Výsledek je vidět v panelu vpravo dole.



The screenshot shows the Visual Studio Code IDE. The Explorer panel on the left shows the file `nsdtest.py` selected. The main editor displays the same Python code as in the previous screenshot. The Terminal panel at the bottom right shows the execution of the script:

```

PS C:\Users\student\Documents\programovani> & C:\Users\student\AppData\Local\Programs\Python\Python311/python.exe c:\Users\student\Documents\programovani\nsdtest.py
NSD(4477,1073)=37
PS C:\Users\student\Documents\programovani>

```

1.7.3 Interpret Python v režimu REPL

V příkazové řádce můžeme spustit interpret Python i bez argumentu určujícího, který kód se má spustit. V tom případě se dostaneme do prostředí označovaného zkratkou REPL (read-evaluate-print loop). Příklad použití ukazuje následující záznam komunikace, kde interpret reaguje na příkazy zadané z klávesnice (ty začínají ">>>")

```
Python 3.10.12 (main, Jun 11 2023, 05:26:28) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 2
>>> x = 1.5
>>> x=(x+a/x)/2
>>> x
1.4166666666666665
>>> x=(x+a/x)/2
>>> x
1.4142156862745097
>>> x=(x+a/x)/2
>>> x
1.4142135623746899
>>> x=(x+a/x)/2
>>> x
1.414213562373095
>>>
```

Jistě jste poznali opakované použití vztahu pro výpočet \sqrt{a}

$$x_{n+1} = \frac{1}{2} \left(x_n + \frac{a}{x_n} \right).$$

Za pozornost stojí chování REPL, který, pokud má zadání podobu výrazu (např. `x`), vytiskne výslednou hodnotu výrazu, zatímco pokud je zadání příkaz (např. přiřazovací), nic se netiskne.

1.7.4 Prostředí Jupyter

Další, velmi pohodlnou možností bude prostředí pracovního sešitu Jupyter. Toto prostředí umožňuje kombinovat strukturovaný text s kusy kódu programu následovaných výstupem tohoto kódu. Oproti konzoli, může ten mít i podobu obrázku, animace nbo i zvuku. Důležité ale snadné je pochopit, že pořadí provádění kódu je určeno tím, jak uživatel spouští a přeskakuje jednotlivé buňky, nikoli přímo pořadím buněk v sešitu.

Prostředí Jupyter je m.j. dostupné v rámci Visual Studio Code a také (září 2023) jako volná služba [Google Colab](#)

Následující buňka obsahuje zdrojový kód a výsledek jeho běhu.

```
[1]: # program pro Vypocet NSD
a = 4477 # vezmi dvě čísla: 4477
b = 1073 # a 1073

print( 'NSD(',a,',',b,')=', sep=' ', end=' ' ) # vypíše: NSD(4477,1073)=

while not b==a: # dokud jsou obě čísla různá
    if a>b: # zjistí, které je větší
        a = a-b # a odečti menší od většího
    else:
        b = b-a
```

(continues on next page)

(pokračujte na předchozí stránce)

```
print( a )
NSD(4477, 1073)=37
```

Velkou výhodou takového postupu je, že ačkoli kód již proběhl a výsledky běhu tohoto kusu kódu jsou již zobrazeny pod ním, data se kterými se počítá (přesněji tzv. globální proměnné) jsou i nadále k dispozici. Můžeme toho využít a např. zjistit, kolik je $4477/a$ takto

```
[2]: 4477/a
```

```
[2]: 121.0
```

Uveďme však, že oblíbenost jazyka Python spočívá též v tom, že pro kdejaký myslitelný problém je k dispozici potřebný nástroj. Například pro nalezení NSD (anglicky GCD) stačí napsat

```
[3]: import math
print( math.gcd(4477, 1073) )
37
```

Zde je `math` název knihovny, kde jsou shromážděny mnohé užitečné funkce. Ty co tam nenajdete, jsou v dalších snadno dostupných knihovnách.

V našem kurzu budeme používat obojí přístup. Někjaké funkce si napíšeme, abychom se to naučili, jiné budeme používat hotové a připravené v dostupných knihovnách a budeme s jejich pomocí řešit složitější úlohy.

1.8 Kontrolní úlohy

1. Vyzkoušejte všechny tři metody spuštění programu pro NSD pro dvojici čísel 11322 a 6035.
2. Spočítejte v REPL hodnotu $\sqrt{2}$ na základě příkladu výše. Nemáte-li zatím nainstalovaný Python, zkuste si to na <https://trinket.io/console> Nezapomeňte, že k jednou již zadanému výrazu nebo příkazu se můžete vrátit klávesou `<up>`.
3. Totéž si zkuste v prostředí `Jupyter lite`. Pro otevření nového sešitu zvolte v menu `File -> New -> Notebook` nebo klikněte na ikonu Python (Pyodide) v odstavci Notebook v záložce Launcher.

Spouštění programů v Pythonu

Již víme, že v Pythonu se programem rozumí textový soubor (většinou s příponou `.py`) který obsahuje pravidla jazyka Python respektující popis kroků, které je třeba vykonat.

Vyzkoušíme si jak takový program několika různými způsoby spustit. Abyste nemuseli příslušný kód programu ručně přepisovat, naleznete jej zde [test1.py](#).

Seznámíme se se třemi způsoby, jak spustit existující `*.py` soubor

- Na příkazové řádce počítače s OS Windows
- V IDE Visual Studio Code
- V prostředí cloudové služby Google Colab

Je běžným způsobem práce, že vyjdete z nějakého existujícího kódu, ten upravíte pro své potřeby a následně spustíte. V dnešní době je i v úvodním kurzu programování nutno dodat, že původní kód může obsahovat nějaké zákeřnosti a jeho spuštěním jim můžete dovolit páchat nekalosti. V tomto smyslu je práce pomocí Google Colab zdaleka nejbezpečnější.

2.1 Spouštění `*.py` na příkazové řádce

Práce na příkazové řádce sice právem připomíná počítačový pravěk, ale nikterak nezastarala a dnes představuje způsob, kterým se spouštějí i ty nejrozsáhlejší výpočty. Protože uživatelé Linuxu tohle umí, vysvětlíme si to pro prostředí Windows, kde jde o méně obvyklý způsob práce.

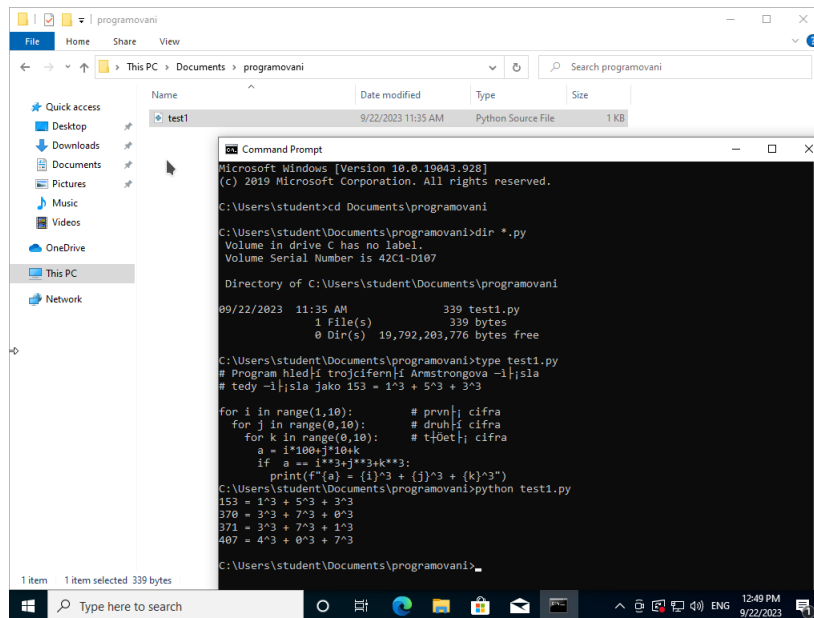
Nejprve je potřeba spustit program, který nám ovládat počítač příkazy umožní. Použijeme ten starší, který se jmenuje `cmd.exe`, novější a složitější PowerShell budeme ignorovat. Zadáním `cmd.exe` do vyhledávací kolonky start menu dokážete tento program spustit. Objeví se vám okénko, které obsahuje tzv. prompt, tedy výzvu k zadání příkazu. Většinou má prompt podobu výpisu běžného adresáře následovaného znakem větší.

Na následujícím obrázku vidíte výsledek provedení čtyř příkazů

- `cd novy_adresar ...` změní běžný adresář. To je užitečné, když si chceme ušetřit práci, protože soubory z běžného adresáře oslovujeme jen jménem, zatímco k souborům v ostatních adresářích musíme přidat relativní nebo absolutní cestu.

- `dir [cesta/maska] ...` vypíše seznam souborů v běžném adresáři. Pokud je uvedena maska (např. `*.py`) vypíše jen ty soubory, jejichž jméno má požadovaný tvar. Lze zadat i cestu k jinému, než běžnému adresáři.
- `type soubor ...` vypíše obsah souboru.
- `python program.py ...` spustí kód v jazyce python zapsaný v souboru, který je uveden jako parametr.

Na pozadí je stejný adresář otevřen i souborovým manažerem Windows, povšimněte si podrobností, které sděluje o souboru `test1.py` a srovnajte je s výpisem příkazu `dir`.



Ačkolik takto často pracovat nebudete, je nezbytné výše uvedenému rozumět. Jediným krokem, který jsme výše ignorovali je psaní kódu. To lze svěřit běžnému textovému editoru, rozumnou volbou na Windows je program Notepad++. Spojení textového editoru s pomůckami pro práci s kódem a jeho laděním představuje tzv. integrované vývojové prostředí (IDE).

2.2 Spouštění *.py ve IDE

Tentýž kód můžeme spustit z IDE, konkrétně budeme používat **VS Code**.

Na obrázku níže vidíte program `test1.py` otevřený v jeho editoru (pravý horní panel) a výsledek jeho běhu (pravý dolní panel nesoucí záhlaví "terminal" a vzhledově podobný příkazové řádce výše). Běh byl spuštěn tlačítkem > vpravo nahoře. Všimněte si, že v terminálovém okně jsou uvedeny absolutní cesty (běžného adresáře, interpretu pythonu a spouštěnému programu).

```

1 # Program hledá trojčiferná Armstrongova čísla
2 # tedy čísla jako 153 = 1^3 + 5^3 + 3^3
3
4 for i in range(1,10): # první cifra
5     for j in range(0,10): # druhá cifra
6         for k in range(0,10): # třetí cifra
7             a = i*100+j*10+k
8             if a == i**3+j**3+k**3:
9                 print(f"(a) = {i}^3 + {j}^3 + {k}^3")

```

```

PS C:\Users\student\Documents\programovani> & C:/Users/student/AppData/Local/Programs/Python/Python311/python.exe c:/Users/student/Documents/programovani/test1.py
153 = 1^3 + 5^3 + 3^3
370 = 3^3 + 7^3 + 0^3
371 = 3^3 + 7^3 + 1^3
407 = 4^3 + 0^3 + 7^3
PS C:\Users\student\Documents\programovani>

```

2.3 Spouštění *.py v Jupyteru

U jednodušších kódů je možnost obsah souboru *.py zkopírovat do jedné buňky sešitu prostředí Jupyter a buňku "spustit". Potíže budou mohou nastat, pokud kód např. vyžaduje čtení ze standardního vstupu (ovšem funkce `input(. . .)` by mohla fungovat.) Do prostředí Google Colab je potřeba dodat potřebné vstupní soubory a naopak výstupní soubory bezpečně uložit, protože virtuální stroje v tomto prostředí jsou vytvořeny ve výchozí podobě bez uživatelských dat a po odpojení se data zase zahodí. To se netýká vlastního kódu a jeho výstupu obsaženého v sešitě *.ipynb.

Pak je tu také možnost využít příkazovou řádku obsaženou v Jupyteru. Její příkazy se uvozují vykřičníkem. Protože ten běží nad prostředím Linuxu, používáme např. místo `dir` příkaz `ls`. Protože běh programu se odehrává na serveru, zatímco uživatelská komunikace v prohlížeči, není (zatím?) možné spouštěnému programu zadávat vstup z klávesnice.

[1]: *# soubor se kódem programu uložíme z webu na lokální disk*

```
! wget https://utf.mff.cuni.cz/~ledvinka/PrProgFyz/2023/py/test1.py
```

```
--2023-09-22 16:29:26-- https://utf.mff.cuni.cz/~ledvinka/PrProgFyz/2023/py/test1.py
Resolving utf.mff.cuni.cz (utf.mff.cuni.cz)... 195.113.23.4
Connecting to utf.mff.cuni.cz (utf.mff.cuni.cz)|195.113.23.4|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 330 [text/x-python]
Saving to: 'test1.py'
```

```
test1.py          100%[=====>]          330  --.-KB/s   in 0s
```

```
2023-09-22 16:29:26 (13,4 MB/s) - 'test1.py' saved [330/330]
```

[2]: *# přesvědčíme se, že obsahuje požadovaný kód*

```
! head test1.py
```

```
# Program hledá trojčiferná Armstrongova čísla
# tedy čísla jako 153 = 1^3 + 5^3 + 3^3
```

(continues on next page)

```
for i in range(1,10):      # první cifra
    for j in range(0,10):  # druhá cifra
        for k in range(0,10): # třetí cifra
            a = i*100+j*10+k
            if a == i**3+j**3+k**3:
                print(f"{a} = {i}^3 + {j}^3 + {k}^3")
```

[4]: # a konečně kód sputíme

```
!python3 test1.py
```

```
153 = 1^3 + 5^3 + 3^3
370 = 3^3 + 7^3 + 0^3
371 = 3^3 + 7^3 + 1^3
407 = 4^3 + 0^3 + 7^3
```

Proměnné, výrazy, základní matematické funkce

3.1 Vyčíslení vzorečku

Jak přesvědčíme počítač, aby nám vyřešil primitivní číselnou úlohu:

O kolik si zkrátím cestu, když půjdu přes fotbalové hřiště úhlopříčkou místo po obvodu?

Ano, jde o to spočítat

$$a + b - \sqrt{a^2 + b^2}$$

pro normované rozměry fotbalového hřiště.

V jazyce Python, který budeme používat na naší přednášce to dokážeme třemi řádky

```
[1]: a = 105
      b = 68

      print( a + b - (a**2 + b**2)**0.5 )
      47.9040368357156
```

Tento krátký *program* obsahuje

- tři **příkazy** (dva přiřazovací (=) a jeden pro textový výstup (`print`))
- dvě **proměnné** (a a b)
- pět číselných **konstant** (tzv. literálů) – 105, 68, 2, 2, 0.5
- jeden numerický **výraz** zapsaný pomocí operací +, -, * a **

Povšimněte si, že jsme záměrně použili proměnné, místo abychom konstanty opakovali, např.

```
print( 105 + 68 - (105**2 + 68**2)**0.5 )
```

Python není jediný programovací jazyk. Pro zajímavost porovnejme tento *program* s ekvivalentním programem v jazyce C:

```
#include <stdio.h>
#include <math.h>

int main() {
    double a = 105;
    double b = 68;

    printf("%g\n", a+b-sqrt(a*a+b*b));

    return 0;
}
```

Pozorování: přívětivější zápis kódu (ale i další důvody) učinily z Pythonu dnes nejpoužívanější jazyk. Stručnost kódu ale naznačuje, že se něco může odehrávat skryto očím nepoučeného uživatele tohoto jazyka. Na našem kurzu na to budeme často muset dávat pozor.

Další problém:

Jak moc se blíží tyto rozměry zlatému řezu?

```
[4]: print(a/b, (a+b)/a )
```

```
1.5441176470588236 1.6476190476190475
```

Pozorujeme, že

- hodnoty proměnných zůstaly v našem prostředí zachovány i po skončení vyhodnocování předchozí buňky

3.2 Proměnné

Slouží k uchování dat pro další použití. Jaké proměnné použijeme závisí konkrétním algoritmu/výpočtu. To uvidíme v dalších hodinách.

Při zápisu složitějších matematických výrazů je vhodné delší nebo opakující se podvýrazy uložit do pomocných/dočasných proměnných.

$$f = \frac{1 - \sqrt{1 - x^2}}{1 + \sqrt{1 - x^2}}$$

tedy můžeme rozdělit na dva přiřazovací příkazy

```
odmocnina = (1-x**2)**0.5
f = (1-odmocnina)/(1+odmocnina)
```

V Pythonu se mají proměnné zapisovat malými písmeny (+podtržítka), tedy místo

```
Ekin = 0.5*m*v**2
```

bychom měli např. psát

```
e_kin = 0.5*m*v**2
```

Uvidíte, že ne vždy se toto pravidlo dodržuje.

3.3 Zápis čísel

Součástí kódu jsou různé hodnoty. Pravidla, která používáme snadno uhadnete z následujících případů.

```
cele_cislo1 = 123
cele_cislo2 = 1000000
cele_cislo3 = 1_000_000      # čitelnější zápis

realne_cislo1 = 123.45
realne_cislo2 = 0.0101
realne_cislo3 = 6.02214076e23 # Avogadrova konstanta
realne_cislo4 = 6.62607015e-34 # Planckova konstanta
```

Platí, že pokud se v zápisu čísla vyskytuje `.` (tečka) nebo `e` (velké nebo malé), jde o reálné číslo. S jejich vlastnostmi se seznámíme podrobněji později, již teď je rozumné vědět, že přesnost výpočtů s reálnými čísly je asi 15 cifer, zatímco operace probíhající výhradně s celými čísly nemají omezenou přesnost.

Je velmi pohodlné, že Python umí komplexní čísla.

```
komplexni_cislo1 = 2+3j
komplexni_cislo2 = 0.2-1.3j
```

Vždy je chápáno jako dvojici čísel reálných (tedy $i + 1j$), jak si takové tvrzení ověřit se naučíme zanedlouho. Pozor, $1+j$ není zápis čísla ale součet jedničky a proměnné j .

3.4 Pojmenované konstanty

Za konstantu můžeme považovat hodnotu, která

- Nejméně po dobu běhu programu nemění svoji hodnotu

Konstantou tedy může být jak číslo π , tak např. konkrétní rozměry nějakého zařízení, ačkoli v sousední laboratoři mají větší.

Konstanta může být reprezentována konkrétní číselnou hodnotou, např. `105`. Je ale nepohodlné a asi i nesprávné mít tuto hodnotu roztroušenou na různých místech kódu. Proto je zvykem tuto hodnotu označit slovy, definovat konstantu

```
STRANA_A = 105
```

a dále místo čísla `105` používat identifikátor `STRANA_A`.

Bývá zvykem, že

- Hodnoty konstant nelze měnit

To v Pythonu ale neplatí, tento jazyk předpokládá neomylného programátora. Ten se drží pravidla

- Hodnoty konstant nesmíme měnit

Zvyk/pravidlo v Pythonu toto řeší tak, že proměnné s významem konstant se mají psát velkými písmeny, proměnné malými písmeny. Opět, ne vždy to budeme dodržovat, už jen proto, že rozdělení na konstanty (jen velká písmena) a proměnné (malá první písmena identifikátoru, libovolná další) není často úplně jednoznačné.

3.4.1 Shrnutí:

V programování existují dva koncepty: *proměnná* a *konstanta*.

- **Proměnné** postupně nabývají různých hodnot, např. proměnná `soucet` může pro řadu 1, 2, 3, ..10 nabývat hodnot 0, 1, 3, 6, ..., 55. Měly by se psát malými písmeny.
- **Konstanty** mají jednu pevně danou hodnotu. Měly by se psát velkými písmeny. Ani jedno ale jazyk Python nevynucuje. V mnoha případech se psaní velkými písmeny nedodrhuje, např. π se jmenuje `math.pi`.

Detailně je to popsáno v textu [Python Constants](#)

3.5 Aritmetické výrazy

Výrazy vznikají složením operandů a aritmetických operátorů. Ty nejdůležitější shrnuje tato tabulka

zápis	hodnota	komentář
<code>1 + 1</code>	2	
<code>2 * 3</code>	6	
<code>13 / 5</code>	2.6	
<code>13 // 5</code>	2	celočíslné dělení
<code>13 % 5</code>	3	zbytek po "-"
<code>5**3</code>	125	mocnění

V ní používáme celočíselné operandy. Kromě celých čísel máme po ruce i čísla reálná (zvaná `float`) a čísla komplexní.

zápis	hodnota	komentář
<code>6.6743e-11</code>	6.6743×10^{-11}	(lze psát i <code>6.6743E-11</code>)
<code>(1+1j)**2</code>	$2i$	protože $(1 + i)^2 = 2i$

Reálná (i komplexní) čísla mají na rozdíl od čísel celých omezený počet cifer (přesnost)

```
>>> print(1+0.1-1)
0.10000000000000009
```

3.6 Logické výrazy

Při rozhodování, jak při běhu programu postupovat budeme potřebovat logické výrazy.

zápis	hodnota	komentář
<code>1+1 == 2</code>	True	jedno rovnítko nestačí
<code>1+1 != 3</code>	True	viz symbol \neq
<code>1 < 2</code>	True	
<code>1 > 2</code>	False	
<code>1 <= 2</code>	True	
<code>1 >= 2</code>	False	

Tato porovnání kombinujeme pomocí `and`, `or` a `not`

```
>>> 1>2 or 3>2
True

>>> 1>2 and 3>2
False

>>> 1<2 and not 3<2
True
```

3.7 Priorita aritmetických operací

Operace, se kterými jsme seznámili, mají takto uspořádané přednosti:

```
**
*, /, //, %
+, -
<, <=, >, >=, !=, ==
not
and
or
```

Proto v následujícím výrazu se operace odehrají od konce, ačkoli, pokud preference operací nevyžadují jinak, vyhodnocují se výrazy zleva doprava.

```
>>> False or True and not 1 == 19 - 2 * 3**2
False
```

Samozřejmě, k úpravě pořadí operací se používají závorky $(1+2)*3$.

- sčítání + a odčítání - má prioritu nejmenší
- následuje násobení * a dělení /
- mocnění ** má mezi binárními operacemi prioritu nejvyšší
- k úpravě pořadí vyhodnocování operací se používají závorky

Pozn. Všimněte si závorek níže:

Úloha: *Jaké jsou tedy správné rozměry hřiště se stejnou plochou, ale správným poměrem stran?*

```
[12]: phi = (1+5**0.5)/2

p = (a*b*phi)**0.5
q = p/phi

print(p, q, p/q, (p+q)/p)

107.48377868159572 66.42862846449752 1.618033988749895 1.618033988749895
```

Pro pořádek si zde uvedeme úplnější dokumentaci priority všech operací, které v kurzu potkáme.

- Vyhodnocování podvýrazu v závorkách (...) nenaruší žádná operace před nebo za závorkami.
- ** představuje operaci mocnění. Její vysoká priorita umožňuje, že $3*2**2*5==60$. Asociativita je zprava, tedy $a**b**c == a**(b**c)$. Ovšem i když máme dispozici operaci mocnění, i v Pythonu je Hornerovo schéma pro vyhodnocování polynomu rychlejší!

- $-a$ představuje tzv. unární mínus. Podobně můžeme psát $+a$. (Zájemci mohou prozkoumat též unární binární negaci celých čísel, např. ~ 6 .)
- Násobení zapisujeme $*$. Dělení $/$ dá i dvou celých čísel podíl jako číslo reálné, zatímco celá část podílu se zapisuje $//$ (pozor, zatímco $5//4==1$, je $-5//4==-2$, tedy "zaokrouhleno" směrem dolů. Pokud bychom po celočíselném dělení i/j požadovali (anti-)symetrii při $i \rightarrow -i$, použijeme výraz $\text{int}(i/j)$). Zbytek po dělení se zapisuje $\%$. Funguje to i pro reálné operandy, např. `print(5.75//2.0, 5.75%2.0)` vypíše `2.0 1.75`.
- Následuje sčítání $+$ a odčítání $-$, nižší prioritu už budou mít jen bitové a logické operace:
- Jen výjimečně potkáme operace bitového posunu vpravo $a >> n$ (což je rovno $a // 2^{*n}$) a vlevo $a << n$ ($== a * 2^{*n}$).
- Další bitové operace jsou (vyjmenovány s klesající prioritou) $a \& b$, $a \wedge b$, $a | b$. Jejich použití je dost výjimečné, ale např. pro celá čísla k je $1 - (k \& 1) * 2$ rychlejší variantou $(-1) ** k$.
- Teprve nyní následují operace porovnání $a > b$, $a \geq b$, $a < b$, $a \leq b$, $a == b$, $a != b$, ta poslední s významem $a \neq b$. Stejnou prioritu má `in` v testu `2 in [1, 2, 3]` resp. `x not in [1, 2, 3]` a konečně i test identity `is` a `is not`, který mj. umožňuje zjistit, zda se dvě proměnné aktuálně odkazují na tatáž data testem `x is y`.
- Konečně s klesající prioritou tu máme operátory `not`, `and`, `or`, což umožňuje psát přirozeně `p==0 or q!=0 and r/(q*p)>1` bez závorek okolo jednotlivých porovnání.
- Pro úplnost uvedme, že pokud bychom na `if` a `else` v podmíněném výrazu nahlíželi jako na operátory, pak mají ještě nižší prioritu: `math.sin(x)/x if x!=0 else 1.0`.

Pozor, logické operace `and`, `or` podléhají zkrácenému vyhodnocování, jak je popsáno níže.

3.8 Průběh vyhodnocování výrazů

Výpočet výrazu probíhá zleva doprava, přičemž pokud to prioritá operátorů vyžaduje, vyzvednuté hodnoty se uschovají dokud je nebude možno použít. To je důležité, když ve výrazu budeme volat funkce, protože to určuje pořadí jejich volání. Zde si to ilustrujeme pomocí funkce `input`, která se ptá na vstup z klávesnice. Ještě potřebujeme vědět, že pro řetězce se násobením myslí `'2'*3 == '222'` a operací plus jejich spojování `'1'+ '222' == '1222'`. Vyzkoušejte si to pro jiné zadané hodnoty.

```
input("x=") + input("y=") * int( input("z=") )
```

```
x=1
y=2
z=3
'1222'
```

Vidíme, že postupně se vyhodnotí všechny operandy a to zleva doprava a to i tehdy, když je nejprve vzhledem k prioritě operací spočítat pravější část výrazu. Z tohoto pravidla je ale výjimka:

3.9 Neúplné vyhodnocování logických výrazů

Zatímco pro určení operace součtu dvou čísel $a+b$ musíme k tomu, abychom určili výsledek, vyhodnotit oba operandy, logické operace se takto nechovají, protože u operací `and`, `or` je v polovině případů znám výsledek operace již po vyhodnocení prvního operandu. To proto, že např. `True or x` dá stejný výsledek pro obě logické hodnoty x . Při výpočtu se proto hodnota x vůbec nezjišťuje a výraz `True or x` nabývá hodnoty `True` pro *jakékoli* x . Proto `True or "nesmysl"/0` nezpůsobí žádnou chybu!

Mohlo by se zdát, že se tak děje kvůli urychlení výpočtu, nicméně povaha logických výrazů často vede k tomu, že zkrácené vyhodnocování logických výrazů vítáme hlavně proto, že nám umožní neřešit rizikové chování výrazů:

```
if x>0 and y>0 and x**0.5+y**0.5<1:
    ...
```

Takto psaná podmínka nemá díky zkrácenému vyhodnocování logických výrazů potíže s vyhodnocováním pro záporná x .

Dalším místem, kde o této vlastnosti logických binárních operací `and` a `or` musíme vědět jsou výrazy obsahující funkce. Podle příslušných pravidel totiž nebudou příslušné části výrazů vyhodnocovány a funkce v nich obsažené se nevolají. Lze toho využít k urychlení výpočtu ale může nás to i překvapit, pokud daná funkce má za úkol ještě něco jiného, než vrátit vypočtenou hodnotu.

Rozmyslete si, proč se ve druhém případě Python neptá na hodnotu y :

```
input("x=") == "1" and input("y=") == "2"
```

```
x=1
y=2
True
```

Tedy došlo k vyhodnocení obou podvýrazů ve tvaru `input(...) == ...`. Naopak, pokud níže zadáme jako první jinou odpověď než 1, druhý podvýraz `input("y=") == "2"` se nebude vyhodnocovat na y se nás program nezeptá.

```
input("x=") == "1" and input("y=") == "2"
```

```
x=2
False
```

```
[5]: # Zde si vyzkoušejte příklady výše
input("x=") + input("y=") * int(input("z="))
```

```
[5]: '1222'
```

3.10 Operandy

Operace, které jsme zmínili, pracují s jedním nebo dvěma svými operandy. Co přesně se odehraje záleží na typu (obou) operandů. Například `+` znamená jednu sčítání čísel, pokud jsou oba operandy čísla (např. $40+2$), ale u řetězců jde o jejich spojování (např. `jmeno+' '+prijmeni`). Operandy mohou mít více podob:

- Hodnoty proměnných. V příkazu `x = a + b` se sčítání účastní hodnoty proměnných a a b .
- Číselné konstanty. Z jejich zápisu se určí i jejich typ `int`, `float`, `complex`, např. `2.0` je typu `float`.
- Řetězcové konstanty, např. první operand ve výrazu `'MUDr. ' + prijmeni`.

- Výraz — např. u výrazu $a^{**2} + b^{**2}$ jsou oběma operandy operace $+$ jiné výrazy. Výraz může ale nemusí být v kulatých závorkách.
- Hodnoty vrácené nějakou funkcí, např. `math.sin(x) + 1`. Na konverzi typů lze také nahlížet jako na volání funkce.
- Další...

Pokud je operandem nějaká proměnná, zůstane její hodnota nezměněna. Pokud je operandem výsledek nějaké funkce nebo binární operace, je tento použit a protože není již potřeba, přestává existovat. Jím zabírané místo uvolněno k dalšímu použití.

3.11 Knihovna math

Zatím jsme odmocňování řešili operací $x^{0.5}$. Odmocnina ale představuje důležitý příklad **standardní matematické funkce**. Ty jsou ve většině programovacích jazyků dostupné přímo nebo ve vhodné *knihovně*. V Pythonu je to knihovna `math` a odmocnina se jmenuje `sqrt`. To, že ji chceme používat musíme předem oznámit.

```
[2]: import math

a = 105
b = 68

print( a + b - math.sqrt(a**2 + b**2) )

47.9040368357156
```

Zde je seznam nejdůležitějších funkcí v knihovně `math` a také dvě, které Python umí i bez knihovny `math` a to `abs` a `round`.

Za povšimnutí stojí mj. funkce `atan2(x, y)`, která počítá úhel průvodiče bodu o kartézských souřadnicích $[x, y]$ v rozsahu $(-\pi, \pi)$, zatímco `arctan(y/x)` by i pro záporná x vrátil úhly menší než pravé.

```
[ ]: # konstanta pi
print( math.pi )

# trigonometrické funkce
print( math.sin(a) )
print( math.cos(a) )
print( math.tan(a) )

print( math.asin(b/a) )
print( math.acos(b/a) )
print( math.atan(b/a) )
print( math.atan2(b,a) )

# hyperbolické funkce
print( math.sinh(a) )
print( math.cosh(a) )
print( math.tanh(a) )

print( math.asinh(a) )
print( math.acosh(a) )
print( math.atanh(b/a) )
```

(continues on next page)

(pokračujte na předchozí stránce)

```

# zaokrouhlování
print( math.floor(p) )
print( math.ceil(p) )
print( round(p) )
print( round(p,2) )

# abs
print( abs(q-p) )
print( math.fabs(q-p) )

# exp, log
print( math.exp(p) )
print( math.log(p) )
print( math.log10(p) )
print( math.log10(p)/math.log10(math.e) )

# faktorial
print( math.factorial(12) )

```

Chtělo by se napsat, že kromě funkcí obsahuje knihovna `math` i definice konstant. Např. π získáme napsáním `math.pi`

Populární vlastnost Pythonu, umožňující měnit věci podle uživatelových představ, ale také znamená, že i tuto hodnotu můžeme změnit:

```

[14]: print( math.cos( math.pi ) )

math.pi = 3
# od teď jsme ztraceni, např.
print( math.cos( math.pi ) )

# takže to hned opravíme
math.pi = 3.1415926535897932385
print( math.cos( math.pi ) )

-1.0
-0.9899924966004454
-1.0

```

Pokud není z názvu jasné, co funkce počítá, lze si v interaktivním prostředí požádat o nápovědu zapsáním otazníku. Bohužel, její interpretace vyžaduje někdy například tušit, že slovo `Integral` zde znamená "celé číslo", nikoli pojem z matematické analýzy:

```

[33]: math.ceil?

Docstring:
ceil(x)

Return the ceiling of x as an Integral.
This is the smallest integer >= x.
Type:      builtin_function_or_method

```

S pomocí uvedených funkcí lze vyčíslit běžné výrazy, např.

$$\ln\left(x + \sqrt{1 + x^2}\right)$$

```
[9]: x = math.sinh(2)
print(x, math.log(x+math.sqrt(1+x**2)))
3.626860407847019 2.0
```

Poznámka:

Pokud vám knihovna `math` nestačí je tu ještě `scipy.special`

```
[11]: import scipy.special
print(scipy.special.gamma(10.0), math.factorial(9))
362880.0 362880
```

3.12 Zabudované funkce

Takto (*builtin*) se v Pythonu označují funkce, jenž jsou k okamžitě dispozici při spuštění i bez importu nějaké knihovny. Ačkoli mnohé zatím nebudeme umět použít, je zde vhodné místo, kde se o nich zmínit.

Mezi zabudované funkce patří

- Nejzákladnější aritmetické funkce `abs(x)`, `round(x)`, `pow(x, y) == x**y`, `max(x, y, ...)`, `min(x, y, ...)`, `round(x)`
- Funkce `type` vrací typ svého argumentu. Budeme ji používat při výkladu typů a také ke kontrole argumentů funkcí.
- Funkce pro práci se seznamy, např. `len(a)`, `sum(a)`, `min(a)`, `max(a)` vrací počet prvků seznamu, jejich součet a minimální/maximální hodnotu prvku.
- Funkce `input` (vzývá) vypíše výzvu a vrátí řetězec, který zadáme jako vstup na klávesnici.
- Následující funkce pro práci s řetězci si probereme při výkladu řetězců:
 - Nejdůležitější je `len(s)` vrací počet znaků řetězce
 - `ord('ž')` vrací unicode kód jednoznakového řetězce.
 - K ní inverzní je funkce `chr(n)` vrací znak s daným kódem.
- Později také probereme konverzi typů, např. když `list("ABC")` převede řetězec na seznam znaků. Konverzi lze chápat jako volání funkcí. Proto mezi builtin funkcemi najdete `bool`, `complex`, `int`, `list`, `set`, `str`. Funkce `repr` se chová podobně jako `str`, jen někdy vrátí podrobnější informace, než ona.
- Další funkce, viz [dokumentace](#)

3.13 Kontrolní úlohy

1. Vyzkoušejte, kolik desetinných míst třetí Stirlingova formule

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n}\right)$$

pro $n = 15$. (Vyčíslete obě strany a vizuálně porovnejte výsledky, použijte funkci `math.factorial` a konstantu `math.e`.)

2. Nalezněte efektivní povrchovou teplotu Slunce, pokud by to byla koule o poloměru $6.95700 \times 10^8 \text{m}$ vyzařující jako dokonale černé těleso výkonem $P = \sigma T^4 = 3.83 \times 10^{26} \text{W}$. Použijte $\sigma = 6.9570010^{-8} \text{Wm}^{-2}\text{K}^{-4}$.

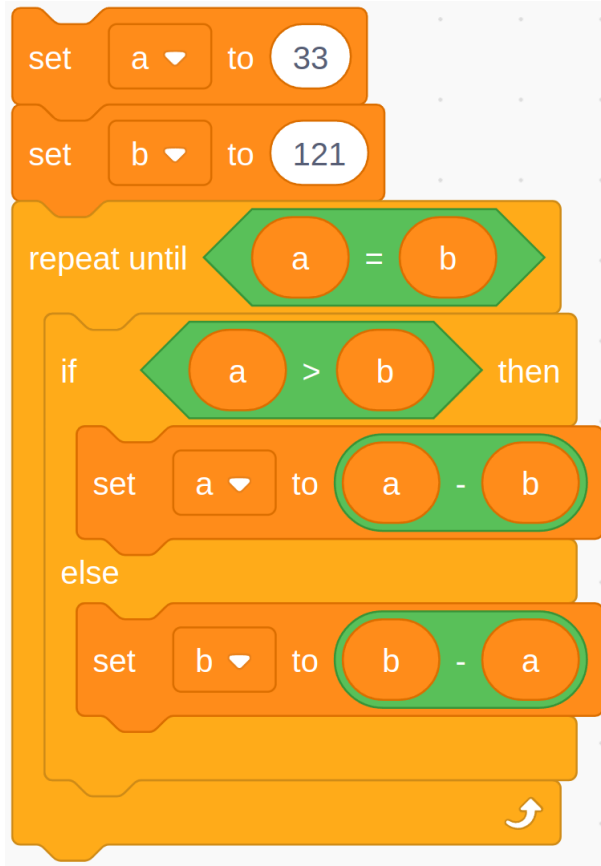
Na našem prvním příkladě pro výpočet největšího společného dělitele jsme viděli, že cíle lze dosáhnout rozdělením problému na menší části, které měly následující povahu

- uschovávání hodnot (např. mezivýsledků) do proměnných
- rozhodnutí, jak dále postupovat (která z proměnných a,b je větší?)
- opakování souboru kroků
- tisk informací pro uživatele (`print`)

To jsou příklady tzv. příkazů. Za desetiletí vývoje programování se ukázalo, že i nejuniverzálnější jazyk nepotřebuje takových příkazů jen několik a složitost programů spočívá v jejich vhodném složení do většího celku.

4.1 Příkazy a jejich posloupnosti

Již víme, že kód sestává z posloupnosti příkazů. Ty zapisujeme textem dodržujícím jistá pravidla. Pro ilustraci si ukažme, že téhož lze dosáhnout graficky, například v jazyce `scratch` diagram



popisuje kód v Pythonu vypadající takto:

```
a = 33
b = 121

while not a==b:
    if a>b:
        a = a-b
    else:
        b = b-a
```

Vidíme tedy, že místo skládání puzzle v Pythonu jen vhodně odsazujeme řádky s jednotlivými příkazy.

4.2 Přiřazovací příkaz

Oznamuje, že proměnná má nabývat hodnotu určenou nějakým výrazem.

```
promenna = vyraz
```

tedy například

```
energie = hmotnost * rychlost_svetla**2
```

V jazyce Python je ale přiřazovací příkaz také místem, kde proměnné vznikají. Prvním přiřazením do proměnné se s jejím jménem sváže hodnota určená výrazem na pravé straně znaku =. Jak ještě několikrát zmíníme, proměnná

nepředstavuje tak jako v jiných počítačových jazycích popisku nějakého místa paměti pro uložení dat, ale odkaz na taková data. Dalším přiřazením do téže proměnné se tedy změní nejen hodnota, ale také místo, kde v paměti data sídlí. To je velmi důležité při práci s poli a tam se k tomuto faktu znovu vrátíme.

V důsledku přiřazení se může změnit i typ dat.

Pokročilejší výklad

Chování přiřazení si můžeme demonstrovat za pomoci zabudované funkce Pythonu `id(...)`, která sděluje na jaké adrese jsou data proměnné uložena.

```
[9]: x = 1.2
print(f"{id(x)=}")
y = x
print(f"{id(y)=}")
x = x/y
print(f"{id(x)=}")
x = 1.2
print(f"{id(x)=}")
```

```
id(x)=140717650747248
id(y)=140717650749072
id(x)=140717650741360
id(x)=140717650740496
```

Povšimněte si, že přiřazení `x=y` způsobí, že obě proměnné odkazují na tatáž data. Pokud bychom použili `x=y+0`, nebylo by tomu tak, ačkoli by `x` a `y` nabývaly stejné hodnoty. Stejně adresy tedy znamenají více než stejné hodnoty.

- Ke zjištění totožnosti slouží operátor `is`
- Ke zjištění rovnosti slouží operátor `==`

```
[13]: x1 = 1.2
y1 = x1

x2 = 1.2
y2 = x2+0

print(f'x1 is y1 = { }')
print(f'x2 is y2 = { }')
print()
print(f'x1 == y1 = { }')
print(f'x2 == y2 = { }')
```

```
x1 is y1 = True
x2 is y2 = False
```

```
x1 == y1 = True
x2 == y2 = True
```

Další komplikace je v tom, že z hlediska `is` se celá čísla a řetězce chovají zcela resp. poněkud jinak než čísla reálná:

```
[12]: x1 = 1.0
x2 = 1.0+0j
print( f"{x1 is x2 = }" )
```

(continues on next page)

```
s1 = "ab"
s2 = "a"+"b"
print( f"{s1 is s2 = }" )

i1 = 10
i2 = 10+0
print( f"{i1 is i2 = }" )

i1 = 400
i2 = 400+0
print( f"{i1 is i2 = }" )

x1 is x2 = False
s1 is s2 = True
i1 is i2 = True
i1 is i2 = False
```

Konec pokročilejšího textu

4.2.1 Vícenásobné přiřazení

Velmi pohodlná je varianta přiřazovacího příkazu zahrnujícího dvě nebo i více přiřazení zároveň. Její užitečnost si můžeme demonstrovat na prohození obsahu dvou proměnných. Uvidíme, že v programování jde o častý úkaz.

Obvyklý způsob, jak prohodit dvě proměnné je

```
pom_prom = a
a = b
b = pom_prom
```

Python umožňuje to zapsat přehledněji

```
a, b = b, a
```

Protože takový zápis zpřehledňuje kód lze jej doporučit, i když, samozřejmě, nejde o zjednodušení z hlediska rychlosti operací.

Pozn. V našem kurzu nebude příliš času si vysvětlit, že jde o techniku související se složeným datovým typem *n-tic* (*tuple*) a jejich rozbalováním (*unpacking*) v rámci přiřazení.

4.3 Příkaz import

Již jsme potkali konstrukci

```
import math
```

Jde o příkaz, který zařídí, že můžeme používat funkce z této knihovny za použití zápisu `print(math.sin(math.pi/3))` a pod.

Pro úplnost uvedme, že tento příkaz se může vyskytovat v dalších podobách. Například

```
from math import sin, cos, pi
```

nám dovolí používat ještě jednodušší podobu zápisu `print(sin(pi/3))`

Ještě častěji potkáme variantu

```
import numpy as np
```

která nám umožní místo delšího `numpy` psát kratší `np`. Drobnost, ale velmi používaná a srozumitelná.

4.4 Příkaz print a volání podprogramu

Dávno již používáme příkaz `print`, např.

```
print( x, x*x, x*x-a )
```

by nám srozumitelně sdělilo, jak dobře jsme nedávno hledali odmocninu z `a`.

Jde o speciální variantu příkazu, kterému se odborně říká *výrazový příkaz*, ale pro jednoduchost budeme teď používat termín volání podprogramu (jde o podmnožinu výrazových příkazů skládajících se právě z jedné funkce).

Podprogram (procedura jak uvidíme vlastně funkce) `print` je již připravený kus programu, který vezme zadané argumenty a zařídí, že se jejich textová podoba objeví na požadovaném místě (zatím v konzoli nebo jako výstup buňky v sešitu Jupyter).

Jako jiný příklad výrazového příkazu použijme funkci `input`. Ta sice vrací řetězec, který po výzvě zadáme na klávesnici, nicméně můžeme tuto vrácenou hodnotu ignorovat a psát

```
[19]: input('stiskněte klávesu <enter>')
      print('děkuji')
```

```
děkuji
```

Jde o program s dvěma příkazy, oba mají podobu volání podprogramu.

4.5 Větvení - podmíněný příkaz if

Jako obvykle máme jednoduchou variantu s jednou větví

```
if podminka:
    prikaz1
    prikaz2
    ....
```

a variantu s

```
if podminka:
    prikaz1
    prikaz2
    ....
else:
    prikazA
```

(continues on next page)

```
prikazB
....
```

Použití odsazování (indentace) pro podpříkazy strukturovaných příkazů si vyžádalo ještě variantu s **elif**

```
if podminka:
    prikaz1
    prikaz2
    ....
elif podminka2:
    prikazA
    prikazB
    ....
```

Tu lze kombinovat s **else** na konci řetězce podmínek.

Uvedme konkrétní příklad:

potřebujeme-li prohodit dvě proměnné tak, aby bylo $a \geq b$, můžeme napsat

```
if a < b:
    a, b = b, a
```

4.6 Cyklus while

```
while podminka:
    prikaz1
    prikaz2
    ....
```

Pokud podmínka není splněna hned na počátku, žádný z příkazů těla cyklu se neprovede.

Jde o klíčový příkaz, který umožní opakovat jisté operace, dokud nedosáhneme požadovaného stavu určeného podmínkou. Například:

```
s = 0
n = 1
while n < 1000:
    s = s + 1/n
    n = n * 2
```

spočte součet $1 + \frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{512}$.

Důležité: mezi příkazy těla cyklu by neměl chybět takový, který někdy změní hodnotu podmínky. Jinak se bude cyklus opakovat bez přestání do té doby, dokud nenastane nějaká chyba, běh programu nepřeručíme nebo neukončíme. (Rozdíl mezi přerušit a ukončit ještě potkáme.)

4.7 Cyklus for

V počítačových jazycích býval důležitý cyklus, který zařídil, že nějaká proměnná v cyklu nabývala hodnoty z nějakého intervalu celých čísel. Tvůrci počítačových jazyků ovšem tento koncept různě rozšiřovali. V Pythonu je to méně takto: Vezmi nějaký seznam hodnot a zaříd' že tzv. řídicí proměnná cyklu postupně nabude všech hodnot z tohoto seznamu. To se později ještě zdokonalilo v tom, seznam může být definován jen předpisem, jak takové hodnoty zjistit, ale není nezbytné jej skutečně vytvářet. To umožní, že počet opakování cyklu `for` může být vyšší, než kolik je dostupné paměti na uložení takového seznamu.

Na počátku našeho kurzu se ovšem vrátíme k základům programování a ukážeme si jak vypsát seznam celých čísel od 1 do 100:

```
for i in range(1,101):
    print(i)
```

Hlavička cyklu v podobě `for i in range(1,101):` zařídí, že bude provedena následující sekvence operací

```
i = 1
print(i)
i = 2
print(i)
i = 3
print(i)
...
i = 99
print(i)
i = 100
print(i)
```

Musíme vzít na vědomí, že funkce `range` se chová tak, že její první argument představuje hodnotu, kterou se začne, zatímco druhý argument hodnotu **před** kterou se skončí. Je to nepohodlné, jak ale uvidíme, v rámci jazyka Python odůvodněné. Můžeme teď psát, že základní varianta cyklu `for` má podobu

```
for index in range(initial, final+1):
    prikaz1
    prikaz2
    ....
```

Zajímavým detailem takovéto podoby cyklu `for` je fakt, že na jeho konci nabývá i hodnoty `final`, zatímco svojí funkcí podobný cyklus

```
i = initial
while index < final:
    prikaz1
    prikaz2
    ....
    index = index + 1
```

skončí i s hodnotou `final+1` (v obou případech předpokládáme `initial < final`).

Funkce `range` může mít jen jeden argument, tj. `range(stop)`, potom např. `range(5) -> 0, 1, 2, 3, 4`.

Funkce `range` může mít i třetí argument, tj. `range(start, stop, step)`, potom např. `range(2, 12, 2) -> 2, 4, 6, 8, 10`.

Cyklus `for` je velmi důležitý i v této základní podobě. Proto si vyzkoušejte napsat takové argumenty `range` aby následující kód vypsal:

- 1,2,3
- 1,3,5,7
- -4,-3,-2,-1
- -1,-2,-3,-4 (použijte argument `step=-1`)

```
[15]: # prostor pro testování cyklů for
```

```
for i in range(2,12,2):
    print(i,end=' ')

```

```
2 4 6 8 10
```

Pozn. Ve skutečnosti je `range(...)` funkce, která vrací návod, jak a jaký interval má proměnná `i` procházet. Místo ní může být jakýkoli výraz u kterého dává smysl pocházet jeho hodnoty, např. seznam:

```
seznam = [1,2,3]
for x in seznam:
    print(x)

```

4.7.1 Vnořené cykly

Protože příkazy můžeme kombinovat, lze v jednom cyklu mít cyklus další. Je vhodné použít jinou řídicí proměnnou, výjimky z tohoto pravidla patří do pokročilého programování.

```
[18]: # malá násobilka
```

```
for i in range(1,11):
    for j in range(1,11):
        print(f'{i*j:4}',end=' ')
    print()

```

```

1  2  3  4  5  6  7  8  9 10
2  4  6  8 10 12 14 16 18 20
3  6  9 12 15 18 21 24 27 30
4  8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100

```

4.8 Příkaz skoku break

Pokud je cyklus svojí povahou složitější, může být výhodné zjišťovat podmínku jeho ukončení nějakým výpočtem uvnitř a následně použít příkaz `break`, který vyskočí z cyklu a běh programu pokračuje dalším příkazem, jaký by následoval po obvyklém ukončení cyklu.

4.8.1 Cyklus repeat

Příkaz cyklu testující podmínku na konci cyklu nikoli na před jeho započítím (jako je `repeat` v Pascalu nebo `do` v C) není v Pythonu k dispozici. Místo toho můžeme použít nekonečný cyklus `while True:` a jeho opakování ukončit po testu na konci. Například

```
while True:
    x = int( input('zadejte kladné číslo:') )
    if x>0:
        break

print('děkuji')
```

Cvičení: přepište tento kód bez použití `break` a rozmyslete si, zda vám to přijde elegantnější. Pozn. je více možností, jedna např. opakuje volání funkce `input`, jiná nastaví na počátku vhodnou hodnotu `x`.

4.9 Indentace (odsazování) kódu

Viděli jsme, že např. při použití příkazu `while` potřebujeme odlišit, které příkazy se mají opakovat a které provádět po skončení cyklu, a že za tím účelem jsou příkazy těla cyklu odsazeny.

Máme tu dvě pravidla

- Jako náповěda, že máme uvažovat o změně odsazování slouží dvojtečka na konci řádků s příkazy `if`, `else`, `elif`, `while`, `def` atd.
- Odsazení podřízených příkazů musí být větší a stejné. Doporučuje se používat stejné odsazení v podobě čtyř mezer.

Komplikaci představuje neviditelný znak `<tab>`, který odsakuje na začátek dalšího tabulačního sloupce. Jeho šířka je obvykle osm znaků. Python nedovoluje kombinovat tabulátory a mezery:

```
tomas@dronte:~/tmp/prog$ python3 spaces.py
File "/home/tomas/tmp/prog/spaces.py", line 3
  print(2)
TabError: inconsistent use of tabs and spaces in indentation
tomas@dronte:~/tmp/prog$ cat spaces.py
if 1 < 2:
    print(1)
    print(2)
tomas@dronte:~/tmp/prog$ hexdump -C spaces.py
00000000  69 66 20 31 20 3c 20 32  3a 0a 09 70 72 69 6e 74  |if 1 < 2:..print|
00000010  28 31 29 0a 20 20 20 20  20 20 20 20 70 72 69 6e  |(1).      prin|
00000020  74 28 32 29 0a                                |t(2).|
00000025
tomas@dronte:~/tmp/prog$
```

Tento záznam komunikace s počítačem obsahuje tři příkazy

- pokus o spuštění programu `spaces.py`. Ten skončí výpisem chyby `TabError: inconsistent use of tabs and spaces in indentation`
- vypsání obsahu souboru `spaces.py` příkazem `cat`, které neukazuje nic podezřelého
- podrobné vypsání obsahu souboru `spaces.py` příkazem `hexdump`, které ukazuje, že řádek `print(1)` začíná tabulátorem (*ascii* znak 09), nikoli osmi mezerami.

Závěr: Pro indentaci příkazů se doporučuje používat jen mezery.

4.10 Pokračování řádků

Někdy se vše nevejde na jeden řádek. Ve fyzice to jsou zejména dlouhé výrazy. Možností je více, nám stačí jediné – výraz rozdělte uvnitř závorek. Následující příklad m.j. ukazuje, že odsazení pokračovacího řádku může být libovolné a nemusí se podřizovat příkazu `i f`.

```
if abs(x) < 0.2:
    y = x*(1 + x2*(-0.3333333333333333 + x2*(0.2
        + x2*(-0.14285714285714285 + x2*0.1111111111111111 )))
```

4.11 Více příkazů na jeden řádek

Někdy je žádoucí mít na jednom řádku více příkazů. V tom případě slouží k oddělení příkazů středník. Používejte střídmě.

4.12 Kontrolní úlohy

1. Opravte indentaci v programu pro malou násobilku

```
# malá násobilka
for y in range(1,11):
for x in range(1,11):
print(f' {x*y:4}' ,end='')
if x==1:
print(' |',end='')
if y==1:
print()
print('-----+',end='')
print()
```

tak, aby jeho výstup byl

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50

(continues on next page)

(pokračujte na předchozí stránce)

6		12	18	24	30	36	42	48	54	60
7		14	21	28	35	42	49	56	63	70
8		16	24	32	40	48	56	64	72	80
9		18	27	36	45	54	63	72	81	90
10		20	30	40	50	60	70	80	90	100

Povšimněte si, že $x==1$ se testuje uvnitř cyklu `for x in ...`, zatímco $y==1$ se testuje v rámci cyklu `for y in ...`. Poslední `print()` ukončuje každý z vytištěných řádků.

Proměnné a typy dat

Tušíme, že počítače pracují nejen s čísly. A navíc, není číslo jako číslo.

```
[4]: promenna1 = 10          # celé číslo
      promenna2 = 10.0      # reálné číslo
      promenna3 = 10.0+0.0j # komplexní číslo
      promenna4 = '10'     # řetězec

print(promenna1, promenna2, promenna3, promenna4)

10 10.0 (10+0j) 10
```

Ve výše uvedeném případě můžeme ze všech jedniček napsat dvojky, čímž se *hodnoty* všech čtyřech proměnných změní. To, co se ale nezmění, něco jako povaha těchto hodnot, je tzv. *typ*. Může se zdát, že jde o nedůležitou kategorii, a chápat ji jako jen jako přívažek hodnoty, nicméně při rozvažování o programu hraje typ důležitou roli.

- V závislosti na typu proměnných označuje tentýž zápis velmi odlišné operace
 - $1+2$ bude 3 , protože do operace $+$ vstupují čísla,
 - $'1'+ '2'$ bude $'12'$, protože operandy "sčítání" jsou řetězce,
 - $\langle 1, 2 \rangle + \langle 10, 20 \rangle$ bude jednou $\langle 11, 22 \rangle$ a jindy $\langle 1, 2, 10, 20 \rangle$ podle typu, který symbolizují závorky $\langle \dots \rangle$.

Statické a dynamické typování

Jak se s takovou nejednoznačností může počítačový jazyk vyrovnat? Jsou dvě možnosti:

- Jazyk používá *statické typy* takže ještě před spuštěním kódu, když narazí na identifikátor proměnné, může si být jistý, jaký typ dat v ní je uschován, a vytvořit kód plně využívající dovedností daného počítače. (Pokud jsou například sčítána celá čísla, vystačí s jedinou instrukcí mikroprocesoru). Sem patří jazyky jako FORTRAN a C.
- Jazyk používá *dynamické typy*, kdy typ proměnné se zjišťuje až za běhu programu. Neexistuje pak pochopitelně žádná jedna instrukce procesoru, která by všechny možné situace zvládla. To je příklad jazyka Python. (Námi používaný interpret CPython si místo instrukcí procesoru vytváří návod (pseudokód) jak, se zachovat ať bude typ dat jakýkoli, což zpomaluje výpočet. Jsou ale i další, pokročilejší varianty, které zpomalení zmenšují.)

Rychlost výpočtu, ilustrovaná výše zmínkou o počtu instrukcí potřebných k provedení operace + není jedinou komplikací, kterou jazyk s dynamickými typy přináší.

Protože v Pythonu získá proměnná typ spolu hodnotou při exekuci přiřazovacího příkazu, musí při chápání významu operací s takovou proměnnou čtenář kódu **více přemýšlet**. např. pro kód `p = a + b` na otázku, jaký typ dat obsahuje `p`, nemůžeme odpovědět, aniž známe *typ* `a` a `b`. Na to nemusíme (nemůžeme) samozřejmě znát jejich konkrétní hodnotu, ale právě jen jejich typ. Právě ten rozhodne, co znamená `+`. Z dobrého kódu by mělo být zřejmé, jakého typu ta která proměnná je.

Ještě uvidíme, že do hádání typu proměnných a výrazů se v moderních IDE použít i editor kódu, který označuje nesprávné operace a napovídá, jaké operace jsou s daným typem možné.

Při psaní kódu bychom měli vědět, jakého typu proměnná je. Zjistit typ proměnné lze za běhu pomocí funkce `type` a tedy kód

```
print(a)
print(type(a))
```

(v textové podobě) vypíše nejprve hodnotu proměnné `a` a potom i její typ.

Čtyři proměnné, jejichž hodnotu jsme si již vypsali výše, mají následující typy (spojení `<class x>` zatím překládáme jako "druh datové nádoby na uchovávání `x`"):

```
[5]: print(type(promenna1), type(promenna2), type(promenna3), type(promenna4))
```

```
<class 'int'> <class 'float'> <class 'complex'> <class 'str'>
```

Vidíme zde následující typy proměnných/dat:

- `int` ... celé číslo
- `float` ... reálné číslo
- `complex` ... komplexní číslo
- `str` ... řetězec

Klíčové je mít na paměti, že

5.1 Proměnná: typ + hodnota

Proměnné vstupují (např.) do operací i se svým typem.

Typ proměnné určuje, jaký význam bude mít zapsaná operace a jakého typu bude výsledek

```
1 + 1 --> 2
'1'+'1' --> '11'
```

Hodnota proměnné pak určuje hodnotu výsledku operace.

Bez znalosti typu obou operandů nelze rozhodnout, zda je daná operace legální, např. `a/b` dává smysl jen pro číselné typy. Python má velmi laxní *statickou* kontrolu kódu, takže taková chyba bývá často odhalena až při běhu programu. (Brzy se seznámíme s příklady nástrojů, které umožňují takovou chybu odhalit dříve.)

5.2 Typ výsledku operace je dán typem operandů

Typ proměnných musíme brát v úvahu, když provádíme operace. Následující tabulka ukazuje, co se stane když použijeme operaci `*` mezi různými typy. Povšimněte si, že kromě obvyklého chování čísel tu máme i množení řetězců, `'a' * 4` \rightarrow `'aaaa'`.

Operace `*` mezi číselnými typy

*	int	float	complex
int	int * int \rightarrow int	int * float \rightarrow float	int * complex \rightarrow complex
float	float * int \rightarrow float	float * float \rightarrow float	float * complex \rightarrow complex
complex	complex * int \rightarrow complex	complex * float \rightarrow complex	complex * complex \rightarrow complex

Operace `*` mezi celými čísly a řetězci

*	int	str
int	int * int \rightarrow int	int * str \rightarrow str
str	str * int \rightarrow str	str * str \rightarrow Error

5.2.1 Binární operátory pro číselné výpočty

Zde je seznam nejčastějších binárních operátorů, jaké budeme používat

symbol	význam
+	součet
-	rozdíl
*	součin
/	podíl
**	mocnění
//	celá část podílu
%	zbytek po dělení

Operace zbytku po dělení je definovaná i pro reálná čísla:

```
[4]: import math
a = 314
b = a % (2*math.pi)

print(math.sin(a) , math.sin(b))

-0.15859290602857282 -0.15859290602856096
```

Všechny operace výše s výjimkou operace `/` mají tu vlastnost, že výsledek je celé číslo, jen když jsou oba operandy celé čísla

```
int * int --> int
float * int --> float
int * float --> float
float * float --> float
```

tedy

```
13//12 --> 1
13//12.0 --> 1.0
```

U dělení bylo v Pythonu od verze 3 upuštěno od této vlastnosti. Může totiž být snadnou příčinou chyb, protože zápis

```
energy = 1/2 * m * v * v
```

začíná výpočtem zlomku 1/2, který má v čitateli i jmenovateli celé číslo. Pokud bychom (jak je tomu v C/C++) měli `1/2 == 0`, byla by za všech okolností takto spočtená kinetická energie nulová. V verzi Python 3 se tak můžete spolehnout, že `1/2 == 0.5` protože

```
int / int --> float
```

Nebude ale na škodu, když budete takovéto zlomky psát preventivně např. jako `1.0/2` nebo `0.5`, kdyby si někdo váš kód vypůjčil do programu psaného v jiném jazyce.

Kontrolní otázky

1. Jak zjistíte typ dat obsažených v proměnné?
2. Jakým výrazem spočteme, kolik vagónů (celé číslo) je potřeba k přepravě `m` tun materiálu (celé číslo), je-li nosnost každého `n` (celé číslo)?

Příkaz print a něco o řetězcích

6.1 Příkaz print

Jen stručně: Zavoláním příkazu `print` se na textovém výstupu programu objeví textová podoba jeho argumentů. Ty jsou odděleny mezerou a na konci se ještě přidá nový řádek. Proto

```
print(a, '+', b, '=', a+b)
print(a, '-', b, '=', a-b)
```

vypíše pro `a=5` a `b=2` tyto dva řádky

```
5 + 2 = 7
5 - 2 = 3
```

Lze ale změnit oddělovač (separátor) například na prázdné místo (řetězec `' '`)

```
print(a, '+', b, '=', a+b, sep='')
```

kdy se vypíše

```
5+2=7
```

Podobně lze uvedením `end=' '` zabránit vypsání nového řádku na konci výstupu příkazu `print`. Nebo jej volbou `end=' '` přeměnit na mezeru. Nastavení `end=...` lze libovolně kombinovat s požadovanou volbou `sep=...`

Cvičení: Pro vhodné `a,b,c` zařídte aby `print` vypsalo

```
12345 * 9 + 6 = 111111
```

```
[3]: # zde si to vyzkoušejte
a, b, c = 12345, 9, 6
print()
```

6.2 Řetězce

Ačkoli bude těžiště našeho snažení ve výpočtu numerických hodnot, budeme občas muset pracovat i nečíselnými informacemi. Již jen v okamžiku, kdy potřebujeme své výsledky uložit pro další použití nastává několik situací, které spadají mezi práci s řetězci:

- textová prezentace číselného výstupu vyžaduje tzv. formátování, tedy někdy potřebujeme rozhodnout, zda máme výsledek zapsat jako 101.5, 101.4997214, 1.014997214e2
- pokud jde o tabelovaná data, je potřeba řešit mezery mezi sloupci
- občas potřebujeme přidat komentář, co který sloupec znamená
- nebo přidat mezi čísla nějaký další vysvětlující text
- vytvářet názvy souborů programem

Pokud naopak chceme číselné hodnoty z textového souboru načíst, může situace vyžadovat předzpracování dat na úrovni řetězců. Navíc řetězce obecně představují srozumitelný nenumernický datový typ a lze na nich realizovat zajímavé algoritmy.

Pro naše potřeby se omezíme na minimum. V úvodní části představíme nezbytné základy a až se seznámíme s poli, vrátíme se k některým dalším operacím s řetězci.

6.2.1 Řetězcové literály

Když napíšeme sekvenci znaků mezi uvozovky ("abc") nebo apostrofy ('abc') a přiřadíme ji do proměnné

```
jednotka = 'kg'
```

stane se `jednotka` identifikátorem proměnné, která je typu `str`.

Dědictvím jazyka C je způsob vkládání speciálních znaků do řetězce.

```
retezec = "a\tb\n1\t2"
print(retezec)
```

obsahuje speciální znak přechodu na nový řádek `'\n'` tabulátor `'\t'`, takže vytiskne

```
a  b
1  2
```

Pokud chceme do řetězce vložit apostrof, je nejsnazší jej "obalit" uvozovkami

```
doba_konani = "podzim '23"
```

a obráceně

```
zvolani = '"miluji" Python'
```

Ještě flexibilnější je trojice uvozovek nebo apostrofů. Ta umožňuje do řetězcového literálu čitelně vložit i nové řádky

```
html_text = '''<html>
<body>
  <h1>Hola!</h1>
</body>
</html>
'''
```

je ekvivalentní zápisu

```
html_text = '<html>\n <body>\n  <h1>Hola!</h1>\n </body>\n</html> \n'
```

(To, že je první řádek nepohodlně odsazen lze řešit použitím tří apostrofů následovaných znakem *backslash*: `html_text='''\` a pak můžeme pokračovat na novém řádku, aniž by řetězec tímto novým řádkem začínal.)

Formátované řetězce

Poměrně nedávno se v Pythonu objevily f-strings (formátované literály, interpolované řetězce). Ty umožňují do řetězce vložit identifikátor hodnoty proměnné (nebo celý výraz) a popsat, jak má výsledné číslo vypadat (počet platných cifer, zarovnání atd). Ačkoli je mnoho dalších způsobů, jak dosáhnout stejného výsledku, budeme používat výhradně tento.

Pokud před zápis řetězcové konstanty zapíšeme písmeno f, vyhledají se v tomto řetězci výrazy uzavřené ve složených závorkách a ty jsou nahrazeny jejich hodnotou (tzv. *string interpolation*)

Jednoduchý příklad:

```
str1 = '{a}+{b}={a+b}'
str2 = f'{a}+{b}={a+b}'
print(str1)
print(str2)
```

vypíše pro hodnoty proměnných a=1 a b=2

```
{a}+{b}={a+b}
1+2=3
```

Vidíme tedy, že písmeno f před literálem řetězce způsobí, že výrazy v závorkách jsou nahrazeny jejich hodnotou (samozřejmě převedenou do podoby řetězce).

Navíc v novějších verzích Pythonu (≥ 3.8)

```
print(f'{a+b=}')

```

vypíše

```
a+b=3
```

```
[1]: a, b = 1, 2
print(f'{a}+{b}={a+b}')
```

```
1+2=3
```

Lze ovlivnit, kolik znaků výpis čísla zabere, jak bude centrován a nebo např. kolik desetinných míst chceme. Např. ve jménech souborů pak chceme, aby místo mezer byly u čísel nuly.

Další informace jsou např. [zde](#) a [zde](#)

Následující příklad ukazuje základní formátování celých čísel a čísel reálných.

```
[12]: print("celé číslo")
i = 123
print(f'{{i}}      :__{{i}}__')      # podtržítka slouží jen pro zviditelnění mezer ve_
↳ výstupu
print(f'{{i:6}}    :__{{i:6}}__')    # 6 míst doprava
print(f'{{i:<6}}   :__{{i:<6}}__')    # 6 míst doleva
print(f'{{i:^6}}   :__{{i:^6}}__')    # 6 míst vycentrovat

print("\nběžné číslo")
x = 123.4567890123456
print(f'{{x}}      :__{{x}}__')
print(f'{{x:8.2e}} :__{{x:8.2e}}__')
print(f'{{x:8.2f}} :__{{x:8.2f}}__')
print(f'{{x:8.2g}} :__{{x:8.2g}}__')

print("\nkulaté číslo")
x = 12.4
print(f'{{x}}      :__{{x}}__')
print(f'{{x:8.2e}} :__{{x:8.2e}}__')
print(f'{{x:8.2f}} :__{{x:8.2f}}__')
print(f'{{x:8.2g}} :__{{x:8.2g}}__')

print("\nmalé číslo")
x = 12.4e-12
print(f'{{x}}      :__{{x}}__')
print(f'{{x:8.2e}} :__{{x:8.2e}}__')
print(f'{{x:8.2f}} :__{{x:8.2f}}__')
print(f'{{x:8.2g}} :__{{x:8.2g}}__')

print("\nvelké číslo")
x = 12.4e+12
print(f'{{x}}      :__{{x}}__')
print(f'{{x:8.2e}} :__{{x:8.2e}}__')
print(f'{{x:8.2f}} :__{{x:8.2f}}__')
print(f'{{x:8.2g}} :__{{x:8.2g}}__')
```

celé číslo

```
{{i}}      :__123__
{{i:6}}    :__ 123__
{{i:<6}}   :__123 __
{{i:^6}}   :__ 123 __
```

běžné číslo

```
{{x}}      :__123.4567890123456__
{{x:8.2e}} :__1.23e+02__
{{x:8.2f}} :__ 123.46__
{{x:8.2g}} :__ 1.2e+02__
```

kulaté číslo

```
{{x}}      :__12.4__
{{x:8.2e}} :__1.24e+01__
```

(continues on next page)

(pokračujte na předchozí stránce)

```
{x:8.2f}:__ 12.40__
{x:8.2g}:__ 12__
```

malé číslo

```
{x}      :__1.24e-11__
{x:8.2e}:__1.24e-11__
{x:8.2f}:__ 0.00__
{x:8.2g}:__ 1.2e-11__
```

velké číslo

```
{x}      :__12400000000000.0__
{x:8.2e}:__1.24e+13__
{x:8.2f}:__12400000000000.00__
{x:8.2g}:__ 1.2e+13__
```

Cvičení: Vraťte se k příkladu "12345 * 9 + 6 = 111111" a použijte formátovaný řetězec

Základní operace s řetězci

Podobně jako u číselných proměnných, pro založení proměnné obsahující řetězec použijeme přiřazovací příkaz, např.

```
str_a = 'retezec a'
```

Místo konstanty (literálu) 'retezec a' může na pravé straně stát výraz. Ten se může skládat z binárních operací

- +... spojování řetězců ('a'+ 'b' ----> 'ab')
- *... opakování řetězců ('a' *3 ----> 'aaa')

a z funkcí, které vracejí řetězec.

Funkce str a chr

Tou nejdůležitější je funkce str provádějící konverzi hodnoty libovolného typu na řetězec.

Poznámka: Podobnou roli má i funkce repr. Většinou vrací totožný řetězec jako funkce str, pokud ne, pak by funkce repr by měla vrátit řetězec obsahující více informací.

```
[6]: import math
str(math.pi)
```

```
[6]: '3.141592653589793'
```

Další zajímavou funkcí je chr. Ta převádí celé číslo na znak. Počítače totiž i znaky chápou jako čísla. Pravidla podle kterých jsou znaky očíslovány jsou věcí dohody. Důležitý standard pro čísla 0..127 se jmenoval ASCII, souvisel ještě s dálkopisným přenosem textů a přirozeně přešel i do říše počítačů. Znaky mezinárodní abecedy pak zahrnuje v Pythonu používaný Unicode. Protože jde o kód binární, mají následující řádky 16 nikoli 10 znaků.

Unicode představuje rozšíření ASCII a místo 128 přiřazuje čísla již >100 000 znakům. Znaky pod 32 jsou neviditelné řídicí znaky jako je nový řádek nebo tabulátor, další znaky si můžete prohlédnout na [Wikipedii](#) nebo můžete snadno upravit kód níže tak, aby místo hodnoty 127 vypsál i znaky s kódem až např. 1000.

```
[2]: print('      | 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15')
print('-----| '+'-'*48, end='')
```

(continues on next page)

```

for i in range(32,127):
    if i%16==0:
        print(f'\n{i:>4} | ',end='')
    print(chr(i),end=' ')

```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
32		!	"	#	\$	%	&	'	()	*	+	,	-	.	/	
48		0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64		@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80		P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
96		`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112		p	q	r	s	t	u	v	w	x	y	z	{		}	~	

Úloha: Vyzkoušejte, jaký znak se skrývá pod číslem 9825. (Použijte `print` a `chr`.)

Závěrečná poznámka: Ačkoli se řetězce skládají ze znaků, nemáme dovoleno řetězce měnit, i když chceme změnit jen jediný znak, musíme vytvořit novou hodnotu a do proměnné přiřadit. Řetězce ještě potkáme, až se naučíme pracovat s poli.

7.1 Výstup textových dat do souboru

Doposud jsme používali příkaz `print` za účelem výstupu na *konzoli*. Často potřebujeme výstup programu uložit do nějakého souboru. Můžeme

- Výstup programu zkopírovat z konzole myší do textového editoru.
- Přesměrovat výstup programu do souboru na příkazové řádce pomocí symbolu `>`

```
python program.py > data.txt
```

Uvidíme, že potom vše, co původně program vypisoval, skončí v souboru `data.txt`.

- Modifikovat program tak, aby výstup posílal do souboru. Za tím účelem využijeme argument příkazu `print` pojmenovaný `file`. Ten ale musí mít podobu souboru otevřeného pro zápis, nikoli jen jména souboru. Proto se používá konstrukce

```
[4]: with open("data.txt", "w") as out_file:  
    print(f"1+1="), file=out_file  
    print(f"1+2="), file=out_file
```

Zde jsme použili nový příkaz `with`. Ten je určen k automatickému úklidu důsledků akce, kterou má ve svém (prvním) argumentu. V případě práce se soubory jde o to, že příkaz

```
my_file = open('data.txt', 'w')
```

soubor otevře (např. pro zápis nebo čtení) a ponechá jej za tím účelem otevřený až do okamžiku, než zavoláme operaci

```
my_file.close()
```

(v důsledku objektové natury Pythonu je `close` metoda třídy soubor a nikoli globální funkce.)

V prostředí jupyter/google colab/ipython pak zůstane soubor otevřen i po vyhodnocení dané buňky. To může být nežádoucí a proto se doporučuje použít příkaz `with`, který dobu otevření souboru ohlídá.

O tom, že daný soubor existuje se můžeme přesvědčit příkazem shellu `ls` (pod linuxem) nebo `dir` (pod windows). Jaká data soubor obsahuje zjistíme vyspáním jeho obsahu na konsoli příkazem `cat` (pod linuxem) resp. `type` (pod windows).

```
[8]: !ls -l data.txt
!echo -----
!cat data.txt

-rwxr-xr-x 1 ledvinka ledvinka 12 říj 16 14:32 data.txt
-----
1+1=2
1+2=3
```

Pod windows by příkazy shellu musely vypadat takto

```
!dir data.txt
!echo -----
!type data.txt
```

a výstup by pak byl např.

```
Volume in drive C has no label.
Volume Serial Number is 8C21-A167

Directory of c:\Users\student\Documents\programovani

10/16/2023  12:52 PM                14 data.txt
              1 File(s)                14 bytes
              0 Dir(s)  18,584,666,112 bytes free
-----
1+1=2
1+2=3
```

Pokud se divíte, proč pod linuxem a windows mají soubory různou velikost (12 resp. 14 byte), pak za tím je různá podoba konce řádek na obou platformách ("`\n`" resp. "`\n\r`").

Úlohy

1. Vytvořte programem soubor `triangl.txt` obsahující text v podobě deseti řádek

```

                10
              8 9 10
            6 7 8 9 10
          4 5 6 7 8 9 10
        2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9 10
        2 3 4 5 6 7 8 9 10
          4 5 6 7 8 9 10
            6 7 8 9 10
              8 9 10
                10
```

Pozn. Nejprve odlaďte program aby vypisoval požadovaný text "na obrazovku", teprve poté přidejte `file=soubor`.

2. Alternativně vypište do souboru tabulku dělitelností

```

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
2 . * * * * * * * * * *
3 . . * * * * * * * * *
4 * . . * * * * * * * *
5 * * . * * * * * * * *
6 * * * . * * * * * * *
7 * * * * . * * * * * *
8 * * * * * . * * * * *
9 * * * * * * . * * * *
10 * * * * * * * . * * *
11 * * * * * * * * . * *
12 * * * * * * * * * . *
13 * * * * * * * * * * .
14 * * * * * * * * * * *
15 * * * * * * * * * * *
16 * * * * * * * * * * *
17 * * * * * * * * * * *
18 * * * * * * * * * * *
19 * * * * * * * * * * *
20 * * * * * * * * * * *

```

7.2 Vstup textových dat ze souboru

Protože naše znalosti nedovolují zatím zpracovat složitější podobu vstupních dat, budeme uvažovat úplně nejjednodušší variantu, kdy na každém řádku vstupu bude jediné číslo. Program pak vypíše rozdíl hodnot na posledních dvou řádcích.

V prvním kroku ale musíme nějaká vstupní data pořídit. Abychom se dozvěděli i něco zajímavého z matematiky, data budou mít podobu posloupnosti $a_k = k(k-1)(k-2)/6$ pro $k = 0, 1, 2, \dots, n$.

```

[8]: n = 13
     fn_out = 'posloupnost.txt'

     with open(fn_out, 'w') as f:
         for k in range(n+1):
             print(k*(k-1)*(k-2)//6, file=f)

```

Nyní jsme vytvořili soubor `posloupnost.txt`, který obsahuje čísla 0, 0, 0, 6, 24, 60, 120, 210, 336, 504, 720, 990, 1320, 1716 na každém řádku jedno. V interaktivním režimu se o tom můžeme přesvědčit např. příkazem `! head -6 posloupnost.txt`, který vypíše prvních 6 řádků tohoto souboru.

Protože se ale učíme číst ze souboru, tuto drobnost si napíšeme

```

[18]: fn_in = 'posloupnost.txt'
      print_lines = 6

      with open(fn_in) as fin:
          for line in fin:
              if print_lines <= 0:
                  break

              current_a = int(line)

```

(continues on next page)

```
print(current_a)
print_lines = print_lines - 1
```

```
0
0
0
1
4
10
```

Zde otevíráme soubor voláním `open(fn_in)`, přičemž se neobtěžujeme uvést druhý parametr, protože má výchozí hodnotu 'r', tedy *otevři soubor pro čtení*. Snadno se o tom přesvědčíme zadáním `?open` (zkuste si to).

Proměnná `print_lines` se stará o to, abychom vytiskli nanejvýš požadovaný počet řádek (a kdo to preferuje, může ve stylu C poslední řádek pozměnit na `print_lines -= 1`).

Protože to použijeme za chvíli, řádek, který v rámci příkazu `for` načteme do řetězce `line`, před vypisáním konvertujeme do podoby celého čísla a uložíme do proměnné `current_a`.

Poznámka: Protože by soubor nemusel zamýšlený počet řádek obsahovat, nezvolili jsme variantu obsahující cyklus `for k in range(print_lines)`. Protože by nyní šlo o celočíselný cyklus, místo příkazů `for line in fin` bychom museli použít neprobíranou variantu čtení ze souboru v podobě `line = fin.readline()`.

Nyní můžeme tento krátký kód rozšířit a přidat výpočet rozdílu sousedních řádků. Musíme přidat název výstupního souboru a proměnnou, která si bude pamatovat hodnotu posledního řádku. Tu před započítáním cyklu vhodně inicializujeme.

```
[10]: fn_in = 'posloupnost.txt'
      fn_out = 'diference1.txt'

previous_a = None

with open(fn_in) as fin:
    with open(fn_out, 'w') as fout:
        for line in fin:
            current_a = int(line)
            if previous_a is not None:
                print(current_a-previous_a, file=fout)
            previous_a = current_a
```

Provedením toho kódu jsme vytvořili soubor `diference1.txt`, který obsahuje čísla 0, 0, 6, 18, 36, 60, 90, 126, 168, 216, 270, 330, 396, jak bychom zjistili provedením příkazu shellu `!cat diference1.txt`.

Nás bude zajímat, co by stalo, kdybychom postup opakovali. Máme více možností. Jedna spočívá v tom, že program výše spustíme s `fn_in = 'diference1.txt'` a vhodně pozměněným `fn_out`.

Další možností je přímo napsat program, který počítá a vypisuje tyto druhé diference. To je čtenáři ponecháno jako cvičení.

Konečně je tu možnost, která představuje zajímavou metodu práce s programy na příkazové řádce a kterou si nyní vyzkoušíme.

7.3 Přesměrování vstupu a výstupu programu

Když programy místo z buněk Jupyterového sešitu spouštíme z příkazové řádky, jsme zvyklí, že se dočkáme textového výstupu, který se následně objeví na konzoli. Například

```
tail -2 posloupnost.txt
```

```
220
286
```

zachycuje situaci, kdy na příkazové řádce spustíme program `tail` a požádáme jej, aby nám vypsal poslední dva řádky souboru `posloupnost.txt`.

Pokud příkaz pozměníme na

```
tail -2 posloupnost.txt > poslednidva.txt
```

místo na obrazovce skončí tyto dvě čísla v souboru `poslednidva.txt`.

Podobně se chová na příkazové řádce znak `<`, který podstrčí obsah souboru spouštěnému programu, jako by to byl vstup z klávesnice. V našem případě budeme chtít provést následující příkazy

```
python3 dif.py < posloupnost.txt > diference1.txt
python3 dif.py < diference1.txt
```

První příkaz spustí z pomoci interpretu jazyka python program obsažený v souboru `dif.py`, který spočte to, co jsme již dokázali: načte soubor `posloupnost.txt`, spočte první diference a pošle je do souboru `diference1.txt`. Druhý příkaz pak vezme `diference1.txt` a vypíše první diference dat obsažených v tomto souboru.

Za tím účelem musíme vytvořit soubor `dif.py` obsahující mírně pozměněnou verzi kódu výše:

```
import os

previous_a = None

for line in os.stdin:
    current_a = int(line)
    if previous_a is not None:
        print(current_a-previous_a)
    previous_a = current_a
```

Nejvýraznější změnou je absence obou příkazů `open`. Je to tím, že program vypisuje pomocí základní varianty příkazu `print` bez specifikace výstupního souboru a čte z tzv. *standárního vstupu*. Tím by za normálních okolností, kdybychom nepoužívali přesměrování pomocí `<` byla klávesnice.

Je vhodné zde zmínit základní fakt související se spouštěním programů v příkazové řádce. Každý program dostane při svém spuštění tři otevřené textové soubory - `stdout`, který je otevřen pro zápis. To, co do něj program zapíše se objeví na obrazovce na řádcích následujících spuštění programu. V pythonu tam končí výstup příkazu `print` s výchozí hodnotou nepovinného parametru `file=...`

Pokud jsme použili přesměrování `> vyst_soubor.txt`, je před spuštěním programu vytvořen prázdný soubor `vyst_soubor.txt` a výstup programu končí v něm. Po dobu běhu programu obsahuje `vyst_soubor.txt` částečnou podobu výstupu spuštěného programu, po skončení běhu programu je `vyst_soubor.txt` uzavřen a obsahuje vše, co program poslal na `std_out`. - `stdin`, který je otevřen pro čtení a skončí v něm to, co píšeme na klávesnici. Pokud používáme přesměrování `< vst_soubor.txt`, přicházejí místo klávesnice data z tohoto souboru.

Vstupní soubor má konečnou délku. Pokud probíhá vstup z klávesnice, existuje speciální kombinace kláves (ctrl-Z pod windows, jinde ctrl-D), která má význam konce vstupu. - stderr, který je otevřen pro zápis a to, co do tohoto souboru program pošle se také objeví na obrazovce. Důvod k existenci tohoto kanálu je, že i pokud přesměrujeme stdout do souboru pomocí >, chybová hlášení se stále objevují na obrazovce.

Tolik teorie. Abychom si to mohli vyzkoušet potřebujeme mít zdrojový text v souboru dif.py. Protože pracujeme ve virtuálních sezeních prostředí Jupyter, začínáme s prázdným adresářem a všechny soubory si musíme vytvořit. Soubor posloupnost.txt jsme vytvořili spuštěním kódu výše. Soubor dif.py vytvoříme přesměrováním příkazu shellu echo -e "text1\n\ntext2", který vypíše dva řádky, na prvním text1 a na druhém text2.

```
[15]: !echo -e 'import sys\n\nprevious_a = None\n\nfor line in sys.stdin:\n    current_a = \n    ↪int(line)\n    if previous_a is not None:\n        print(current_a-previous_a)\n    ↪previous_a = current_a' > dif.py
!cat dif.py
```

```
import sys

previous_a = None

for line in sys.stdin:
    current_a = int(line)
    if previous_a is not None:
        print(current_a-previous_a)
    previous_a = current_a
```

Příkazem cat dif.py jsme ověřili, že soubor obsahuje zamýšlený kód. Znak vykřičník, jak víme, zařídí, že tyto příkazy nejsou podtrženy interpretu Pythonu, ale shellu systému, pod kterým běží Jupyter.

Nyní již můžeme spočítat druhé diference:

```
[12]: !python3 dif.py < posloupnost.txt > diference1.txt
!python3 dif.py < diference1.txt
```

```
0
1
2
3
4
5
6
7
8
9
10
11
```

Cvičení: V úvodním programu jsme použili celočíselné dělení, což vůbec nevadilo, protože výraz $k(k-1)(k-2)$ je pro celé k dělitelný šesti.

Změňte // na /. Přesvědčte se, že nyní dojde při konverzi int(line) k chybě. Prohlédněte si soubor posloupnost.txt a odhalte rozdíl oproti použití operace /. Vyzkoušejte, že konverze na typ float, tedy výraz ve tvaru float(line) problém řeší.

Cvičení: zkuste postup zopakovat pro jiný polynom třetího stupně, než je $k(k-1)(k-2)/6$ a přesvědčte se, že opět tvoří druhé diference aritmetickou řadu.

Cvičení: Přesvědčte se, že čtvrté diference polynomu třetího stupně jsou nulové. Za tím účelem proveďte čtyři příkazy

shellu, které vytvořit kromě souboru `diference1.txt` také `diference2.txt` a `diference3.txt` aby nakonec vypsaly čtvrté diference.

Cvičení: Program `dif.py` lze spustit místo v shellu Jupyteru na příkazové řádce vašeho počítače. Vyzkoušejte, že počítá diference čísel zadaných z klávesnice místo z přeměřovaného souboru a že když zadáte poslední řádek vstupu, lze o tom program informovat klávesami `ctrl-Z` pod windows nebo jinde `ctrl-D`.

7.3.1 Zřetězení programů v příkazové řádce (*pipe*)

V posledním cvičení se provede několik příkazů, které postupně vytvářejí pomocné soubory (konkrétně `diference1.txt` také `diference2.txt` a `diference3.txt`). Protože nás tyhle mezivýsledky nezajímají, můžeme využít vlastností shellu, který dovoluje zřetězit programy tak, že výstup jednoho se stává vstupem druhého. Nejprve standardní ukážka:

```
[13]: ! head -4 posloupnost.txt | tail -2
```

```
0
1
```

Protože soubor `posloupnost.txt` obsahuje řádky `0, 0, 0, 1, 4, 10, ...`, když příkaz `head -4` vypíše jeho první čtyři řádky, jsou to tyto `0, 0, 0, 1`. Použitím přeměřování, které se způsobí spojením příkazů symbolem `|`, se tyto čtyři řádky stanou vstupem příkazu `tail -2`. Jeho je výstup, tedy poslední dva řádky jeho vstupu, jsou pochopitelně `0, 1`.

Nyní je zřejmé, že druhé diference dat v souboru `posloupnost.txt` můžeme spočítat i bez použití mezisouboru `diference1.txt` takto:

```
[ ]: !python3 dif.py < posloupnost.txt | python3 dif.py
```

Cvičení: Vyzkoušejte takto spočítat i třetí a čtvrté diference.

Shrnutí: Vstup a výstup dat z programu

Data, která program produkuje mohou podle potřeby skončit na obrazovce nebo v souboru. Do souboru je můžeme poslat přeměřováním výstupu na příkazové řádce nebo přímo v programu soubor vytvořit a psát do něj variantou příkazu `print(..., file=...)`.

Čtení dat může také probíhat přímo ze souboru, z konzole nebo z přeměřovaného vstupu. Je však komplikováno tím, že načtená data mají podobu řetězců. Je na řádce více jak jeden číselný údaj, musíme se naučit takový řetězec rozebrat na jednotlivé údaje a ty pak konvertovat na čísla. To se naučíme až za nějaký čas.

Více programů můžeme spustit tak, že výstup jednoho se stane vstupem druhého.

Při psaní programů je pohodlné a správné konkrétní výpočty izolovat do k tomu určených funkcí:

- Pokud takový výpočet potřebujeme v programu na více místech, ušetříme si práci
- Výpočet izolovaný do podoby funkce lze snáze testovat
- Rozdělení na kódu menší části (kdysi se říkalo podprogramy) nám umožňuje snáze rozmyslet o jeho fungování

Hned na první lekci jsme potkali algoritmus pro největší společný dělitel v podobě funkce, jakou nám poradila AI. Představuje dobrou ilustraci toho

- jak vypadá uvedení funkce v kódu,
- jak se používá,
- že je to výhodné.

```
[3]: # spočtáme gcd tří čísel

def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

p, q, r = 4633, 4715, 4879

c = gcd(p, gcd(q, r))

print(c)

41
```

V prostředí Jupyter máme funkci k dispozici od okamžiku provedení buňky obsahující zavedení funkce příkazem `def`. Jednak tuto funkci můžeme *použít* při dalším počítání, také toho ale můžeme využít v jejím testování.

```
[4]: gcd(4455, 9977)
```

[4]: 11

8.1 Definice funkce

Práce s funkcemi zahrnuje dva odlišné kroky. Samozřejmě je to zavolání funkce, podobně jako kus výrazu `math.sin(x)` nějak spočte sinus x a tuto hodnotu pak dále ve výrazu použijeme. Pokud ale požadujeme novou funkci, musíme dodat její definici. Tato definice říká,

- co je funkci potřeba dát za informaci, aby se mohla do výpočtu pustit
- jak to má udělat

Pro ilustraci použijeme jednoduchý příklad, tzv. Fibonacciho posloupnost tvořenou celými čísly $F_0 = 0$, $F_1 = 1$, $F_k = F_{k-1} + F_{k-2}$. Její procházení je otravné, takže použijeme funkci, která se bude jmenovat *fib* a když jí jak argument dáme nějaké číslo n ona vrátí hodnotu F_n . Pro jednoduchost tato naše první funkce vždy spočte příslušný člen posloupnosti jejím procházením znovu od začátku.

```
[6]: def fib(n):
    """Počítá n-tý člen Fibonacciho posloupnosti"""

    assert n >= 0, "Použitý postup neumí záporné argumenty"
    if n == 0:
        return 0;

    a = 0          # inicializace lokálních proměnných
    b = 1
    for i in range(2, n+1):
        a, b = b, a+b    # jeden krok Fibonacciho řady

    return b

print(fib(20))
```

6765

Zde nejprve pár pozorování:

- Zápis definice funkce
 - začíná hlavičkou, která říká, jak se funkce jmenuje a jaké parametry potřebuje, aby mohla spočítat, co se po ní požaduje
 - dále pokračuje příkazy, s jakými jsme se již seznámili
 - ale je mezi nimi jeden nový příkaz `return`
- Seznam formálních parametrů je v závorce za identifikátorem funkce. Protože podobně jako v algebře předpokládáme, že argumenty mohou nabývat různých hodnot, označíme je identifikátory tak, aby jednou mohlo být `n==2` a jindy `n==20`.
- V Pythonu stále ještě spíše není zvykem uvádět typ argumentů. (Ukážeme se, že to jde a že to má smysl)
- Opět je to odsazení (indentace) textu, které určuje, jaké příkazy jsou součástí definice funkce.
- Je zvykem na prvním řádku pod hlavičkou funkce uvést dokumentační řetězec. Tento řetězec se obvykle zobrazí jako nápověda, když v napíšeme název funkce. Trojité uvozovky dovolují jako mít jako součást řetězce nové řádky i uvozovky.

- Je vhodné zkontrolovat parametry. Kontrola v podobě `assert n>=0` kromě záporných n také vyloučí mnohé nečíselné parametry, které způsobí chybu při porovnávání s nulou. (Např. `fib(True)` ale testem projde a funkce nakonec vrátí hodnotu 1.)
- Nevadí nám, že pracujeme s velkými čísly, protože v Pythonu (ne však v *numpy*) mohou mít celá čísla libovolnou velikost, takže `fib(100)==354224848179261915075`.
- Příkaz `return b` zařídí, že provádění funkce se ukončí a ta vrátí (odkaz na) uvedenou hodnotu.
- Ani lokální proměnné se nedeklarují.

Výše definovanou funkci `fib` můžeme použít v dalších výpočtech:

```
[7]: # demonstrace hodnot poměru sousedních členů Fibonacciho posloupnosti
phi = (1+5**0.5)/2
for k in range(5,28,5):
    pomer = fib(k+1)/fib(k)
    print( f"{k:3} {pomer:15.12f} {pomer-phi:9.2g}" )

 5  1.6000000000000000  -0.018
10  1.61818181818182   0.00015
15  1.618032786885   -1.2e-06
20  1.618033998522    9.8e-09
25  1.618033988670   -7.9e-11
```

8.2 Volání funkce

Když máme funkci `fib` stojí za to ověřit známou vlastnost podílu sousedních členů

```
k = 100
print(fib(k+1)/fib(k))
```

Kód sice není zrovna efektivní, ale je naprosto zřejmé, co že to má `print` vypsát.

Funkce `fib` bude volána dvakrát, jednou s argumentem 101, podruhé 100.

Důležité: Těm závorkám za identifikátorem funkce rozumějme jako *spust' výpočet funkce* Proto, když některé funkce nepotřebují argument, stále potřebují aspoň prázdné závorky. Příklad:

```
print()
```

vytiskne prázdný řádek. Jako příkaz bez závorek by nemělo v programu žádný efekt.

8.3 Nepovinné argumenty

Následující funkce vrací řetězec odpovídající danému celému číslu v příslušném základu pozičního zápisu čísel. U druhého argumentu má uvedenu výchozí hodnotu 10.

```
[1]: def int2str(a, radix=10):

    assert 2 <= radix <= 10
    assert type(a) == int

    if a<0:
```

(continues on next page)

```
    return '-' + int2str(-a,radix)

s = ''
while True:
    s = str(a%radix) + s
    a = a // radix
    if a==0:
        break

return s

a = 97
print( int2str(a) )
print( int2str(a,10) ) # stejné jako výše
print( int2str(a,2) )

97
97
1100001
```

Jak příklad ukazuje, tuto hodnotu nemusíme uvádět, pokud nám výchozí hodnota vyhovuje.

Pokud má funkce více argumentů s uvedenou výchozí hodnotou,

```
def f(a, b=0, c=1, d=2):
    print(f' {a=}, {b=}, {c=}, {d=}')
```

můžeme při volání

- uvést všechny v uvedeném pořadí $f(1, 2, 3, 4)$
- uvést jen několik prvních $f(1, 2)$ nebo $f(1, 2, 3)$
- uvést pojmenované argumenty $f(1, d=3)$
- několik prvních argumentů a vybrané pojmenované $f(1, 2, d=3)$

Pozn. Rozmyslete si, že víte, co v každém z výše uvedených příkladů procedura f vytiskne.

8.4 Předávání argumentů

V Pythonu nemůžeme ovlivnit, jakým způsobem se funkcím předávají argumenty. Je vždy ekvivalentní přiřazovacímu příkazu. Podobně jako u přiřazení se nevytváří kopie předávaných dat ale jen další odkaz na ně. Uvidíme, že u např. u polí a seznamů to může mít nezamyšlené důsledky.

8.5 Proměnné lokální a globální

Doposud jsme uvažovali jen programy tvořené sadou příkazů bez definic vlastních funkcí. Identifikátory proměnných, které jsme v programu použili odkazovaly na proměnné, které počaly svůj život prvním přiřazením a přestaly existovat až při ukončení programu. To jsou tzv. **globální proměnné**.

V okamžiku zavolání funkce se odehraje důležitá změna. Kartotéka s globálními proměnnými se odsune stranou a založí se nová kartotéka, nazvěme ji kartotékou lokálních proměnných. Ta zpočátku obsahuje jen identifikátory dané seznamem argumentů funkce. Když uvnitř funkce přiřadíme do identifikátoru, který v ní není se založí nová **lokální proměnná**. A to i tehdy, pokud proměnná s tímto jménem je v kartotéce globálních proměnných.

Obráceně, pokud uvedme nějaký identifikátor jinde než jako cíl přiřazovacího příkazu, nejprve se hledá v kartotéce lokálních proměnných. Teprve, když se tam daný identifikátor nenajde, podívá se Python do kartotéky globálních proměnných. Samozřejmě, pokud ani tam není, jde o chybu.

Odborně se místo o kartotéky používá termín **scope**, řekněme rozsah (platnosti proměnných). Volání funkce tedy přepíná z globálního do lokálního rozsahu.

Někdy můžeme chtít uvnitř funkce přiřadit do globální proměnné. K tomu slouží příkaz `global`. Důsledky jeho použití demonstrujeme na dvojici kódů

<pre> y = 'původní y' def f(x): "y bude globální" global y y = x print(repr(y)) f('změna') print(repr(y)) ----- 'původní y' 'změna' </pre>	<pre> y = 'původní y' def f(x): "y bude lokální" y = x print(repr(y)) f('změna') print(repr(y)) ----- 'původní y' 'původní y' </pre>
--	--

Vidíme, že ve druhém případě je `y` lokální proměnná a její nastavení se nedotkne globální proměnné, i když nese stejný identifikátor.

8.5.1 Rozbor dějů při volání funkce

Uvažujme krátký program:

```
[32]: def symmetric_ratio(x,y):
    "Vrací hodnotu x/y+y/x"
    s = x**2+y**2
    return s/(x*y)

x = 1
y = 4
print( symmetric_ratio(x,y+1), symmetric_ratio(x+1,y) )

5.2 2.5
```

Program obsahuje následující kroky:

- Vytvoření funkce `symmetric_ratio`. Jde o globální identifikátor
- Vytvoření proměnné `x` spojené s jejím nastavením na 1
- Vytvoření proměnné `y` spojené s jejím nastavením na 4
- Spočtení hodnot `x` a `y+1`
 - přepnutí rozsahu z globálního do lokálního rozsahu funkce `symmetric_ratio`
 - vytvoření lokální proměnné `x`. Protože jde o argument, je obsazena předanou hodnotou 1.
 - vytvoření lokální proměnné `y`. Protože jde o argument, je obsazena předanou hodnotou 4.
 - (Dokumentační řetězec se nepřekládá.)
 - vytvoření lokální proměnné `s` a její nastavení na odpovídající hodnotu (26).
 - spočtení 26/5
 - vrácení této hodnoty spojené s přepnutím do globálního rozsahu
- Spočtení hodnot `x+1` a `y`
 - přepnutí rozsahu z globálního do lokálního rozsahu funkce `symmetric_ratio`
 - vytvoření lokální proměnné `x`. Protože jde o argument, je obsazena předanou hodnotou 2.
 - ...
 - spočtení 20/8
 - vrácení této hodnoty spojené s přepnutím do globálního rozsahu
- vytisknutí obou vrácených hodnot

Přepínání rozsahů znamená, že globální `x` a lokální `x` jsou jiné proměnné a program výše je ekvivalencí následujícímu kódu. (Jako simulaci konce existence lokálních úproměnných zde používáme příkaz `del`, který zruší daný identifikátor. Jinak se s ním moc nepotkáme a tak zcela stačí, pokud budete rozumět následujícímu kódu bez oněch řádků, kde se vyskytuje.)

```
[33]: # provádějí se příkazy hlavního programu
x = 1
y = 4

# výpočet a předávání argumentů prvního volání      #####
```

(continues on next page)

(pokračujte na předchozí stránce)

```

param_x = x #
param_y = y+1 #
# vlastní výpočet popsany kódem funkce #####
local_s = param_x**2+param_y**2 #
result = local_s/(param_x*param_y) #
# konec existence lokálních proměnných i argumentů #
del param_x, param_y, local_s #
# konec běhu funkce #####
print_arg1 = result
del result

# výpočet a předávání argumentů druhého volání #####
param_x = x+1 #
param_y = y #
# vlastní výpočet popsany kódem funkce #####
local_s = param_x**2+param_y**2 #
result = local_s/(param_x*param_y) #
# konec existence lokálních proměnných i argumentů #
del param_x, param_y, local_s #
# konec běhu funkce #####
print_arg2 = result
del result

print(print_arg1,print_arg2)
del print_arg1, print_arg2

```

5.2 2.5

8.6 Funkce nebo procedura?

Zrovna na příkazu `print` vidíme, že existují funkce, které nic nevrací. V Pythonu je tohle nic speciální hodnota `None`.

Všimněte si, že hlavička definice nerozlišuje, zda jde o funkci nebo proceduru, vždy je použito `def`. Toto rozlišení nastane v rámci příkazů uvnitř funkce:

- Funkce obsahuje příkaz `return` bez "argumentu"
- Funkce končí bez příkazu na konci

Obě tyto situace jsou ekvivalentní provedení příkazu

```
return None
```

Zde je příklad kombinující obě tyto možnosti:

```

def print_fib(n):
    if n<0:
        print('Neumím záporná n')
        return

```

(continues on next page)

```
print(fib(n))
```

Kdybychom příkaz `return` vynechali, funkce by sice napsala 'Neumím záporná n', ale následovalo by stejně volání funkce `fib` a ta by pak ohlásila chybu díky příkazu `assert`.

Často nemívají procedury žádný `return`, protože nepříjemné situace jsou ošetřeny příkazem `assert`.

```
[21]: def print_triangle(n):
    """Vytiskne trojúhelník z 1,3,5,..n resp. 2,4,6,..,n hvězdiček"""

    assert n>0 and type(n)==int, "Argument print_triangle(n) musí být celé kladné číslo"

    for k in range((n-1)%2+1,n+1,2):
        print(' '*(n-k)//2 + '*'*k)
```

```
print_triangle(5)
```

```
*
***
*****
```

8.7 Lokální proměnné a procedurální programování

Lokální proměnné jsou klíčovou součástí procedurálního programování. Program rozdělíme na menší podproblémy, a když při jejich řešení potřebujeme stav mezivýpočtů někam uložit, neobtěžujeme s tím ostatní. Používáme proměnné, jejichž efekty jsou omezeny jen na tuto funkci.

Důvod pro to je velmi praktický - u složitějších programů je pracné hlídat nezamýšlené interakce mezi jeho jednotlivými částmi a lokalita proměnných tohle usnadňuje. V Pythonu se takové výhody soustředí zejména na to, že pokud explicitně neoznámíme, že chceme modifikovat globální proměnnou, uvnitř funkce vytváříme a nadále modifikujeme pouze soukromou (lokální) proměnnou, která se klidně může jmenovat jako důležitá proměnná globální. Až se seznámíme se statickou kontrolou typů, uvidíme, že i zde jsou lokální proměnné pohodlnější, protože je tím snazší udržet typ konzistentní proměnné, čím kratší je doba jejího života.

První vlastnost si ilustrujme na jednoduchém příkladu. Bývá zvykem celočíselnou proměnnou označovat `i`. Proto dává smysl psát

```
def sum100(n):
    s = 0
    for i in range(0,2*n,2):
        s = s+10**i
    return s

i = 10
while i>0:
    print(sum100(i))
    i = i-1
```

kde používáme `i` jak v hlavním programu tak ve funkci `sum100`. Pokud by `i` nebylo v rámci funkce `sum100` lokální proměnnou, nechoval by se program správně. Snadno se přidáním řádku `global i` na začátek `sum100` přesvědčíme.

8.8 Lokální vs globální proměnné

Víme, že přiřazením do doposud nepoužitého identifikátoru vzniká nová proměnná. Dalším přiřazením se pak mění její hodnota.

Obě tato pravidla musíme nyní upřesnit.

- Přiřazením do identifikátoru doposud nepoužitého **uvnitř funkce** vzniká nová **lokální** proměnná.
- Pokud přiřadíme do identifikátoru, který již existuje pro globální proměnnou, stejně zakládáme proměnnou lokální a tato onu proměnnou globální uvnitř funkce **zastíní**.
- Použitím identifikátoru globální proměnné (nikoli ovšem přiřazením do něj) používáme onu globální proměnnou. U jednoduchých reálných a celočíselných proměnných tedy můžeme přečíst jejich hodnotu ale nikoli ji změnit. U polí to bude časem složitější.
- Chceme-li z vnitřku funkce přiřazením změnit globální proměnnou, musíme tento náš záměr učinit zjevným použitím příkazu `global`

Protože v interaktivním režimu (např. Jupyter) můžeme i po havárii funkce zadávat další příkazy, přináší modifikace globálních proměnných z vnitřku procedur a funkcí riziko. Když totiž taková funkce havaruje např. v důsledku špatných hodnot argumentů, může zanechat globální proměnné v porušeném, nekonzistentním stavu. V takovém případě musíme před další práci uvést globální proměnné do vhodného stavu.

Stejně jako v jiných jazycích můžeme překlepem odkázat na nějakou existující globální proměnnou místo proměnné zamýšlené. V Pythonu je to ale častější problém, protože uvnitř funkce máme přístup ke všem právě existujícím globálním proměnným.

8.9 Životní cyklus proměnných

V Pythonu je život proměnných určen časem provádění příkazu, nikoli řádkem, na kterém je uveden.

- Globální proměnná začne existovat prvním přiřazením do daného identifikátoru. Od tohoto okamžiku ji mohou používat další příkazy hlavního programu a všechny jím volané funkce v libovolné hloubce. Globální proměnná existuje, dokud program neskončí (nebo ji neodstraní příkaz `del`).
- Lokální proměnná začíná existovat prvním přiřazením do daného identifikátoru uvnitř funkce. Její život končí příkazem `return` nebo posledním řádkem procedury. (Neprobíráme vnořené funkce a příkaz `nonLocal`.)

V běžných jazycích představují proměnné jen jména známá překladači a určující pojemnování kousků paměti. V přeloženém programu pak z proměnných zůstanou jen čísla paměťových buněk. Jejich jména jsou případně potřeba jen pro ladění programu. V Pythonu je proměnná něco jako položka v telefonním seznamu účastníka divoké párty. Jak program běží, objevují se nové položky v seznamu proměnných. Aktuální seznam proměnných získáme voláním funkce `dir()`.

8.10 Příkaz `def`

Narozdíl od jazyků jako je C nebo Fortran je v Pythonu definice funkce aktivní příkaz. Podobá se příkazu přiřazovacímu. Například

```
q = x + y
```

spočte hodnotu $x+y$, tato spočtená data se někde nachází a na tuto spočtenou hodnotu přiřazení namíří odkaz vedoucí z identifikátoru `q`. Podobně

```
def p(x,y):
    return x + y
```

přeloží kód funkce (vyzvedni x, vyzvedni y, sečti a vrať). Takto přeložený kód (=data) se někde nachází a *def* na tento kód namíří odkaz vedoucí z identifikátoru *p*.

Tento detail se projeví v okamžiku, kdy funkce obsahuje nepovinné argumenty, jejichž hodnota je určena výsledkem volání jiné funkce, řekněme

```
def hmotnost_kulicky(polomer, material = hustota_materialu("ocel"))
    ...
```

V tom případě se funkce *hustota_materialu* zavolá v okamžiku, kdy provádění kódu dorazí na řádek s *def* *hmotnost_kulicky* nikoli až když funkci voláme, např. `print(hmotnost_kulicky(0.4))`.

Podobně jako do stejné proměnné můžeme několikrát přiřadit a je to poslední přiřazení, které nakonec určuje hodnotu proměnné, můžeme vícekrát po sobě definovat stejně se jmenující funkci, přičemž platí naposled provedená definice funkce.

8.11 Funkce jako argument jiné funkce

Důležité je rozumět dvěma velmi odlišným přiřazením:

```
u = math.sin(x)
v = math.sin
```

Obě přiřazení jsou platné příkazy. Po jejich provedení *u* bude hodnota sinu *x*, takže bude rozumné použít příkaz `print(u)` a vytiskneme tak hodnotu *u*, *v* bude funkce, takže bude rozumné použít příkaz `print(v(y))`, čímž spočteme sinus *y*, který pak vytiskneme.

Tento rozdíl je důležitý právě v situacích, jaké fyzik při programování snadno potká. Řekněme, že chceme numericky počítat hodnotu určitého integrálu $\int_0^\phi \sin(x)dx$. Mohlo by nás napadnout, že toho dosáhneme zavoláním vhodné funkce a vypsáním výsledku takto:

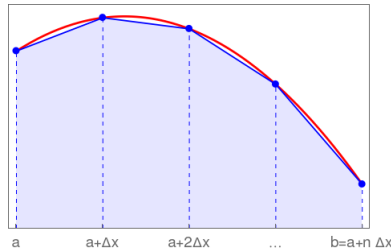
```
print( integral(math.sin(x),0,phi) )    # tohle je špatně
```

Toto je špatně, přičemž mohou nastat dvě typické situace

- proměnná *x* existuje a má číselnou hodnotu, takže se spočte její sinus. To je nějaké číslo, které ale nic neřká o tom jaké hodnoty má sinus jinde a nelze z něj tedy spočítat hodnotu určitého integrálu.
- proměnná *x* neexistuje nebo nemá číselnou hodnotu, takže dojde k chybě při výpočtu sinu.

Aby totiž mohla vůbec funkce `integral` spočítat hodnotu určitého integrálu, musí se ve vhodných bodech zeptat integrované funkce, jaká je její hodnota. Na takové ptaní musí specifikovat hodnotu argumentu a funkci pak zavolat. V Pythonu toho lze dosáhnout snadno, jak ukáže následující příklad.

Ještě si vyložíme, jak spočítat určitý integrál, zatím to berte tak, že napíšeme funkci, která spočte přibližnou hodnotu $\int_{-1}^1 \sqrt{1-x^2}dx$. Učláme to tak, že funkce `vyska_pulkruznice` bude vracet hodnotu integrandu a funkce `integral` bude počítat přibližnou hodnotu určitého integrálu libovolné funkce, kterou dostane jako svůj první argument. Udělá to tak, že spočte součet hodnot integrandu ve vhodných bodech integračního intervalu a součet pak vynásobí délkou dílku (například setiny), na který interval rozdělíme. (Až si vyložíme lichoběžníkovou metodu, uvidíme, proč se v součtu objeví průměr hodnot na krajích intervalu, zatím jen obrázek, ilustrující modrými puntíky, ve kterých bodech funkce `integral` volá funkci *f*)



```
[3]: import math

def integral(f, a, b, n=100):
    "Spočte integral f(x) dx pro x=a..b lichoběžníkovou metodou"
    dx = (b-a)/n
    soucet = (f(a) + f(b))/2
    for i in range(1,n):
        x = a + i*dx
        soucet = soucet + f(x)
    return dx*soucet

def vyska_pulkruznice(x):
    return math.sqrt(1-x*x)

# zkusíme spočíst pi jako dvojnásobek plochy pod kružnicí
print( 2*integral(vyska_pulkruznice,-1,1, n=1000) )

3.141487477002142
```

Protože předávání argumentu je podobné přiřazení $f = \text{vyska_pulkruznice}$, takže uvnitř funkce *integral* argument *f* se chová jako funkce, přičemž zastupuje předanou funkci.

Po dobu běhu funkce *integral* bude první argument nabývat hodnoty *vyska_pulkruznice*, nebo např *sinus*, když, abychom ověřili, že funkce počítá správné hodnoty, zkusíme vyčíslit `integral(math.sin,0,math.pi)` a jindy, řekněme, *rychlost_castice*, což bude námi napsaná funkce jedné proměnné, když budeme chtít spočíst dráhu. Nyní se jen musíme naučit správnou numerickou metodu, napsat lepší variantu funkce *integral* a budeme umět integrovat libovolnou (hezkou) funkci. Bude to podobné, jako správně napsaná funkce *sinus* umí spočíst jeho hodnotu pro libovolné *x*.

Poznámka: To, že funkce jsou vlastně proměnné odkazující na objekt typu funkce vysvětluje, jak máme chápat výstup příkazu

```
print(fib)
```

Vypadá to, že jde o chybu a nejspíš jsme zapomněli uvést argument. Jenže Python to jako chybu nechápe a rád odpoví

```
<function fib at 0x3243f48>
```

To divné číslo říká, kde v paměti jsou informace o přeložené funkci uložena. To, že jsou poměrně komplexní ukáže například to, že obsahují atribut `__doc__`:

```
print(fib.__doc__)
```

tedy vypíše

```
'Počítá n-tý člen Fibonacciho posloupnosti'
```

8.12 Rekurze

Zajímavou možností v procedurálním programování představuje rekurzivní volání. Rozumí se tím situace, kdy

- funkce f volá samu sebe
- funkce f volá další funkci, která sama nebo skrze ještě další funkce zavolá opět f

Nejprve si vše ilustrujeme na obvyklém příkladě, kterým je faktoriál funkce, protože ten splňuje

$$n! = n(n-1)!$$

Následující program tuto vlastnost využívá:

```
[3]: def faktorial(n):
    if n<=1:
        return 1

    n1f = faktorial(n-1)
    return n*n1f

f = faktorial(7)
print(f)

5040
```

Především, aby rekurzivní výpočet mohl skončit, je nezbytné, aby funkce obsahovala větev, která rekurzi nepoužívá a aby u ní výpočet časem skončil.

Druhým důležitým předpokladem je zajímavý charakter lokálních proměnných, o kterém jsme dosud nemluvili. Nejen, že lokální proměnné a argumenty funkce existují jen po dobu jejího volání, ale v případě, že funkce zavolá sama sebe, vnikne nová sada argumentů a lokálních proměnných (chovají se stejně, jen argumenty se inicializují při volání zatímco lokální proměnné až za běhu funkce.)

Abychom si popsali, co se při rekurzi děje, musíme na chvíli připustit, že při své činnosti program není naplňován po řádcích ale po instrukcích. (Nevadí zde, že v Pythonu nejde přímo o instrukce procesoru počítače.) Zjednodušený seznam instrukcí našeho krátkého programu pro vypsání 7! je

```

    # def faktorial(n):
    #     if n<=1:
10     LOAD      (n)
11     LOAD_CONST (1)
12     COMPARE_OP (<=)
13     JUMP_IF_FALSE (to 16)

    #         return 1
14     LOAD_CONST (1)
15     RETURN_VALUE

    #     n1f = faktorial(n-1)
16     LOAD      (n)
17     LOAD_CONST (1)
18     BINARY_OP (-)
19     CALL      (faktorial)
20     STORE     (n1f)
```

(continues on next page)

(pokračujte na předchozí stránce)

```

#   return n*n1f
21  LOAD      (n)
22  LOAD      (n1f)
23  BINARY_OP (*)
24  RETURN_VALUE

# f = faktorial(7)
30  LOAD_CONST (7)
31  CALL      (faktorial)
32  STORE     (f)

# print(f)
33  LOAD      (f)
34  CALL      (print)

```

Číslo na začátku řádku označuje pořadí instrukce, jak je uložena v paměti, velkými písmeny jsou symbolické názvy každé operace/instrukce, v závorkách pak jejich argumenty/operandy. Povšimněte si, že instrukce skoku a volání mají jako argumenty pořadí instrukcí, kde má výpočet pokračovat. (Více nebudeme potřebovat, ale protože fyzika často zajímá, "jak to funguje", ještě pár slov: Instrukce předpokládají jakýsi zásobník, kam se operacemi LOAD ukládají hodnoty a operacemi jako je *, - nebo <= se ze dvou hodnot na vrcholu zásobníku stane jedna. Tu lze pak vyzvednout instrukcí STORE a uložit do proměnné. Po provedení instrukce RETURN_VALUE zůstane tato hodnota na vrcholu zásobníku.)

Z hlediska pochopení průběhu volání při rekurzi nás zajímá jen instrukce CALL. Jde o volání funkce. V programu se vyskytuje 3×. Jedenkrát se volá funkce `print` a na dvou místech se volá funkce `faktorial`. To je důležité, protože je na místě se ptát, jak to program dělá, že se při provádění instrukce návratu z funkce (`RETURN_VALUE`) výpočet vrátí na instrukci za tím správným CALL. Zjednodušená odpověď je, že mezi lokálními proměnnými každé funkce je ještě jedna skrytá, říkáme jí `return_address`, která je instrukcí CALL nastavena tak, že instrukce návratu z funkce si tuto hodnotu může přečíst a pokračovat prováděním instrukce za tím správným CALL.

Následující tabulka schematicky zachycuje stav těchto individuálních lokálních dat v okamžiku, kdy zrovna funkce `faktorial` volaná s $n=3$ spočetla $2!$, uložila tuto hodnotu do lokální proměnné `n1f` a chystá se ji vynásobit 3 (to je hodnota `n`) a vrátit tento součin:

n	n1f	return_address
7		32
6		20
5		20
4		20
3	2	20

Slovní popis tabulky je

hlavní program zavolal `faktorial(7)`

. funkce `faktorial` s hodnotou argumentu $n=7$ spočetla 7-1 a zavolala `faktorial(6)`

... funkce `faktorial` s hodnotou argumentu $n=5$ spočetla 5-1 a zavolala `faktorial(4)`

.... funkce `faktorial` s hodnotou argumentu $n=4$ spočetla 4-1 a zavolala `faktorial(3)`

..... funkce `faktorial` s hodnotou argumentu $n=3$ spočetla 3-1, zavolala `faktorial(2)`, dostala výsledek a ten uložila do `n1f`

Důležitý poznatek: při rekurzivním volání funkcí existuje tolik nezávislých sad lokálních proměnných, jaká je hloubka

rekurze. Součástí stavu výpočtu funkce pak je i informace, kde má výpočet pokračovat, až bude funkce hotová. To samozřejmě platí i bez rekurze, protože tatáž funkce může být volána z různých míst programu a je nezbytné, aby pokračovala ve výpočtu přesně za tím místem (instruckí *call*), kde jsme o výpočet hodnoty funkce požádali. (Protože lokální proměnné zabírají nějaké místo, pak pokud je počet opakování rekurzivního volání velký, může výpočet havarovat pro nedostatek takové paměti - zásobníku. Python konkrétně ohlásí `RecursionError: maximum recursion depth exceeded`.)

Poznámka: Toto je jedno z míst, na které mívá počítačovní záškodníci. Pokud se jim podaří paměť modifikovat tak, aby se po skončení výpočtu nějaké funkce program vrátil jinam než je zamýšleno (řekněme místo zpět do funkce *stav_konta* do vhodného místa funkce *pripis_na_konto*), mohou napáchat rozličné škody.

Jako další ukázkou rekurzivního algoritmu použijeme výpis celého čísla v pozičním systému. Jde o to, že například číslo $n=5471$ lze vypsát tak, že nejprve vypíšeme $n//10 = 547$ a potom $n\%10 = 1$. Rekurze se pak objeví přirozeně, protože stejný postup aplikujeme na číslo 547. Totéž platí i ve dvojkové či hexadecimální soustavě. Jako obvykle u hexadecimálního zápisu použijeme písmena a-f pro hexadecimální cifry 10-15. Za tím účelem se v kódu objeví `chr(ord('a') + c-10)`, tedy znak s kódem `ord('a')` plus o kolik je cifra c větší jak 10.

```
[9]: def print_i2(i, end='\n'):
    assert i>=0

    if i>2:
        print_i2(i//2, end='')

    print(i%2, end=end)

print_i2(253)

#####

def print_i10(i, end='\n'):
    assert i>=0

    if i>10:
        print_i10(i//10, end='')

    print(i%10, end=end)

print_i10(253)

#####

def print_i16(i, end='\n'):
    assert i>=0

    if i>16:
        print_i16(i//16, end='')

    c = i%16
    if c>9:
        c = chr( ord('a') + c-10 )
```

(continues on next page)

(pokračujte na předchozí stránce)

```
print(c, end=end)
```

```
print_i16(253)
```

```
11111101
```

```
253
```

```
fd
```

Úloha - Procvičení nepovinných argumentů funkcí

1. Změňte hlavičku funkce `print_i10` na `def print_i10(i, end_char='\n')`:

(tedy přejmenujte druhý argument této funkce) a následně pozměňte vnitřek funkce, tak aby fungovala jako dopsud

2. Následně změňte hlavičku funkce `print_i10` na `def print_i10(i, end_char):`:

(tedy vynechte automatickou hodnotu druhého argumentu) a příslušně upravte její použití, aby dávalo stejné výsledky jako dopsud

Úloha - Zobecněte všechny tři funkce do jediné s dodatečným argumentem `radix`, jehož výchozí hodnota je 10, která zkontroluje, že jeho hodnota je mezi 2 a 36 a pak číslo vypíše.

Funkce, co něco počítají

Velmi často narazíme na situaci, kdy je potřeba spočítat hodnotu nějakého výrazu závislého na parametrech. Funkce jsou správným způsobem zápisu takového kódu, a v programování je tedy chápeme podobně jako v matematice.

Pokud tedy potřebujeme pracovat s výrazem

$$\sum_{k=0}^n \frac{x^k}{1+k^2}$$

je rozumné si jej označit nějakou zkratkou. Pokud tak činíme v rámci programování, znamená to jako součást našeho programu definovat funkci a tu pak na příslušném místě použít.

Nutno zdůraznit, že takový postup je správný, i když funkci používáme jen na jednom místě: * Jakmile jde o netriviální vzoreček, protože to povede k zpřehlednění kódu. V případě delšího vzorečku navíc bude režie související s voláním funkce zanedbatelná. * Kdykoli izolace takového výpočtu znamená zpřehlednění kódu (\implies méně chyb). * Pokud výpočet vzorce vyžaduje pomocné proměnné, stanou se z nich lokální proměnné a nepřekáží. * Protože funkci lze většinou testovat nezávisle na dalším kódu.

Již víme, jak funkci napsat a že vzoreček výše můžeme přepsat takto

```
[ ]: def moje_funkce(x,n):
    "Sčítá n členů mocninné řady x^k/(k^2+1)"
    soucet = 1
    xk = 1
    for k in range(1,n+1):
        xk = xk*x
        soucet = soucet + xk/(1+k*k)

    return soucet

moje_funkce(1/2,1)
```

```
1.25
```

Protože přepisování vzorečků do podoby funkcí v nějakém programu je častá činnost, rozeberme si podrobněji, co takový proces zahrnuje.

Nejprve musíme identifikovat vlastnosti vzorečku, které určují podobu kódu

- konstanty (a zda je chápat jako lokální nebo globální)
- parametry + typy parametrů (celá čísla, reálná, komplexní, vektor, pole, ...)
- druhy a meze cyklů resp. podmínky jejich ukončení
- optimalizace

S uvážením výše uvedených hledisek přepsat vzorec do podoby kódu.

Optimalizaci budeme uvažovat jen naprosto elementární:

- opakující se podvýrazy
- pokud se ve vztahu vyskytují hodnoty $1, x, x^2, \dots, x^n$ nepotřebujeme mocnění, stačí nám v každém kroku spočítat (viz výše) $xk = xk * x$
- podobně $1!, 2!, 3!, \dots, n!$ počítáme pomocí opakovaným násobením $kfakt = kfakt * k$
- v obou případech nesmíme zapomenout na inicializaci např. $xk=1$
- další varianty téhož jsou např. pokud se v \sum_k nebo \prod_k objevují $x^{2k+1}, k!!$ atp.

Méně elementární optimalizací je postupný výpočet $\sin(x), \sin(2x), \dots, \sin(nx)$. Ten lze pomocí součtových vzorců redukovat na násobení (ano, i když musíme navíc počítat i $\cos(kx)$, stejně se to vyplatí)

To, kdy místo jednoho vzorce vezmeme jiný, efektivnější, již nespadá do programování. Příklad potkáte na Prosemináři matematické fyziky u výpočtu velkých Fibonacciho čísel:

$$F_n = F_{n-1} + F_{n-2} \quad \text{vs.} \quad F_n = [\phi^n - (-\phi)^{-n}] / \sqrt{5}.$$

9.1 Příklad jednoduché funkce dané vzorečkem

$$h(n) = \sum_{k=1}^n \frac{1}{k}$$

Pozorování: je to funkce celočíselného argumentu a vrací reálné číslo. V Pythonu není povinností typ argumentu ani výsledku explicitně uvádět. Proto vystačíme s kódem

```
[ ]: def harmonic(n):
    """Vrací hramonická čísla h_n pro celé n>=0"""
    s = 0.0
    for k in range(1,n+1):
        s = s+1/k

    return s
```

Při psaní tohoto kódu byl zřejmý postup výpočtu doplněn o minimální *docstring*. Bylo také nutno nějak zvolit název funkce a identifikátory lokálních proměnných a zvážit, zde se projeví, když n není celé, a jestli to musíme hlídat. (Rozmyslete+vyzkoušejte.)

Potenciální chyba: pro záporná n vrací bez protestu 0.

9.2 Složitější funkce

Pro obvod elipsy s poloosami a, b snadno najdeme vzorec

$$O = 2\pi a \left[1 - \sum_{k=1}^{\infty} \left(\frac{(2k-1)!!}{(2k)!!} \right)^2 \frac{\epsilon^{2k}}{2k-1} \right]$$

kde např. $7!! = 1.3.5.7$ a $\epsilon^2 = 1 - b^2/a^2$. Protože sčítat nekonečně členů nelze, obsahuje funkce dodatšný parametr `rtol` stanovující akceptovatelnou relativní chybu. Podmínka $2 \cdot \text{abs}(ds) < \text{rtol} \cdot (1 - \text{eps2})$ je odhadnuta na základě vzorce pro zbytek geometrické řady s přidáním rezervou $2x$.

Navíc s použitím tzv. eliptických funkcí je možné psát $O = 4aE(\epsilon^2)$, kde E nalezeneme jako funkci `ellipe` v `scipy.special` místo v `math`.

Vidíme dva různé postupy dávající v rámci tolerance stejný výsledek.

```
[ ]: import math
import scipy.special

def obvod_elipsy(a, b, rtol=1e-10):
    if b>a:
        a,b = b,a

    assert 3e-12 < rtol < 0.1, "Parametr rtol mimo rozumný rozsah."
    eps2 = 1-b**2/a**2
    max2k = 1_000_000

    s = 1
    f2 = 1
    dve_k = 2
    while dve_k<max2k:
        f2 = f2*((dve_k-1)/dve_k)**2 * eps2
        ds = f2/(dve_k-1)
        s = s-ds
        dve_k = dve_k+2
        if 2*abs(ds)<rtol*(1-eps2):
            return 2*math.pi*a*s

    assert False, "Soucet rady nekonverguje"

def obvod_elipsy_presne(a, b):
    """Výpočet obvodu elipsy s použitím eliptických integrálů z knihovny scipy"""
    eps2 = 1-b**2/a**2
    return 4*a*scipy.special.ellipe(eps2) # zajímavost: funguje i pro eps2<0, proto_
    ↪ nezkoumáme zda a>b

obvod_elipsy(1,0.005)/obvod_elipsy_presne(1,0.005) - 1
6.292877330338342e-11
```

```
[ ]: # Ukázka argumentů, pro které assert funkce selže
ae = 1
be = 0.0001
obvod_elipsy(ae,be)/obvod_elipsy_presne(ae,be) - 1
```

9.3 Polynomy a Hornerovo schéma

Další obvyklou formou v níž může být užitečný vzorec zapsán je polynomiální aproximace.

Zde si ukážeme, že pro výpočet hodnoty polynomu nepotřebujeme mocnění, pokud použijeme chytré závorky protože existuje tzv. Hornerovo schéma

$$a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \dots + a_n x^n = a_0 + x \left(a_1 + x \left(a_2 + x \left(a_3 + \dots + x \left(a_{n-1} + x a_n \right) \dots \right) \right) \right).$$

Komentář ke kódu

- funkce kontroluje platnost předpokladů pomocí `assert`
- výrazy mohou pokračovat na dalším řádku, pokud je nový řádek uvnitř závorek
- Hornerovo schéma v proměnné `eps2`
- Duhové závorky v editoru
- Koeficienty se naučíte získat v jiných předmětech

```
[ ]: def obvod_elipsy_maleeps(a, b):
    if b>a:
        a,b = b,a

    eps2 = 1-b**2/a**2
    assert eps2<0.2, f"aproximace vyžaduje eps2<0.2, ale eps2={eps2:4.2g}"

    return 2*a*(3.141592653589794 +
        eps2*(-0.7853981633973549 + eps2*(-0.14726215563710238 +
        eps2*(-0.06135923157099829 + eps2*(-0.03355582977261537 +
        eps2*(-0.021140163218417073 + eps2*(-0.014533859052562739 +
        eps2*(-0.010604501590171097 + eps2*(-0.008077869381533908 +
        eps2*(-0.006330939782841333 + eps2*(-0.005106344943641083 +
        eps2*(-0.004691669404826085 - 0.003988570536164187*eps2))))))))))

obvod_elipsy_maleeps(1, 0.9)/obvod_elipsy_presne(1,0.9) - 1
2.220446049250313e-16
```

9.4 Testujeme funkce

Poté, co kód funkce napíšeme je na místě vyzkoušet, jestli jsme někde neudělali chybu. Víme, že toto nebudeme zkoumat na úrovni matematického důkazu, ale spíše empiricky. Základním důvodem nepřeskočit testování nějaké funkce je fakt, že nám to ušetří práci později, až nebude fungovat program, který funkci používá.

Již jsme viděli, příklad situace, kdy máme dvě alternativní funkce počítající tentýž výraz. Pak můžeme zkoumat

- zda dávají stejné výsledky
- kdy je která efektivnější (rychlejší)

Většinou ale máme ale jen jednu funkci, pak můžeme

- Nakreslit graf funkce pro vhodný rozsah parametrů. Stojí za to, tak učinit podrobně v okolí problematických bodů.
- Srovnat hodnoty ve vybraných bodech s hodnotam z tabulky či jiných zdrojů (např. Wolfram Alpha).

- Zkontrolovat zda funkční hodnoty splňují nějakou identitu, která pro danou funkci platí, např. $\Gamma(x+1) = x\Gamma(x)$.

9.5 Příklad demonstrující doktrínu strukturovaného programování

Následují dva programy vykreslující statistiku interferenčního obrazce. Ten druhý demonstruje doktrínu strukturovaného procedurálního programování tím, že kód je rozdělen na hlavní program a dvě funkce.

Interferenční obrazec je dán hustotou pravděpodobnosti, kterou popíšeme nějakou modelovou funkcí, např.

$$p(x) = \left(\frac{\sin x}{x} \right)^2.$$

Program generuje náhodné hodnoty respektující $p(x)$ a to tak, že pro náhodné x spočte $p(x)$ a pak si "hodí kostkou" a s pravděpodobností $p(x)$ danou hodnotu x přijme (a tedy s pravděpodobností $1 - p(x)$ tuto hodnotu x zahodí).

Program je doplně kódem pro Jupyter uvedeným v bodech 2-4, který výstup programu uložený v souboru "data.txt" vykreslí a zobrazí.

```
[ ]: ## 1. kód vytvářející textový soubor s tabulkou čísel

import random
import math

# rozměry stínítka interferenčního obrazce
a = 18

deset_na_p = 10

with open("data.txt","w") as f:
    for p in range(1,6):
        n = 0 # počet dopadů na stínítka
        while n < deset_na_p:
            x = -a + 2*a*random.random() # náhodná souřadnice x z intervalu <-a,a>

            if math.sin(x)**2 > random.random()*x*x: # dopadne zde ?
                y = -0.4 + 0.8*random.random() + p # náhodná souřadnice y
                print(x, y, file=f)
                n = n + 1
        print(end="\n\n", file=f)
        deset_na_p = deset_na_p * 10

## 2. kód vytvářející skript pro gnuplot

gnuplot_skript = """
set term pngcairo size 1000,600 enhanced font "Arial,12.0" # font je nutno specifikovat!
set output "obrazek.png"

plot "data.txt" linecolor "black" pt 7 ps 0.4
"""

with open("prikazy.gp","w") as f:
    print(gnuplot_skript, file=f)
```

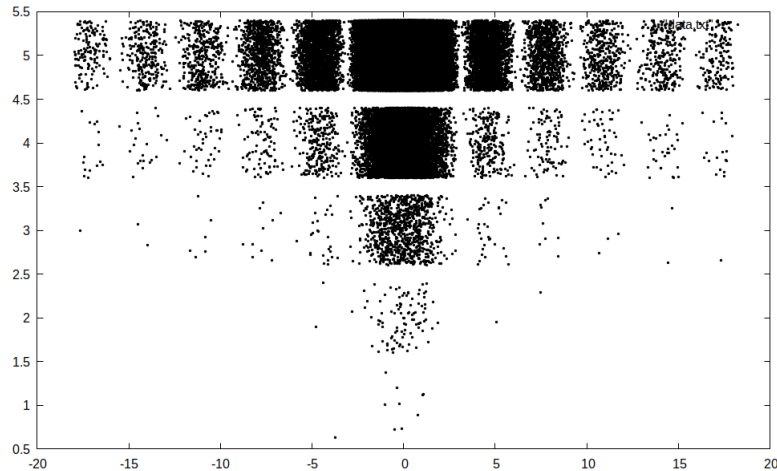
(continues on next page)

```
## 3. kód spouštějící gnuplot

!gnuplot prikazy.gp

## 4. kód zobrazující vytvořený obrázek

from PIL import Image
display(Image.open("obrazek.png"))
```



```
[ ]: ## 1. kód vytvářející textový soubor s tabulkou čísel
```

```
import random
import math

# rozměry stínítka interferenčního obrazce
a = 18

def pravdepodobnostni_funkce(x):
    "funkce p(x)= ( sin(x)/x )^2"
    # POZOR: funkce musí mít obor hodnot spadající do intervalu (0,1)
    if x==0:
        return 1
    return (math.sin(x)/x)**2

def nahodny_bod():
    "generuje nahodna x,y respektujici hustou pravdepodobnosti p(x)"
    b = 0.4
    while True:
        x = -a + 2*a*random.random()          # náhodná souřadnice x z intervalu <-a,a>

        hod_kostkou = pravdepodobnostni_funkce(x) > random.random()
        if hod_kostkou:
            y = -b + 2*b*random.random()      # náhodná souřadnice y z intervalu <-b,b>
```

(continues on next page)

(pokračujte na předchozí stránce)

```

    return x,y

with open("data.txt","w") as f:
    for p in range(1,6):
        for n in range(10**p):
            x,y = nahodny_bod()
            print(x, y+p, file=f)
        print(end="\n\n", file=f)

## 2. kód vytvářející skript pro gnuplot

gnuplot_skript = """
set term pngcairo size 1000,600 enhanced font "Arial,12.0" # font je nutno specifikovat!
set output "obrazek.png"

plot "data.txt" linecolor "black" pt 7 ps 0.4 notitle
"""

with open("prikazy.gp","w") as f:
    print(gnuplot_skript, file=f)

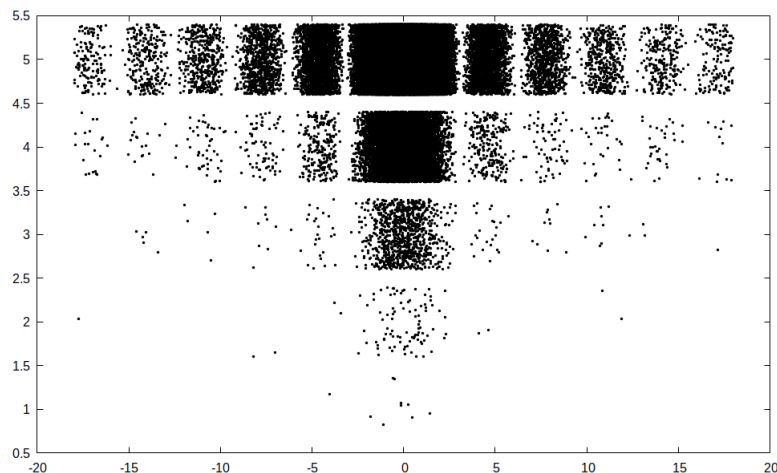
## 3. kód spouštějící gnuplot

!gnuplot prikazy.gp

## 4. kód zobrazující vytvořený obrázek

from PIL import Image
display(Image.open("obrazek.png"))

```



Porovnejme oba kódy abychom ocenili, že rozdělení kódu do funkcí přináší vyšší srozumitelnost

```

import random
import math

# rozměry stínítka interferenčního obrazce
a = 18

deset_na_p = 10

with open("data.txt","w") as f:
    for p in range(1,6):
        n = 0
        while n < deset_na_p:
            x = -a + 2*a*random.random()

            if math.sin(x)**2 > random.random()*x*x:
                y = -0.4 + 0.8*random.random() + p
                print(x, y, file=f)
                n = n + 1
        print(end="\n\n", file=f)
        deset_na_p = deset_na_p * 10

# rozměry stínítka interferenčního obrazce
a = 18

def pravdepodobnostni_funkce(x):
    "funkce p(x)= ( sin(x)/x )^2"
    # POZOR: funkce musí mít obor hodnot spadající do intervalu (0,1)
    if x==0:
        return 1
    return (math.sin(x)/x)**2

def nahodny_bod():
    "generuje nahodna x,y respektujici hustou pravdepodobnosti p(x)"
    b = 0.4
    while True:
        x = -a + 2*a*random.random()      # náhodná souřadnice x ∈ <-a,a>

        hod_kostkou = pravdepodobnostni_funkce(x) > random.random()
        if hod_kostkou:
            y = -b + 2*b*random.random()  # náhodná souřadnice y ∈ <-b,b>
            return x,y

with open("data.txt","w") as f:
    for p in range(1,6):
        for n in range(10**p):
            x,y = nahodny_bod()
            print(x, y+p, file=f)
        print(end="\n\n", file=f)

```

9.6 Funkce definované rekurentními vztahy

Dobře známe rekurzivní definici faktoriálu:

$$n! = n (n - 1)!, \quad 0! = 1$$

Tu do podoby funkce přepíšeme takto

```

def faktorial(n):
    "Výpočet faktoriálu rekurzí"
    assert n >= 0
    if n == 0:
        return 1
    return n * faktorial2(n-1)

```

Podobně můžeme psát pro Fibonacciho posloupnost

$$f_n = f_{n-1} + f_{n-2}, \quad f_0 = 0, \quad f_1 = 1$$

tedy

```

[ ]: def fib(n):
    assert n>=0 and type(n) == int
    if n<2:
        return n

    return fib(n-1) + fib(n-2)

print(fib(35))

```


9227465

Pro větší hodnoty argumentu je ale funkce velmi pomalá a v určitý okamžik bychom ji překonali ručními výpočty. To proto, že bychom použili lineární algoritmus. Jeho varianta v podobě funkce vypadá takto:

```
[ ]: def fib(n):
    "Fibonacciho posloupnost, n-tý člen"

    assert n >= 0
    if n == 0:
        return 0

    a, b = 0, 1
    for i in range(2, n+1):
        a, b = b, a+b
    return b
```

Pokud bychom neměli k dispozici vícenásobné přiřazení (může se vám hodit např. v C), pak by kód vypadal takto

```
[ ]: def fib(n):
    "Fibonacciho posloupnost, n-tý člen"

    assert n >= 0
    if n == 0:
        return 0

    a = 0
    b = 1
    for i in range(2, n+1):
        c = a + b
        a = b
        b = c
    return b
```

Mohlo by se zdát, že zbytečně věnujeme spoustu času nudným Fibonacciho číslům, nicméně ve fyzice (nejen té matematické) potřebujeme často např. Legendery polynomy. Ty jsou podobně jako jiné důležité funkce dány formulí podobnou Fibonacciho:

$$P_k(x) = \frac{2k-1}{k} x P_{k-1}(x) - \frac{k-1}{k} P_{k-2}(x)$$

přičemž tato rekurentní formule začíná

$$P_0(x) = 1, \quad P_1(x) = x$$

```
[ ]: def legendreP(n, x):
    "Legendery polynom"

    assert n >= 0
    if n == 0:
        return 1

    a = 1
```

(continues on next page)

```

b = x
for i in range(2, n+1):
    c = ((2*k-1)*x*b-(k-1)*a)/k
    a = b
    b = c
return b

```

Abychom mohli pohodlně ocenit pohlednost těchto polynomů použijeme v následujícím kódu funkci `grafFunkce`, která namaluje funkci, kterou jí předáme jako argument. Jak to dělá se ale dozvíme později.

```

[1]: import matplotlib.pyplot as plt
import numpy as np

def grafFunkce(f, a=0, b=1, n=300, show=False, arg1=None, arg2=None, **opts):
    """
    Namaluj graf f(x) pro x z intervalu <a,b>

    Jako volitelné argumenty lze užít např.
    c/color, ls/linestyle, lw/linewidth
    viz: https://matplotlib.org/stable/api/\_as\_gen/matplotlib.pyplot.plot.html

    show = True .... vynutí vykreslení grafu, nečeká se na konec buňky.

    arg2 = a ... graf funkce f(x,a)
    arg1 = b ... graf funkce f(b,x)

    """

    xVals = np.linspace(a,b,n)
    if arg1 is None and arg2 is None:
        yVals = [f(x) for x in xVals]
    elif arg1 is None:
        yVals = [f(x, arg2) for x in xVals]
    elif arg2 is None:
        yVals = [f(arg1, x) for x in xVals]
    else:
        yVals = [f(arg1, arg2, x) for x in xVals]

    plt.plot(xVals, yVals, **opts)

    if show:
        plt.show()

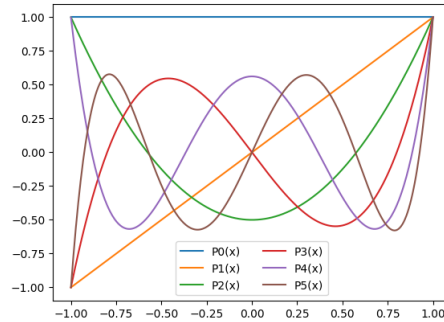
```

```

[ ]: for k in range(0,6):
    grafFunkce(legendreP, -1, 1, arg1=k, label=f"P{k}(x)") # graf

leg = plt.legend(loc='lower center', ncols=2) # jen pro informaci

```



9.7 Hledání kořenů funkcí, metoda půlení intervalu

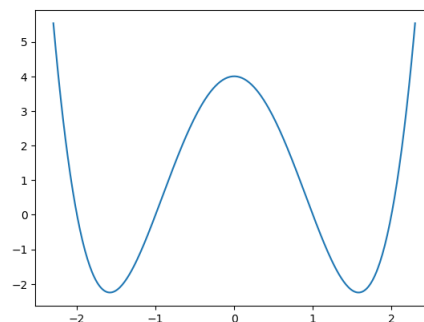
Mnoho funkcí je definován jako řešení nějaké rovnice. Například odmocnina. Proto si ukážeme dvě metody hledání kořenů a jak je použít v definici nové funkce.

Začneme metodou jednoduchou, spolehlivou a často postačující svým výkonem. Je založena na faktu, že spojitá funkce musí mít mezi dvěma body, kde nabývá opačného znaménka nějaký kořen. Spočte proto, jaké znaménko má funkce uprostřed dného intervalu a podle znaménka pak zúží hledání na levou nebo pravou polovinu intervalu. Pak se postup opakuje, dokud není interval dost malý, aby nám libovolný jeho bod stačil jako odhad kořene funkce. Protože délka intervalu se s počtem kroků zmenšuje jako 2^{-n} , nebudeme takových půlení při rozumném počátečním odhadu intervalu potřebovat příliš ($2^{-10} = 1/1024 \approx 10^{-3}$, tedy $2^{-50} \approx 10^{-15}$).

Definujeme si nejprve vhodnou funkci:

```
[3]: def f(x):
    x2 = x*x
    return (x2 - 4)*(x2 - 1)

grafFunkce(f, -2.3, 2.3)
```



```
[4]: def koren_f(a,b, eps=1e-12):
    fa = f(a)
    fb = f(b)

    assert fa*fb<0, "Metoda půlení intervalu vyžaduje opačná znaménka na koncích intervalu"

    while abs(b-a)>eps:
        c = (a + b) / 2 # spocti pulku intervalu
        fc = f(c)      # spocti funkci honotu tam
```

(continues on next page)

```

if fc ==0:
    return c

if fa*fc > 0:
    a = c      # podle znamenka uprav meze intervalu
    fa = fc    # není nutné
else:
    b = c
    fb = fc

return (a+b)/2

```

```
koren_f(0,1.5)
```

```
[4]: 1.0000000000000001137
```

V následujícím kódu modifikujeme funkci `f(x)` tak, aby kreslila puntíky v místech, kde se ptáme na její funkční hodnotu. Můžeme pak vidět, jak metoda postupuje.

```
[10]: f_kresli_xy = False
```

```

def f(x):
    x2 = x*x
    y = (x2 - 4)*(x2 - 1)
    if f_kresli_xy:
        plt.plot([x,x],[0,y],'-')
    return y

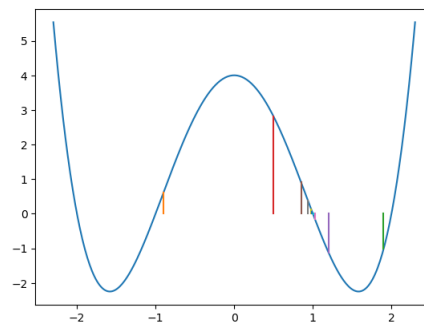
```

```
grafFunkce(f, -2.3, 2.3)
```

```
f_kresli_xy = True
```

```
koren_f(-0.9,1.9)
```

```
[10]: 1.000000000000000453
```



Na předchozím příkladě byl zajímavý také způsob, jak jsme funkci `f` předali informaci, zda má malovat úsečky znázorňující postup půlení. Použili jsme globální proměnnou `f_kresli_xy`. Připomeňme si, že pokud tuto neměníme, nemusíme uvnitř funkce psát `'global f_kresli_xy'`.

9.8 Newtonova metoda hledání kořene (metoda sečen)

Ta spočívá v nahrazení funkce její tečnou v bodě x_k , popsanou rovnicí $f_{\text{tec}}(x) = f(x_k) + f'(x_k)(x - x_k)$, pro niž lze snadno spočítat hodnotu kořene.

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

Tento postup opakujeme dokud nedostaneme dostatečně přesný odhad kořene.

Zatím se následujícímu kódu budeme věnovat jen z hlediska toho, že

- definuje dvě funkce, které vrací $f(x)$ a $f'(x)$
- ve dvou krocích provede Newtonův postup hledání kořene
- vytiskne výsledek

Doporučení: změňte `x0` podívejte se jak se změní obrázek.

Až se seznámíme s poli a knihovnou `matplotlib`, vrátíme se k tomuto kódu z hlediska programování:

- stanoví rozpětí vertikální osy, která bude na grafu vidět. Polynom 4. stupně nabývá rychle velkých hodnot a je třeba rozsah grafu takto omezit.
- namaluje graf funkce $f(x)$
- namaluje graf tečny (vzorec rovnice tečny je uveden výše, o výpočet příslušných hodnot se postará funkce `tecna0(x)`.)
- namaluje osu x
- namaluje puntíky tam, kde se nacházejí x_0, x_1, x_2 . (Tak se ilustruje chování Newtonovy metody.)

```
[16]: def f(x):
      return (x*x-4)*(x*x-1)

      def f1(x):
          return x*(4*x*x-10)

      x0 = 0.5

      x1 = x0 - f(x0)/f1(x0)

      x2 = x1 - f(x1)/f1(x1)
      print(f"{x0} --> {x1} --> {x2:6.4f}")

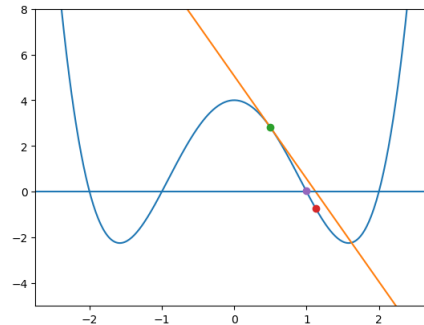
      def tecna0(x):
          return f(x0) + f1(x0)*(x-x0)

      plt.ylim(-5,8) # nastavení rozsahu osy y
      grafFunkce(f, -2.5, +2.5) # graf f(x)
      grafFunkce(tecna0, -2.5, +2.5) # graf tečny
      plt.axhline() # namaluj osu x
      plt.plot([x0], [f(x0)], 'o') # puntík na souřadnice {x0, f(x0)}
      plt.plot([x1], [f(x1)], 'o') # {x1, f(x1)}
      plt.plot([x2], [f(x2)], 'o') # {x2, f(x2)}
```

(continues on next page)

```
plt.show()
```

```
0.5 --> 1.125 --> 0.9942
```



Úloha Změňte v úloze výše hodnotu x_0 a podívejte se, jaký kořen (pokud vůbec) ve dvou krocích najde.

Úloha Vyzkoušejte, že v úplné blízkosti kořene metoda tečen každým krokem zhruba zdvojnásobí počet platných míst.

Ted, když jsme vyzkoušeli, jak metoda tečen funguje můžeme napsat funkci, která najde kořen nějaké funkce.

Požadujeme

- musí hledat kořen libovolné funkce
- bod, kde začít, dostane také jako argument
- funkce musí poznat, že kořen nenašla

```
[135]: def rootNewton(ff1, x0, rtol=1e-8, atol=1e-12, maxIter=20):
        """
        Provede nanejvýš maxIter kroků Newtonovy metody.
        ff1 ... Funkce a její derivace (předávají se společně).
        x0 ... počáteční odhad kořene
        rtol, atol ... požadovaná relativní absolutní chyba kořene
        Výchozí hodnoty by obvykle měly dát 14 cifer pro kořen v okolí 1
        """
        n = 0
        x = x0
        while True:
            n = n+1
            assert n < maxIter, "Překročen maximální počet iterací při hledání kořene."

            f, f1 = ff1(x)
            dx = -f/f1
            x = x + dx
            delta = max(atol, abs(rtol*x))
            if abs(dx) < delta:
                return x

def f_f1(x):
    "Vrací f(x) a f'(x). f má kořeny -2,-1,1,2."
    x2 = x*x
    return (x2-4)*(x2-1), x*(4*x2-10)
```

(continues on next page)

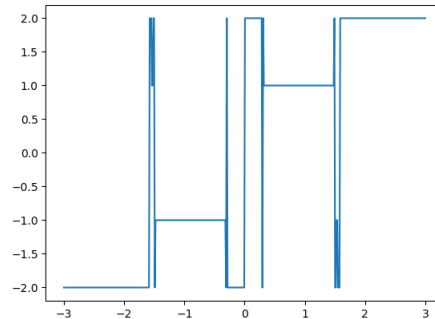
(pokračujte na předchozí stránce)

```
rootNewton(f_f1, 0.5)
```

[135]: 1.0

Zajímavost: Můžeme si namalovat, u kterého kořene skončí Newtonova metoda, když začneme s různými hodnotami počátečního odhadu polohy kořene. Vyplývá z toho, že je dobrá znát přibližně polohu kořene.

[40]: grafFunkce(rootNewton, -3, 3, arg1=f_f1, n=498)



9.9 Inverzní funkce

Nyní uvedené metody hledání kořenu použijeme. Jedním ze způsobů, jak definovat funkci je zadání rovnice, kterou řeší. Místo abstraktní formulace vezměme konkrétní příklad:

Hledáme hodnotu y takovou, že pro dané $z \in (-e^{-1}, \infty)$ řeší rovnici $z = ye^y$.

Máme dvě možnosti

- použijeme nějakou funkci pro hledání kořene definované výše
- napíšeme vše znovu (díky specializaci může být pak funkce rychlejší)

Z hlediska programování je zajímavější první varianta, protože musíme nějak zařídit, aby se hodnota z předala funkci, která bude počítat $f(x) = xe^x - z$ a její derivaci $f'(x) = (x + 1)e^x$. Zvolíme nejjednodušší variantu, použijeme globální proměnnou.

[147]: `import math`

```
param_z_W = 0.0
```

```
def f_f1_W(x):
    "Vrací f(x) a f'(x) pro f=x e^x-z"
    ex = math.exp(x)
    return x*ex - param_z_W, (x+1)*ex
```

```
def lambertW(x):
    "Lambertova funkce W(x). Řeší rovnici y e^y == x. Def. obor je x > -1/e."
    xe1 = x*math.e+1
    if xe1 == 0:
```

(continues on next page)


```
[155]: import math

for x in [1.0, -8.0, math.pi, 0.1]:
    print(f"{x.hex():>22} ({x})")

0x1.0000000000000000p+0 (1.0)
-0x1.0000000000000000p+3 (-8.0)
0x1.921fb54442d18p+1 (3.141592653589793)
0x1.9999999999999999ap-4 (0.1)
```

Například π je tedy nahrazeno aproximací

$$\pi \approx \frac{7074237752028440}{2251799813685248} = \frac{7074237752028440}{2^{51}}.$$

vzhledem k tomu, že čitatel může mít nejvýš 16 dekadických cifer, dá se říct, že reálná čísla a výpočty s nimi jsou k dispozici jen (necelými) 16 platnými ciframi. Protože mocniny desítky nejsou mocninami dvojky, racionální čísla $0.1 = 1/10$, $0.99 = 99/100$, atp. mající konečný dekadický zápis, mají binární zápis nekonečný (podobně jako třeba číslo $1/3$) a tedy nejsou v proměnné typu float uložena přesně.

Skutečnost, že místo reálných čísel máme jen velmi malou podmnožinu čísel racionálních znamená, že výsledek každé operaci je většinou znám jen v zaokrouhlené podobě. Obvykle platí, že výsledek takové operace je zatížen podobnou chybou jako veličiny do operace vstupující. Důležitou výjimkou jsou operace sčítání, odečítání (a složitější, například trigonometrické funkce sčítání implicitně obsahující):

```
1.0000000000123456
-1.0000000000000000
-----
0.0000000000123456 -> 1.23456E-11
```

Výsledek tedy najednou představuje číslo s pouhými šesti platnými ciframi. Přesně tento problém nastane při použití známého vzorečku pro kořen kvadratické rovnice. Stačí si prohlédnout tabulky integrálů nebo speciálních funkcí, aby člověk zjistil, jak často se rozdíl blízkých veličin počítá, např. $1 - \sqrt{1 - x^2}$ pro malá x . Existují situace, kdy ztráta přesnosti je extrémní:

```
[157]: from math import cos

for i in range(7):
    x = 2/10**i
    f1 = (1 - cos(3*x) - cos(4*x) + cos(5*x))/(1 - cos(x))**2
    f2 = 4*(3 + 5*cos(x) + 3*cos(2*x) + cos(3*x))
    print( f"{x=:6} {f1=:15.12g} {f2=:15.12f} {f1-f2:8.1e}" )

x= 2.0 f1=-0.325979034705 f2=-0.325979034705 1.2e-15
x= 0.2 f1= 45.9554059445 f2=45.955405944498 3.9e-13
x= 0.02 f1= 47.9792035724 f2=47.979203573004 -6.0e-10
x= 0.002 f1= 47.9997750673 f2=47.999792000357 -1.7e-05
x=0.0002 f1= 48.0171463987 f2=47.999997920000 1.7e-02
x= 2e-05 f1= 2775.55710226 f2=47.999999979200 2.7e+03
x= 2e-06 f1= 0 f2=47.999999999792 -4.8e+01
```

Povšimněte si, že zdrojem chyby ve výpočtu f_1 je, jak bylo zmíněno výše, odečítání dvou blízkých čísel. Výraz f_2 je ekvivalentní f_1 , ale k tomuto problému v něm v okolí $x = 0$ nedochází.

Je několik postupů jak obejít tento problém. Vynecháme-li ty, co vyžadují hluboké znalosti reprezentace reálných čísel v počítači, je potřeba nalézt alternativní formu daného výrazu. Někdy to jsou aplikace rozvojit, ovšem zmiňovaný výraz

$1 - \sqrt{1 - x^2}$ pro malé $|x|$ stačí upravit jako $x^2/(1 + \sqrt{1 - x^2})$ a podobně u oné kvadratické rovnice stačí úprava výrazu

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}},$$

kde při výpočtu volíme ten vztah, kdy oba sčítance mají stejná znaménka a tedy nedojde ke ztrátě platných cifer.

9.10.1 Kontrolní úlohy

1. Zaokrouhlovací chyby v triviálních vzorcích

- Pro $m = 1000$ kg a $v = 100$ km/h spočtete

$$E_{\text{kin}} = \frac{1}{2}mv^2$$

Zvažte možnost definovat konstanty $\text{km}=1000$ a $\text{hodina}=3600$ (a pokud chcete dodržet etiketu, použijte pro konstanty velká písmena).

- Totéž pro Einsteinův vzorec

$$E_{\text{kin}} = \frac{mc^2}{\sqrt{1 - \frac{v^2}{c^2}}} - mc^2$$

- Povšimněte si, že hodnoty nebudou moc souhlasit. Použijte trik, který byl doporučen pro odmocniny a ukažte, že relativistická kinetická energie dává pro automobily neměřitelně odlišnou kinetickou energii, jako teorie newtonovská.

9.11 Lokalizace chyb

Pokud při výpočtu dojde k chybě, jsme na to upozorněni. Pokud program spoštíme z příkazové řádky, je program ukončen.

Pokud pracujeme v interaktivním režimu a pokud dojde k chybě uvnitř nějaké funkce, je provádění kódu ukončeno na globální úrovni, tedy lokální proměnné volaných funkcí zmizí a nejsou dostupné. Globální proměnné zůstávají ve stavu v jakém se nacházely v okamžiku chyby.

Ilustrujme si to na následujícím příkladu. Definujeme funkci

$$f(x, y, n) = \sum_{k=1}^n \frac{x^k}{k - y}$$

a poté zkusíme spočítat $g(x, y) = f(x, -y, n)/f(x, y, n)$ pro $y = 1$ a dostatečně velké n .

```
[ ]: # demonstrace výpisu chyby

def f(x, y, n):
    """Demonstrace lokalizace chyb."""

    s = 0.0
    xk = x
    for k in range(1, n+1):
        s = s + xk/(k-y)
```

(continues on next page)

(pokračujte na předchozí stránce)

```

    xk *= x

    return s

def g(x,y):
    p = f(x,-y,50)
    q = f(x,+y,50)
    return p/q

print(g(0.5,1))

```

Chybová hlášení závisejí na způsobu, jímž kód výše spustíme. Pro naše potřeby je nejsrozumitelnější hlášení v prostředí Jupyter:

```

-----
ZeroDivisionError                                Traceback (most recent call last)
Cell In[1], line 18
     15     q = f(x,+y,50)
     16     return p/q
--> 18 print(g(0.5,1))

Cell In[1], line 15, in g(x, y)
     13 def g(x,y):
     14     p = f(x,-y,50)
--> 15     q = f(x,+y,50)
     16     return p/q

Cell In[1], line 7, in f(x, y, n)
      5 xk = x
      6 for k in range(1,n+1):
----> 7     s = s + xk/(k-y)
      8     xk *= x
     10 return s

ZeroDivisionError: float division by zero

```

Na prvním řádku se dozvíme, že došlo k dělení nulou a že funkce, které byly volány naposledy, jsou uvedeny na konci výpisu. Hlášení sděluje, že v rámci příkazu `print(g(0.5,1))` došlo chybě, protože při výpočtu `q = f(x,+y,50)` došlo ve funkci `f` k dělení nulou při vyhodnocení pravé strany přiřazovacího příkazu `s = s + xk/(k-y)`.

V našem případě je chyba zřejmá, v $y = 1$ není $g(x, y)$ definována. V situacích, kdy nám chyba z kódu zřejmá není, musíme chybu hledat za běhu programu. Dva klíčové postupy jsou

- Program doplníme o vhodné množství příkazů `print`, které na obrazovku nebo do soboru pošlou dostatek informací o průběhu výpočtu, abychom z nich chybu dokázali najít. Pokud do funkce `f` přidáme na začátek cyklu příkaz `print(f'f({x},{y},{n}): pricitam {xk}/{(k-y)}')` budou poslední dva řádky ladícího výstupu

```

f(0.5,-1,50): pricitam 8.881784197001252e-16/51
f(0.5,1,50): pricitam 0.5/0

```

- Použijeme nástroje pro ladění dostupné v IDE jako je VS Code nebo PyCharm, kde se výpočet pozastaví ještě uvnitř funkce, kde k chybě došlo a máme možnost zkoumat aktuální stav lokálních proměnných.

```

1 # demonstrace výpisu chyby
2
3 def f(x,y,n):
4     """Demonstrace lokalizace chyb."""
5
6     s = 0.0
7     xk = x
8     for k in range(1,n+1):
9         s = s + xk/(k-y)
10
11
12
13
14
15 def g(x,y):
16     p = f(x,-y,50)
17     q = f(x,+y,50)
18     return p/q
19
20 print(g(0.5,1))

```

Exception has occurred: ZeroDivisionError
float division by zero

File "/media/Disk1/vyuka/Prog2023/py/traceback_VScode.py", line 9, in f
s = s + xk/(k-y)

File "/media/Disk1/vyuka/Prog2023/py/traceback_VScode.py", line 17, in g
q = f(x,+y,50)

File "/media/Disk1/vyuka/Prog2023/py/traceback_VScode.py", line 20, in <module>
print(g(0.5,1))

ZeroDivisionError: float division by zero

CALL STACK

- f traceback_VScode.py
- g traceback_VScode.py
- <module> traceback_VScode.py

Obrázek: Snímek prostředí VS Code zachycující ohlášení dělení nulou. V levém sloupci dole je seznam funkcí zavolaných v okamžiku havárie, zde tedy, že `g` zavolala `f`. Pro funkci, kterou zde vybereme je nahoře dostupná informace o jejích lokálních proměnných. V prostřední části levého sloupce je možno přidat výraz, který nás zajímá. Zde vidíme, že pro aktuální hodnoty lokálních proměnných funkce `f` vede výraz $xk/(k-y)$ k dělení nulou.

To, že program rozdělíme na funkce přináší při hledání chyb výhodu. Pokud identifikuje problematickou funkci, můžeme napsat samostatný kód, který ji separátně testuje. Je zřejmé, že takové testování se zkomplikuje, pokud problematická funkce komunikuje s jinými prostřednictvím globálních proměnných.

[]:

Než si ukážeme klíčové příklady použití polí, seznámíme se s antickým algoritmem pro pořízení seznamu prvočísel:

10.1 Eratosthenovo síto

Eratosthenes z Kyrény nám radí vyškrtnat ze seznamu všech celých čísel nulu (Pythonu nás nutí brát nulu jako první celé číslo), jedničku a pak všechny dvou- a více- násobky původně nevyškrtnutých čísel. Co zbyde nevyškrtnuté je prvočíslo.

V následujícím kódu máme funkci, která vrací pole informací o tom, zda číslo i je prvočíslo. To má podobu tabulky

0	1	2	3	4	5	6	7	8	9	...
0	0	1	1	0	1	0	1	0	0	...

V tabulce je na horním řádku pořadové číslo v poli/seznamu – tzv. *index* a ve spodním řádku je pak hodnota určená indexem. V matematice to zapisujeme a_i . Pověšměte si, že index je celé číslo a hodnoty jsou stejného typu. Takovouto organizaci dat v programu nazýváme pole.

Představujeme si, že tahle tabulka nějak bydlí v paměti počítače (podrobněji dále) a když zadáme index, počítač se umí podívat, jaká hodnota indexu přísluší. Tato operace se nejen v jazyce Python zapisuje `promenna[i]`. Index i je podle této představy celé číslo, přičemž Python spadá to té podmnožiny počítačových jazyků, kde se prvním prvkem pole/seznamu rozumí ten s indexem 0, tedy `promenna[0]`.

Následující kód definuje a volá funkci `eratosthenes(n)`. Jak víte malé `e` píšeme nikoli z neúcty, ale proto, že úzus použitého jazyka rezervuje identifikátory počínající velkým písmenem pro třídy (a konstanty).

```
[ ]: import numpy as np

def eratosthenes(n):
    """Vrací pole [False,False,True,True,False,True,...] délky n+1
       indikující, že mezi čísly 0..n jsou 2,3,5,... prvočísla"""
```

(continues on next page)

```

je_prvocislo = np.full(n+1, True)
je_prvocislo[0] = False
je_prvocislo[1] = False

p = 2
while p*p <= n:
    if je_prvocislo[p]:
        for nasobek_p in range(2*p, n+1, p):
            je_prvocislo[nasobek_p] = False
        p = p + 1

    return je_prvocislo

n_max = 35
vysledek = eratosthenes(n_max)
print( vysledek )
print( "\nNalezená prvočísla:" )
for i in range(n_max+1):
    if vysledek[i]:
        print(i, end=' ')

[False False True True False True False True False False False True
 False True False False False True False True False False False True
 False False False False False True False True False False False]

Nalezená prvočísla:
2 3 5 7 11 13 17 19 23 29 31

```

Sám algoritmus je v kódu snadno čitelný. Má za úkol demonstrovat, jakým způsobem budeme v jazyce Python pracovat s poli. Není to způsob jediný, ale poskytne nám nejméně náročnou a ve světě programování pro fyziky běžnou představu o práci s poli.

Již první řádek říká, že budeme používat knihovnu `numpy`. Protože jsme zvyklí, že i pro funkci sinus musíme do nějaké knihovny, není to úplně překvapivé. Je ale třeba hned přiznat, že Python má jiný základní typ pro pole a seznamy, který nevyžaduje další knihovny. Jen se nám tolik nehodí vzhledem k omezenému rozsahu naší přednášky.

Příkaz

```
import numpy as np
```

říká, že chceme používat knihovnu `numpy`, ale místo (např.) `numpy.sin` chceme psát kratší `np.sin`.

Program výše je rozdělen na funkci a kus programu, který funkci volá a vypisuje výsledky. Takto definovaná funkce nám dovolí ji použít dále ve výkladu.

Protože celých čísel je nekonečně, parametr `n` funkce určuje, u jakého největšího čísla budeme prověřovat prvočíselnost. Uvnitř funkce je nejdůležitější řádek

```
je_prvocislo = np.full(n+1, True)
```

Dává vzniknout lokální proměnné v podobě pole $n + 1$ hodnot `True`. S touto proměnnou pak funkce celou dobu pracuje a tu i vrátí jako výsledek.

Jak se na proměnnou sluší, můžeme ji modifikovat, proto v dalších dvou řádcích nastavíme, že čísla 0 a 1 nejsou prvočísla. Např. `je_prvocislo[0] = False` změní první položku pole, která byla původně inicializována na `True`.

Následuje Eratosthenovo vyškrtávání: - pro všechna čísla p , - pokud nejsou moc velká (pak cyklus ukončíme), - a pokud je p prvočíslo, - vyškrtáme všechny jeho násobky, - které představují platné indexy pole `je_prvocislo`, proto `range(2*p, n+1, p)`.

- Vyškrtané pole, obsahující `True` jen na místě prvočísel pak funkce vrátí.

Vraťme se ještě jednou k volání funkce `np.full(n+1, True)`. Tušíme, že číslo n nemusí být malé a tedy samotné vytváření pole není samozřejmou operací. Někde v paměti musí být potřebné místo. Zde se projeví klíčová vlastnost knihovny `numpy`. Ta poskytuje pole tvořená určeným počtem prvků téhož typu. Typ prvků nelze měnit, musíme jej znát dopředu. Počet prvků je také lepší neměnit. Z hlediska spotřeby paměti je velikost pole dána součinem počtu prvků a velikosti jednoho prvku. (Pro pokročilejší: odhad počtu bytů potřebných pro uložení proměnné `x` typu `numpy.ndarray` spočteme jako součin atributů `x.size * x.itemsize`.)

Výsledek, který funkce `eratosthenes` vrátí, kód výše použije k vypísání prvočísel $p \leq n_{\max}$. Kód obsahuje podmíněný příkaz `if vysledek[i]`: který testuje, zda prvek pole s indexem i zastupuje prvočíslo a je tedy vhodné jej vytisknout. (Často lze potkat i ekvivalentní zápis `if vysledek[i]==True`.)

Pole a seznamy jsou velmi podobné struktury. Abychom si to přiblížili, demonstruje následující funkce pořízení seznamu prvočísel. I když jde o tutéž informaci, jakou vrací funkce `eratosthenes`, její podoba je dost odlišná. Zapsána jako tabulka vypadá takto

0	1	2	3	4	5	6	7	8	9	...
2	3	5	7	11	13	17	19	23	29	...

```
[ ]: def seznam_prvocisel(n):
    """Vrací seznam prvočísel <= n typu numpy.ndarray"""
    e = eratosthenes(n)

    count = 0
    for i in range(n+1):
        if e[i]:
            count += 1

    seznam = np.full(count, 0)
    j = 0
    for i in range(n+1):
        if e[i]:
            seznam[j]=i
            j += 1

    return seznam

seznam25 = seznam_prvocisel(25)
print(seznam25)

n_max = 1_000_000
print('Počet prvočísel <=', n_max, 'je', len(seznam_prvocisel(n_max)))

[ 2  3  5  7 11 13 17 19 23]
Počet prvočísel <= 1000000 je 78498
```

Tento kód demonstruje možný postup konverze pole logických hodnot `je/není prvočíslo` na seznam prvočísel. Protože u polí, jak se je učíme, musíme před jejich vytvoření vědět, jak jsou velká, funkce `seznam_prvocisel` hned poté, co zavolá funkci `eratosthenes`, spočte, kolik prvočísel ve vráceném poli je. Učiní tak v rámci cyklu v indexu i , kdy projde všechny prvky pole `e[i]` a kdykoli narazí na `True`, zvětší proměnnou `count` o jednu.

Z pedagogických důvodů tedy funkce `seznam_prvocisel` opět používá jen dvě důležité operace - vytvoření pole funkcí `np.full` - přístup k prvku pole `e[i]`, `seznam[j]`

S nimu zkusíme vystačit, jak jen to půjde.

Sadno bychom to mohli učinit jediným příkazem, např. `count = np.count_nonzero(e)`, protože knihovna `numpy` obsahuje více jak 500 různou měrou užitečných funkcí a mezi nimi je i funkce `numpy.count_nonzero`. Poměrně často lze za pomoci dokumentace a/nebo editoru IDE funkci provádějící žádanou operaci pohodlně nalézt a použít správné argumenty. My se ale používat `numpy` učíme, takže strpíme, že mnohé z funkcí, které v našem kurzu píšeme, jsou již napsané a dostupné v některé z knihoven. V tomto případě bychom vlastně ani znalost funkcí knihovny `numpy` nepotřebovali, protože počet prvočísel v poli `e` lze zapsat i jako `count = sum(e)`.

Pro čtenáře znalého typu `list` doplňuji dvě další podoby funkce `seznam_prvocisel`:

```
[ ]: def prime_list1(n):
    """Vrací seznam prvočísel <= n typu list"""
    e = eratosthenes(n)

    seznam = []
    for i in range(n+1):
        if e[i]:
            seznam.append(i)

    return seznam

def prime_list2(n):
    """Vrací seznam prvočísel <= n typu list"""
    e = eratosthenes(n)

    return [i for i in range(n+1) if e[i]]

assert len(prime_list1(1000000)) == len(prime_list2(1000000)) == 78498
```

Ačkoli `list` je základním typem jazyka Python, námi používané pole budou dána datovým typem `numpy.ndarray` (z angl. *n-dimensional array*). Ten je spolu se desítkami/stovkami užitečných funkcí k dispozici v knihovně `numpy`; v rámci kurzu budeme samozřejmě používat jen naprosté minimum z nich.

- Svými vlastnostmi se pole z knihovny `numpy` více podobají polím, jaká se používají v ostatních jazycích (C, Fortran, ...).
- Jejich použití pro výpočty ve fyzice je mnohem běžnější.

Dále tedy budeme používat slovo pole jako synonymum pro `numpy.ndarray`.

10.1.1 Demonstrace běhu algoritmu s poli

Doporučuji si projít následující kód v nástroji pythontutor.com. Kvůli velikosti textu obsahuje 0/1 místo `False/True` a také seznamy `list` protože tento nástroj `numpy` "neumí".

```
def eratosthenes(n):
    """Vrací pole [False,False,True,True,False,True,...] délky n+1
       indikující, že mezi čísly 0..n jsou 2,3,5,... prvočísla"""
    je_prvocislo = [0, 0] + [1]*(n-1)

    p = 2
    while p*p <= n:
```

(continues on next page)

(pokračujte na předchozí stránce)

```

    if je_prvocislo[p]:
        for nasobek_p in range(2*p,n+1,p):
            je_prvocislo[nasobek_p] = 0
    p = p + 1

    return je_prvocislo

n_max = 35
vysledek = eratosthenes(n_max)
print( vysledek )
print( "\nNalezená prvočísla:")
for i in range(n_max+1):
    if vysledek[i]:
        print(i, end=' ')

```

10.2 Pole – celek a část. Přístup k prvku pole (a[i])

Pole představuje celek složený z více stejných prvků. Podle okolností budeme chtít: - Pracovat s celkem (např. `print(seznam25)`). - Pracovat s jediným prvkem (např. `seznam[j]=i`) - Pracovat s několika prvky najednou (viz řezy polí dále)

Přístup k prvku pole, tedy možnost pracovat s jediným prvkem pole je důležitý krok v principu programovacího jazyka. Je-li prvek např. reálné číslo a jazyk poskytuje nástroje pro práci s takovými čísly (např. sčítání), dokážeme opakováním takových operací u každého prvku např. sčítat vektory. Prvek pole můžeme nastavovat i číst.

Nastavení prvku pole jsme viděli u Eratosthenova algoritmu, kde bylo potřeba jej nastavit tak, aby signalizoval, že dané číslo není prvočíslo. Takováto operace změny prvky má většinou podobu přiřazení, tedy `a[i] = nova_hodnota`.

Obrácená operace, tedy čtení hodnoty prvku se zapisuje stejně, ale tato kombinace indentifikátoru pole a indexu v hranatých závorkách se objevuje v nějakém výrazu. Může jít např. o aritmetický výraz na pravé straně přiřazovacího příkazu nebo o logický výraz, který se objevuje v příkazech `if`, `while` atd.

Připomeňme, že v hranatých závorkách se může vyskytovat libovolný celočíselný výraz, jehož hodnota padne do mezí stanovených počtem prvků pole. Například výraz `a[i+1]*a[i-1]==a[i]**2` může být součástí testu, zde se posloupnost chová jako geometrická řada. Při takovém testu ale nesmíme zapomenout, že hodnoty indexu `i` musejí splňovat `i>0` a `i+1<len(a)`.

10.3 První položka pole má index 0

Jazyky jako je Python, C, Java, ... nám vnucují, že první položka pole má index 0. Tedy v kódu funkce `eratosthenes` příkaz `je_prvocislo[0] = False` nastavuje první prvek pole.

Je snadné si na to zvyknout, jenže bohužel pravděpodobně budete zároveň pracovat v jazycích (Fortran, Mathematica, Matlab), kde je počáteční index jednička. I na toto neustálé střídání konvencí si nakonec zvyknete (budete muset).

10.4 Bereme na vědomí typy list a tuple

Již víme, že pole budou užitečná. Jsou natolik užitečná, že se v Pythonu vyskytují dva základní typy polím velmi podobné. Protože nemůžeme stihnout jejich vlastnosti ani povrchně probrat, ale zároveň jsou v jazyce Python příliš všudypřítomné, shrneme si aspoň nejzákladnější informace.

Společnou vlastností `ndarray`, `list`, `tuple` je možnost vybrat prvek pomocí hranatých závorek, např. `a[i+1]`.

- Typ `list` vytvoříme, když napíšeme seznam libovolného (i nulového) počtu hodnot do hranatých závorek

```
a = [1,2,3]
```

- Typ `tuple` vytvoříme, když napíšeme seznam hodnot do kulatých závorek

```
a = (1,2,3)
```

Protože jedna hodnota v závorce má význam uzávkovaného výrazu, je nutné psát před zavírací závorkou čárku `a1=(1,)`. Kdybychom to potřebovali lze prázdný tuple psát `a0=()`.

Narozdíl od typu `list` nelze do jednotlivých prvků tuple dosadit nové hodnoty. Právě kvůli této odlišnosti byl tento typ zaveden, v jazyce Python to hraje důležitou roli.

Typ tuple jsme již potkali, protože funkce, které vracejí více hodnot, vracejí ve skutečnosti právě typ tuple. Ačkoli jsme si navykli na pohodlnější způsob psaní `return x,y`, ten je ekvivalentní explicitnímu uvedení kulatých závorek `return (x,y)`.

Kontrolní otázka: U *builtin* funkce `divmod(a,b)` si můžeme představovat, že končí příkazem `return a//b, a%b`. Jaký výsledek očekáváte u příkazu `print(type(divmod(7,4)))`?

10.5 Unpacking

Byli jsme zvyklí u funkcí, které vracejí více hodnot použít vícenásobné přiřazení

```
podil, zbytek = divmod(p,q)
```

Jde o speciální případ procesu zvaného *unpacking*, který má např. pro tři hodnoty podobu

```
prom1, prom2, prom3 = seznamu_podobny_vyraz
```

Na pravé straně může být cokoli, co má tři části: `ndarray`, `list`, `tuple`, `range`, `...`. Pokud počet hodnot na levé a pravé straně nesouhlasí, dojde k chybě.

10.6 Pole nebo seznam?

Ještě jednou se vrátíme k odlišnostem mezi polem a seznamem. Ten není úplně zřetelný, jak demonstruje příklad výše, kde v obou případech používáme tentýž typ `numpy.ndarray`.

10.6.1 Pole

V problémech souvisejících s fyzikou budeme spíše potkávat *pole*. Polem obvykle rozumíme sadu prvků stejného typu naskládaných do jednoho celku, přičemž celkový počet prvků tvořících pole je obvykle předem známý. Např. u funkce `eratosthenes` výše určujeme nejprve délku pole, které funkce vrátí. Během výpočtu pole naplníme informací o tom, zda dané celé číslo je prvočíslem.

Se kterým z prvků chceme pracovat určujeme indexem, takže místo identifikátoru proměnné, např. `x`, který jsme používali ve výrazech nebo na pravé straně přiřazovacího příkazu píšeme například `a[i+1]`, kde `a` je identifikátor proměnné typu pole a `i+1` je příklad celočíselného výrazu, který dává hodnotu indexu určujícího, se kterým prvkem pole míníme pracovat. Celý příkaz může mít například podobu

```
soucet = soucet + a[i]*b[i]
```

Pokud konkrétně tuto operaci provedeme pro všechny prvky stejně dlouhých polí `a` a `b` (a pokud je sčítací proměnná na začátku vynulovaná) spočteme takto skalární součin dvou vektorů

$$\vec{a} \cdot \vec{b} = \sum_i a_i b_i$$

Pole se dvěma indexy kromě matic může také představovat obrázek, tzv. bitmapu. Obrázky mohou mít různé šifky a výšky (počítáno pixely), ale cítíme, že obrázek není seznam pixelů.

10.6.2 Seznam

Další funkce `seznam_prvocisel(n)` naopak obsahovala čísla, nikoli logické hodnoty. Tato čísla (v souladu s názvem funkce) představovala (seřazený) seznam prvočísel $\leq n$. To mělo ilustrovat odlišnost mezi polem a seznamem.

Tedy, seznamem můžeme mít jednak obecně na mysli data, kde informace, v nich obsažená, zahrnuje také délku seznamu a dává dobrý smysl do seznamu několik položek přidat a délku tak změnit. A i po takové změně má stále smysl se např. ptát kolik položek je dvou seznamech stejných. Oproti tomu pole obecně jsou spíše podobná vektorům, kde nemůžeme změnit dimenzi \vec{x} ze 3 na 5 a \vec{y} ze 3 na 7 a tvářit se, že jsme jen něco přidali. Zkuste nyní spočítat $\vec{x} \cdot \vec{y}$!

Seznamem však také můžeme konkrétně mít typ `list`, jak jsme zmínili, jde o jeden ze základních typů, jaké Python nabízí. Ten je většinou vhodnější pro pořizování seznamů (v obecném smyslu výše) než pole knihovny `numpy`, na která se v kurzu soustředíme. Typ `list` budeme v kurzu používat co nejméně. Sice tím přijdeme o zajímavou část jazyka Python, ale mnohé odlišnosti `list` od `numpy` polí jsou dost matoucí a v rámci výuky musíme dát přednost srozumitelnosti.

Proto, i když to přinese delší kód, budeme i seznamy ubytovávat v polích z knihovny `numpy`.

10.7 Vytvoření pole (`numpy.full` a další)

Připomeňme si, že proměnné v Pythonu odkazují na data, jaká jim přiřadíme a podobně je to i s argumenty funkcí. Pole, se kterými chceme pracovat se tedy nejprve někde musejí vzít, abychom je mohli svázat se jménem nějaké proměnné a s její pomocí je dále zpracovávat nebo obrábět.

Mezi spoustou možností, jak vytvořit pole zmiňme jako nejdůležitější tuto:

```
my_array = numpy.full( count, fill_value )
```

Pokud například bude `count==10` a `fill_value==2`, bude `my_array` odkazovat na takovouto tabulku čísel:

0	1	2	3	4	5	6	7	8	9
2	2	2	2	2	2	2	2	2	2

Snadno se o tom přesvědčíme příkazem `print(my_array)`.

Důležitá vlastnost polí z `numpy` je to, že typ hodnot v nich uložených je určen v okamžiku vzniku pole. Nyní jsme zvolili celé číslo a tak v dalších operacích můžeme měnit hodnoty prvků, ale nemůžeme měnit jejich typ. Navíc se o této vlastnosti dozvíme jen z toho, že výsledek operace nebude totožný s naším přáním. Například `my_array[3] = math.pi` bude provedeno jako `my_array[3] = int(math.pi)`, tedy ve čtvrtém prvku pole se objeví celé číslo 3.

Shrnuto, s použitím `numpy` se kód

```
my_array = np.full( 4, 2 )
```

chová podobně jako např. v C

```
int my_array[4] = {2,2,2,2};
```

tedy určuje typ prvků pole a jejich počet. (Kdybychom v kurzu probírali podrobněji typ `list`, viděli bychom, že zde je možno mít v poli prvky různých typů a dokonce jejich typ měnit za běhu programu.)

Cvičení: Vytvořte pole `pole_3x1` o délce 3 a naplněné hodnotou `1.0`. Co bude výstupem příkazu `print(pole_3x1)`? Jak se liší od situace, kdybychom měli pole tří celých čísel 1?

10.7.1 Vytvoření pole - další možnosti

V našem kurzu bychom vystačili s použitím jediného způsobu vzniku pole a to `numpy.full(count, fill_value)`.

Protože knihovna `numpy` byla motivována snadným psaním kódu, obsahuje mnoho dalších způsobů, jak můžeme vytvořit pole. I my tak potkáme ještě několik dalších možností. Zkusíme v našem kursu vystačit se dvěma cestami, jak vytvořit pole s jedním indexem.

- Vytvoříme pole o dané délce vyplněné předepsanými hodnotami
 - Již víme, že `numpy.full(count, fill_value)` vytvoří pole naplněné prvky téže hodnoty a tedy i téhož typu. Hodnoty pole pak budeme podle potřeby modifikovat, typ hodnot v průběhu doby existence pole změnit nejde.
 - Zkratka `numpy.zeros(n)` je pohodlnější a rychlejší ekvivalent `numpy.full(n, 0.0)`. Tuto variantu vytvoření pole daných rozměrů budeme užívat nejčastěji.
 - Funkce `numpy.linspace(a, b, num)` vrátí pole čísel rovnoměrně pokrývající interval $\langle a, b \rangle$. Viz též kód funkce `my_linspace` níže.
- Vytvoříme pole o hodnotách a délce daných jinými daty, obvykle provedením *konverze typu*. (Pozor: z historických důvodů nemá tato konverze tvar `nazev_typu(puvodni_data)`, kde název typu by měl být `numpy.ndarray`.)
 - Velmi užitečné je určit hodnoty výčtem, např. `numpy.array([x0, x1, x2, x3, ...])` vytvoří pole o tolika prvcích, kolik jich je v hranatých závorkách uvedeno a o hodnotách, jaké v závorce uvedeme. Jde o konverzi hodnoty `[x0, x1, x2, x3, ...]`, která, jak víme, je typu `list`.
Pokud bychom odhad společného typu nahradili pro jednoduchost operací `sum` (která dává smysl jen pro číselné typy), mohli bychom takovou konverzi `numpy.array(lst)` připodobnit funkcí založenou na `np.full` Python def `list2array(lst):` `"""konverze 1-d seznamu na pole numpy."""`
`n = len(lst) val = sum(lst) array = np.full(n, val) for i in range(n): arr[i] = lst[i] return arr`
 - Již jsme viděli, že `range` představuje návod na procházení intervalu celočíselné aritmetické řady. I na něj lze aplikovat konverzi, tedy např. psát `numpy.array(range(1, 7))`. Tím získáme totéž, jako zápisem `numpy.array([1, 2, 3, 4, 5, 6])`.

- Do této kategorie spadá i důležité vytvoření duplikátu pole, tedy `y = x.copy()`. Brzy uvidíme, proč potřebujeme takové duplikáty. Samozřejmě, i zde bychom vystačili s aplikací `np.full`, např. `Python def array_copy(old_array): """Vytvoření kopie 1-d pole numpy.""" n = len(old_array) new_array = np.full(n, arr[0]) for i in range(n): new_array[i] = old_array[i] return new_array`

Obě cesty obsahují nástrahu: `numpy.full(4, 1)` vytvoří stejné pole jako `numpy.array([1, 1, 1, 1])`. Jde o čtveřici celých nikoli reálných čísel. Pokud tedy jako hodnotu prvků v `numpy.full` uvedeme celé číslo nebo ve výčtu `numpy.array([. . .])` uvedeme samá celá čísla, nemusí se chování pole shodovat s vašimi představami obvykle založenými na vlastnostech reálných čísel. To proto, že přiřazením do prvku pole nemůžeme změnit společný typ všech hodnot pole. Operací s celým pole to ale samozřejmě půjde. Připomeňme, že jistou míru bezpečí před tímto druhem chyby představuje funkce `numpy.zeros(n)` ekvivalentní `numpy.full(n, 0.0)`.

10.7.2 Neplatné hodnoty indexů polí (IndexError)

Především, pokud mluvíme o polích poskytovaných knihovnou `numpy`, jejich indexy by měly být celá čísla (případně další od nich odvozené složitější konstrukce). Pokud tedy omylem zatoužíme po prvním prvku pole a místo indexu `0` použijeme reálné číslo `0.0`, dočkáme se chybového hlášení, které nám to připomene (text chybového hlášení je zkrácen)

```
IndexError: only integers, slices (`:`), ... and integer or boolean arrays are valid
↪ indices
```

Jiná chyba nastane, pokud použijeme hodnotu celého čísla, která je za hranicemi pole. V tom případě je nám jeho délka v chybovém hlášení připomenuta

```
IndexError: index 20 is out of bounds for axis 0 with size 20
```

U matic hlášení sdělí, zda jsme špatně zadali index řádku (axis `0`) nebo index sloupce (axis `1`).

Otázka: Jak to, že když má pole podle chybového hlášení výše velikost 20, vede index o hodnotě 20 k chybě?

10.7.3 Záporné indexy

Již víme, že `s[0]` je první prvek pole/seznamu. Pokud bychom napsali `s[-1]` znamenalo by to v jazycích, jako je např. C, pokus zjistit hodnotu nacházející se v paměti počítače těsně před prvním prvkem pole. Obvykle je to nezamyšlená, chybná operace a podle situace vede okamžitě nebo po chvíli k ukončení běhu programu (lepší situace) nebo ke špatnému výsledku výpočtu.

V Pythonu jsou záporné indexy použity ve významu indexování od konce pole/seznamu, takže použití `s[-1]` znamená poslední prvek pole, `s[-2]` je ten předposlední atp.

10.8 Přiřazovací příkaz s poli

U polí se silně projeví důležitá vlastnost přiřazovacího příkazu a povaha proměnných v jazyce Python. Jde o to, že * Proměnné Pythonu představují štítky s odkazy na aktuální data * Přiřazením do proměnné měníme odkaz, ne hodnotu dat! * Na tatáž data tak může odkazovat více proměnných * U polí lze změnit nejen na co proměnná odkazuje, ale také hodnoty jednotlivých prvků, např. `a[1]=4`. Tím se ale mění hodnota druhého prvku pole odkazovaného proměnnou `a`, ale i všech dalších proměnných, které na toto pole odkazují. Jde o zamýšlenou vlastnost jazyka Python, který dokonce zavádí kategorii *mutable* (pozměnitelných) typů. Doposud nám nevadilo, že hodnota na hodnotu 4 odkazují všechny proměnné, které přiřazením získaly tuto hodnotu. To proto, že onu čtyřku není možno nijak pozměnit, provedením operace, např. `a=a+1` začne proměnná `a` odkazovat na pětku aniž se na čtyřce něco změní. Celá čísla (typu `int`) jsou **immutable**.

Uvažte kód

```
import numpy as np
a = np.full(3,1.0)
b = a
a[1] = 4
b[2] = 9

print(a, b)
```

Rozmyslete si nejprve, co očekáváte za výstup. Poté kód spusťte a zjistěte, zda splňuje vaše očekávání. Pokud ne, vraťte se k záhlaví "Přiřazovací příkaz s poli".

Rada:

Místo přiřazení

```
nove_pole = stare_pole
```

automaticky používejte tvar

```
nove_pole = stare_pole.copy()
```

Málokdy v našem kurzu totiž potřebujeme dvě proměnné odkazující pod různými jmény na totéž pole. (Tento postup, nefunguje pro seznamy seznamů, např. `[[1,2],[3,4]]`, nicméně typ `list` je v našem kurzu v nemilosti, takže to můžeme opomenout a používat metodu `copy()` k získání duplikátu vektorů i matic.)

Na toto chování polí při přiřazení musíme pamatovat také, když předáváme funkci argument typu pole a tento uvnitř funkce měníme. Předání argumentu lze chápat jako přiřazení a uvnitř funkce tak původní pole vystupuje pod novým jménem, ale jde o tatáž data, a většinou proměnnou, která představovala při volání funkce její argument, měnit nechceme.

10.8.1 Další elementární operace s poli

POčet prvků (délku) pole zjistíme (mimo jiné) výrazem `len(s)`. Proto, kdybychom se to nenaučili za chvíli jednodušeji, `s[len(s)-1]` je poslední prvek seznamu.

Ačkoli je neefektivní připojovat k seznamu uloženému v `numpy.ndarray` další hodnoty, pokud to potřebujeme máme k dispozici operaci `promenna_typu_pole.resize(nova_velikost)`.

10.9 Typy prvků polí v numpy

Popularita této knihovny je také dána tím, že její správné použití přinese podstatné urychlení oproti zabudovanému typu `list`. Toho je dosaženo dvěma kroky - Všechny prvky pole jsou stejného typu - S tímto typem umí procesor efektivně pracovat

První vlastnost má důležité důsledky při vytváření polí z jiných hodnot. Pokud použijeme konverzi ze seznamu hodnot různého typu, společný typ se odvodí z nejširšího typu z následující řady `int<float<complex<string<...`, jaký se nalezne mezi konvertovanými hodnotami. Proto `numpy.array([1,2,3])` vytvoří pole celých čísel, zatímco `numpy.array([1,2,3.0])` vytvoří pole čísel reálných.

10.9.1 Celá čísla v numpy

S celými čísly je v programování potíž, protože jejich zápis sice vyžaduje konečný počet cifer, ale ne nezbytně počet malý. Python zvládá tuto potíž skrývat, i pokud je výsledek operace velmi dlouhé číslo. I tak se tato potíž může projevit, viz např. chybové hlášení pro příkaz `print(math.factorial(2000))`.

Aby mohl počítač s celými čísly efektivně pracovat musejí být interně uložena v dvojkovém tvaru. (Rozmyslete a vyzkoušejte si, že číslo dvacet má dvojkový zápis 10100_2 , např. příkazem `print(0b10100)`.) Počet bitů celého čísla se kterým procesor pracuje naráz navíc souvisí s počtem vodičů, jimiž se dopravují data uvnitř výpočetní jednotky procesorů. Shrnuto, dnešní počítače pracují efektivně s celými čísly o délce 8, 16, 32 a 64 bitů. To představuje následující intervaly, v kterých celé číslo o takové bitové šířce může ležet:

```
[ ]: for n in [8,16,32,64]:
      print(f'{n:2}-bitové celé číslo bez znaménka: 0..{2**(n-1):<30}')
      print(f'                               se znaménkem: {-2**(n-1)}..{2**(n-1)-1}')
```

8-bitové celé číslo bez znaménka: 0..255
 se znaménkem: -128..127

16-bitové celé číslo bez znaménka: 0..65535
 se znaménkem: -32768..32767

32-bitové celé číslo bez znaménka: 0..4294967295
 se znaménkem: -2147483648..2147483647

64-bitové celé číslo bez znaménka: 0..18446744073709551615
 se znaménkem: -9223372036854775808..9223372036854775807

Musíme nyní diskutovat, jak psát funkce, které pracují s poli. Máme tu několik rolí, ve kterých tato pole mohou vystupovat:

- Vstupní argument, jehož hodnotu uvnitř funkce neměníme.
- Vstupní argument, jehož hodnotu uvnitř funkce měníme, ale nechceme aby to bylo "vidět venku".
- Vstupně-výstupní argument, který měníme a chceme, aby to bylo vidět venku
- Výstupní hodnota (např. funkce `geom_posloupnost(a0, q, n)` může vrátit n členů g. posloupnosti).
- Globální proměnná může vystupovat (někdy i zároveň) v obou výše uvedených rolích.

11.1 Argumenty typu pole – vstupní argument (jen pro čtení)

Začneme funkcí, která spočte součet kladných prvků seznamu/pole. Níže uvedený kód řeší následující otázky:

- jaký je první a poslední index prvku pole
- zda je prvek kladný
- nulování, přičítání a navrácení součtu

```
[ ]: def sum_positive(a):  
    """Součet kladných prvků seznamu"""  
    soucet = 0  
    for i in range(len(a)):  
        if a[i]>0:  
            soucet = soucet + a[i]  
  
    return soucet  
  
pole = np.array([-2,-1,0,1,2])  
print( 'sum =',sum_positive( pole ) )
```

```
sum = 3
```

Tento kód se záměrně podobá tomu, jak bychom je psali např. v C, tedy zejména používá indexy. Modernější přístup využije toho, že for umí procházet přímo prvky pole:

```
[ ]: def sum_positive_v2(a):
    """Součet kladných prvků seznamu"""
    soucet = 0
    for x in a:
        if x>0:
            soucet = soucet + x

    return soucet

pole = np.array([-2,-1,0,1,2])
print( 'sum =',sum_positive_v2( pole ) )
```

```
sum = 3
```

Vidíme, že obě verze nemění hodnotu argumentu a.

Ne všechny funkce, které mají za vstup pole budeme muset psát. Existuje např. funkce sum, která počítá součet prvků pole. Využijeme toho k dalšímu výkladu.

Uvažujme následující variantu funkce sčítající pozitivní členy pole:

```
[ ]: def sum_positive_v3(a):
    """Součet kladných prvků seznamu"""
    for i in range(len(a)):
        if a[i]<0:
            a[i] = 0

    return sum(a)

pole = np.array([-2,-1,0,1,2])
print( 'sum =',sum_positive_v3( pole ) )
print( 'pole =', pole )
```

```
sum = 3
pole = [0 0 0 1 2]
```

Zde se projeví vedlejší účinky volání takové funkce: proměnná, kterou dostala jako argument byla pozměněna.

Cvičení: Opravte funkci sum_positive_v3 tak, aby neměnila hodnotu proměnné, která je jí předána jako argument. (Využijte např. metodu copy().)

11.2 Volání funkcí s argumentem typu pole

Volání funkce `sum_positive` výše se podobá např. volání funkce `abs`, u které můžeme např. psát `abs(x)`, `abs(-5)`, `abs(x-y)`. Můžeme tedy - Předat funkci jako argument proměnnou `sum_positive(x)` - Předat funkci jako argument výraz - něco jako konstantu `sum_positive(np.array([-1,1,2]))` - nebo výraz `sum_positive(-x)` (uvidíme, že $-\{a_k\} = \{-a_k\}$) - ještě složitější výraz ...

Dobrym důvodem pro předávání polí funkci je např., pokud z nich tato funkce dokáže namalovat obrázek.

11.3 Malování obrázků (knihovna `matplotlib`)

Jednou z velmi pohodlných vlastností Pythonu je, že máme k dispozici i knihovnu pro malování jednoduchých obrázků. Zmíníme ji na tomto místě, protože u klíčových funkcí knihovny hrají pole roli vstupních argumentů, které určují např. souřadnice malovaných bodů a čar.

Grafy funkcí kreslíme za pomoci funkce `plot` z knihovny `matplotlib.pyplot`. Z rozsáhlého seznamu možností zvolíme jen nejjednodušší minimum, zbytek naleznete v [dokumentaci](#)

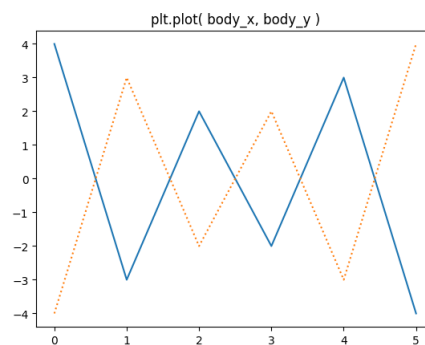
Pozn. Až na řádek, kde otáčíme najednou znaménko všech prvků pole zápisem `-body_y`, pokud víte jak, můžete v kódu použít pro data jednodušší typ `list`.

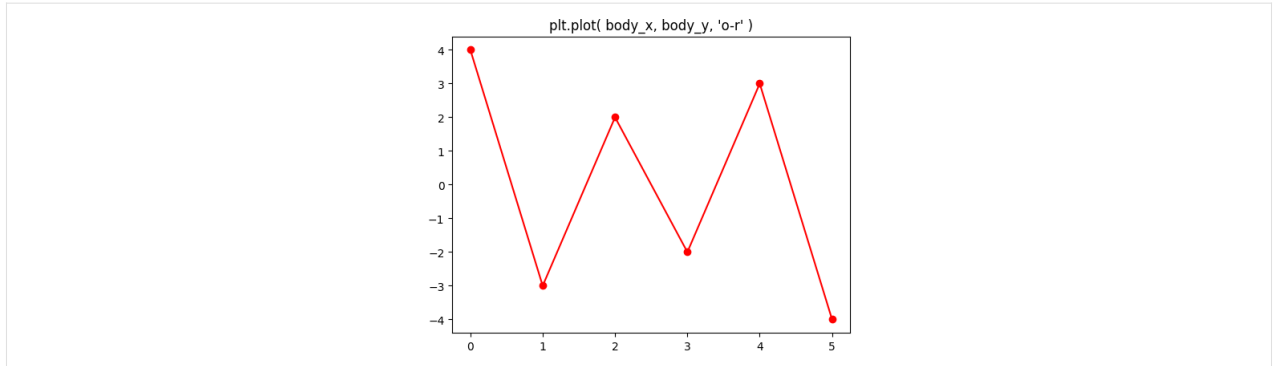
```
[ ]: import matplotlib.pyplot as plt

body_x = np.array( [0,1,2,3,4,5] )
body_y = np.array( [4,-3,2,-2,3,-4] )

plt.plot( body_x, body_y )
plt.plot( body_x, -body_y, ':' )
plt.title("plt.plot( body_x, body_y )")
plt.show()

plt.plot( body_x, body_y, 'o-r' )
plt.title("plt.plot( body_x, body_y, 'o-r' )")
plt.show()
```





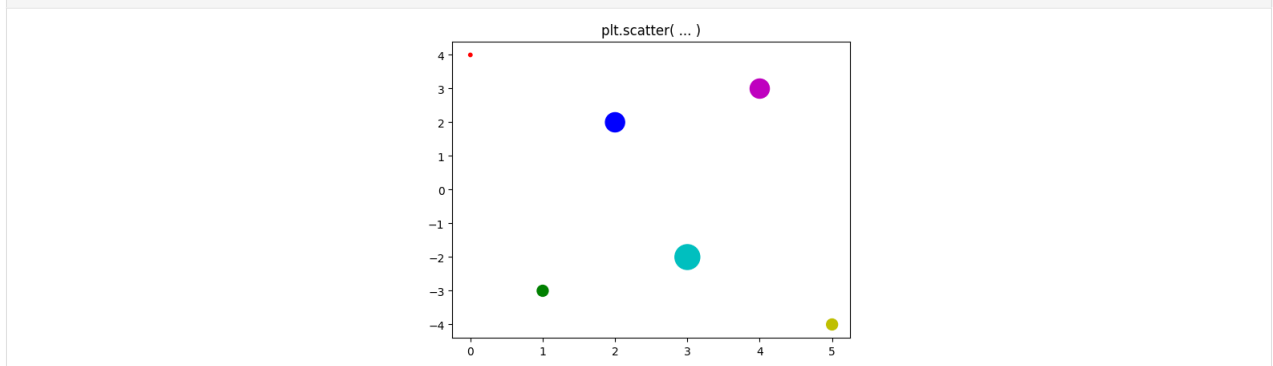
Dokumentace funkce `plot` je rozsáhlá, zmiňme jen, že první a druhý argument jsou pole x-ových a y-ových souřadnic bodů. Pokud nevedeme další argumenty, vykreslí `plot` lomenou čáru. Další argument umožňuje modifikovat styl barvu atd, např. `'o'` je tečkovaná a `--` čárkovaná čára, zatímco `'o-r'` je plná čára (`-`) spojující puntíky (`o`) obojí v červené barvě (`r`), atp.

Občas ještě budeme potřebovat funkci `scatter` pro malování puntíků různých velikostí a barev.

```
[ ]: import matplotlib.pyplot as plt

body_x = np.array( [0,1,2,3,4,5] )
body_y = np.array( [4,-3,2,-2,3,-4] )
plochy = np.array( [10,100,300,500,300,100] )
barvy = np.array( ['r','g','b','c','m','y'] )

plt.scatter( body_x, body_y, s=plochy, c=barvy )
plt.title("plt.scatter( ... )")
plt.show()
```



11.4 Univerzální funkce a výrazy s poli

Povšimněme si, že knihovna `numpy` přináší pohodlnou práci s poli, která obsahují hodnoty určené matematickými funkcemi.

Například graf funkce pořídíme takto:

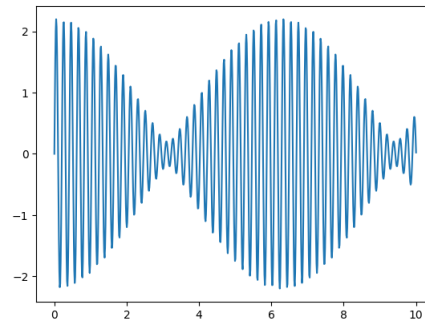
```
[ ]: import numpy as np
import matplotlib.pyplot as plt
```

(continues on next page)

(pokračujte na předchozí stránce)

```
x = np.linspace(0, 10, 800)
y = np.sin(30*x)+1.2*np.sin(31*x)
```

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



Popis kódu

- Funkce `numpy.linspace` pohodlně vytvoří 800 bodů rovnoměrně pokrývajících daný interval. Jakmile tak získáme rovnoměrně rozložené hodnoty pokrývající daný interval reálných čísel, můžeme využít schopností `numpy` sčítat, násobit a tp. celá pole a také spočítat hodnotu nějaké funkce pro všechny elementy pole "najednou".
- `numpy` poskytuje m.j. funkci `np.sin`, která spočte sinus pro všechny prvky pole, které dostane jako argument. Vrábí tedy stejně dlouhé pole, jako je argument.

Další matematické funkce, které najdeme v `numpy`:

- Funkce jedné proměnné: `sqrt`, `cbt`, `exp`, `log`, `log10`, `log2`, `log1p`, `real`, `imag`, `sin`, `cos`, `tan`, `arcsin`, `arccos`, `arctan`, `sinh`, `arcsinh`, ...
- Funkce dvou polí (prvek po prvku z obou argumentů): `hypot`, `arctan2`, `copysign`, `gcd`, `lcm`, `maximum`, `minimum`
- Funkce pole a skalárního argumentu: `round`
- `numpy` naučí operace `+` a `*` (a další) sčítat resp. násobit pole prvek po prvku, tedy ve smyslu $c_i = a_i + b_i$ resp. $c_i = a_i * b_i$. Zatímco u sčítání jde o operaci totožnou se sčítáním vektorů, u násobení ne, skalární součin by dal jedno číslo a o to zrovna zde nestojíme.
- viděli jsme ale, že je dovoleno násobit vektor skalární hodnotou ($30*x$, tj. $c_i = k * a_i$) a podobně bychom mohli ke všem prvkům vektoru přičíst konstantu, kdybychom napsali `x+1` ve smyslu $c_i = a_i + q$

Z toho, co víme o operacích `+` a `*` u seznamů, které pro ně také mají svůj (velmi odlišný) význam, nás nepřekvapí, že když plus "vidí" dva operandy typu `ndarray` udělá jednu akci, když vidí operand typu `ndarray` a řekněme `float`, udělá něco jiného. Jde o tzv. přetěžování operátorů, kdy se podle typu operandů zavolá ta či ona funkce. Konkrétní realizaci přetěžování operátorů v jazyce Python si vysvětlíte v navazujících kursech.

Ve výsledku tak můžeme používat operace `+`, `-`, `*`, `/`, `**`, `//`, `%` a zapisovat tak celé výrazy obsahující pole. Pokud sčítáme dvě pole `p1 + p2`, musejí mít `p1` a `p2` stejnou délku, můžeme však také přičíst ke všem prvkům pole stejnou skalární hodnotu, tak že napíšeme `p1 + 1`. Podobně je to i u dalších operací.

```
[ ]: p = np.linspace(0, 4, 5)
      print('  p =', p)                # pole
      print(' 2*p =', 2*p)            # skalar * pole
      print(' 4-2*p =', 4-2*p)        # skalar - skalar * pole
```

(continues on next page)

(pokračujte na předchozí stránce)

```
print(' ... =', np.sin(p)**2+np.cos(p)**2) # pole**2 + pole**2
```

```
p = [0. 1. 2. 3. 4.]
2*p = [0. 2. 4. 6. 8.]
4-2*p = [ 4.  2.  0. -2. -4.]
... = [1. 1. 1. 1. 1.]
```

V následujícím příkladě využijeme toho, že `np.sin` vytvoří stejně dlouhé pole jako je jeho argument. Funkce pro malování totiž vyžaduje aby byl stejný počet x-ových a y-souřadnic malovaných bodů.

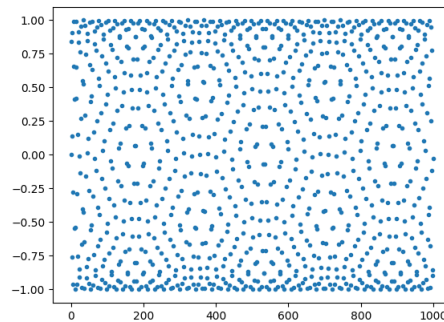
```
[ ]: # obsazení pole hodnotami danými funkcí
# příklad ufunc: np.sin
```

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

tabulka_int = np.arange(0,1000)

tabulka_sin = np.sin(tabulka_int)

plt.plot(tabulka_int, tabulka_sin, '.')
plt.show()
```



Úloha: Funkce $\sin(x)$ má periodu 2π . Funkce $\sin\left(\frac{\pi x}{22}\right)$ má periodu $2 \times 22/7$. Vyzkoušejte, jak různé racionální aproximace čísla $\pi \approx 3, 22/7, 333/106$ mění obrázek výše.

11.5 Argumenty typu pole – výstup

Protože jsou argumenty v Pythonu předávány jako odkazy, můžeme je někdy měnit. Má to ale své podmínky:

- argument musí být *mutable* typu (to pole splňují)
- nesmíme použít přiřazení do proměnné ale jen do její části
- můžeme také používat metody, jako je třeba `append` u typu `list`

Nejprve si ilustrujeme správný postup:

```
[ ]: def vynuluj_zaporne(a):
    for i in range(len(a)):
        if a[i]<0:
            a[i] = 0

pole1 = np.array( [1,-1,2,-2,3,-3,4] )
vynuluj_zaporne(pole1)

print(pole1)

[1 0 2 0 3 0 4]
```

```
# This is formatted as code
```

Nyní totéž špatně. Následující funkce nemodifikuje jednotlivé prvky pozměňováním $a[i]$ postupně pro všechna i . Místo toho spočte správný výsledek najednou na základě vlastností funkce $(x + |x|)/2$ a tento **dosadí** do a .

```
[ ]: def vynuluj_zaporne_spatne(a):
    print('hodnota a na zacatku',a)
    a = (a+np.abs(a))/2
    print('hodnota a na konci ',a)

pole1 = np.array( [1,-1,2,-2,3,-3,4] )
vynuluj_zaporne_spatne(pole1)

print(pole1)

hodnota a na zacatku [ 1 -1  2 -2  3 -3  4]
hodnota a na konci   [1.  0.  2.  0.  3.  0.  4.]
[ 1 -1  2 -2  3 -3  4]
```

Vidíme, že `pole1` zůstane nezměněno, ačkoli argument `a` po aplikaci vzorce nabyl uvnitř funkce správných hodnot.

Klíčovou chybou je začátek příkazu

```
a = ...
```

Podstata přiřazovacího příkazu v jazyce Python totiž určuje, že tím změníme, na co lokální symbol `a` ukazuje, nikoli valstní hodnoty na které před přiřazením `a` odkazovalo. Od tohoto okamžiku tedy máme původní `pole1` se zápornými čísly a pak nová data na které odkazuje proměnná `a`. Tato data jsou lokální v dané funkci na jejím konci zmizí, zatímco `pole1` se nezmění.

Časem se dovíme, že v problém lze řešit též použitím tzv. řezů `a[:] = ...`, kdy přepíšeme najednou více prvků pole, aniž by `a` začalo odkazovat na jiné pole. Spíše ale v takovéto situaci je správné použít příkaz `return`.

Takové funkce buď pole vytvoří nebo spočtou jako modifikaci pole vstupního `a` a pak použijí příkaz `return`. Například

```
return (a+np.abs(a))/2
```

Jako ukázkou si napíšeme funkci, která počítá tabulku funkčních hodnot $f(x_i)$ pro x_i rovnoměrně pokrývající interval $\langle a, b \rangle$. Argumenty funkce budou čtyři: funkce `f`, meze intervalu `a, b` a počet bodů pokrývajících tento interval ve kterých se má funkční hodnota spočít.

Všimněte si, že funkce `function_table` má pole jako svůj výstup, zatímco `plot` zase jako vstup a že je možné tyto role propojit.

Pozn. Časem totéž dosáhneme s použitím `numpy.linspace`.

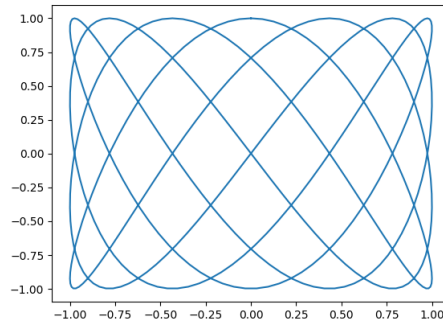
```
[ ]: def function_table(f, start, end, count=400):
    tab = np.full(count, 0.0)
    for k in range(count):
        x = start + (end-start)*k/(count-1)
        tab[k] = f(x)

    return tab

from math import sin,cos,pi
import matplotlib.pyplot as plt

p,q = 4,7

plt.plot( function_table(sin,0,p*2*pi), function_table(cos,0,q*2*pi) )
plt.show()
```



Protože Python stejně pracuje s odkazy, není vrácení polí nic speciálního. Uvědomme si však proč by to mohl být problém. Pokud funkce vrací pole, například náhodných čísel, pak v příkazu

```
print( sum_positive( seznam_nahodnych_cisel(1_000_000) ) )
```

musí někde chvíli existovat seznam milionu čísel a to od chvíle, kdy je vytvořen až do okamžiku, kdy skončí sčítání. Poté ale již seznam není potřeba a protože milion čísel není málo, je potřeba uvolnit paměť, kterou zabíral. Věc by se ještě mohla zkomplikovat tím, kdyby si (například za účelem ladění) funkce `sum_positive` poznamenala odkaz na poslední sčítané pole. Potom by úklid milionu prvků nastal až po dalším zavolání této funkce, protože ta by si při něm poznamenala odkaz na nové pole, čímž by ten starý přestal existovat.

Právě neviditelnost těchto operací určuje mnohé vlastnosti a asi i popularitu jazyka Python a odlišuje jej např. od C, kde tohle musíme explicitně řešit. Cenou, kterou za pohodlí platíme bývá m.j. rychlost.

Jednou z možností, jak organizovat práci s poli je mít je uložené v jedné nebo několika globálních proměnných.

Program pak sestává z procedur, která s těmito daty vhodně manipulují. Nemusejí ale typicky řešit jejich předávání – data celou dobu sídlí v k tomu určených proměnných.

Uvažujme následující jednoduchý příklad programu.

Hlavní program

- inicializuje tabulku prvočísel prostřednictvím volání funkce `eratosthenes`
- vyrobí tabulky hodnot, přičemž použije opakovaně funkci `goldbach_count`
- namaluje obrázek

Podprogramy

- funkce `eratosthenes` obsadí globální proměnnou pojmenovanou `je_prvocislo`. Protože do této proměnné přiřazuje, je v ní uvozena jako `global je_prvocislo`
- funkce `goldbach_count` namaluje spočte počet párů prvočísel dávajících daný součet. Proměnnou `je_prvocislo` silně používá ale nemodifikuje

Povšimněte si, že použití globální proměnné v kódu na rozdíl od mnoha jiných jazyků, nemusí předcházet řádku, kde je proměnná vytvořena. Vznik proměnné je totiž odlišný od deklarace, jaká se používá např. v C.

Pozn. Až se to naučíme, je `goldbach_count` dobře akcelarovatelná použitím `@njit`. (Ještě většího zrychlení výpočtu lze ovšem dosáhnout za použití nápadů slavných matematiků. Čtení o Goldbachově hypotéze.)

```
[ ]: # Program počítá, kolik dvojic prvočísel dává daný součet (sudý a větší než 2)
# Souvisí s Golbachovu hypotézou, která říká, že se vždy nějaká dvojice najde
# Program demonstruje použití/sdílení globálních proměnných více funkcemi
# (a nikoli optimální algoritmus vykreslující "Goldbachovu kometu")
```

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

def goldbach_count(k):
    """Počet dvojic prvočísel dávajících v součtu hodnotu k.
    Používá globální proměnnou je_prvocislo,
    kterou vytvoří procedura erarosthenes."""
    assert k%2==0 and k>2, "k musí být sudé a větší než 2"

    p_max = len(je_prvocislo)-1
    assert k<p_max, "Příliš krátký seznam prvočísel"

    count = 0
    for p in range(2,k-1):
        if je_prvocislo[p] and je_prvocislo[k-p]:
            count = count+1

    return count

def eratosthenes(n):
    """Vytváří pole [False,False,True,True,False,True,...] délky n+1
    indikující, že mezi čísla 0..n jsou 2,3,5,... prvočísla"""
    global je_prvocislo
    je_prvocislo = np.full(n+1, True)
    je_prvocislo[0] = False
    je_prvocislo[1] = False

    p = 2
    while p*p <= n:
        if je_prvocislo[p]:
            for nasobek_p in range(2*p,n+1,p):
                je_prvocislo[nasobek_p] = False
            p = p + 1

n_max = 10000

eratosthenes(n_max)
```

(continues on next page)

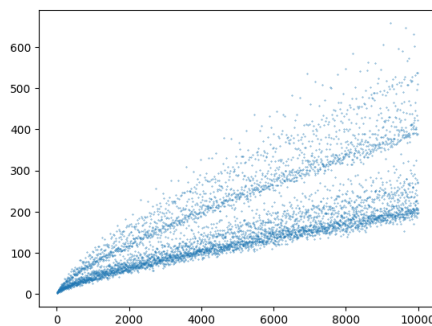
```

delka_pole = n_max//2-2
suda = np.full(delka_pole,0)
pocty = np.full(delka_pole,0)

for k in range(delka_pole):
    s = 2*k+4 # cisla 4,6,8,...
    suda[k] = s
    pocty[k] = goldbach_count(s)

plt.scatter( suda, pocty,s=0.1)
plt.show()

```



11.6 Globální proměnná v roli statické lokální proměnné

Někdy musíme použít globální proměnnou i když s ní pracuje jen jedna jediná funkce a zasloužila by si tedy být proměnnou lokální. Mohli jsme to potkat již dříve, kdybychom psali funkci, která potřebuje uložit mezi svými voláními nějaký stav. I u numerického kódu je občas potřeba si pamatovat hodnoty mezi voláními funkce: někdy kvůli urychlení, jindy z principu. Často pak mají tyto hodnoty podobu polí.

Jednoduchým příkladem globální proměnné poskytující službu úschovny pro nějakou funkci je varianta funkce, která počítá obvod elipsy vzorečkem.

Hlavním bodem výpočtu je součet mocninné řady, přičemž koeficienty řady jsou dost složité na to, aby, když už je jednou spočteme, bylo výhodné si je uložit pro příště. Již dříve jsme použili jsme následující vzorec pro obvod elipsy s poloosami a, b

$$O = 2\pi a \left[1 - \sum_{k=1}^{\infty} \left(\frac{(2k-1)!!}{(2k)!!} \right)^2 \frac{\epsilon^{2k}}{2k-1} \right]$$

Nepředstavuje sice optimální návod na výpočet obvodu elipsy, ale umožní nám demonstrovat obvyklou možnost zrychlení tohoto typu výpočtů. Sumu v závorce můžeme zapsat $\sum_{k=1}^{\infty} a_k \epsilon^{2k}$, přičemž

$$a_k = a_{k-1} \frac{(2k-1)(2k-3)}{(2k)^2}, \quad a_1 = \frac{1}{4}$$

Pokud bychom opakovaně potřebovali počítat obvody elips s různou výstředností, mohlo by být výhodné mít uschované hodnoty a_k . (Lepší postup pro numerický výpočet obvodu elipsy najdete [zde](#) a je implementován v knihovně `scipy`, kterou použijeme pro kontrolu správnosti našeho výpočtu.)

```
[ ]: import numpy as np

def obvod_elipsy(a, b, rtol=1e-10):
    global oe_ak_ready

    if b>a:
        a,b = b,a

    eps2 = 1-b**2/a**2
    eps2k = eps2
    atol = 0.5*rtol*(1-eps2)

    s = 1
    for k in range(1,oe_ak_ready+1):
        ds = oe_ak[k]*eps2k
        s -= ds

        if ds<atol:
            return 2*np.pi*a*s

        eps2k *= eps2

    for k in range(oe_ak_ready+1,oe_max_k):
        oe_ak[k] = oe_ak[k-1]*(2*k-1)*(2*k-3)/(4*k*k)
        oe_ak_ready = k

        ds = oe_ak[k]*eps2k
        s -= ds

        if ds<atol:
            return 2*np.pi*a*s

        eps2k *= eps2

    assert False, "Soucet rady nekonverguje"

oe_max_k = 1000
oe_ak = np.full(oe_max_k,0.25)
oe_ak_ready = 1

import scipy.special
def obvod_elipsy_presne(a, b):
    """Výpočet obvodu elipsy s použitím eliptických integrálů z knihovny scipy"""
    eps2 = 1-b**2/a**2
    return 4*a*scipy.special.ellipe(eps2)    # zajímavost: funguje i pro eps2<0

rel_chyba = obvod_elipsy(1,0.2)/obvod_elipsy_presne(1,0.2) - 1

print(f'{rel_chyba = } {oe_ak_ready = }')
```

```
rel_chyba = 6.183942247162122e-11 oe_ak_ready = 331
```

Povšimněte si, že příkaz `global oe_ak_ready` se týká jen jedné ze třech globálních proměnných používaných funkcí `obvod_elipsy`. Již víme, že je to proto, že jen do této globální proměnné uvnitř funkce dosazujeme. Přitom modifikujeme i druhou proměnnou `oe_ak`, ve které postupně budujeme tabulku koeficientů. Je to ale jen přiřazení, před kterým jsou globální proměnné "ochráněny" existencí příkazu `global`!

Poznámka: Bývá zvykem, když vše, co funkce potřebuje interně ke svému provozu má podobu lokálních proměnných. Jejich existence je ale běžně omezena na dobu běhu funkce, zatímco zde potřebujeme, aby tabulka hodnot koeficientů existovala i mezi voláními funkce. Za tímto účelem například v jazyce C/C++ máme modifikátor `static`, kterým říkáme, že lokální proměnná má existovat pořád. Protože v Pythonu ale statické lokální proměnné neexistují, používají někteří programátoři jako náhražku atributy funkce. Protože funkce existuje počínaje provedením příkazu `def funkce(...)`, je možné jak uvnitř funkce, tak na globální úrovni použít zápis `funkce.id_promenne`. Hodnota uložená v tomto atributu existuje nezávisle na lokálních proměnných funkce.

11.7 Globální proměnná v roli konstanty

Většina programovacích jazyků zná koncept konstanty. Jde o veličinu, která má po dobu běhu programu neměnnou hodnotu. My si snadno představíme např. π , ale v typickém programu bývá mnohem víc konstant. Pro kompilované jazyky přinášejí konstanty možnost urychlit běh kódu, triviálně například v příkazu `y = 2*pi*r` provést násobení $2 * \pi$ již během překladač a nezdržovat s ním při vlastním výpočtu.

Python nezná konstanty. (Dokonce i hodnotu `math.pi` můžete změnit.)

To ovšem nic nemění na tom, že některé veličiny v programu mají význam konstant. Pokud se taková 'konceptní' konstanta objevuje ve funkci, znamená to, že ji při každém zavolání funkce je potřeba vytvořit. To se projeví zejména, pokud je konstanta typu pole. To v principu může zdržovat a jako řešení tedy máme možnost nahradit ji proměnnou globální. V Pythonu je dokonce doporučováno takové proměnné zapisovat velkými písmeny, aby bylo čtenáři (případně i editoru vývojového prostředí, nikoli však interpretu jazyka Python) jasné, že jde o konstantu.

Nejprve jednoduchý příklad: počítáme objem mnohostěnu s danou délkou hrany a počtem stěn. Pravidelné mnohostěny mohou mít jen 4, 6, 8, 12 a 20 stěn. Pro každý z nich použijeme známý objem mnohostěnu s jednotkovou hranou (např. u krychle je to 1) a ten vynásbíme třetí mocninou délky hrany. Dokonce i v Pythonu se pozná, že u verze 1 takové funkce vždy znovu a znovu inicializujeme tabulku objemů Platónských těles.

```
[ ]: import numpy as np

def objem_mnohostenu_v1(n, a):
    """Počítá objem pravidelného mnohostěnu s n stěnami a délkou hrany a"""

    tabulka = np.array([0,0,0,0, 0.117851130197757921, 0, 1.0, 0,
                       0.471404520791031683,0,0,0,7.66311896062463197,
                       0,0,0,0,0,0,2.18169499062491237])

    objem1 = tabulka[n]
    assert objem1 != 0, f"Pravidelný {n}-stěn neexistuje!"
    return objem1 * a**3

def objem_mnohostenu_v2(n, a):
    """Počítá objem pravidelného mnohostěnu s n stěnami a délkou hrany a"""

    objem1 = OBJEMY_MNOHOSTENU[n]
```

(continues on next page)

(pokračujte na předchozí stránce)

```

assert objem1 != 0, f"Pravidelný {n}-stěn neexistuje!"
return objem1 * a**3

OBJEMY_MNOHOSTENU = np.array([0,0,0,0, 0.117851130197757921, 0, 1.0, 0,
                               0.471404520791031683,0,0,0,7.66311896062463197,
                               0,0,0,0,0,0,2.18169499062491237])

%timeit objem_mnohostenu_v1(20,2)
%timeit objem_mnohostenu_v2(20,2)

6.22 µs ± 2.34 µs per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
928 ns ± 8.64 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

```

Podobná situace může nastat, když nějaká funkce má za úkol spočítat měřenou veličinu na základě kalibračních dat, která jsou známá v okamžiku psaní kódu. Pak lze příslušnou tabulku, řekněme závislost odporu termistoru na teplotě, uložit do globální proměnné typu pole. Protože v Pythonu neexistují konstanty, je později možné doplnit program o proceduru, která načte lepší kalibrační data např. ze souboru.

11.8 Numerická kvadratura

Kvadratura je historicky slovo označující výpočet hodnoty určitého integrálu. Obvykle se použije relace

$$\int_a^b f(x)dx \approx \sum_{i=0}^n w_i f(x_i)$$

Ukážeme si lichoběžníkové pravidlo, pro $n + 1$ bodů ležících ekvidistantně na intervalu $\langle a, b \rangle$

$$x_i = \frac{i}{n}, \quad w_i = \frac{b-a}{n} \begin{cases} \frac{1}{2} & i = 0, n \\ 1 & i \neq 0, n \end{cases}$$

Odvodíme si Simpsonovo/Keplerovo pravidlo, ale hlavně numerickou kvadraturu vezmeme jako příklad použití globálních "konstant".

K simpsonovi: - Lichoběžník neumí správně integrovat ani kvadratické polynomy - lineární substituce umožní uvažovat jen interval $\langle -1, 1 \rangle$ - symetrické rozložení x_i a sudé w_i automaticky integruje správně všechny liché mocniny - uhadneme $w_i = \{1/3, 4/3, 1/3\}$ jako váhy integrující správně 1 a x^2 pro $x_i = \{-1, 0, 1\}$.

Ke Gaussovi - Není důvod vyžadovat ekvidistanční pokrytí intervalu - víc stupňů volnosti, větší mocniny lze integrovat přesně

Následuje příklad programu, který počítá numerickou hodnotu integrálu $\int_0^1 \sqrt{1+x^3} dx$. Používá numerickou metodu, která používá seznam bodů, kde se vyčíslují hodnoty integrandu a seznam vah, jakými tyto hodnoty přispějí k hodnotě určitého integrálu. Seznam poloh i vah představují konstanty, které se nezměnily od dob C. F. Gausse.

Co si pamatovat:

- Lichoběžníkové pravidlo je $f(x_i) \cdot [0.5, 1, 1, 1, 1, 0.5] \cdot dx$
- vědět, že je dost neefektivní
- že ani Simpsonovo vylepšení $f(x_i) \cdot [1, 4, 2, 4, 2, 4, 1] \cdot dx/3$ není nic moc
- vědět, že ekvidistanční vzorkování funkce je většinou naivní a že správné jméno je Gauss

Program níže ilustruje

- "konstanty" v Pythonu
- defaultní parametry
- naivní určení chyby porovnáním hrubého a jemného kroku

```
[ ]: import numpy as np

GAUSS8 = np.array([
    [0.183434642495649805,0.525532409916328986,
     0.796666477413626740,0.960289856497536232],
    [0.181341891689180991,0.156853322938943644,
     0.111190517226687235,0.0506142681451881296]])

GAUSS16 = np.array([
    [0.0950125098376374,0.2816035507792589,0.4580167776572274,
     0.6178762444026438,0.7554044083550030,0.8656312023878318,
     0.9445750230732326,0.9894009349916499],
    [0.09472530522753425,0.0913017075224618,0.08457825969750125,
     0.07479799440828835,0.06231448562776695,0.0475792558412464,
     0.03112676196932395,0.01357622970587705]])

TRAPEZOID6 = np.array([[0.2,0.6,1.0],[0.2,0.2,0.1]])

SIMPSON11 = np.array([[0.0, 0.2, 0.4, 0.6, 0.8, 1.0],
                      [1/15,1/15,2/15,1/15,2/15,1/30]])
# sorry, zde fejkujeme sudy počet zdvojeným výpočtem v nule, proto 1/15

def quadrature(f, a, b, method=GAUSS8):
    k = (b+a)/2
    l = (b-a)/2

    s = 0
    for i in range(len(method[0])):
        s = s + ( f(k+l*method[0,i]) + f(k-l*method[0,i]) ) * method[1,i]

    return s*(b-a)

def print_integral(f, a, b, method=GAUSS8):
    intf = quadrature(f, a, b, method=method)

    mid = (a+b)/2
    intf2 = quadrature(integrand, a, mid, method=method) + quadrature(integrand, mid, b,
↪method=method)

    ee = abs(intf - intf2)

    print(f"{intf} ± {ee:.7.2}")

def integrand(x):
    return np.sqrt(1+x**3)
```

(continues on next page)

(pokračujte na předchozí stránce)

```
print_integral(integrand, 0, 1, method=TRAPEZOID6)
print_integral(integrand, 0, 1, method=SIMPSON11)
print_integral(integrand, 0, 1)
print_integral(integrand, 0, 1, method=GAUSS16)
```

```
1.1149921123536037 ± 0.0027
1.111445816918736 ± 2e-06
1.1114479705504803 ± 1.8e-11
1.1114479705325755 ± 2.2e-16
```

Cvičení: Funkce `integrand` používá `np.sqrt` a další operace, které dávají smysl nejen se skalárními hodnotami, ale i s vektory `numpy.ndarray`. Zkuste přepsat funkci `quadrature` za použití `numpy`. Použijte též funkci `np.dot(a, b)`, která, pokud dostane jako argumentu stejně dlouhé vektory, spočte $\sum_i a_i b_i$, takže kód lze zjednodušit až do podoby

```
return (b-a)*np.dot(..., method[1])
```

Elementární algoritmy pracující s poli

Na přednášce si vysvětlíme několik jednoduchých příkladů realizujících základní činnosti s poli.

12.1 Obsazování polí hodnotami

Asi nejzákladnějším problémem je obsazení pole hodnotami určenými indexem. Například může jít o aritmetickou posloupnost v matematickém zápise $x_j = a + j h$.

Již jsme zmínili, že téhož lze dosáhnout použitím funkce `numpy.linspace(a,b,num)`, také existuje funkce `numpy.arange([start,]stop[,step])`, která ovšem nezahrne do pole hodnotu `stop`.

```
[ ]: import numpy
print(f"{numpy.arange(1,2,0.1) = }")
print(f"{numpy.linspace(1,2,11) = }")

numpy.arange(1,2,0.1) = array([1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9])
numpy.linspace(1,2,11) = array([1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2. ])
```

Tyto funkce jsou natolik užitečné, že si jejich přibližné varianty napíšeme jako názornou ukázkou, jak vytvářet pole.

```
[ ]: def my_arange(start, stop=None, step=1):
    """Return evenly spaced values within a given interval. See numpy.arange(...)"""
    if stop==None:
        stop = start
        start = 0

    num = int( (stop-start)/step )
    if num<0:
        num = 0

    result = np.zeros(num)
    for i in range(num):
```

(continues on next page)

```

    result[i] = start + i*step

    return result

def my_linspace(start, stop, num=50):
    """Return evenly spaced numbers over a specified interval. See numpy.linspace(...)"""
    result = np.zeros(num)
    for i in range(num):
        result[i] = (start*(num-1-i) + i*stop)/(num-1)

    return result

```

Již jsme viděli pohodlnější postup za pomoci univerzálních funkcí. Pokud takový postup nechceme použít, je obvyklým postupem vytvoření nového pole aplikací funkce na každý prvek původního pole.

```

[ ]: # Demonstrace  $y[i] = f(x[i])$ 

import numpy as np
import matplotlib.pyplot as plt

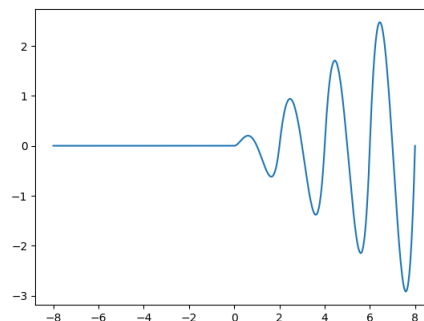
# nejprve nějaká funkce jedné proměnné
def funkce(x):
    if x<0: return 0
    t = x % 2.0
    return x*t*(1-t)*(2-t)

# vytvoření hodnot  $x[i]$ 
tabulka_x = np.linspace(-8,8,500)

# výpočet hodnot  $y[i] = f(x[i])$ 
tabulka_y = np.zeros(len(tabulka_x)) # vytvoříme prázdné pole vhodné délky
for i in range(len(tabulka_x)):
    tabulka_y[i] = funkce(tabulka_x[i])

# obrázek
plt.plot(tabulka_x, tabulka_y)
plt.show()

```



12.2 Vyhledávání v poli

Pole můžeme procházet nejen proto, abychom z něj spočetli pole jiné, ale také proto, abychom zjistili, zda se v něm daný prvek (zde číslo) nachází.

Napišeme si funkci `slow_in(x, a)`, která zjistí, zda se prvek `x` nachází v poli `a`. Jako obvykle v takové situaci, když prvek najdeme, může funkce okamžitě vrátit `True` a nemusí procházet zbylé prvky pole.

Funkci použijeme tak, že změříme čas, který potřebuje k nalezení posledního čísla v seznamu, a porovnáme to s rychlostí existující funkce volané v rámci testu `x in a`, jak jej nabízí jazyk Python a implementuje knihovna `numpy`.

```
[ ]: # vyhledání prvku
```

```
def slow_in(x, a):
    for y in a:
        if y==x:
            return True
    return False

%timeit slow_in(999,tabulka_int)
%timeit 999 in tabulka_int
```

```
221 µs ± 11.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
5.7 µs ± 1.59 µs per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Opět vidíme, že funkce psané v Pythonu jsou oproti funkcím např. v C značně pomalejší, ale použití lepšího algoritmu to může dohnat. Pokud je totiž seznam seříděný, lze v něm vyhledávat pomocí "půlení intervalu".

Za pozornost v následujícím kódu stojí:

- Vlastní algoritmus binárního hledání
- Použití `//` tedy celočíselného dělení
- Přiřazení `left=mid+1` resp. `right=mid-1` která zaručí, že pokud prvek v seznamu není, skončíme nakonec s `left > right` a tedy cyklus `while` nemůže být nekonečný.

```
[ ]: # binární vyhledání prvku v seříděném seznamu
```

```
def binary_in(x, a):
    left = 0
    right = len(a)-1
    if x<a[left] or x>a[right]:
        return False

    while left <= right:
        mid = (left+right)//2
        if a[mid]==x:
            return True
        if a[mid]<x:
            left = mid+1
        else:
            right = mid-1

    return False
```

(continues on next page)

```
tabulka100k = np.arange(100_000)
```

```
%timeit binary_in(946,tabulka100k)
%timeit 946 in tabulka100k
```

```
18.4 µs ± 5.45 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
56.3 µs ± 9.79 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

12.3 Ověření vlastností hodnot uložených v poli

Následující dvě funkce ukazují, nejjednodušší příklady toho, jak projít seznam a prověřit nějakou jeho vlastnost. V prvním případě je to individuální vlastnost prvků, ve druhém pak vlastnost sousedních prvků v poli. Proto v druhém případě musíme použít indexy prvků, abychom sousedskost dokázali zapsat. V prvním případě nás zajímá jen hodnota jednotlivých prvků a tak indexy nepoužijeme (ale mohli bychom).

```
[ ]: def all_positive(a):
    """Jsou všechny prvky seznamu a kladné?"""
    for x in a:
        if not x>0:
            return False
    return True
```

```
print( all_positive(tabulka_sin) )
print( all_positive(tabulka_sin+1) )
```

```
False
True
```

```
[ ]: def is_sorted(a):
    """Představuje seznam a neklesající posloupnost?"""
    for i in range(1,len(a)):
        if not a[i-1] <= a[i]:
            return False
    return True
```

```
print( is_sorted(tabulka_int) )
print( is_sorted(tabulka_sin) )
```

```
True
False
```

12.4 Hledání optimálního prvku seznamu

Důležitou třídou problémů je hledání optimálního prvku seznamu. Obvykle procházíme prvek po prvku a kontrolujeme, zda není aktuální prvek lepší než ten nejlepší, co jsme doposud potkali. (Název funkce odráží fakt, že existuje funkce `max`.)

```
[ ]: def slow_max(a):
    wrk = a[0]
    for x in a:
        if x > wrk:
            wrk = x
    return wrk

slow_max(tabulka_sin)
```

```
0.999990471552965
```

12.5 Hledání optimálního prvku seznamu s podmínkou

V následujícím kódu řešíme kromě běžného procházení všech prvků pole také to, že žádný prvek nemusí splňovat požadovaná kritéria. Tento případ funkce musí nějak signalizovat. Využíváme toho, že Python poskytuje typ/hodnotu `None`.

Hodnotu `None` ale nemůžeme porovnávat a tedy podmínka aktualizace doposud nalezené optimální hodnoty musí obsahovat dvě podmínky. Aktualizace nastane jednak když `wrk` is `None` a nebo když `x > wrk`. Připomeňme si, že díky **zkrácenému vyhodnocování logických výrazů** nenastane chyba, protože, když je první podmínka splněna, or se neobtěžuje s vyhodnocováním druhé.

```
[ ]: def max_negative(a):
    wrk = None
    for x in a:
        if x < 0:
            if wrk is None or x > wrk:
                wrk = x
    return wrk
```

```
max_negative(tabulka_sin)
```

```
-3.014435335948845e-05
```

Ještě následuje kód, který připomíná, že na to, abychom našli dva největší prvky pole, nemusíme nezbytně pole třídit.

```
[ ]: def max2(a):
    assert len(a)>1
    m1 = a[0]
    m2 = a[1]
    if m1<m2:
        m1,m2 = m2,m1

    for i in range(2,len(a)):
        x = a[i]
```

(continues on next page)

```

    if x > m1:
        m2 = m1
        m1 = x
    elif x>m2:
        m2 = x
    return m1,m2

```

```
max2(tabulka_sin)
```

```
(0.999990471552965, 0.9999900726865629)
```

12.6 Jedno pole, dva cykly

Zde si ukážeme příklady algoritmů, které nevystačí s jediným cyklem. (Ještě uvidíme, že často existují algoritmy rychlejší.)

Začneme hledáním duplicit.

```

[ ]: def bez_duplicit(a):
    n = len(a)
    for i in range(1,n):
        for j in range(i):
            if a[i]==a[j]:
                return False

    return True

[ bez_duplicit(tabulka_int), bez_duplicit(tabulka_int%999) ]

```

```
[True, False]
```

Nadšenci do jazyka Python by možná navrhli mnohem (>30x) rychlejší realizaci téhož postupu. Například můžeme využít možnosti porovnávat celé pole s nějakou hodnotou, získat pole booleovských hodnot True/False a pak spočítat, kolikrát se ve výsledku objeví True, tedy kolik prvků tuto podmínku splňuje.

```

[ ]: def bez_duplicit2(a):
    for x in a:
        if np.count_nonzero( a == x ) > 1:
            return False

    return True

[ bez_duplicit2(tabulka_int), bez_duplicit2(tabulka_int%999) ]

```

```
[True, False]
```

Takovýto postup je již za hranicemi toho, co v našem kurzu stihneme. Musíme vystačit jen s několika málo operacemi, jejichž kombinací dokážeme napsat programy realizující běžné algoritmy. Přesto výše uvedené urychlení výpočtu obsahuje jednu samozřejmou myšlenku, kterou musíme zmínit. Obvykle je to nejvnitřnější cyklus, který nejvíce ovlivňuje rychlost složitějšího výpočtu. Proto má smysl hledat optimální podobu kódu nejdříve tady. Navržený postup zdánlivě vnitřní cyklus odstraní, je to ale jen proto, že několik cyklů je schovaných uvnitř operace `==` a funkce `numpy.count_nonzero`. Zrychlení hledání duplicit tedy nesouvisí s tím, že bychom vnitřní cyklus odstranili ale s tím, že je zapsán v jazyce C.

K hledání duplicit v seznamu se ještě vrátíme, protože algoritmus výše není většinou optimální a budeme rozvažovat jaké jsou alternativy.

V našem začátečním kurzu se jednak učíme napsat správný kód (a to první varianta `bez_duplicit` splňuje) ale také si časem povíme, jak teoreticky posuzovat rychlost algoritmů, abychom mohli vybrat ten lepší. Z tohoto pohledu uvidíme, že `bez_duplicit` a `bez_duplicit2` mají stejnou asymptotickou složitost $O(N^2)$.

Jako aplikaci takového teoretického rozvažování si ukážeme, že pro dlouhé seznamy je lepší pole před hledání duplicit seřadit, protože duplicitu pak můžeme odhalit porovnáním sousedních prvků.

Vraťte se k výše zmíněným algoritmům pro duplicity až se v následujících kapitolách seznámíte s pojmem časová složitost. Až probereme typ slovník (`dict`) napište funkci, která jej používá ke zjišťování duplicit.

Tak jako jsme výše potkali neprobíranou část knihovny `numpy` i vlastní jazyk Python obsahuje části, které neprobíráme.

Zmiňme proto, že jeden možný test duplicit je založen na datové struktuře Pythonu – množině (`set`). Množina z principu daný prvek obsahuje nebo ne, nemůže tam ale být vícekrát. Proto konverzí `set` (pole) získáme množinu, jejíž počet prvků lze pak porovnat s počtem prvků původního pole. Je zřejmé, že možnost konverze seznamu na množinu znamená, že Python obsahuje

```
[ ]: def bez_duplicit3(a):
      return len(set(a)) == len(a)

[ bez_duplicit3(tabulka_int), bez_duplicit3(tabulka_int%999) ]

[True, False]
```

Pochopíme, že je výhodné hledat duplicity v seřazeném seznamu. Naučíme se i teoreticky spočítat, kolikrát zhruba je takový postup rychlejší a to i když započteme čas potřebný pro seřazení seznamu. Zde jen kód, protože je naprosto srozumitelný, teorie později...

```
[ ]: def bez_duplicit4(a):
      n = len(a)
      a = sorted(a)

      for i in range(1,n):
          if a[i-1] == a[i]:
              return False

      return True

[ bez_duplicit4(tabulka_int), bez_duplicit4(tabulka_int+[33]) ]
```

```
[ ]: from numba import njit

@njit
def bez_duplicit5(a):
    n = len(a)
    for i in range(1,n):
        for j in range(i):
            if a[i]==a[j]:
                return False

    return True

[ bez_duplicit5(tabulka_int), bez_duplicit5(tabulka_int+[33]) ]
```

```
[ ]: %timeit bez_duplicit(tabulka_sin)
      %timeit bez_duplicit2(tabulka_sin)
      %timeit bez_duplicit3(tabulka_sin)
      %timeit bez_duplicit4(tabulka_sin)
      %timeit bez_duplicit5(tabulka_sin)
```

```
[ ]: bez_duplicit([1,5,8,3,1,4,6])
```

Problém: Najděte v seznamu dvě čísla, pro která je jejich podíl co nejbližší dané hodnotě.

Opět použijeme dva cykly, chytřejší algoritmus pro tento umělý problém nestojí za vymýšlení. To že mluvíme o podílu znamená, že nesmíme dělit nulou a že dělení není komutativní, takže budeme mít oba cykly úplné. Proto přichází v úvahu, nepoužívat cykly přes indexy.

```
[ ]: def nejblizsi_podil(a, q):
      """V seznamu a najde dva prvky které mají podíl co nejbližší q. Může jít 2x o tentýž
      ↪prvek."""

      best_x = a[0]
      best_y = a[0]
      best_q = 1

      for y in a:
          if y!=0:
              for x in a:
                  xy = x/y
                  if abs(xy-q) < abs(best_q-q):
                      best_x = x
                      best_y = y
                      best_q = xy

      return best_x, best_y, best_q

nejblizsi_podil(tabulka_sin, 0.3)
```

Úloha: Napište verzi s indexy, tedy `for i in range(n) ...`, která navíc garantuje, že číselník a jmenovatel budou jiné prvky pole.

Úloha: Přidejte podmínku, že jmenovatel musí být kladný.

Úloha: Napište variantu této `nejblizsi_podil(a, seznam_citatele, seznam_jmenovatele)`, která bude hledat optimální podíl $q \approx m/n$, kde m bude náležet do seznamu číselníků a n do seznamu jmenovatelů.

Úloha: Napište funkci, která pro danou hodnotu odporu nalezne nejlepší dvojici rezistorů z řady E6, tedy s hodnotami odporu 1, 1.5, 2.2, 3.3, 4.7, 6.8, 10, 15, 22, 33, 47, 68, 100, ... $10^6\Omega$ - spojených sériově - spojených paralelně nebo sériově

Zde se učíme psát funkce, které pracují s poli. Ještě si o seřazování seznamů podle velikosti budeme povídat, ale již nyní si můžeme představit jeden snadno pochopitelný (ovšem, jak uvidíme, neefektivní) postup:

12.7 Buble-sort

Pole procházíme od začátku na konec a kdykoli potkáme špatně seřazenou dvojici sousedních prvků pole, prohodíme je. Snadno si rozmyslíme, že toto procházení pole musíme opakovat, přičemž, protože největší prvek takto probublá na konec seznamu, v dalším procházení můžeme vždy jedno testování ušetřit.

```
[ ]: def buble_sort_inplace(a):
    """procedura setřídí pole, které dostane jako parametr. Nic nevrací. Učíme se na ní.
    ↪ práci s poli, jinak je buble sort na nic."""

    n = len(a)

    for i in range(n-1,0,-1):
        for j in range(i):
            if a[j]>a[j+1]:
                a[j], a[j+1] = a[j+1], a[j]

k_setrideni = tabulka_sin.copy()
buble_sort_inplace(k_setrideni)
print(is_sorted(tabulka_sin), is_sorted(k_setrideni))
```

False True

```
[ ]: import matplotlib.pyplot as plt
import matplotlib.animation as animation
import numpy as np
from IPython.display import HTML

def add_frame(a, indices, color):
    """Přidej do globální proměnné frames snímek zachycující stav pole `a`.
    Vybrané sloupce `indices` obarví `color`."""

    # sloupečky volíme šedivé
    # jména barev viz: https://matplotlib.org/stable/gallery/color/named_colors.html
    c = np.full(len(a), 'darkgray') # numpy strings => musí být nejdelší řetězec

    # a požadované (dva) pak obarvíme
    c[indices] = color # numpy dovolí nastavit najednou všechny

    x = np.arange(len(a))
    frames.append(plt.bar(x, a, color=c))

def buble_sort_demo(a):
    """Bubble sort doplněný voláním funkce add_frame pro vytvoření animace."""

    n = len(a)
    for i in range(n-1,0,-1):
        swap_count = 0
```

(continues on next page)

```

for j in range(i):
    if a[j]>a[j+1]:
        add_frame(a, [j,j+1], ['red','green'])
        a[j], a[j+1] = a[j+1], a[j]
        swap_count += 1
        add_frame(a, [j,j+1], ['green','red'])
    else:
        add_frame(a, [j,j+1], ['green','red'])
if swap_count==0:
    break

# příprava na vytvoření animace má dva kroky
frames = [] # vytvořím prázdný seznam snímků
fig = plt.figure() # vytvořím prázdný obrázek

# vytvoříme pole náhodných čísel
k_setrideni = 12+60*np.random.random(12) #12+abs(60*np.cos(50*x+22))

# setřídíme
bubble_sort_demo(k_setrideni)

# zobrazíme animaci
fps = 2 # frames per second
ani = animation.ArtistAnimation(fig, frames, interval=1000//fps)
# ještě po sobě uklidíme, protože matplotlib nechal otevřený obrázek
plt.close()

# zobrazíme animaci v buňce Jupyteru
HTML(ani.to_html5_video())
<IPython.core.display.HTML object>

```

12.8 Řezy polí

Mimořádně užitečnou zkratkou pro práci s poli nebo maticemi jsou řezy. Nejprve si připomeňme funkci `range`. Používáme ji jako specifikaci hodnot, kterých má nabývat řídicí proměnná cyklu. Pokud váháte, zopakujte si, co bude výstupem cyklu

```

for i in range(1,10,2):
    print(i)

```

Seznamy i pole v Pythonu dovolují pracovat s více prvky pole najednou, pokud jako index pole uvedeme dvě nebo tři celočíselné výrazy oddělené dvojtečkou. Pokud uvedeme dvojtečku, ale nikoli hodnotu rozumí se hodnotou počátek (před první dvojtečkou) nebo konec pole (za první dvojtečkou) nebo jednička (za druhou dvojtečkou).

Nejprve si ukažme, co znamená takový řez pole, pokud se objeví jako část výrazu. Například:

```

[ ]: pole = np.linspace(0,9,10)
print('pole = ',pole)
print('pole[ 1: 3] = ',pole[1:3])

```

(continues on next page)

(pokračujte na předchozí stránce)

```
print('pole[ : 3] =',pole[:3])
print('pole[-3:-1] =',pole[-3:-1])
print('pole[-3: ] =',pole[-3:])
print('pole[ 1::2] =',pole[1::2])

pole = [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
pole[ 1: 3] = [1. 2.]
pole[ : 3] = [0. 1. 2.]
pole[-3:-1] = [7. 8.]
pole[-3: ] = [7. 8. 9.]
pole[ 1::2] = [1. 3. 5. 7. 9.]
```

V přiřazovacím příkaze mohou nastat dvě situace. - Do pole přiřazujeme pole stejného charakteru (počtu prvků) `pole[0:4] = pole[10:18:2]` - Do všech prvků vektoru přiřazujeme tutéž hodnotu `pole[0:4] = 0`

Užitečnost obsazení celého řezu pole jednou hodnotou potkáme již u Eratosthenova síta, kde proškrtávání násobků prvočísla p můžeme zapsat jedním přiřazením

```
if je_prvocislo[p]:
    je_prvocislo[2*p:n+1:p] = False
```

Takový řez pole se může objevit všude, kde jsme zvyklí pracovat s polem, tedy například jako argument funkcí.

```
[ ]: import numpy as np

def eratosthenes(n):
    """Vrací pole [False,False,True,True,False,True,...] délky n+1
       indikující, že mezi čísla 0..n jsou 2,3,5,... prvočísla"""
    je_prvocislo = np.full(n+1, True)
    je_prvocislo[0] = False
    je_prvocislo[1] = False

    p = 2
    while p*p <= n:
        if je_prvocislo[p]:
            je_prvocislo[2*p:n+1:p] = False
            p = p + 1

    return je_prvocislo

n_max = 35
vysledek = eratosthenes(n_max)
#print( vysledek )
print( "\nNalezená prvočísla:")
for i in range(n_max+1):
    if vysledek[i]:
        print(i, end=' ')

Nalezená prvočísla:
2 3 5 7 11 13 17 19 23 29 31
```

```
[ ]: phi = np.linspace(0,2*np.pi,129)
x = np.cos(phi)
```

(continues on next page)

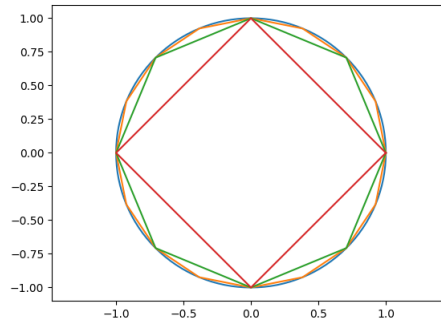
(pokračujte na předchozí stránce)

```

y = np.sin(phi)

plt.axis("equal")
plt.plot(x,y)
plt.plot(x[::8],y[::8])
plt.plot(x[::16],y[::16])
plt.plot(x[::32],y[::32])
plt.show()

```



Úloha: Rozmyslete si, co dá výraz `pole[::-1]+pole` když jsme již výše vytvořili `pole = np.linspace(0,9,10)`.

Pozor, v knihovně numpy jsou řezy chápány jako odkazy. Pokud tedy funkce argument typu `pole` mění, dojde ke změně původního `pole`.

Podobně se řezy polí mohou objevit na levé straně přiřazovacího příkazu. Zde je fakt, že měníme prvky daného `pole` zřejmý.

```

[ ]: zaporna_cisla = np.linspace(-10,-1,10)
print(zaporna_cisla)
vynuluj_zaporne(zaporna_cisla[::3])
print(zaporna_cisla)
zaporna_cisla[3] = -11
print(zaporna_cisla)

[-10.  -9.  -8.  -7.  -6.  -5.  -4.  -3.  -2.  -1.]
[ 0.  -9.  -8.  0.  -6.  -5.  0.  -3.  -2.  0.]
[-11. -11. -11.  0.  -6.  -5.  0.  -3.  -2.  0.]

```

12.9 Stěhujeme prvky pole

Součástí numerických algoritmů bývá i velmi nudná záležitost – stěhování polí a jejich částí. Protože to lze pokazit, musíme se o několika detailech zmínit.

Následující kód ilustruje - ztotožnění vektorů (ne nezbytně záměrné) - stěhování celých polí - stěhování částí polí - stěhování překrývajících se částí polí - špatně - dobře - pohodlně - varování: řezy polí (na rozdíl od seznamů) jsou odkazy, nikoli kopie

```

[ ]: def nove_pole(n=10):
    global pole
    pole = np.array(range(n))

```

(continues on next page)

(pokračujte na předchozí stránce)

```

print('ztotožení vektorů (ne nezbytně záměrné):')
nove_pole()

pole2=pole
pole2[1]=-1
print(pole,pole2)

print('stěhování celých polí:')
nove_pole()
pole2=pole.copy()
pole2[1]=-1
print(pole,pole2)

print('stěhování částí polí')
nove_pole()
pole[:3] = pole[-3:]
print(pole)

print('stěhování překrývajících se částí polí : špatně')
nove_pole()
for i in range(7):
    pole[i+3] = pole[i]
print(pole)

print('stěhování překrývajících se částí polí : dobře')
nove_pole()
for i in range(6,-1,-1):
    pole[i+3] = pole[i]
print(pole)

print('stěhování překrývajících se částí polí : pohodlně')
nove_pole()
pole[3:] = pole[:-3]
print(pole)

print('pozor, řezy polí jsou odkazy, nikoli kopie.')
nove_pole()
pole[0:5], pole[5:10] = pole[5:10], pole[0:5]
print(pole)

print('musíme tedy vytvořit kopii !')
nove_pole()
pole[0:5], pole[5:10] = pole[5:10], pole[0:5].copy()
print(pole)

```

```

ztotožení vektorů (ne nezbytně záměrné):
[ 0 -1  2  3  4  5  6  7  8  9] [ 0 -1  2  3  4  5  6  7  8  9]
stěhování celých polí:
[0 1 2 3 4 5 6 7 8 9] [ 0 -1  2  3  4  5  6  7  8  9]

```

(continues on next page)

(pokračujte na předchozí stránce)

```

stěhování částí polí
[7 8 9 3 4 5 6 7 8 9]
stěhování překrývajících se částí polí : špatně
[0 1 2 0 1 2 0 1 2 0]
stěhování překrývajících se částí polí : dobře
[0 1 2 0 1 2 3 4 5 6]
stěhování překrývajících se částí polí : pohodlně
[0 1 2 0 1 2 3 4 5 6]
pozor, řezy polí jsou odkazy, nikoli kopie.
[5 6 7 8 9 5 6 7 8 9]
musíme tedy vytvořit kopii !
[5 6 7 8 9 0 1 2 3 4]

```

```

[ ]: print('ztotožení vektorů (ne nezbytně záměrné):')
nove_pole()
print(pole)

pole2=pole

pole2[1]=-1
print(pole,pole2)

id(pole)==id(pole2)

```

```

ztotožení vektorů (ne nezbytně záměrné):
[0 1 2 3 4 5 6 7 8 9]
[ 0 -1  2  3  4  5  6  7  8  9] [ 0 -1  2  3  4  5  6  7  8  9]

True

```

```

[ ]: print('stěhování celých polí:')
nove_pole()
pole2=pole.copy()
pole2[1]=-1
print(pole,pole2)

```

```

stěhování celých polí:
[0 1 2 3 4 5 6 7 8 9] [ 0 -1  2  3  4  5  6  7  8  9]

```

```

[ ]: id(pole)==id(pole2)

False

```

```

[ ]: print('stěhování částí polí')
nove_pole()
print(pole)
pole[:3] = pole[-3:]
pole[-1] = 11

```

(continues on next page)

(pokračujte na předchozí stránce)

```
print(pole)
```

```
stěhování částí polí
[0 1 2 3 4 5 6 7 8 9]
[ 7 8 9 3 4 5 6 7 8 11]
```

```
[ ]: print('stěhování překrývajících se částí polí : špatně')
nove_pole()
for i in range(7):
    pole[i+3] = pole[i]
print(pole)
```

```
stěhování překrývajících se částí polí : špatně
[0 1 2 0 1 2 0 1 2 0]
```

```
[ ]: print('stěhování překrývajících se částí polí : dobře')
nove_pole()
for i in range(6,-1,-1):
    pole[i+3] = pole[i]
print(pole)
```

```
stěhování překrývajících se částí polí : dobře
[0 1 2 0 1 2 3 4 5 6]
```

```
[ ]: print('stěhování překrývajících se částí polí : pohodlně')
nove_pole()
pole[3:] = pole[:-3]
print(pole)
```

```
stěhování překrývajících se částí polí : pohodlně
[0 1 2 0 1 2 3 4 5 6]
```

```
[ ]: print('pozor, řezy polí jsou odkazy, nikoli kopie.')
nove_pole()
pole[0:5], pole[5:10] = pole[5:10], pole[0:5]
print(pole)
```

```
pozor, řezy polí jsou odkazy, nikoli kopie.
[5 6 7 8 9 0 1 2 3 4]
```

```
[ ]: print('musíme tedy vyrobit kopii !')
nove_pole()
pole[0:5], pole[5:10] = pole[5:10], pole[0:5].copy()
print(pole)
```

```
musíme tedy vyrobit kopii !
[5 6 7 8 9 0 1 2 3 4]
```

12.10 Polynomy jako pole koeficientů

Snadno představitelnou aplikací polí různé délky je seznam koeficientů polynomu. Ze své definice má polynom $\sum_{k=0}^n a_k x^k$ nějaký řád n a je tak určen koeficienty a_0, a_1, \dots, a_n , volíme tedy, že seznam $[-8, 0, 2]$ odpovídá polynomu $2x^2 - 8$. Zde se hodí, že Python má indexy polí počínající nulou.

Pro nás je zajímavé, že běžné operace, jako je sčítání a násobení dvou polynomů opět vytvoří polynom. (U dělení bychom potřebovali pracovat s racionálními funkcemi.)

Pojďme si pár takových funkcí napsat jako procvičení práce s poli. Praktický význam je dost omezený, protože polynomy určené koeficienty jsou pro vyšší řády velmi náchylné k zaokrouhlovacích chybám.

Komentář k vybraným funkcím:

- Ve funkci `hodnota_polynomu(p,x)` používáme Hornerovo schéma. Aby byl kód co nejpodobnější jiným jazykům, píšeme `p[n-1]` místo v Pythonu dovoleného `p[-1]`.
- Sčítání polynomů musí brát v úvahu, že sčítané polynomy mohou mít různý řád. Příklady níže ukazují vybrané tři způsoby, jak to zohlednit.
- Užitečné funkce pro práci s polynomy naleznete v části knihovny `numpy.polynomial`.

```
[ ]: def hodnota_polynomu(p,x):
    """Vrátí hodnotu p(x)"""
    n = len(p)
    f = p[n-1]
    for k in range(n-2,-1,-1):
        f = f*x+p[k]

    return f

def soucet_polynomu(p,q):
    lenp = len(p)
    lenq = len(q)

    r = np.zeros( max(lenp,lenq) )

    for k in range(lenp):
        r[k] = p[k]

    for k in range(lenq):
        r[k] += q[k]

    return r

def soucet_polynomu_v2(p,q):
    lenp = len(p)
    lenq = len(q)

    if lenp >= lenq:
```

(continues on next page)

(pokračujte na předchozí stránce)

```

    r = p.copy()
    for k in range(lenq):
        r[k] += q[k]
else:
    r = q.copy()
    for k in range(lenp):
        r[k] += p[k]

    return r

def soucet_polynomu_v3(p,q):
    lenp = len(p)
    lenq = len(q)

    r = np.zeros( max(lenp,lenq) )
    r[:lenp] = p
    r[:lenq] += q

    return r

def soucet_polynomu_v4(p,q):
    lenp = len(p)
    lenq = len(q)
    minpq = min(lenp,lenq)

    r = np.zeros( max(lenp,lenq) )

    r[:minpq] = p[:minpq]+q[:minpq]

    if lenp>lenq:
        r[lenq:] += p[lenq:]
    elif lenq>lenp:
        r[lenp:] += q[lenp:]

    return r

p1 = np.array([1,2,3,4])
p2 = np.array([5,-2,11])

for f in [soucet_polynomu,soucet_polynomu_v2, soucet_polynomu_v3, soucet_polynomu_v4]:
    p12 = f(p1,p2)
    for x0 in range(11):
        assert hodnota_polynomu(p12,x0) == hodnota_polynomu(p1,x0) + hodnota_polynomu(p2,
↪x0)

```

Úloha: Napište funkci vracující součin dvou polynomů

Samozřejmě, pole mohou mít víc indexů. My si probereme několik příkladů, kdy půjde o pole, které má význam matice. Ty mají dva indexy, jeden vybírá řádek, druhý sloupec. Z výkladu bude jasné, jak postupovat, kdybychom potřebovali pole s více indexy.

Seznámíme se jen s následujícími operacemi a to jen pro pole s jedním a dvěma indexy:

- vytvoření prázdného pole inicializovaného nulami
- přístup k prvku pole
- vytvoření kopie pole
- řezy pole
- aplikace funkcí a aritmetických operací na pole
 - se stejným počtem a rozsahem indexů
 - s jiným počtem indexů

13.1 numpy - vytvoření matice

Protože matice má obecně obdélníkový tvar, musíme zadat dvě dimenze. Ty je potřeba zadat jako seznam dvou celých čísel (pro znalé: místo seznamu by mělo jít o *tuple*, ale těm se v kurzu snažím vyhýbat).

```
matice_3x5 = np.zeros([3,5]) # 3 řádky, 5 sloupců
```

V základním kurzu budeme zjišťovat rozměry pole stejně jako u seznamu prostřednictvím funkce *len* (časem objevíte, že tvar pole je k dispozici v atributu *shape*, který je typu *tuple*).

Jak ale zjistit oba rozměry matice? Protože (zjednodušuji) se matice skládá z řádků, vrátí `len(matice_3x5)` hodnotu 3. Pokud si vybereme první řádek (index 0) a zjistíme jeho délku, tedy `len(matice_3x5[0])`, dostaneme počet sloupců: 5. Bereme první řádek, protože ten existuje i u matice dimenze 1 a matice mají všechny řádky stejně dlouhé.

```
[ ]: import numpy as np

p6 = np.zeros(6)
m3x5 = np.zeros( (3,5) )

print(p6)
print(type(p6), len(p6) )
print(m3x5)
print(type(m3x5), len(m3x5), len(m3x5[0]))
```

```
[0. 0. 0. 0. 0. 0.]
<class 'numpy.ndarray'> 6
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
<class 'numpy.ndarray'> 3 5
```

```
[ ]: print(len(m3x5[2]))
```

```
5
```

```
[ ]: pradku, psloupcu = m3x5.shape
print( pradku, psloupcu)
```

```
3 5
```

```
[ ]: np.linspace(0,1,11).shape
```

```
(11,)
```

Následující funkce ilustrují jednoduchý způsob jak zjistit, co je dané pole zač.

Atribut `shape` vrací *tuple*. Počet položek, tj. `len(a.shape)` říká, kolik má pole indexů. Pro názornost ale místo použití *shape* zjišťujeme rozměry vektoru a matice níže za pomoci funkce *len*. (Kdo chce pracovat s typem *tuple*, jistě uhadne, jak v těchto funkcích nahradit funkci *len* atributem *shape*.)

```
[ ]: import numpy as np

def is_ndarray(a):
    return type(a) == np.ndarray

def is_numpy_vector(v):
    if type(v) != np.ndarray:
        return False

    return len(v.shape) == 1

def is_numpy_matrix(m):
    if type(m) != np.ndarray:
        return False

    return len(m.shape) == 2
```

(continues on next page)

(pokračujte na předchozí stránce)

```
def vector_dim(v):
    return len(v)

def matrix_columns(m):
    return len(m[0])

def matrix_rows(m):
    return len(m)

[ is_ndarray(m3x5), is_numpy_vector(p6), is_numpy_vector(m3x5), is_numpy_matrix(m3x5),
↪vector_dim(p6) ]

[True, True, False, True, 6]
```

```
[ ]: [1,2,3].shape
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-57-2a3b0c02476c> in <cell line: 1>()
----> 1 [1,2,3].shape

AttributeError: 'list' object has no attribute 'shape'
```

Praktická rada: Povšimněte si, jak je psaná např. funkce `is_numpy_matrix`. Vzhledem k tomu, že na jejím druhém řádku nastane `return` pro všechny typy jiné, než `numpy.ndarray`, chytrá IDE, např. VScode, počínaje třetím řádkem 'vědí', jakého je `m` typu a nabídnou nám jeho atributy a metody při psaní kódu.

Cvičení: Napište znovu a bez nahlížení do kódu výše všech 6 funkcí `is_ndarray(a)`, `is_numpy_vector(v)`, `is_numpy_matrix(m)`, `vector_dim(v)`, `matrix_columns(m)`, `matrix_rows(m)`. (Není to zbytečná práce, procvičíte si na nejjednodušších úlohách pochopení organizace čísel do matic.)

```
[ ]: m42 = np.zeros([4,2])
```

```
print(m42)
```

```
[ matrix_rows(m42), matrix_columns(m42) ]
```

```
[[0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]]
```

```
[4, 2]
```

13.1.1 Prvky matice

Matice má dva indexy, přičemž obvykle první udává řádek a druhý sloupec. Stále platí, že první index je 0. Máme-li tedy matici

$$a = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix}$$

vypadá tato v našem programu takto

```
a[0,0]  a[0,1]  a[0,2]  a[0,3]
a[1,0]  a[1,1]  a[1,2]  a[1,3]
a[2,0]  a[2,1]  a[2,2]  a[2,3]
```

Pozn. Matice lze v principu vytvářet jako seznam seznamů, např. `a = [[1,0,0],[0,1,0],[0,0,1]]`, pak ale musíme psát `a[i][j]`. Protože se ovšem kopie takovýchto matic vytvářejí jinak než kopie vektorů, dáme přednost *numpy*, kde je méně komplikací.

Následující funkce ilustruje vytváření matice.

```
[ ]: # vytvoření jednotkové matice
# (samozřejmě, existuje np.eye, ale my se to teď učíme)

def matrix_1(dim):
    a = np.zeros((dim,dim))

    for i in range(dim):
        a[i,i] = 1.0

    return a

matrix_1(3)

array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Cvičení: Napište funkci

```
def is_diagonal_matrix(m):
    """Vrátí False, pokud je některý z nediagonálních prvků matice nenulový."""
    ...
```

Cvičení: Napište funkci

```
def diagonal_matrix(d):
    """Funkce vytvoří nulovou matici n x n, kde n je dimenze vektoru určeného argumentem_
    ↪ d a obsadí diagonálu prvky d"""
    ...
```

nahrazující `np.diag(seznam_diag_prvku)` stejným způsobem, jako `matrix_1(3)` napodobuje `np.eye(3)`.

Příklad použití: `diagonal_matrix([1,2,3])` musí vrátit

```
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

Cvičení: Otestujte obě funkce.

13.2 Vytvoření kopie

Protože přiřazení ztotožňuje odkazy, musíme často vytvářet kopie. Pro numpy stačí na vektory i matice použít metodu `copy`, tedy

```
[ ]: ma1 = np.zeros([2,2])
ma2 = ma1.copy()
ma3 = ma2

ma3[1,0] = 1000

print ma1, "\n"
print ma2, "\n"
print ma3
```

```
[[0. 0.]
 [0. 0.]]
```

```
[[ 0.  0.]
 [1000.  0.]]
```

```
[[ 0.  0.]
 [1000.  0.]]
```

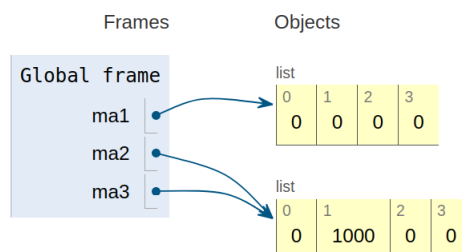
Pro úplnost, vizualizace provádění podobného kódu, který ale kvůli omezením služby pythontutor.com musí používat seznamy

```
ma1 = [0,0,0,0]
ma2 = ma1.copy()
ma3 = ma2

ma3[1] = 1000

print ma1, "\n"
print ma2, "\n"
print ma3
```

dá takovouto strukturu odkazů



I když by šlo matice vytvořit skrze typ `list`, výše použitý příklad s vektory je bvěrnější podobou toho, co se odhrává, když použijeme `numpy.ndarray`.

13.3 Řezy matic

Protože má matice dva indexy, můžeme řezy specifikovat několika způsoby. - `a[i, :]` představuje *i*-tý řádek matice - `a[:, i]` představuje *i*-tý sloupec matice - `a[:2, :2]` představuje levou horní 2x2 podmatici - atd

Opět můžeme do řezu maticí dosadit. - `a[řez1,řez2] = matice - a[řez1,řez2] = skalární_hodnota - a[řez1,řez2] = vektor_hodnota_kompatibilní_s_řez2`

Pokud určíme pevně jeden z indexů máme na levé straně vektor a platí pravidla, která jsme zmiňovali u řezů s jedním indexem.

U matic musí v prvním případě navzájem souhlasit počet řádků a sloupců. Věš třetím případě proběhne přiřazení jednoho řádku do obdélníkové matice se stejnou délkou řádku. V dokumentaci `numpy` se o druhé a třetí variantě píše jako o rozšíření typu (*broadcasting*), kdy se skalár povýší na vektor, nebo matici, vektor na matici atp.

```
[ ]: m3x4 = np.arange(12).reshape((3,4))
print(m3x4)
m3x4[:2, :2] = 11

print(m3x4)

[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[[11 11  2  3]
 [11 11  6  7]
 [ 8  9 10 11]]
```

13.4 Monte Carlo

Rozsáhlá část počítačové fyziky používá k určení nějaké hodnoty metody založené na náhodných číslech.

Náhodná čísla, která se používají většinou nepocházejí z fyzikálně náhodného procesu ale taková, jaká vzniknou deterministickým matematickým postupem (garantovaná nepředvídatelnost bývá důležitá spíše u šifrování, řekněme komunikace bankou).

Následující obrázek nám ilustruje, že zatímco siny celých čísel nejsou úplně náhodné, hodnoty, které vrací funkce `numpy.random.random` nemají žádnou zřetelnou pravidelnost.

```
[2]: import numpy as np

import matplotlib.pyplot as plt

# obsazení pole hodnotami

tabulka_int = np.array(range(1000))
tabulka_sin = (np.sin(tabulka_int)-1)/2
tabulka_rnd = np.random.random(len(tabulka_int))

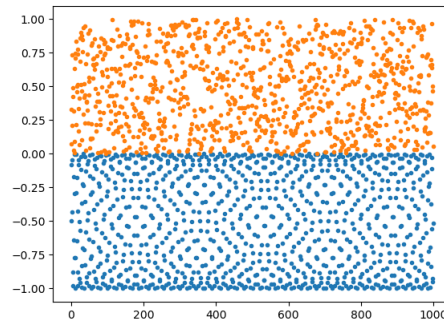
plt.plot(tabulka_int, tabulka_sin, '.')
```

(continues on next page)

(pokračujte na předchozí stránce)

```
plt.plot(tabulka_int, tabulka_rnd, '.')
```

```
plt.show()
```



Existuje víc možností, jak může náhodný vstup dát určitý výsledek. My si ukážeme tu, kdy existenci výsledku garantuje *zákon velkých čísel*.

Klíčový je zde vzorec

$$\int_0^1 f(x)dx = \lim_{n \rightarrow \infty} \frac{1}{n} \sum_{i=1}^n f(x_i)$$

v němž se pro velká n počítá průměrná hodnota $f(x_i)$ pro n náhodně generovaných bodů $x_i \in \langle 0, 1 \rangle$ pro 'rozumnou' funkci f . Lze na něj nahlížet dvěma způsoby:

- máme správný způsob, jak získat správnou posloupnost náhodných čísel x_i a pak vzorec představuje návod, jak spočítat určitý integrál.
- testujeme, zda daná posloupnost čísel x_i splňuje požadavky na 'dobře' náhodná čísla.

My budeme uvažovat první variantu a předpokládat, že správné náhodné hodnoty nám v programu poskytuje generátor náhodných čísel, tedy tento vztah budeme chápat jako vlastnost kvalitního generátoru náhodných čísel.

Všimněme si, že například, když položíme $f(x) = x$, požadujeme vlastně, aby střední hodnota $\bar{x}_i = 1/2$. Tedy nestačí aby čísla byla náhodná, ale musejí být tak náhodná, aby se s jejich rostoucím počtem jejich průměr blížil 0.5. To ovšem splňuje i posloupnost $a_k = \sin k$. Další funkce otestují další kvality generátoru a například $f(x) = x^2$ pro `tabulka_sin` již dá výrazně odlišnou hodnotu od 1/3. Navíc jsou ještě další podmínky, které generátory náhodných čísel musejí splňovat a "výroba" x_i tak není jednoduchá záležitost.

Jedno náhodné číslo z intervalu $\langle 0, 1 \rangle$ získáme zavoláním funkce `random.random()`. (Hádejte, jakou knihovnu musíte importovat!) Podobně jedno náhodné číslo vrátí zavolání `numpy.random.random()`.

Protože ale u zákona velkých čísel potřebujeme hodně náhodných hodnot, použijeme v následujícím příkladě příkaz

```
x = np.random.random(n)
```

čímž získáme najednou stovku nebo také milióny náhodných čísel podle toho, jaká je hodnota n . Z těch potom najednou spočteme všechna $f(x_i)$ a pomocí `numpy.sum(promenna_typu_pole)` i průměr.

```
[ ]: (tabulka_rnd**2).mean()
```

```
0.3314167417864176
```

```
[ ]: # Program demonstruje Monte Carlo kvadraturu
```

```
import numpy as np
```

(continues on next page)

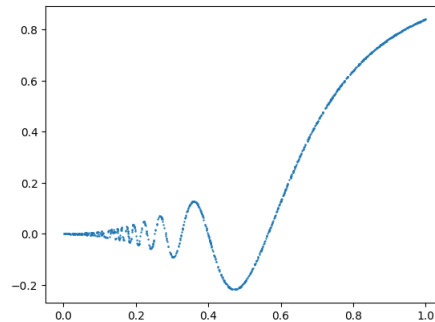
```
import matplotlib.pyplot as plt

for k in range(1,9):
    n = 10**k
    x = np.random.random(n)
    # počítáme integrál x^2 sin(1/x^2) pro x od 0 do 1
    x2 = x**2
    f_x = x2 * np.sin(1/x2)

    exact = 0.21884017626889995
    approx = np.sum(f_x)/n
    err = abs(approx-exact)
    print( f"10^{k:1} {approx:21} {err:7.2}" )
    if k==3:
        plt.scatter(x,f_x,s=0.5)

plt.show()
```

```
10^1  0.10701429276444441  0.11
10^2  0.1893845249078319  0.029
10^3  0.22725216225545983  0.0084
10^4  0.2097998941497601  0.009
10^5  0.2182740510866274  0.00057
10^6  0.218852960970784  1.3e-05
10^7  0.21870513205829425  0.00014
10^8  0.21882691020430017  1.3e-05
```



Cvičení: Přepište program tak, abyste nepoužívali numpy. Náhodná čísla získávejte prostřednictvím `random.random()`, trigonometrické funkce jako `math.sin`.

Zjistěte, kolikrát bude program pomalejší. (Rady: Zmenšete maximální počet náhodných čísel z 10^7 na 10000 a odstraňte volání `print` a `plt.scatter`. Učiníte-li z programu proceduru, lze pak snadno použít `%timeit`.)

13.5 Stručně o modulech

Viděli jsme, že i pro nejjednodušší funkce musíme sáhnout do některé knihovny. Pokud vhodnou funkci nenajdeme, musíme si ji napsat. Pokud ji pak chceme použít v několika různých programech, používali jsme doposud zkopírování kódu příslušné funkce do nového programu. Máme ale možnost vytvořit naši vlastní knihovnu a tu importovat do všech programů stejně jako ostatní knihovny. To si hned ukážeme, ale je potřeba varovat, že

- Python rozlišuje moduly a balíky modulů. Ty druhé potkáte až v nějakém v kurzu jazyka Python.
- Moduly mohou být uloženy na různých místech souborového systému. Pro nás bude důležité, že příkaz `import` začíná hledat modul ve stejném adresáři, kde je spouštěný program.

Modul je soubor s příponou `.py` obsahující zdrojový kód funkcí, příkazů pro inicializaci proměnných atp.

Jak takový modul vypadá si ukážeme na následujícím příkladě. Jde o modul, který poskytuje jedinou funkci pro výpočet faktoriálu, ta však vyžaduje, aby se přepočítala tabulka hodnot, kterou funkce interně používá. Je zvykem používat podtržítka jako signál uživateli modulu, že daná proměnná nebo funkce je určena jen pro interní použití.

Tato přednáška používá sešity prostředí Jupyter a tak musíme nejprve v aktuálním adresáři vytvořit soubor `myfactorial.py` obsahující kód modulu. Mohli bychom jej stáhnout z webu (`!wget https://server/dir/myfactorial.py`), ale aby jeho obsah byl zřejmý, vytvoříme jej následujícími příkazy Pythonu:

```
[1]: file_contents = """\
# Toto je modul myfactorial.py

def float_fact(n):
    "Vrací přepočítanou hodnotu n! typu float"
    assert n>=0
    return __fact_table[n]

def __init_fact_table():
    "Privátní funkce pro inicializaci tabulky"
    global __fact_table

    print('inicializuji tabulku hodnot faktoriálů')

    __fact_table = [1.0]
    kf = 1.0
    for k in range(1,170):
        kf *= k
        __fact_table.append( kf )

# Zavolej inicializaci tabulky faktoriálů
__init_fact_table()
"""

with open("myfactorial.py","w") as py:
    print(file_contents, file=py)
```

Nejprve si ověříme, že soubor by vytvořen a na jeho začátku je očekávaný komentář.

```
[2]: ! head -1 myfactorial.py
# Toto je modul myfactorial.py
```

Jako obvykle použijeme příkaz `import` a název souboru bez přípony `.py`.

Kód našeho modulu se od obvyklé situace liší tím, že vypisuje informaci o probíhající inicializaci. To je čistě ze vzdělávacích důvodů, abychom viděli, že příkazy obsažené v modulu se spouští při jeho prvním importování. Podruhé již ne.

```
[3]: import myfactorial
```

```
inicializují tabulku hodnot faktoriálů
```

```
[4]: import myfactorial
```

Nyní můžeme funkci exportovanou z modulu použít. (Vyzkoušejte například, že je pro vyšší hodnoty argumentu rychlejší než faktoriál z knihovny `math`.)

```
[5]: print( myfactorial.float_fact(69) )
```

```
1.711224524281413e+98
```

Na závěr uvedme, že psaní modulů lze doporučit z více důvodů

- sdílení stejného kódu více programy
- možnost kód jednou odladit a pak již neměnit
- vede ke kratším, přehlednějším programům a tedy zmenšuje riziko chyby

Seznámili jsme se již s nezákladnějšími operacemi s řetězci.

- Vytvoření řetězce explicitním zápisem (literálem) `s = 'retezec v apostrofech'` a `s = "retezec v uvozovkach"`
- Vytvoření řetězce konverzí `s = str(math.pi)`
- Vytvoření řetězce obsahujícího znak s daným ASCII/Unicode kódem `chr(65)`

Také jsme potkali speciality

- Speciální znaky `"a\tb\n1\t2"`
- Formátované řetězce `f'Pro n={2**i} je vysledek {spoctiTo(i)}'`
- Víceřádkové řetězce

```
s = '''\  
<html>  
<body>  
<h1>Hola!</h1>  
</body>  
</html>  
'''
```

Dále jsme viděli elementární operace s řetězci

- Spojování `'a'+'b'` (dá `'ab'`)
- Množení `'a'*5` (dá `'aaaaa'`)

14.1 Řetězce vs. pole znaků

Je přirozené nahlížet na řetězce jako na pole znaků. Můžeme tedy použít složené závorky a z řetězce přečíst znak na dané pozici. První znak řetězce má index 0, druhý 1, atd.

```
[ ]: s = 'ABCD'

for i in range(len(s)):
    print(f's[{i}] == "{s[i]}"')

s[0] == "A"
s[1] == "B"
s[2] == "C"
s[3] == "D"
```

Abychom si procvičili práci s indexy u řetězců, napíšeme si nejprve funkci převádějící řetězec na celé číslo. Samozřejmě, činíme tak jen z výukových důvodů, protože takovou službu již poskytuje konverze `int(s)`, kde `s` je řetězec obsahující dekadický zápis čísla.

```
[ ]: def atoi(s):
    'převéde řetězec na celé číslo'
    i = 0 # index cifry
    n = 0 # postupně se kumulující výsledek

    for i in range(len(s)):
        assert '0' <= s[i] <= '9', "Neplatná cifra"
        n = n*10 + ord(s[i])-ord('0')

    return n

def atoi2(s):
    'převéde řetězec na celé číslo, tato varianta funkce nepoužívá indexy'
    i = 0 # index cifry
    n = 0 # postupně se kumulující výsledek

    for c in s:
        assert '0' <= c <= '9', "Neplatná cifra"
        n = n*10 + ord(c)-ord('0')

    return n

print( atoi('123456') )

123456
```

Stejně jako u polí můžeme i u řetězců použít řezy, tedy např.

```
[ ]: s = 'ABCD'
print(f' {s} = {s} ')
print(f' s[1:3] = {s[1:3]} ')
print(f' s[1:] = {s[1:]} ')
print(f' s[:-1] = {s[:-1]} ')

```

```
s = 'ABCD'
s[1:3] = 'BC'
s[1:] = 'BCD'
s[:-1] = 'ABC'
```

Zajímavou pomůckou pro zapamatování je, že u řetězců (protože operace + znamená spojování) platí `s[:n] + s[n:] == s` pro libovolnou (i zápornou) hodnotu `n`.

To vše ale platí, pokud informaci z řetězce čteme. Není ale dovoleno modifikovat konkrétní znak. Řetězce (typ `str`) představuje totiž v logice jazyka Python *immutable* typ. Nebude nám to asi vadit, protože většinou budeme po možnosti modifikovat jednotlivé znaky řetězce toužit v situaci, kdy je cíle možno dosáhnout zavoláním některé z metod typu `str`.

Nejprve si zběžně ujasněme termín *metoda*. Souvisí s principy tzv. objektového přístupu k programování, my se zatím učíme přístup procedurální a tak lze zkráceně říci, že jde o funkci, jejíž první argument se místo do závorek za název (identifikátor) funkce nebo procedury připojuje před její název tečkou. Zatímco

- `len('111939')` je příkladem volání funkce, kde řetězec je její argument a vrací celé číslo 6, tedy počet znaků, který tvoří daný řetězec
- `'111939'.count('9')` je příkladem volání metody. První argument se nachází před tečkou a volání vrátí hodnotu 2, protože devítka je v řetězci 2×.

Pokud by se nám nelíbilo, že `count` musíme psát za tečku, můžeme defiovat stejnojmennou funkci dvou argumentů

```
def count(x, a):
    return x.count(a)
```

a poté psát např. `print(count('111223', '2'))`.

Použití metod přináší i jisté pohodlí. Pokud dokáže editor kódu uhodnout typ identifikátoru (nebo výrazu v závorkách) před tečkou, může nabídnout po napsání tečky seznam metod. Pokud jsou rozumně pojmenované, můžeme bez hledání v dokumentaci dosáhnout potřebného efektu.

14.2 Vybrané metody typu `str`.

Operací, které jsou takto dostupné jako metody typu `str` hodně. Podrobně vše najdete v [dokumentaci](#), zde jen stručné shrnutí. Tyto funkce občas uvidíte použité v příkladech, ale je jasné, že jejich znalost nebude u zkoušky potřeba.

- Mnoho funkcí se zabývá modifikací velkých a malých písmen v řetězci, např. `lower`, `upper`, `capitalize`, `swapcase`. Příklad: `'xxxii'.upper() == 'XXXXII'`.
- Další funkce testují, zda řetězec splňuje nějaká kritéria, např. `isalpha`, `isdecimal`, `islower`.
- Odstranění mezer na začátku a/nebo konci řetězce zajistí metody `strip`, `lstrip`, `rstrip`. Příklad: `' 1 2 3 '.strip() == '1 2 3'`.
- Metody `find` a `replace` zajistí elementární vyhledávání a nahrazování v řetězci.

```
print( '1\t2\t3'.replace('\t','...') )
```

vytiskne

```
1...2...3
```

Většinou však použijete složitější variantu používajíc tzv. **regulární výrazy**, která dovolí pohodlně provést i takovou operaci, jako nalezení letopočtů zapsných v římskými číslicemi v textu a jejich nahrazení dekadickou hodnotou. Jde o rozsáhlé téma, které sice nespádá do "základů programování", nicméně je natolik důležité, že o existenci musíte vědět. Proto jej zmíníme v kapitole "Co jsme nestihli".

- Podobné operaci hledání je rozdělení řetězce na seznam řetězců. Opět můžeme vzít jako oddělovač jediný znak (např. čárku), pak nám stačí metoda `split` typu `str`. Důležitý příklad souvisí s interpretací řádku čísel načtených ze souboru.

Příklad Následující kód nejprve získá data o hranicích souhvězdí z webu a následně je načte. Protože bez podpory dalších knihoven mámě v Pythonu načtená data v podobě řetězce musíme tento převést na číselné údaje.

Klíčová je zde funkce `str.split(sep)`, která rozdělí řetězec na místech určených hodnotou argumentu `sep`. V našem případě je to jednak mezera a poté dvojtečka (viz formát dat).

Další užitečnou vlastností jazyka, kterou několikrát využijeme je *unpacking*. Již víme, že ten v nejjednodušší podobě znamená přiřazení prvků pole do tolika proměnných, kolik je délka pole, např. `a, b, c = [1, 2, 3]`.

Pozn. Pokud program spouštíte na počítači, kde nemáte instalovaný program `wget`, soubor `bound_edges_18.txt` můžete do běžného adresáře uložit webovým prohlížečem.

```
[ ]: # Hranice souhvězdí ze stránky https://pbarbier.com/constellations/boundaries.html
! wget https://pbarbier.com/constellations/bound_edges_18.txt
```

```
--2023-12-11 10:29:31-- https://pbarbier.com/constellations/bound_edges_18.txt
Resolving pbarbier.com (pbarbier.com)... 104.244.120.9
Connecting to pbarbier.com (pbarbier.com)|104.244.120.9|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 89114 (87K) [text/plain]
Saving to: 'bound_edges_18.txt'

bound_edges_18.txt 100%[=====] 87.03K 487KB/s in 0.2s

2023-12-11 10:29:32 (487 KB/s) - 'bound_edges_18.txt' saved [89114/89114]
```

```
[ ]: # Nakreslí lomené "čáry" vyznačující dané souhvězdí na obloze
# data v souboru mají formát/význam
#
# 587:586 P+ 11:31:00 +11:00:00 11:52:00 +11:00:00 VIR LEO
# KE1:KE2 ED RA1 DE1 RA2 DE2 CO1 CO2
#
# KE1,KE2 ... Vertex keys
# ED ... Edge type and direction -- [M]eridian or [P]arallel
# RA1, DE1 ... Right ascension and declination (B1875) of 1st vertex
# RA2, DE2 ... Right ascension and declination (B1875) of 2nd vertex
# CO1, CO2 ... Constellations delimited by this edge

import matplotlib.pyplot as plt

def hex2dec(xms):
    "Převádí úhel v šedesátkové (hexagesimální) soustavě x:minuty:vteřiny na reálné číslo"
    x, m, s = xms.split(':')
    return int(x) + int(m)/60 + int(s)/3600

def plot_constellation_bdry(name):
```

(continues on next page)

(pokračujte na předchozí stránce)

```
'Nakreslí lomenou "čáru" vyznačující dané souhvězdí na obloze'

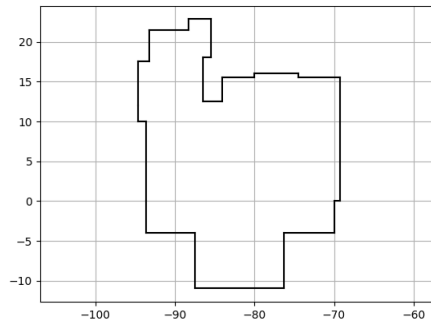
hrs2deg = -360/24 # konverze hodin na stupně u rektascense

with open('bound_edges_18.txt') as file:
    for line in file:
        ke12, ed, ra1, de1, ra2, de2, co1, co2 = line.split(' ')

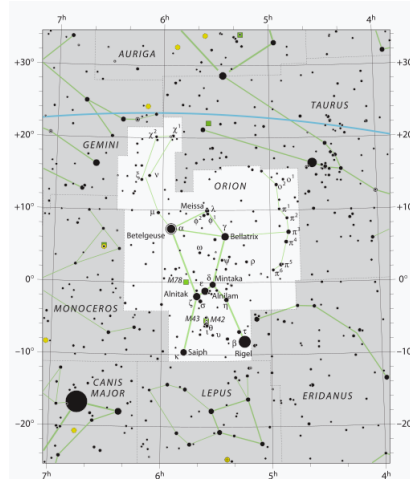
        if co1==name or co2==name:
            ra1 = hex2dec(ra1) * hrs2deg
            de1 = hex2dec(de1)
            ra2 = hex2dec(ra2) * hrs2deg
            de2 = hex2dec(de2)

            plt.plot([ra1,ra2],[de1,de2],'k')

plot_constellation_bdry('ORI')
plt.axis("equal")
plt.grid()
plt.show()
```



Srovnáním našeho obrázku s tím na [wikipedii](#) vidíme nedostatky našeho zobrazení sféry na rovinu (naše poledníky jsou vertikály). Orientace a škálování vodorovné osy na tomto obrázku pak vyjasní, proč jsem museli použít konstantu $\text{hrs2deg} = -360/24$.



14.2.1 Regulární výrazy

Jde o jazyk určený k popisu řetězců. Například má-li být identifikátor neprázdná posloupnost písmen, číslic a podtržítka nezačínající číslicí, použijeme regulární výraz

```
[a-zA-Z_][a-zA-Z0-9_]*
```

Dnes se běžně používá v textových editorech ke specifikaci hledaného řetězce a pokud v programu potřebujeme upravovat řetězce (například při načítání dat v nepohodlném tvaru) je regulární výraz ve většině případů správným nástrojem jaký použít.

Pravidel pro psaní regulárních výrazů je hodně, ty nejdůležitější jsou - `a` znamená znak `a` - `[abc]` znamená znak `a` nebo `b` nebo `c` - `[a-d]` znamená znak `a` nebo `b` nebo `c` nebo `d` - `[^xy]` znamená libovolný znak kromě `x` a `y` - `ab*` znamená `a` nebo `ab` nebo `abb` nebo `abbb` atd. - `ab+` znamená `ab` nebo `abb` nebo `abbb` atd. - `ab?` znamená `a` nebo `ab` - `a(bb)?c` znamená `ac` nebo `abbc`

Závorky navíc označují skupiny, jejichž hodnotu můžeme dále použít.

Regulární výrazy jsou dnes součástí nástrojů pro vyhledávání v editorech a jsou také vhodným nástrojem, pokud ve vašem programu potřebujete modifikovat řetězce nebo textové soubory. Následující příklad se pokouší řešit převod desetinné čárky v číslech na desetinnou tečku. Funkce `re.sub` (*sub* jako *substitute*) je zde použita se třemi parametry:

- první představuje vyhledávaný text popsaný regulárním výrazem
- druhý pak text, který jej má nahradit. Speciální výrazy `\1` a `\2` se nahradí tím, co se napasovalo na první resp. druhou skupinu uvedenou v závorkách v regulárním výrazu.
- třetí argument představuje prohledávaný text.

```
[1]: import re

vstup = """\
1,23, 5,23, 7,02
16,365, 5,96, 11,32
"""

vystup = re.sub(r"([0-9]+),([0-9]+)", r"\1.\2", vstup)
```

(continues on next page)

(pokračujte na předchozí stránce)

```
print(vystup)
```

```
1.23, 5.23, 7.02  
16.365, 5.96, 11.32
```

Poznámka: Na webu [regexr.com](https://www.regexr.com) si můžete vyzkoušet svoje nebo prohlédnout cizí nápady na tvar regulárního výrazu pro daný problém.

Cvičení: Zkuste vzít prvních pár řádek tabulky <https://github.com/dbwebb-se/vlinux/blob/master/example/grep/presidents.txt> a s použitím výrazu `([A-Z])[a-z]+ (([A-Z])[a-z]+)?([A-Z][a-z]+)` upravte program výše, aby převáděl jména na iniciály: George Washington, 1789-1797 --> G. Washington, 1789-1797

Pole a výpočty lineární algebry

Vektory a matice mají ve fyzice nepřehledné množství užití. Proto digitální počítače a jazyky určené k jejich programování braly v úvahu potřeby takových výpočtů, jako je například řešení soustav rovnic.

Zde si na několik příkladech ilustrujeme elementární funkce z lineární algebry, jak se v nich pracuje s poli a jak používat funkce z knihoven.

Jako opakování ověříme, že pro tři konkrétní vektory platí

$$\vec{a} \times (\vec{b} \times \vec{c}) = \vec{b} (\vec{a} \cdot \vec{c}) - \vec{c} (\vec{a} \cdot \vec{b})$$

Napíšeme si funkci pro skalární součin definovaný vztahem

$$\vec{a} \cdot \vec{b} := \sum_i a_i b_i$$

a také funkci pro vektorový součin.

```
[4]: import numpy as np

def scalar_product(a,b):
    "Počítá skalární součin dvou vektorů"
    dim = len(a)

    s = 0.0
    for i in range(dim):
        s = s + a[i]*b[i]

    return s

def vector_product(a,b):
    "Počítá vektorový součin dvou vektorů"
    assert len(a)==len(b)==3

    c = np.zeros(3)
```

(continues on next page)

(pokračujte na předchozí stránce)

```

c[0] = a[1] * b[2] - a[2] * b[1]
c[1] = a[2] * b[0] - a[0] * b[2]
c[2] = a[0] * b[1] - a[1] * b[0]

return c

a = np.array([ 1,2,0 ])
b = np.array([ 2,0,1 ])
c = np.array([ 0,1,2 ])

print( vector_product(a, vector_product(b,c)) )
print( b*scalar_product(a,c) - c*scalar_product(a,b) )

[ 4. -2. -2.]
[ 4. -2. -2.]

```

Problém: Povšimněte si, že obě funkce vracejí správnou hodnotu i tehdy, jsou-li jejich argumenty seznamy typu `list` nikoli pole `numpy.ndarray`. Přesto si rozmyslete, na které operaci by program ohlásil chybu, kdybychom proměnné `a,b,c` inicializovali jako typ `list`, tedy `a = [1,2,3]` atd. Vyzkoušejte, zda Váš odhad vyšel.

Co by se změnilo, kdybychom na čtvrtém řádku měli `s = 0`, tedy celé číslo místo čísla reálného?

Samozřejmě, že v knihovně `numpy` existují funkce, které počítají totéž. Jmenují se `numpy.dot` a `numpy.cross`. Zejména ta první je velmi užitečná a tak ji zanedlouho o něco podrobněji zmíníme znovu.

Cvičení: Napište funkci, která na seznamu vektorů provede [Gramm-Schmidtovu ortogonalizaci](#). (Tento postup akceptuje první vektor v seznamu a od každého dalšího odečítá vhodné násobky těch předchozích, již ortogonálních tak, aby byl každý vektor ortogonální k těm předchozím.)

- Rozmyslete si, jak v programu uložíte počáteční a výsledný seznam vektorů.
- Rozmyslete si, zda chcete mít funkci, která vrací seznam nebo proceduru, která modifikuje odkazem předávaný argument.

15.1 2D a 3D vektory

I u těchto krátkých vektorů sice `numpy` nepřináší zrychlení výpočtů, ale z hlediska psaní kódu je práce s nimi je pohodlnější, protože se hodí, že vektory můžeme sčítat a násobit. Jako příklad si uvedeme klasickou demonstraci vývoje vertikální složky rychlosti ozářením volně padajícího objektu stroboskopem:

```

[21]: # Program vykreslí polohy hmotného bodu
      # podléhajícímu volnému pádu
      # a to v ekvidistantních časových okamžicích

import numpy as np
import matplotlib.pyplot as plt

x0 = np.array([0, 0])
v0 = np.array([1, 9.81])
g = np.array([0, -9.81])

seznam_casu = np.linspace(0, 2, 30)

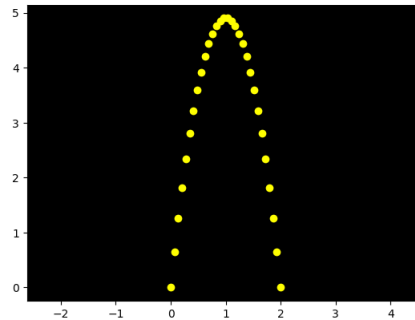
```

(continues on next page)

(pokračujte na předchozí stránce)

```
xy = [ x0 + v0 * t + 0.5 * g * t**2 for t in seznam_casu ]
xy = np.array(xy)

plt.rcParams['axes.facecolor'] = 'black'
plt.axis('equal')
plt.plot( xy[:,0], xy[:,1], 'o', color='yellow')
plt.show()
```

**Diskuse:**

- používáme 2D vektory
- chceme vyrobit seznam takových vektorů představující soubor poloh volně padajícího bodu
- bohužel pak nemůžeme napsat $v_0 * seznam_casu$, protože definovány jsou jen násobení stejně dlouhých vektorů nebo vektoru a skaláru
- proto je zkonstruujeme jako $xy = [f(t) \text{ for } t \text{ in seznam_casu}]$
- funkce `plot` vyžaduje zvlášť x-ové a zvlášť y-ové složky polí, proto používáme řezy, např. `xy[:,0]`

Cvičení: V kódu se používá seznamový výraz (*list comprehension*)

```
xy = [ x0 + v0 * t + 0.5 * g * t**2 for t in seznam_casu ]
xy = np.array(xy)
```

Přepište tuto pohodlnou konstrukci do podoby cyklu, abyste si vyzkoušeli, jaké to je programovat v běžných jazycích, které tuto *vychytávku* nemají.

- vytvořte pole `xy` naplněné hodnotami \vec{x}_0 (použijte `np.full((len(seznam_casu),2), x0)`), tedy pole `xy` bude obsahovat tolik 2D vektorů, kolik je počet prvků proměnné `seznam_casu`
- pro všechny časy `t` z proměnné `seznam_casu`
 - k příslušnému prvku pole `xy` přičtěte $\vec{v}_0 t + \frac{1}{2} \vec{g} t^2$

V uvedeném návodu je drobná zrada: - pokud budete přičítat příkazem `xy[i] = xy[i] + v0 * t + ...` program poběží, ale obrázek nebude dobře

- pokud budete přičítat příkazem

```
xy[i] += v0 * t + ...
```

program ohlásí chybu `Cannot cast ufunc 'add' output ...`

Rozmyslete si, v čem chyba spočívá a opravte ji.

15.2 Matice a vektory

Jak víme, klíčovou operací je násobení matice vektorem. Knihovna numpy to umí skrze funkci `numpy.dot`, ale my se to musíme naučit sami:

```
[33]: import numpy as np

def matrix_dot_vector(a, x):
    dim_x = len(x)
    dim_y = len(a)

    assert dim_x == len(a[0])

    y = np.zeros(dim_y)

    for i in range(dim_y):
        s = 0.0
        for j in range(dim_x):
            s = s + a[i,j]*x[j]
        y[i] = s

    return y

matice = np.array( [
    [1,2,3,4],
    [1,0,0,1]
])

matrix_dot_vector(matice, [-1,3,-3,1])
```

```
[33]: array([0., 0.])
```

Již víme, že knihovna numpy naučí operátory `+`, `*` atp. pracovat s poli. Základní podoby jsou

- `pole * 4`, kdy se každý prvek pole vynásobí čtyřmi.
- `pole * pole2`, kdy obě pole jsou stejně dlouhá a násobení probíhá prvek po prvku.

Proto výraz `matice*matice2` předpokládá, že obě matice mají stejný tvar a vynásobí je prvek po prvku, tedy podle vzorečku $c_{ij} = a_{ij}b_{ij}$.

Při práci s poli s více indexy, kde matice jsou nejjednodušším příkladem, je nutné přinejménším vědět, že je pokud `*` vložíte mezi matici a vektor, nemusí program nahlásit chybu! Knihovna numpy to pochopí tak, že operaci `*` použije na každý řádek. Pozor, nejde o násobení $\mathbb{A} \cdot \vec{x}$ z lineární algebry a tedy dostanete stejný výsledek pro obě pořadí: `matice*pole` i `pole*matice`.

```
[38]: print( 'matice = \n' , matice , "\n" )
print( 'matice*2 =\n', matice * 2 , "\n" )
print( 'matice * [1,2,3,4] =\n', matice * [1,2,3,4], "\n" )
print( '[1,2,3,4] * matice =\n', [1,2,3,4] * matice , "\n" )
print( 'matice * matice =\n', matice * matice , "\n" )

matice =
[[1 2 3 4]
 [1 0 0 1]]
```

(continues on next page)

(pokračujte na předchozí stránce)

```
matice*2 =
[[2 4 6 8]
 [2 0 0 2]]

matice * [1,2,3,4] =
[[ 1  4  9 16]
 [ 1  0  0  4]]

[1,2,3,4] * matice =
[[ 1  4  9 16]
 [ 1  0  0  4]]

matice * matice =
[[ 1  4  9 16]
 [ 1  0  0  1]]
```

Knihovna `numpy` obsahuje velké množství funkcí, které realizují operace z lineární algebry. Z hlediska našeho předmětu - Nehrozí, že bychom vás u zkoušky trápili dotazem: Jak se jmenuje funkce knihovny `numpy`, která vypočte skalární součin dvou vektorů - Potřebujeme v přednášce praktickou ukázkou toho, co to je metoda objektu. Následující řádky se tedy hodí, kdybychom se vás zeptali: Uveďte nějaký příklad proměnné o vhodném typu, kde lze dát příklad volání její metody, a význam takové operace.

Zejména je důležitá funkce `numpy.dot`. Realizuje operace jako jsou - skalární součin vektorů $s = x_i y_i$ - součin matice a vektoru $y_i = \sum_j a_{ij} x_j$ - součin matice a matice $c_{ik} = \sum_j a_{ij} b_{jk}$

Máme tři možnosti

- funkce `dot` z knihovny `numpy`, která má dva argumenty
 - `s = np.dot(vec_x, vec_y)`
 - `vec_y = np.dot(mat_A, vec_x)`
 - `mat_C = np.dot(mat_A, mat_B)`
- použít metodu `dot` prvního operandu, která jako argument dostane druhý operand
 - `s = vec_x.dot(vec_y)`
 - `vec_y = mat_A.dot(vec_x)`
 - `mat_C = mat_A.dot(mat_B)`
- použít operátor `@` (ve významu *dot*)
 - `s = vec_x @ vec_y`
 - `vec_y = mat_A @ vec_x`
 - `mat_C = mat_A @ mat_B`

Cvičení: Napište kód testující devět výše uvedených operací.

15.3 Gaussova-Jordanova eliminace

Řešení soustavy lineárních rovnic

$$A\vec{x} = \vec{b}$$

si ukážeme za použití Gaussovy-Jordanovy eliminace. Ta pomocí sady ekvivalentních úprav redukuje soustavu rovnic do tvaru, kdy výsledná matice soustavy A je jednotková matice a tedy výsledná pravá strana je hledaným řešením. Jak je obvyklé, spojíme A a \vec{b} do jedné obdélníkové matice A' .

Ekvivalentní úpravy jsou tři: - přehazování rovnic, tedy řádků A' , - násobení rovnic nenulovým činitelem, - odečítání násobku jedné rovnice od druhé.

Tato metoda je sice pomalejší, než Gaussova metoda se zpětnou substitucí, ale pro naše potřeby dostatečně dobře ilustruje operace, které s maticemi při řešení soustav rovnic provádíme.

Nejprve si ukažme tyto kroky postupně s použitím řezů matic a výhod interaktivního režimu.

```
[ ]: import numpy as np
# Nastavíme srozumitelný formát výpisu čísel
np.set_printoptions(floatmode="fixed",precision=4)

# Společná matice soustavy a vektoru pravé strany
matice = np.array( [
    [3, 2, 1, 1, 3],
    [0, 2, 3, -1, -4],
    [1, 9, 0, 4, -5],
    [2, 3, 1, 1.0, 0] ] )

# Výpis
matice

array([[ 3.0000,  2.0000,  1.0000,  1.0000,  3.0000],
       [ 0.0000,  2.0000,  3.0000, -1.0000, -4.0000],
       [ 1.0000,  9.0000,  0.0000,  4.0000, -5.0000],
       [ 2.0000,  3.0000,  1.0000,  1.0000,  0.0000]])
```

Abychom získali na diagonále jedničku, je třeba první řádek vydělit třemi. První řádek matice můžeme zapsat jako `matice[0]` nebo `matice[0,:]`.

```
[ ]: matice[0]

array([3.0000, 2.0000, 1.0000, 1.0000, 3.0000])
```

```
[ ]: matice[0] /= 3

matice

array([[ 1.0000,  0.6667,  0.3333,  0.3333,  1.0000],
       [ 0.0000,  2.0000,  3.0000, -1.0000, -4.0000],
       [ 1.0000,  9.0000,  0.0000,  4.0000, -5.0000],
       [ 2.0000,  3.0000,  1.0000,  1.0000,  0.0000]])
```

```
[ ]: matice[2] -= matice[0]
matice[3] -= 2*matice[0]
```

(continues on next page)

(pokračujte na předchozí stránce)

```
matice
array([[ 1.0000,  0.6667,  0.3333,  0.3333,  1.0000],
       [ 0.0000,  2.0000,  3.0000, -1.0000, -4.0000],
       [ 0.0000,  8.3333, -0.3333,  3.6667, -6.0000],
       [ 0.0000,  1.6667,  0.3333,  0.3333, -2.0000]])
```

```
[ ]: # Druhý řádek
matice[1] /= matice[1,1]

matice
array([[ 1.0000,  0.6667,  0.3333,  0.3333,  1.0000],
       [ 0.0000,  1.0000,  1.5000, -0.5000, -2.0000],
       [ 0.0000,  8.3333, -0.3333,  3.6667, -6.0000],
       [ 0.0000,  1.6667,  0.3333,  0.3333, -2.0000]])
```

```
[ ]: matice[0] -= matice[0,1] * matice[1]
matice[2] -= matice[2,1] * matice[1]
matice[3] -= matice[3,1] * matice[1]

matice
array([[ 1.0000,  0.0000, -0.6667,  0.6667,  2.3333],
       [ 0.0000,  1.0000,  1.5000, -0.5000, -2.0000],
       [ 0.0000,  0.0000, -12.8333,  7.8333, 10.6667],
       [ 0.0000,  0.0000, -2.1667,  1.1667,  1.3333]])
```

```
[ ]: # Třetí řádek
matice[2] /= matice[2,2]

matice[0] -= matice[0,2] * matice[2]
matice[1] -= matice[1,2] * matice[2]
matice[3] -= matice[3,2] * matice[2]

matice
array([[ 1.0000,  0.0000,  0.0000,  0.2597,  1.7792],
       [ 0.0000,  1.0000,  0.0000,  0.4156, -0.7532],
       [-0.0000, -0.0000,  1.0000, -0.6104, -0.8312],
       [ 0.0000,  0.0000,  0.0000, -0.1558, -0.4675]])
```

```
[ ]: # Čtvrtý řádek
matice[3] /= matice[3,3]

matice[0] -= matice[0,3] * matice[3]
matice[1] -= matice[1,3] * matice[3]
matice[2] -= matice[2,3] * matice[3]

# Obnovíme výchozí formátování
np.set_printoptions(floatmode="maxprec_equal",precision=8) # set default
```

(continues on next page)

```
# Jsme hotovi
matice
array([[ 1.,  0.,  0.,  0.,  1.],
       [ 0.,  1.,  0.,  0., -2.],
       [-0., -0.,  1.,  0.,  1.],
       [-0., -0., -0.,  1.,  3.]])
```

Následující funkce `GJelim(matice, prava_strana)` provádí tyto kroky najednou. Obsahuje navíc kód, který prohazuje řádky. Lze snadno vidět, že by předchozí postup havaroval, kdyby se na diagonále objevila nula, protože tu násobením na jedničku nepředěláme. Protože jsou rovnice záměnné, můžeme přehodit řádky. Pro regulární matici nějaký musí dát nenulový diagonální prvek. Také nás nepřekvapí, že by mohlo vadit i číslo velmi blízké nule. Podrobná analýza ale ukazuje, že je dobré řádky prohazovat vždy, když lze pod diagonálou najít nějaký větší prvek. Protože by bylo možné ještě navíc prohazovat i sloupce, říká se takovému postupu *částečná pivotace*.

Funkce je doplněna o výpis hodnot během výpočtu (`print_A(...)`), který demonstruje, jak eliminace postupuje.

Na začátku kódu funkce nejprve spojíme matici soustavy a pravou stranu do jediné obdélníkové matice. Nejde o plýtvání časem, stejně bychom si museli pořídít kopie matice i pravé strany, abychom v průběhu eliminace neměnili proměnné, které byly použity při volání.

```
[ ]: import numpy as np

def print_A(A, sleep=0.5):
    "Výpis upravované matice a pravé strany"
    # pro přehlednost skrývám import nepodstatných knihoven
    from IPython.display import clear_output
    import time

    with np.printoptions(floatmode="fixed", precision=3):
        print(A)

    time.sleep(sleep)
    clear_output(wait=True)

def GJelim(matice, prava_strana):
    "Vrátí řešení soustavy rovnic A.x=b nalezené GJ eliminací. Parametry musí být typu ndarray."

    dim = len(prava_strana)

    A = np.zeros([dim,dim+1])

    # následující přiřazení provádí zároveň kontrolu velikosti vstupních polí
    A[0:dim, 0:dim] = matice
    A[0:dim, dim] = prava_strana

    for r in range(dim): # pro všechny řádky <r>
        print_A(A, sleep=2)

        # 1. částečná pivotace (řeší nulovost diagonálního prvku a zvyšuje přesnost)
        best_r = r
        for r1 in range(r+1,dim):
```

(continues on next page)

(pokračujte na předchozí stránce)

```

    if abs(A[r1,r]) > abs(A[best_r,r]):
        best_r = r1
    if best_r != r:
        # A[[best_r,r],r:] = A[[r,best_r],r:] #alternativní varianta prohození
        print_A(A)
        A[best_r,r:], A[r,r:] = A[r,r:], A[best_r,r:].copy()
        print_A(A)

    # 2. normalizace radku
    # 2.1 najdeme prevrácenou hodnotu prvku na diagonale
    faktor = 1 / A[r,r]
    # 2.2 "vsechny" sloupce radku <r> vynasobime timto faktorem
    A[r,r:] *= faktor

    print_A(A)

    # 3. eliminace
    for r1 in range(dim): # pro vsechny radky <r1>
        if r != r1:
            faktor = A[r1,r];
            A[r1,r:] -= faktor * A[r,r:]
            print_A(A)

    print_A(A,sleep=2)
    return A[:,dim].copy()

matice = np.array( [
    [0, 2, 3, -1],
    [3, 2, 1, 1],
    [1, 9, 0, 4],
    [2, 3, 1, 1] ] )

vektor_ps = np.array( [ -4, 3, -5, 0 ] )

x = GJelim(matice, vektor_ps )

print( ' řešení =', x )
with np.printoptions(precision=1):
    print( ' chyba x =', x - [1,-2,1,3] )
    print( 'chyba Ax-b =', matice.dot(x) - vektor_ps )

    řešení = [ 1. -2.  1.  3.]
    chyba x = [-5.6e-16 -8.9e-16  8.9e-16  1.8e-15]
    chyba Ax-b = [-8.9e-16 -8.9e-16 -8.9e-16 -8.9e-16]

```

Dva technické detaily se nám dostaly i do tak jednoduchého programu, jako je řešení soustavy rovnic. Oba souvisí s tím že řezy podle numpy nejsou kopie prvků pole a jen jakési chytré odkazy na původní prvky pole.

Právě proto se v příkazu `return A[:,dim].copy()` vytváří kopie. Jde o to, že `A[:,dim]` je poslední sloupec obdélníkové matice `A`. Ta se zdá být lokální proměnnou funkce `GJelim` a tedy bychom očekávali, že po skončení běhu funkce paměť zabraná `A` je uvolněna. Pokud bychom ale funkci ukončili příkazem `return A[:,dim]` pak by matice `A` nemohla být zapomenuta, protože její poslední sloupeček představuje řešení soustavy rovnic. Teprve až to by nebylo potřeba, mohlo by se uklidit i místo pro matici `A`.

Při prohazování řádek se opět objevuje použití metody `copy()`. Přitom u prohazování např. celých čísel jsme mohli psát `i, j = j, i` a nedošlo k potížím. Řezy numpy jsou ale odkazy na pole a jak je vidět níže, zápis `matice[2]`, `matice[3] = matice[3]`, `matice[2]` způsobí, že řádek obsahující dvojky je přepsán dříve než se jeho hodnota může použít pro prohození. Proto se musí vytvořit "záloha" prostřednictvím `copy()`.

Cvičení: Odstraňte z kódu prohazování řádků a vyzkoušejte, že pro matici

```
matice = np.array( [
    [1e-15, 2, 3, -1],
    [3, 2, 1, 1],
    [1, 9, 0, 4],
    [2, 3, 1, 1] ] )
```

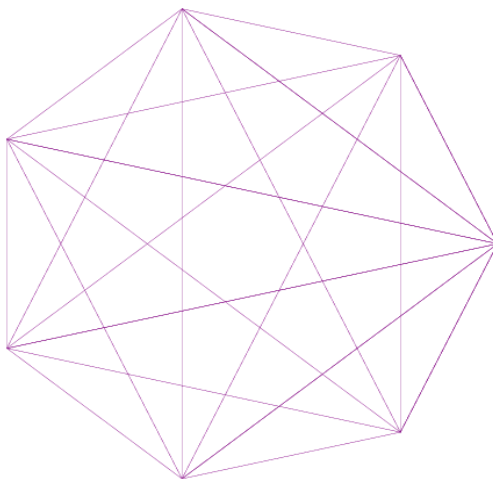
Bude nalezené řešení soustavy rovnic naprosto špatně a že tedy ono prohazová řádků "za lepší" (částečná pivotace) je nezbytné.

Vzhledem k použití celých řezů můžeme eliminaci zapsat jediným příkazem `A[r1,r:] -= A[r1,r] * A[r,r:]`, tedy pomocnou proměnnou `faktor` lze vynechat. Kdybychom ovšem prováděli tuto operaci cyklem, pak protože se hodnota prvku pole `A[r1,r]` během takového cyklu změní, je zapamatování původní hodnoty nezbytné.

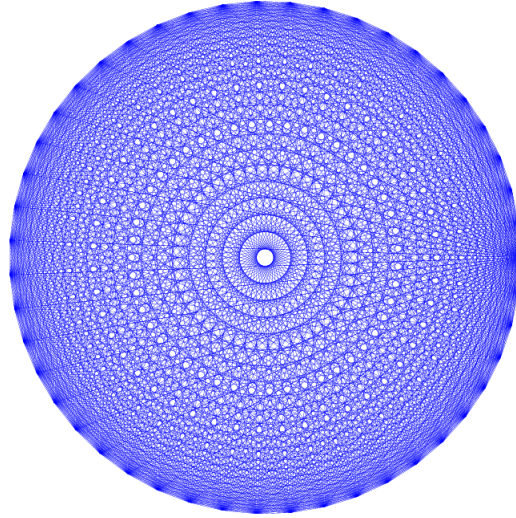
Samozřejmě, řešení soustavy rovnic je k dispozici jako funkce `numpy.linalg.solve`.

Časová náročnost algoritmů

Je zřejmé, že některé výpočty mohou trvat dlouho. Uvažme následující obrázek úplného n -úhelníku pro $n = 7$.



tušíme, že věci se zesložití, pokud vezmeme větší počet vrcholů



Podobně tomu může být i počítačových programů – při růstu nějakého parametru může doba výpočtu překvapivě prudce. Jak něco takového odhadnout?

Pro každý problém zkusíme najít charakteristické N číslo udávající jeho velikost. Ne vždy je to možné ale pro představu pár příkladů:

- Počet položek na skladu v programu pro skladové hospodářství.
- Počet měst, které má navštívit obchodní cestující.
- Počet neznámých v soustavě lineárních rovnic.
- Počet hvězd tvořících hvězdokupu, jejíž vývoj zkoumáme.
- Počet molekul v simulaci.

Nyní se můžeme ptát jak se bude program zabývající se výše uvedenými problémy chovat při růstu onoho charakteristického čísla N . Samozřejmě, pokud neplánujeme růst skladu, zvyšování počtu neznámých atp., nemusíme se tím zabývat. I ve fyzice se ale projevuje tendence počítat složitější a složitější problémy (třeba ty jednodušší už někdo vyřešil) a tak na problém velkého N můžeme narazit.

Při porovnávání různých algoritmů máme možnost porovnat, jak se chovají pro různé velikosti vstupních dat přímo. Někdy závisí doba řešení problému na konkrétních vstupních hodnotách, často ale, jak jsme viděli, u soustav lineárních rovnic, závisí pouze na velikosti problému.

Vzpomeneme-li si na definici algoritmu, víme že jde o posloupnost elementárních operací. Předpokládejme nejdříve, že elementárními operacemi rozumíme základní operce jako jsou sčítání, násobení, větvení, přístup k jednoduché proměnné, přístup k prvku pole volání či návrat z podprogramu atp. Změříme-li dobu, kterou potřebuje počítač k provedení každé z těchto operací a budeme-li předpokládat, že celý program se vykoná za dobu odpovídající součtu těchto jednotlivých operací, víme jak spočítat dobu potřebnou k provedení programu. Uvažujme podprogram pro skalární součin dvou vektorů o délce N

```
def scalar_product(a,b):
    "Počítá skalární součin dvou vektorů"
    dim = len(a)

    s = 0.0
    for i in range(dim):
        s = s + a[i]*b[i]

    return s
```


Obsahuje různé operacem přičemž např. `dim = len(a)` se provádí jednou zatímco `s = s + a[i]*b[i]` se provádí N ×. Mohli bychom tedy nějak zjistit čas provádění jednotlivých operací, ten vynásobit počtem provádění těchto operací, všechny časy sečíst a získat vzorec udávající závislost doby výpočtu na hodnotě N například v podobě

$$T[\text{ns}] = 542N + 1220$$

Jde o naivní přístup, na moderních počítačích je složité odhadnout dobu výpočtu i kdybychom znali prováděné instrukce procesoru, další komplikace přináší svojí povahou jazyk Python.

Můžeme také zkusit provést experiment:

```
[60]: import time
import numpy as np
import matplotlib.pyplot as plt

def scalar_product(a,b):
    "Počítá skalární součin dvou vektorů"
    dim = len(a)

    s = 0.0
    for i in range(dim):
        s = s + a[i]*b[i]

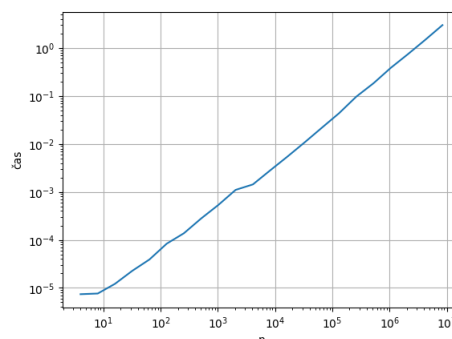
    return s

def time_it(n):
    x = np.linspace(0,1,n)

    start = time.time()
    s = scalar_product(x,x)
    end = time.time()
    return end - start

n = [2**k for k in range(2, 24)]
t = [time_it(k) for k in n]

plt.plot(n,t)
plt.yscale('log')          # logaritmický průběh os
plt.xscale('log')
plt.xlabel('n')
plt.ylabel('čas')
plt.grid()
plt.show()
```



Pohled na obrázek ukazuje, že jde o lineární závislost.

Podobně bychom pro násobení matice vektorem dostali třeba

$$T = 36N^2 + 44N + 26$$

Pokud nás ale zajímá chování pro větší N , je rozhodující první člen (nejvyšší mocnina). Píšeme tedy

$$T \approx 36N^2$$

Určit koeficient přesně je ale i tak velmi obtížné, nejsnazší možností je analýza měření závislosti času výpočtu na N (viz např. graf výše). Pokud chceme mluvit o výkonu algoritmu v okamžiku jeho návrhu ještě před jeho kódováním a nákupem počítače, ukazuje se, že nejjednodušší je prostě psát

$$T = O(N^2)$$

Velké $O(f)$ je označení pro libovolnou funkci g , která splňuje vztah

$$0 < \lim_{N \rightarrow \infty} \left| \frac{g}{f} \right| < \infty$$

Následující tabulka má ilustrovat, že opravdu rozhodující je právě řád, nikoli konkrétní hodnota koeficientu u vedoucího členu.

N	$N \cdot \log_2 N$	N^2	N^3	2^N	$N!$
3	6	9	27	8	6
10	30	100	1 000	1 024	3 628 800
30	150	900	27 000	1.1×10^9	2.65×10^{32}
100	700	10 000	1 000 000	1.27×10^{30}	9.33×10^{157}
1 000	10 000	1 000 000	1×10^9	1.1×10^{300}	4.02×10^{2567}
10 000	140 000	100 000 000	1×10^{15}	2.0×10^{3010}	2.84×10^{35659}

Naše výpočetní možnosti podléhají principiálním omezením, od velkého třesku uplynulo asi 4×10^{26} nanosekund a v pozorovaném vesmíru je $\approx 10^{83}$ atomů.

Vztah $O(f) \cdot O(g) = O(f \cdot g)$ nám pak umožňuje místo rozkladu algoritmu na elementární kroky $O(1)$, uvažovat strukturovaně.

Třeba Gauss-Jordanova eliminace (za předpokladu, že počet pravých stran je $\leq O(N)$) pro každý řádek (tedy $O(N)$ krát) provádí

- hledání největšího prvku na/pod diagonálou $\{O(N)\}$
- normalizace $O(N)$
- odečtení násobku řádku od všech ostatních $O(N^2)$
- na začátku a na konci ještě nějaké kopírování $O(N^2)$

Odsud máme

$$O(N^2) + O(N) * [O(N) + O(N) + O(N^2)] = O(N^3)$$

V tabulce si pak snadno najdeme, pro jaká N jde o zelený, oranžový či hnědý problém.

Podobně jako se s rostoucím N nějak chová čas potřebný k provedení výpočtu, nějak rostou i **paměťové nároky**. Protože nad velikostí datových struktur máme trochu lepší kontrolu, lze v praxi velmi dobře odhadnout, co se vejde a co ne. V rozvahách o schůdnosti různých algoritmů ale také můžeme použít Landauovu notaci velkého O .

Vybrané numerické algoritmy

Naše přednáška není přednáškou o numerických metodách. Přesto jsou zde pravidelně zmiňovány, protože

Základy programování pro fyziky

- musejí přiblížit základy programování na nějakých úlohách. Algoritmy použité v numerických metodách jsou často z hlediska informatiky velice jednoduché a tedy vhodné na programování v úvodním kurzu.
- by měly připravit posluchače na problémy, jaké v rámci programování fyzikové potkají. Mezi ně patří právě základní numerické algoritmy.

17.1 Přehled doposud probraných numerických témat

- Hledání největšího společného dělitele (Euklidův algoritmus s odečítáním a pak s operací modulo)
- Aritmetické výrazy
- Volání knihovních funkcí (knihovny `math` a `scipy.special`)
- Polynomy a Hornerovo schéma
- Vyčíslení sum, součinů
- Funkce definované rekurzivní formulkou
- Hledání kořene (půlení intervalu, Newtonova metoda)
- Důsledky zaokrouhlovacích chyb, reprezentace reálných čísel v počítači
- Eratosthenovo síto
- Organizace dat do polí
- Hromadné vyčíslování skrze univerzální funkce v `numpy`
- Numerická kvadratura (výpočet určitého integrálu lichoběžníkovým pravidlem, ostatní jen jako zajímavost)
- Metoda Monte Carlo na příkladu numerické kvadratury

- Vektory, matice, operace dot (@)
- Řešení soustavy lineárních algebraických rovnic (jen Gaussova-Jordanova eliminace a volání `numpy.linalg.solve`)

Zbývá ale ještě několik důležitých témat doplnit.

17.2 Polynomiální interpolace

Uvažujme, že nějaká veličina y závisí na hodnotě parametru x . Je tedy rozumné mluvit o funkci $y(x)$. Někdy ale může být výpočet funkce velmi náročný, a tak máme možnost ji zjistit jen v omezeném množství hodnot x , jindy toto omezení vyplývá z toho, že funkce byla určena experimentálně pro omezené množství hodnot x .

Předpokládejme, že z těchto důvodů známe její funkční hodnoty jen ve vybraných bodech $x_1 < x_2 < \dots < x_n$. Nastává otázka, co můžeme říci o hodnotě funkce y v nějakém bodě x , kde

- $x_1 < x < x_n$. Pak mluvíme o interpolaci.
- $x < x_1$ nebo $x > x_n$. Pak mluvíme o extrapolaci.

Jde o složitou otázku, nás bude zajímat jen varianta, kdy budeme zkoumat polynom, který v uvedených bodech nabývá stejnou hodnotu. Pokud je jeho řád $n - 1$, pak je určen jednoznačně.

17.2.1 Interpolace jako příklad užití knihovní funkce

Následující kód používá knihovní funkci `scipy.interpolate.lagrange`. Ta vrátí funkci, které když dáme argument, spočte jeho hodnotu.

```
[51]: from scipy.interpolate import lagrange
import math
import numpy as np
import matplotlib.pyplot as plt

f = np.arctan
a, b = -3, 3
nc_list = [(4, 'red'), (8, 'blue'), (12, 'green')]

dx = (b-a)/64
x_plot = np.linspace(a-dx,b+dx,200)
plt.plot( x_plot, f(x_plot) , color='gray', linewidth=0.4)

for n, c in nc_list:
    x = np.linspace(a, b, n)
    y = f(x)
    poly = lagrange(x, y)

    plt.plot( x_plot, poly(x_plot), color=c)
    plt.plot( x, y, 'o', color=c)

plt.title('Polynomiální interpolace')
plt.grid()
plt.show()

for n, c in nc_list:
```

(continues on next page)

(pokračujte na předchozí stránce)

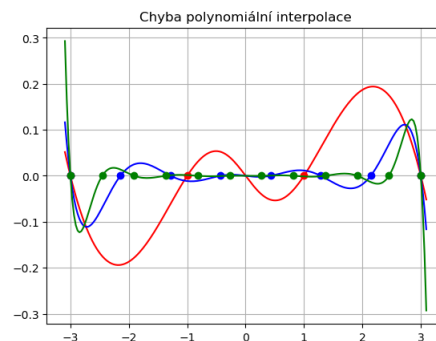
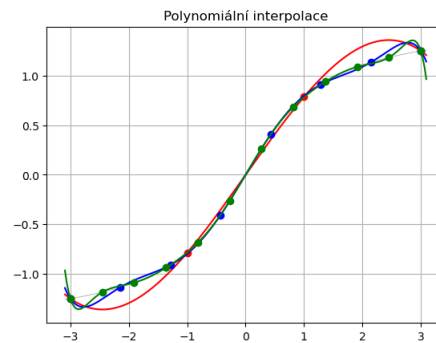
```

x = np.linspace(a, b, n)
y = f(x)
poly = lagrange(x, y)

plt.plot( x_plot, poly(x_plot)-f(x_plot), color=c)
plt.plot( x, np.zeros_like(x), 'o', color=c)

plt.title('Chyba polynomiální interpolace')
plt.grid()
plt.show()

```



Cvičení: pozorujte, jak se chování interpolačního polynomu liší

- když zkracujete délku intervalu,
- když funkci \arctan nahradíte \exp .
- když místo ekvidistantního vzorkování (které dá `np.linspace(a, b, n)`) použijete níže uvedené hodnoty x_i podle Čebyševa.

Lagrangeova interpolace je založena na faktu, že funkci můžeme např. pro $n = 4$ rozložit na součet

$$P_{\text{approx}}(x) = p_1(x) + p_2(x) + p_3(x) + p_4(x)$$

členů tvaru

$$p_2 = \frac{(x - x_1)(x - x_3)(x - x_4)}{(x_2 - x_1)(x_2 - x_3)(x_2 - x_4)} y_2$$

což je polynom, který v je nulový ve všech bodech x_i s výjimkou x_2 , kde nabývá hodnoty y_2 .

Proto má výsledný vzorec tvar

$$P_{\text{approx}}(x) = \frac{(x-x_2)(x-x_3)\dots(x-x_n)}{(x_1-x_2)(x_1-x_3)\dots(x_1-x_n)}y_1 + \frac{(x-x_1)(x-x_3)\dots(x-x_n)}{(x_2-x_1)(x_2-x_3)\dots(x_2-x_n)}y_2 + \dots + \frac{(x-x_1)(x-x_2)\dots(x-x_{n-1})}{(x_n-x_1)(x_n-x_2)\dots(x_n-x_{n-1})}y_n$$

neboli

$$P_{\text{approx}}(x) = \sum_{i=1}^n y_i \prod_{j=1, j \neq i}^n \frac{x-x_j}{x_i-x_j}$$

```
[53]: import numpy as np
import matplotlib.pyplot as plt

def linterp(t, x, y):
    n = len(x)
    assert len(y) == n

    s = 0.0
    for i in range(n):
        p = y[i]
        for j in range(n):
            if i != j:
                p *= (t-x[j])/(x[i]-x[j])
        s += p
    return s

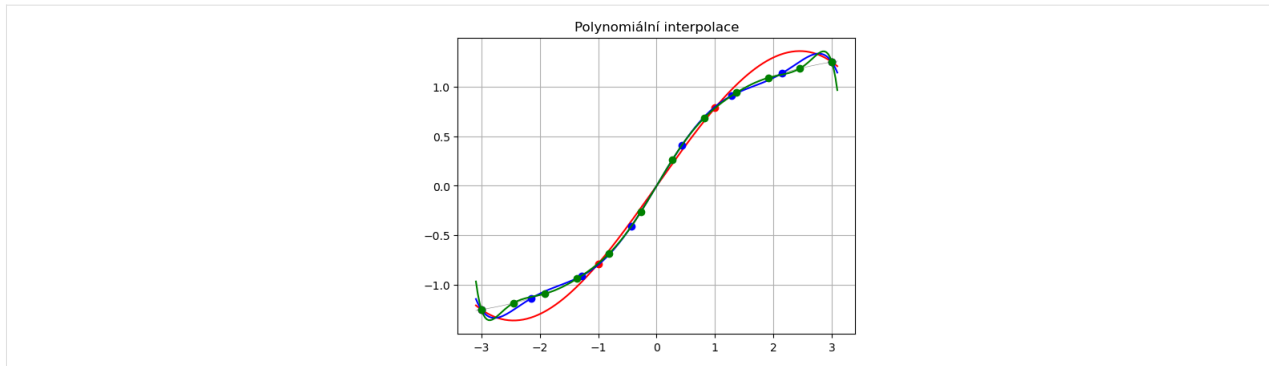
f = np.arctan
a, b = -3, 3
nc_list = [(4, 'red'), (8, 'blue'), (12, 'green')]

dx = (b-a)/64
x_plot = np.linspace(a-dx, b+dx, 200)
plt.plot(x_plot, f(x_plot), color='gray', linewidth=0.4)

for n, c in nc_list:
    x = np.linspace(a, b, n)
    y = f(x)
    y_plot = [linterp(t, x, y) for t in x_plot]

    plt.plot(x_plot, y_plot, color=c)
    plt.plot(x, y, 'o', color=c)

plt.title('Polynomiální interpolace')
plt.grid()
plt.show()
```

Ukazuje se, že pro větší počet bodů, skrz které interpolační polynom prokládáme, je ekvidistantní vzorkování nevhodné. Mnohem lepšího souladu mezi interpolovanou funkcí a její aproximací lze dosáhnout, pokud

- použijeme nerovnoměrné vzorkování intervalu, např. podle Čebyševa,

$$x_i = \frac{a+b}{2} + t_i \frac{b-a}{2}, \quad t_i = \cos \pi \frac{n-k-\frac{1}{2}}{n}, \quad i = 0, 1, \dots, n-1$$

- interval rozdělíme na více částí a opakovaně použijeme polynomiální interpolaci skrz menší počet bodů. Metoda, která se jednotlivé úseky snaží hladce navázat, se jmenuje *spline* (viz ukázka použití CubicSpline níže).

Obojí si můžete vyzkoušet jako cvičení na práci s poli a vést v patrnosti, že se máte podívat do příslušné učebnice, kdybyste potřebovali polynomiální interpolaci větším počtem bodů.

```
[65]: from scipy.interpolate import lagrange, CubicSpline
import math
import numpy as np
import matplotlib.pyplot as plt

f = np.arctan
a, b = -3, 3

dx = (b-a)/64
x_plot = np.linspace(a-dx, b+dx, 200)
plt.plot(x_plot, f(x_plot), color='gray', linewidth=0.4)

n = 8

x = np.linspace(a, b, n)
y = f(x)
poly = lagrange(x, y)
spline = CubicSpline(x, y)

plt.plot(x_plot, poly(x_plot), label='Lagrange')
plt.plot(x_plot, spline(x_plot), '--', label='cubic spline')
plt.plot(x, y, 'o')

plt.title('Polynomiální vs. splajnová interpolace')
plt.grid()
plt.legend()
plt.show()
```

(continues on next page)

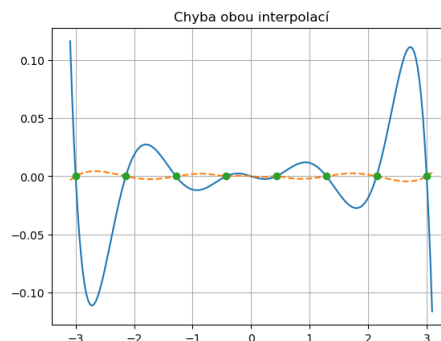
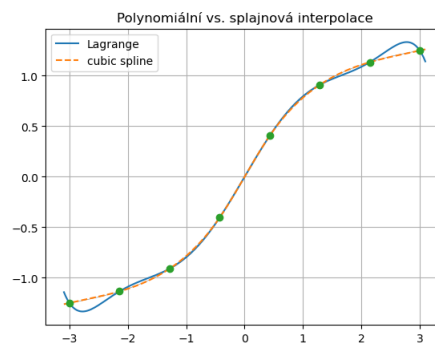
```

x = np.linspace(a, b, n)
y = f(x)
poly = lagrange(x, y)
spline = CubicSpline(x,y)

plt.plot( x_plot, poly(x_plot)-f(x_plot))
plt.plot( x_plot, spline(x_plot)-f(x_plot), '--')
plt.plot( x, np.zeros_like(x), 'o')

plt.title('Chyba obou interpolací')
plt.grid()
plt.show()

```



17.3 Polynomiální regrese

Často nepožadujeme, aby daná křivka určenými body procházela přesně, ale aby minimalizovala vhodně definovanou chybu. Nejsnazší je tzv. metoda nejmenších čtverců, kde máme lineární model

$$\hat{y}(x) = \sum_A p_A f_A(x)$$

závisející na $m < n$ parametrech p_A určujících souřadnice v bázi funkcí $f_A(x)$ a uvažovat chybu v podobě kvadratického výrazu

$$|\vec{y} - \vec{\hat{y}}|^2 = |\vec{y} - \mathcal{F} \cdot \vec{p}|^2,$$

kde $\mathcal{F}_{\setminus A}$ je obdélníková matice hodnot A -té báze funkce v i -tém bodě.

Schematicky můžeme psát, že od minimalizace chyby požadujeme

$$\vec{y} \approx \mathcal{F} \vec{p}$$

kde ale dimenze vektorů $\dim \vec{y} > \dim \vec{p}$, v případě požadování rovnosti by tedy šlo o přeřčenou soustavu lineárních rovnic.

My umíme řešit soustavy rovnic se čtvercovou maticí a je zajímavé, že minimalizace kvadratického výrazu dá lineární soustavu rovnic pro \vec{p} , která z té předchozí vznikne vynásobením transponovanou maticí \mathcal{F}^T

$$(\mathcal{F}^T \mathcal{F}) \vec{p} = \mathcal{F}^T \vec{y}$$

V rámci našeho předmětu nemusíme rozumět původu rovnic, ale musíme je umět převést do kódu. Proto následuje ukázka programu odpovídajícího na otázku, jaký polynom třetího stupně minimalizuje sumu kvadrátů odchylek od průběhu funkce sinus pro sadu n rovnoměrně rozložených bodů na intervalu $\langle 0, \pi/2 \rangle$.

Kód ukazuje dvě varianty. Jedna sestavuje výše uvedenou matici a řeší soustavu rovnic, druhá volá `numpy.polyfit`.

```
[85]: import numpy as np

n = 17

x_samples = np.linspace(0, np.pi/2, n)
y_samples = np.sin(x_samples)

# varianta 1 ... sestavme matici, řešme soustavu
def basis(x):
    return np.array([x*x*x, x*x, x, 1])

matrix_F = np.array( [basis(x) for x in x_samples] )
matrix_FT = matrix_F.transpose()
coeffs1 = np.linalg.solve( matrix_FT @ matrix_F , matrix_FT @ y_samples)
print( coeffs1 )

# varianta 2 ... použijme numpy
coeffs2 = np.polyfit(x_samples, y_samples, 3)
print( coeffs2 )

[-0.11347006 -0.06919881  1.02515468 -0.00154229]
[-0.11347006 -0.06919881  1.02515468 -0.00154229]
```

Cvičení: Nakreslete graf funkce $\sin x$ a polynomu nalezeného regresí.

Cvičení: V principu lze matici F vytvořit příkazem `matrix_F = basis(x_samples).transpose()` Zkuste to! (Nápověda: Ještě je potřeba mírně upravit kód funkce `basis`, protože, zatímco $x*x*x$ i $x*x$ jsou vektory stejně jako x , výraz `1` je skalár.)

17.4 Optimalizace (jako příklad použití knihovního algoritmu)

Hledání minima (nebo maxima) nějaké funkce je velmi složitým problémem. Jedním z důvodů je, že minimalizovaný výraz může mít mnoho lokálních minim, "údolí" v jejich okolí také mohou mít ve více proměnných ošklivý tvar. Existuje proto velké množství přístupů k numerickému hledání extrému funkce více proměnných, nás teď bude zajímat, jak pro již hotový algoritmus náš konkrétní problém upravit.

Vezmeme minulou úlohu, ale místo nejmenších čtverců zvolíme jako míru příhodnosti aproximace maximální nalezenou odchylku. Rázem jde o nelineární problém. Abychom jej mohli knihovní funkci `scipy.optimize.minimize` předložit:

- musíme prozkoumat *dokumentaci*, ze které nahlédneme, že funkce má mnoho parametrů, ale jen dva povinné:
 - jakou funkci chceme minimalizovat, tato funkce má jeden argument s významem pole parametrů,
 - počáteční odhad hodnoty parametrů. Ten kromě informace, kde začít hledat (což i u našeho jednoduchého problému může ovlivnit, které minimum funkce najde), dává funkci `minimize` informaci o tom, kolik neznámých parametrů má hledat. My začneme s hodnotami nalezenými metodou nejmenších čtverců.

Dále z dokumentace zjistíme, že funkce vrací hodnotu nalezených parametrů v atributu `x`, to zda se povedlo nalézt minimum v atributu `success` a chybové hlášení v atributu `message`.

- musíme napsat funkci `maxdev(c)`, která vrací $\max_i |f(x_i) - \sin x_i|$, kde $f(x) = \sum_{i=0}^3 c_i x^i$.
- musíme zavolat funkci `scipy.optimize.minimize` s oběma výše uvedenými argumenty.
- minimalizace může z mnoha důvodů havarovat (v našem příkladě stačí vzít jiné n), proto použijeme `assert`
- nakonec je potřeba z vráceného výsledku extrahovat koeficienty

Neodpustím si komentář, že uvedený postup jak získat polynomičnou aproximaci funkce, je především demonstrací hledání minima funkce více proměnných, nikoli optimálním přístupem.

```
[88]: import numpy as np
      from scipy.optimize import minimize
      import matplotlib.pyplot as plt

      # zadání problému: optimální aproximace sin(x) minimalizující max |chyby|
      n = 27
      x_samples = np.linspace(0, np.pi/2, n)
      y_samples = np.sin(x_samples)

      def poly_eval(coeffs, x):
          "Hodnota polynomu 3. stupně v x, kde x může být np.ndarray"
          return ((coeffs[0]*x + coeffs[1])*x + coeffs[2])*x + coeffs[3]

      def maxdev(coeffs):
          "maximum odchylky polynomu(x) a sin(x)"
          return max(abs(y_samples - poly_eval(coeffs, x_samples)))

      # nejprve chi^2
      coeffs2 = np.polyfit(x_samples, y_samples, 3)
      print( '   chi^2 coeffs =', coeffs2 )

      # zkusíme minimalizaci
      res = minimize(maxdev, coeffs2)
      assert res.success, res.message
      coeffs3 = res.x
```

(continues on next page)

(pokračujte na předchozí stránce)

```

print( 'minimize coeffs =', coeffs3 )

# vypíšeme maximální chybu
print( 'minimize error =', maxdev(coeffs3) )
print( '  chi^2 error =', maxdev(coeffs2) )

# namalujeme grafy chyb
x_plot = np.linspace(0,np.pi/2,200)
y_plot = np.sin(x_plot)

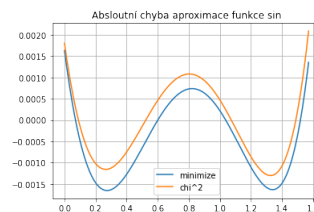
plt.plot( x_plot, y_plot - poly_eval(coeffs3, x_plot), label='minimize')
plt.plot( x_plot, y_plot - poly_eval(coeffs2, x_plot), label='chi^2')
plt.legend()
plt.grid()
plt.title('Absolutní chyba aproximace funkce sin')
plt.show()

```

```

chi^2 coeffs = [-0.11360154 -0.0694899  1.02589085 -0.00179986]
minimize coeffs = [-0.11203843 -0.073022  1.02794336 -0.00163133]
minimize error = 0.0016313314926229042
chi^2 error = 0.002088831061687557

```



17.5 Numerické derivování

Někdy se stane, že potřebujeme spočítat derivaci funkce, jejíž hodnoty známe jen číselně. Ukazuje se, že jde o těžký problém, protože např. v definičním vzorečku

$$y'(x) = \lim_{\epsilon \rightarrow 0} \frac{f(x + \epsilon) - f(x)}{\epsilon}$$

se nepíše, jak moc malé máme vzít ϵ abychom dostali hodnotu derivace s rozumnou přesností. Jedna potíž spočívá v tom, že pokud vezmeme ϵ opravdu hodně malé, bude mít $x + \epsilon$ v důsledku zaokrouhlení v rámci typu float stejnou hodnotu jako x , a tedy vyjde derivace nulová.

Ukazuje se, že vzorec

$$y'(x) = \lim_{\epsilon \rightarrow 0} \frac{f(x + \epsilon) - f(x - \epsilon)}{2\epsilon}$$

dovoluje použít větší ϵ a získat tak výsledek méně zatížený zaokrouhlováním.

Opět, v našem kurzu se především učíme umět napsat kód, ten následující porovná, jak je to s chybami obou vzorců. Funkční hodnoty $x_i = x_0 + i\epsilon$ a $y_i = f(x_i)$ budeme mít uloženy v poli. Protože jde o ekvidistantní vzorkování, můžeme při odečítání sousedních prvků podle vzorců výše výhodně použít řezy polí. Abychom nemuseli řešit derivování u kraje pole, hodnoty derivací v krajních bodech nepočítáme.

```
[110]: import numpy as np

n = 25

xi = np.linspace(-1,1,n)
yi = np.exp(xi)
dy = yi[1:-1]          # exp'(x) = exp(x)

epsilon = xi[1]-xi[0]

dy1 = (yi[2:]-yi[1:-1])/epsilon
dy2 = (yi[2:]-yi[0:-2])/(2*epsilon)

err1 = max( abs( dy1-dy ))
err2 = max( abs( dy2-dy ))

print('Chyby vzorců pro numerickou derivaci:')
print(f'{err1 = :.6.2g}    {err2 = :.6.2g}')
```

```
Chyby vzorců pro numerickou derivaci:
err1 =  0.11    err2 = 0.0029
```

Obrázek níže ilustruje, že ani pro optimální ϵ se druhý vzoreček nepřiblíží přesnosti, na kterou jsme zvyklí, když vyčíslujeme matematické funkce. Numerické derivování není snadné.

Cvičení: Ignorujte kód níže a napište nějaký svůj, který vyrobí podobný obrázek.

```
[106]: import matplotlib.pyplot as plt

def errs(n):
    xi = np.linspace(-1,1,n)
    yi = np.exp(xi)
    dy = yi[1:-1]          # exp'(x) = exp(x)

    epsilon = xi[1]-xi[0]

    dy1 = (yi[2:]-yi[1:-1])/epsilon
    dy2 = (yi[2:]-yi[0:-2])/(2*epsilon)

    err1 = max( abs( dy1-dy ))
    err2 = max( abs( dy2-dy ))

    return np.array([err1,err2])

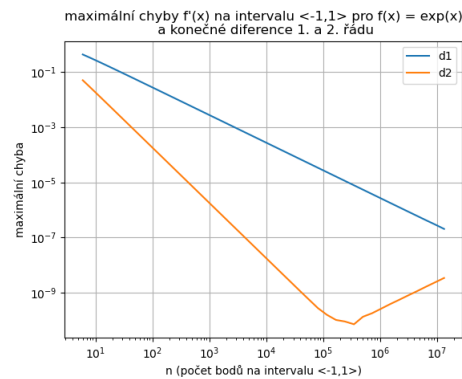
nn = (1.2**np.linspace(10,90,41)).astype(int)
ww = np.array([errs(n) for n in nn])

plt.yscale('log')          # logaritmický průběh os
plt.xscale('log')
plt.xlabel('n (počet bodů na intervalu <-1,1>)\n')
plt.ylabel('maximální chyba')
plt.title("maximální chyby f'(x) na intervalu <-1,1> pro f(x) = exp(x)\n a konečné
↪diference 1. a 2. řádu")
```

(continues on next page)

(pokračujte na předchozí stránce)

```
plt.grid()
plt.plot(nn,ww[:,0],label='d1')
plt.plot(nn,ww[:,1],label='d2')
plt.legend()           # zobraz popisek daný "label=..."
plt.show()
```



17.6 Řešení obyčejných diferenciálních rovnic

Od časů Newtona je mnoho problémů fyziky popisováno diferenciálními rovnicemi.

Nejprve uvažme následující kód. Realizuje tzv. Eulerovu metodu založenou na jednoduché nahrazení derivace

$$\frac{dy}{dt} \doteq \frac{\Delta y}{\Delta t} = \frac{y(t + \Delta t) - y(t)}{\Delta t},$$

tedy konkrétně

$$y(t + \Delta t) = y(t) + \Delta t F(t, y(t)).$$

Tento recept se používá opakovaně a pokud Δt neměníme, získáme ekvidistantně navzorkovaný odhad řešení diferenciální rovnice. Z hlediska programování jde tedy o jeden cyklus a sadu proměnných udržujících zejména informaci o hodnotách t_i, y_i .

Program, který řeší rovnici $dy/dy = -y$ s počáteční podmínkou $y(0) = 1$ je níže. Proč jsme jej již nepotkali na začátku kurzu souvisí se zpracováním získaných hodnot y_i . Zde z nich budujeme seznam a ten pak vykreslíme.

```
[111]: import math

steps_per_print = 100
dt = 0.1/steps_per_print
t_max = 1

n = 0
t = 0 # před cyklem jako obvykle nezapomeneme na inicializaci
y = 1 # zde má navíc význam počátečních podmínek

while t <= t_max + dt/2:
    if n % steps_per_print == 0:
        print(f'{t:6.4f} {y:6.4f} {math.exp(-t):6.4f}')
    dydt = -y
```

(continues on next page)

(pokračujte na předchozí stránce)

```

y = y + dydt * dt
t = t + dt
n = n + 1

```

```

0.0000 1.0000 1.0000
0.1000 0.9048 0.9048
0.2000 0.8186 0.8187
0.3000 0.7407 0.7408
0.4000 0.6702 0.6703
0.5000 0.6064 0.6065
0.6000 0.5486 0.5488
0.7000 0.4964 0.4966
0.8000 0.4491 0.4493
0.9000 0.4064 0.4066
1.0000 0.3677 0.3679

```

Povšimněte si, že

- počáteční podmínky pro diferenciální rovnici se v programu objevá jako inicializace proměnné y
- v cyklu se pak opakovaně ze staré hodnoty y spočte nová. Místo hodnot funkce uložené například v tabulce se tak řešení v programu objeví v jediné proměnné rozložené v čase – v každém opakování cyklu jedna hodnota. Takový postup se nazývá "pochodový" algoritmus, nemusí složitě řešit uložení hodnot funkce a vystačí s minimem proměnných.
- když zmenšíme krok, což zde obstarává konstanta `steps_per_print`, dostaneme přesnější výsledek. Snadno se lze přesvědčit, že pro desetkrát kratší krok bude chyba konečné hodnoty $y(1)$ desetkrát menší.

Tento jednoduchý program jsme mohli napsat hned na první hodině a rozuměli bychom mu. Podobné úlohy řešily i první počítače, jejich stavitelé a programátoři ovšem věděli, že musejí použít numerickou metodu lepší než tu Eulerovu.

O něco komplikovanější kód nám dnes řešení i namaluje:

```

[125]: import numpy as np
import matplotlib.pyplot as plt

steps_per_plot = 1
dt = 0.1/steps_per_plot
t_max = 3

t_data = []
y_data = []

n = 0

t = 0 # před cyklem jako obvykle nezapomeneme na inicializaci
y = 1 # zde má navíc význam počátečních podmínek

while t < t_max:
    if n % steps_per_plot == 0:
        t_data.append(t)
        y_data.append(y)

```

(continues on next page)

(pokračujte na předchozí stránce)

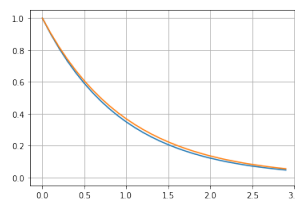
```

dydt = -y
y     = y + dydt * dt
t     = t + dt
n     = n + 1

y_presna = np.exp(-np.array(t_data))
plt.plot(t_data, y_data)
plt.plot(t_data, y_presna)
plt.grid()
plt.ylim(-0.05, 1.05)
plt.show()

print(f'Maximální chyba y(t) = {max(abs(y_data-y_presna)):5.2g}')

```



Maximální chyba $y(t) = 0.019$

Dále si ukážeme, že Newtonovy pohybové rovnice $m\ddot{\vec{x}} = \vec{F}/m$ lze také zapsat jako diferenciální rovnici pro vektorovou veličinu \vec{U} , jejíž polovina složek jsou polohy a druhá polovina rychlosti

$$\frac{d}{dt} \begin{pmatrix} \vec{x} \\ \vec{v} \end{pmatrix} = \begin{pmatrix} \vec{v} \\ \vec{F}(t, \vec{x}, \vec{v})/m \end{pmatrix}.$$

Protože chceme mít po ruce kód, který řeší rovnice pro libovolnou podobu síly, kód přeuspořádáme. Protože v každém kroku numerického řešení diferenciální rovnice nabývá nové hodnoty jednak \vec{U} ale také t , vrací funkce `krok_Euler` dvojici obou nových hodnot. Zatímco by bylo možné pozměnit argument U (je to pole, tedy je *mutable*), čas t je uložen v reálné proměnné (typ `float` je *immutable*) a proměnnou použitou jako aktuální argument pozměnit z vnitřku funkce nelze.

```

[157]: import math
import numpy as np
import matplotlib.pyplot as plt

def krok_Euler(t, U, fce_dUdt, dt):
    """
    Jeden krok Eulerovy metody. Vrací dvojici nové_t, nové_U.
    Argumenty:
    U ..... výchozí hodnota U(t) (musí být typu numpy.ndarray)
    fce_dUdt .. pravá strana dif. rovnice. Argumenty jsou t,U.
    """

    prava_strana = fce_dUdt(t,U)

    U = U + dt*prava_strana
    t = t+dt

```

(continues on next page)

```
return t, U

def pohybova_rovnice_planety(t, U):
    "Předpokládá U = [x,y,vx,vy]. Vrací časovou derivaci dle Newtona."

    GM = 4 * math.pi**2      # grav. konstanta v jednotkách AU, rok, hmotnost_Slunce
    x, y, vx, vy = U

    r2 = x*x+y*y
    r_3 = 1/(r2*math.sqrt(r2))
    ax = - GM * x*r_3
    ay = - GM * y*r_3

    return np.array([vx, vy, ax, ay])

x_data = []
y_data = []

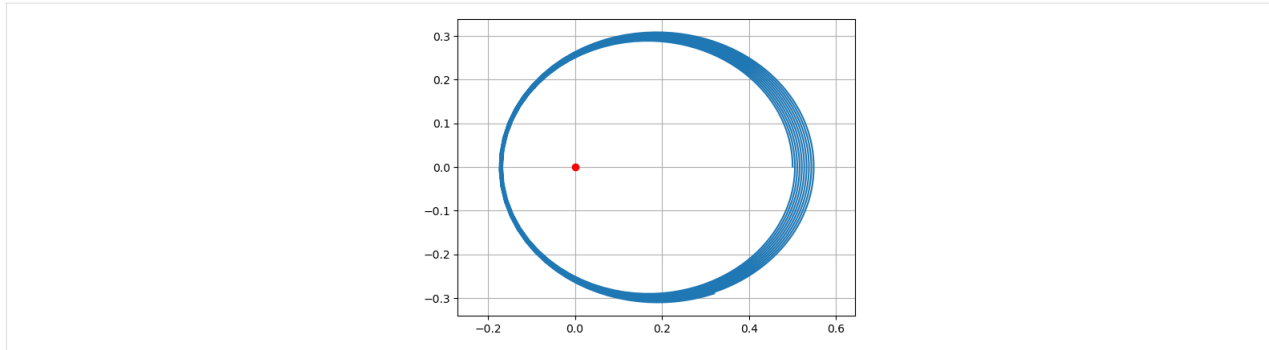
dt = 1/365/24/12 # 5 minut (ani tak nebude s Eulerovou metodou vysledek prilis presny)
tmax = 2 # dva roky

x0 = 0.5 # jednotka délky je AU
y0 = 0
vx0 = 0 # jednotka délky je AU/rok
vy0 = 2*np.pi # kruhová rychlost pro r=1AU

t = 0
Y = np.array([x0, y0, vx0, vy0])

while t<tmax:
    x_data.append(Y[0])
    y_data.append(Y[1])
    t, Y = krok_Euler( t, Y, pohybova_rovnice_planety, dt);

plot_step = len(x_data)//1000+1
plt.plot(x_data[::plot_step], y_data[::plot_step])
plt.plot([0],[0], 'or')
plt.axis('equal')
plt.grid()
plt.show()
```



Toto rozdělení na rovnici `pohybova_rovnice_planety` a numerickou metodu `krok_Euler` je užitečné proto, že nyní můžeme (i bez pochopení detailů) nahradit Eulerovu metodu metodou lepší (u pohybu planety se to projeví potlačením deformace její eliptické orbity).

Pozn. Povšimněte si řádku

```
x, y, vx, vy = U
```

Již víme, že ten využívá vlastnosti přiřazení (nazývané *unpacking*), kdy obsah pole nebo seznamu o čtyřech položkách přiřadíme do čtyřech proměnných. Zatímco např. v metodě `krok_Euler` je totiž výhodné pracovat s polem naráz, abychom např. mohli využít sčítání vektorů, ve funkci `pohybova_rovnice_planety` se projeví, že každá položka pole má jiný význam a vhodné pojmenování prvků pole zpřehlední kód.

```
[155]: import math
import numpy as np
import matplotlib.pyplot as plt

def krok_midpoint(t, U, fce_dUdt, dt):
    """
    Jeden krok Eulerovy metody. Vrací dvojici nové_t, nové_U.
    Argumenty:
    U ..... výchozí hodnota U(t) (musí být typu numpy.ndarray)
    fce_dUdt .. pravá strana dif. rovnice. Argumenty jsou t,U.
    """

    prava_strana = fce_dUdt(t,U)
    Umid = U + dt*0.5*prava_strana

    prava_strana = fce_dUdt(t+dt*0.5,Umid)
    U = U + dt*prava_strana

    t = t+dt
    return t, U

def pohybova_rovnice_planety(t, U):
    "Předpokládá U = [x,y,vx,vy]. Vrací časovou derivaci dle Newtona."

    GM = 4 * math.pi**2      # grav. konstanta v jednotkách AU, rok, hmotnost_Slunce
    x, y, vx, vy = U

    r2 = x*x+y*y
```

(continues on next page)

```

r_3 = 1/(r2*math.sqrt(r2))
ax = - GM * x*r_3
ay = - GM * y*r_3

return np.array([vx, vy, ax, ay])

x_data = []
y_data = []

dt = 1/365/24 # jedna hodina
tmax = 2 # dva roky

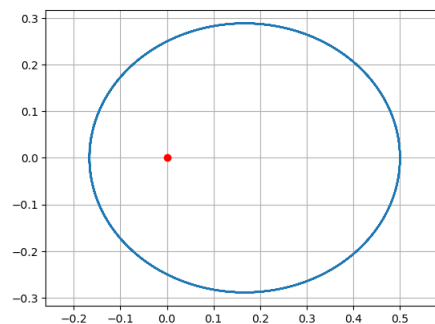
x0 = 0.5 # jednotka délky je AU
y0 = 0
vx0 = 0 # jednotka délky je AU/rok
vy0 = 2*np.pi # kruhová rychlost pro r=1AU

t = 0
Y = np.array([x0, y0, vx0, vy0])

while t<tmax:
    x_data.append(Y[0])
    y_data.append(Y[1])
    t, Y = krok_midpoint( t, Y, pohybova_rovnice_planety, dt);

plot_step = len(x_data)//1000+1
plt.plot(x_data[::plot_step], y_data[::plot_step])
plt.plot([0],[0], 'or')
plt.axis('equal')
plt.grid()
plt.show()

```



Ted, když rozumíme principu řešení diferenciálních rovnic s počáteční podmínkou (*initial-value problem*, IVP), můžeme náš postup porovnat s řešením prostřednictvím knihovní funkce `scipy.integrate.solve_ivp`. Použijeme volání

```
sol = solve_ivp(pohybova_rovnice_planety, (0,tmax), U0, max_step=dt)
```

- První argument určuje řešenou diferenciální rovnici. Jde o stejnou funkci jako v předchozích příkladech.

- Druhý argument má podobu intervalu proměnné t , na kterém numerické řešení hledáme. To znamená, že funkce `solve_ivp` vrátí pole hodnot pokrývajících tento interval.
- Dále musíme zadat počáteční podmínky. K tomu slouží třetí argument.
- Jako obvykle má knihovná funkce několik nepovinných parametrů, my použijeme jediný, `max_step`, kterým si vynutíme zkrácení kroku na vhodnou hodnotu. K tomu můžeme mít dva důvody. Jednak tím snížíme chybu, můžeme ale takto také donutit `solve_ivp`, aby vrátil dostatečně hustě navzorkované řešení např. pro jeho vykreslení - při velkém kroku by se lomená čára znatelně odlišovala od křivky i když by procházela správnými body.

Příkaz `plt.plot(sol.y[0], sol.y[1])` ukazuje, že vrácená hodnota `sol` má řešení uložené jako pole `sol.y` s dvěma indexy. První index určuje jednu z funkcí $x(t), y(t), v_x(t), v_y(t)$. Druhý index pak konkrétní vzorek.

```
[103]: from scipy.integrate import solve_ivp
import numpy as np
import math

def pohybova_rovnice_planety(t, U):
    "Předpokládá U = [x,y,vx,vy]. Vrací časovou derivaci dle Newtona."

    GM = 4 * math.pi**2      # pro grav. konstantu v jednotkách AU, rok, hmotnost_Slunce
    x, y, vx, vy = U

    r2 = x*x+y*y
    r_3 = 1/(r2*math.sqrt(r2))
    ax = - GM * x*r_3
    ay = - GM * y*r_3

    return np.array([vx, vy, ax, ay])

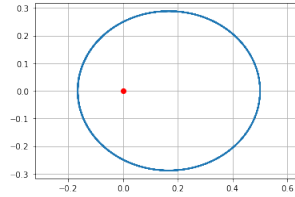
dt = 1/365      # jeden den
tmax = 2       # dva roky

x0 = 0.5       # jednotka délky je AU
y0 = 0
vx0 = 0        # jednotka délky je AU/rok
vy0 = 2*np.pi # kruhová rychlost pro r=1AU

U0 = np.array([x0, y0, vx0, vy0])

sol = solve_ivp(pohybova_rovnice_planety, (0,tmax), U0, max_step=dt)

plt.plot(sol.y[0], sol.y[1])
plt.plot([0],[0], 'or')
plt.axis('equal')
plt.grid()
plt.show()
```



Vidíme, že řešení soustavy diferenciálních rovnic předpokládá, že řešení má podobu $\vec{U}(t)$, které pro každé t nabývá hodnoty \mathbb{R}^n , kde n je počet řešených diferenciálních rovnic. Na druhé straně, z hlediska fyzikálního původu rovnic může být přirozené je mít v poli poskládané jinak.

Zatím jsme použili rozdělení \vec{U} na skalární proměnné:

```
x, y, vx, vy = U
```

Pokud bychom ale studovali pohyb hvězd v trojhvězdě (pro jednoduchost ve 2D), budeme v tomto přístupu mít například

```
x1, y1, x2, y2, x3, y3, vx1, vy1, vx2, vy2, vx3, vy3 = U
```

Počítat potom sílu mezi hvězdami může být nepřehledné. Je tu ale alternativa za použití konverze lineárně uspořádaného pole na matici nebo jinou vhodnou strukturu.

Nejprve si princip demonstrujeme s pomocí řetězců, poté jej použijme uvnitř funkce `pohybova_rovnice_hvezd` ke spočtení zrychlení

$$\vec{a}_i = -G \sum_{j \neq i} M_j \frac{\vec{x}_i - \vec{x}_j}{|\vec{x}_i - \vec{x}_j|^3}$$

Díky operaci `reshape` budeme moci zapsat \vec{x}_i jako `U[0, i]` a \vec{x}_j jako `U[1, j]`. Potom, co práci v této pohodlnější reprezentaci fázového prostoru skončíme, v rámci příkazu `return` provedeme konverzi zpět na vektor.

[125]: `import numpy as np`

```
vektorU = np.array( ['x1', 'y1', 'x2', 'y2', 'x3', 'y3', 'vx1', 'vy1', 'vx2', 'vy2', 'vx3', 'vy3'])
print('vektorU')
print( vektorU )
print( '\nvektorU.reshape((2,3,2))' )
print( vektorU.reshape((2,3,2)) )
```

```
vektorU
['x1' 'y1' 'x2' 'y2' 'x3' 'y3' 'vx1' 'vy1' 'vx2' 'vy2' 'vx3' 'vy3']
```

```
vektorU.reshape((2,3,2))
```

```
[[['x1' 'y1']
  ['x2' 'y2']
  ['x3' 'y3']]
```

```
 [['vx1' 'vy1']
  ['vx2' 'vy2']
  ['vx3' 'vy3']]
```

[114]: `from scipy.integrate import solve_ivp`
`import numpy as np`

(continues on next page)

(pokračujte na předchozí stránce)

```

import math

pocet_hvezd = 3
pocet_souradnic = 2

# součin hmotností hvězd a grav. konstanty (v jednotkách AU, rok, hmotnost_Slunce)
GM_hvezd = np.array([1,1,1]) * 4*math.pi**2

def pohybova_rovnice_hvezd(t, U):
    """
    Předpokládá U = [x1, y1, x2, y2, x3, y3, vx1, vy1, vx2, vy2, vx3, vy3] a pod.
    Vrací časovou derovaci dle Newtona.
    """

    U = U.reshape((2, pocet_hvezd, pocet_souradnic))
    dUdt = np.zeros_like(U)

    for i in range(pocet_hvezd):
        for j in range(i):
            rr = U[0,i] - U[0,j]
            r2 = rr @ rr
            r_3 = 1/(r2*math.sqrt(r2))

            dUdt[1,i] += - GM_hvezd[j] * rr * r_3
            dUdt[1,j] += + GM_hvezd[i] * rr * r_3

    dUdt[0] = U[1]

    return dUdt.reshape(2 * pocet_hvezd * pocet_souradnic)

dt = 1/365          # jeden den
tmax = 0.29         # dva roky

x_1, y_1 = 1, 0
vx_1, vy_1 = 0, -3.05216379919688

x_2, y_2 = -0.5, 0.3193874731184
vx_2, vy_2 = -7.16722802885, 1.52608189959845

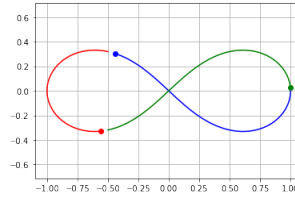
x_3, y_3 = -0.5, -0.3193874731184
vx_3, vy_3 = 7.16722802885, 1.52608189959845

U0 = np.array([x_1, y_1, x_2, y_2, x_3, y_3, vx_1, vy_1, vx_2, vy_2, vx_3, vy_3])

sol = solve_ivp(pohybova_rovnice_hvezd, (0,tmax), U0, max_step=dt)

plt.plot(sol.y[0], sol.y[1], sol.y[0,-1:], sol.y[1,-1:], 'o', color='blue')
plt.plot(sol.y[2], sol.y[3], sol.y[2,-1:], sol.y[3,-1:], 'o', color='red')
plt.plot(sol.y[4], sol.y[5], sol.y[4,-1:], sol.y[5,-1:], 'o', color='green')
plt.axis('equal')
plt.grid()
plt.show()

```



Cvičení: Zapište ekvivalentní kód pomocí řezů, tedy bez reshape. Například \vec{x}_i lze totiž zapsat jako $U[2*i:2*i+2]$.

Protože požité počítační polohy a rychlosti vedou na zajímavý pohyb, vytvoříme si ještě jeho animaci. Konverze obrázků do podoby videosouboru a jeho zobrazení jsou přenechány funkcím z knihoven matplotlib a IPython. Kód spoléhá na ten předchozí ohledně načtení knihoven, definice funkce pohybova_rovnice_hvezd, hodnoty dt a počátečních dat pro pohyb hvězd trojhvězdy U0. Navíc

- Necháme diferenciální rovnici vyřešit na časovém úseku zhruba odpovídajícímu jedné periodě děje (pohyb trojhvězdy není obvykle periodický, zde jde o výjimku). Vhodně zvoleným parametrem `t_eval` funkci `solve_ivp` požádáme, aby vrátila ekvidistantně navzorkované řešení. Co vzorek, to jeden snímek animace, viz příkaz `t_samples = np.linspace(0, tmax, frame_count)`.
- V cyklu `for` připravíme seznam jednotlivých snímků
- Do každého snímku vložíme výsledek čtyřech příkazů `plot`. Ty namalují jednak celkovou trajektorii šedě a pak také barevně tři úseky odpovídající dráze hvězd během určitého časového intervalu. Tak se na animaci dá znázornit rychlost pohybu. (Zvolený interval odpovídá indexům `i0:i1`.)
- Data pro malování polohy hvězd a stop si za pomoci řezů před malováním připravíme (proměnné `point` a `trace`). Díky tou jsou argumenty příkazů `plot` srozumitelnější.
- Seznam snímků pak nechá převést do videosouboru a zobrazit ve výsledkové části.

```
[128]: # import dalších knihoven, nepoužitých v minulém příkladě
import matplotlib.animation as animation
from IPython.display import HTML

# parametry animace
fps = 50          # snímku za sekundu
duration = 5      # trvání 5 sekund
frame_count = int(duration*fps)

# pořídíme řešení na delším časovém intervalu
tmax = 0.90
t_samples = np.linspace(0, tmax, frame_count)
sol = solve_ivp(pohybova_rovnice_hvezd, (0,tmax), U0, max_step=dt, t_eval=t_samples)

# příprava na vytvoření animace
frames = []      # vytvořím prázdný seznam snímků
fig = plt.figure() # vytvořím prázdný obrázek
plt.grid()
plt.axis('equal')

# vytvoření animace snímek po snímku
for i in range(len(t_samples)):
    i0 = max(0,i-fps//2)
    i1 = i+1
```

(continues on next page)

(pokračujte na předchozí stránce)

```

trace = sol.y[:,i0:i1]
point = sol.y[:,i:i1]

this_frame = plt.plot(sol.y[0], sol.y[1], linewidth=0.4, color='gray')
this_frame += plt.plot(trace[0], trace[1], point[0], point[1], 'o', color='blue')
this_frame += plt.plot(trace[2], trace[3], point[2], point[3], 'o', color='red')
this_frame += plt.plot(trace[4], trace[5], point[4], point[5], 'o', color='green')

frames.append(this_frame)

# snímky složíme do animace
ani = animation.ArtistAnimation(fig, frames, interval=1000//fps)
# ještě po sobě uklidíme, protože matplotlib nechal otevřený obrázek
plt.close()

# zobaržíme animaci v buňce Jupyteru
HTML(ani.to_html5_video())

```

[128]: <IPython.core.display.HTML object>

Jako zajímavost uvedme, že kromě animací můžeme v Jupyterových sešitech přehrávat i audiosignál. V následujícím příkladě je opět generován jako řešení diferenciální rovnice. Soustava tří oscilátorů je vybuzena úderem v podobě impulsu ve tvaru $1/[(t-t_0)^2 + \delta t^2]$, který má maximum v čase t_0 a šířku řádově δt .

```

[71]: from scipy.integrate import solve_ivp
import numpy as np
import matplotlib.pyplot as plt
import math
import IPython

def dif_rovnice(t, U):
    """
    Jednoduchá soustava 6 dif. rovnic pro 2 harmonické a jeden anharmonický
    oscilátor s lineárními vazbami, tlumením a impulsním buzením.
    """
    x, y, z, vx, vy, vz = U

    ax = - (2*math.pi*440)**2 * (x-0.1*(y-z)) - 8*vx
    ay = - (2*math.pi*440*4/3)**2 * (y-0.1*(x-z)) - 4*vy
    az = - (2*math.pi*20)**2 * (z+z**3) - 24*vz

    az += 1/((t-0.2)**2+1/4000**2)          # impuls okolo t=0.2

    return [vx, vy, vz, ax, ay, az]

f_sampling = 22050                          # vzorkování 1/2 f_CD
dt = 1/f_sampling
tmax = 2                                    # dvě sekundy
t_samples = np.arange(0,tmax,dt)           # ekvidistantní vzorkování pro audio

U0 = np.zeros(6)

```

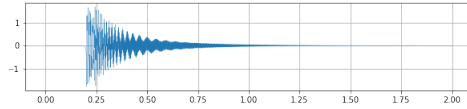
(continues on next page)

(pokračujte na předchozí stránce)

```
sol = solve_ivp(dif_rovnice, (0,tmax), U0, max_step=dt, t_eval=t_samples)

plt.figure(figsize=(10,2))
plt.plot(sol.t, sol.y[0], linewidth=0.3)
plt.grid()
plt.show()

IPython.display.Audio(sol.y[1], rate=f_sampling)
```



```
[71]: <IPython.lib.display.Audio object>
```

```
[ ]:
```

Zatím jsme používali

- jednoduché datové typy (mj. `int`, `float`, `bool`)
- specializované typy (`str`, textový soubor, objekt reprezentující obrázek atp.)
- pole (`numpy.ndarray`)
- seznamy (`list`, `tuple`)

Např. pole představovala jasnou ukázkou situace, kdy z jednoduššího typu budujeme typ komplikovanější. Také jsme viděli, že typ pole otevřel možnost zapsat širokou oblast nových algoritmů. Existují ale další typy, opět otevírající možnost zapisovat pohodlně další oblasti algoritmů. S některými se v této kapitole seznámíme.

18.1 Asociativní pole (`dict`)

V Pythonu jsme zvyklí, že důležité funkce jsou již hotové, takže stačí je jen použít. Podobně je tomu i s datovými strukturami. Jedno z nich jsou tzv. asociativní pole v Pythonu zvané `dict`. Zatímco u polí a seznamů jsme položku určovali její polohou, tedy indexem, o kterou položku máme zájem se u asociativních polí určuje klíčem. Tedy místo `x[2]` můžeme psát `souradnice["Brno"]`. Z hlediska teorie i praxe je důležité, že při růstu počtu položek pole se přístup k prvku (čtení i zápis) chová jako $O(1)$ nikoli jako $O(N)$.

Klíč může být různého druhu, jako příklad použití použijeme klíč v podobě dvojice (*tuple*) řetězců:

```
[ ]: vysledky_zkousek = {} # založíme prázdný slovník

def zapis_vysledek_zkousky(jmeno, prijmeni, vysledek):
    key = (jmeno, prijmeni)

    if key in vysledky_zkousek:
        vysledky_zkousek[key].append(vysledek)
    else:
        vysledky_zkousek[key] = [vysledek]
```

(continues on next page)

```
def slozil_zkousku(jmeno, prijmeni):
    key = (jmeno,prijmeni)

    if key not in vysledky_zkousek:
        return False

    return min(vysledky_zkousek[key]) < 4

zapis_vysledek_zkousky('Petr', 'Novák', 1)
zapis_vysledek_zkousky('Ivan', 'Dlouhý', 4)
zapis_vysledek_zkousky('Karel', 'Holub', 2)
zapis_vysledek_zkousky('Ivan', 'Dlouhý', 2)

print(f"{slozil_zkousku('Karel', 'Holub')} = {}")
print(f"{slozil_zkousku('Jan', 'Kolmý')} = {}")
```

vysledky_zkousek

```
slozil_zkousku('Karel', 'Holub') = True
slozil_zkousku('Jan', 'Kolmý') = False
```

```
{('Petr', 'Novák'): [1], ('Ivan', 'Dlouhý'): [4, 2], ('Karel', 'Holub'): [2]}
```

- přítomnost klíče v seznamu testujeme operací `in`
- novou hodnotu přidáváme přiřazením `my_dict[key] = value`
- stejně modifikuje existující položku slovníku
- zmazání položky například `del my_dict['key']`
- operace se slovníky zahrnují mj. spojování `{'a':1, 'b':3} | {'b':4, 'c':8} == {'a': 1, 'b': 4, 'c': 8}`

Na přednášce si povíme něco o **hešování**. V Pythonu je možnost použít proměnnou/hodnotu jako klíč dána jejím typem, výše jsme například museli jméno a příjmení spojit do proměnné typu tuple nikoli však list.

Důležité je, že slovník můžeme inicializovat přiřazením, například

```
barvy = {
    'aquamarine': '#7FFFD4',
    'azure': '#F0FFFF',
    'black': '#000000',
    'blue': '#0000FF',
    'brown': '#A52A2A',
    'cyan': '#00FFFF',
    'darkblue': '#00008B',
    'darkgray': '#A9A9A9'
}
```

Velmi podobné asociativním polím jsou v Pythonu množiny. U nich není k danému klíči k dispozici jiná informace, než zda do množiny patří či nikoli.

```
[3]: n = 5
print(f"Zadejte {n} různých celých čísel.")
```

(continues on next page)

(pokračujte na předchozí stránce)

```

numbers = set()
for k in range(n):
    a = int(input("Celé číslo: "))
    if a in numbers:
        print(f"Číslo {a} jste již zadali. Máte špatnou paměť.")
        break

    numbers.add(a) # zařaď a do množiny numbers

```

Zadejete 5 různých celých čísel.
Číslo 2 jste již zadali. Máte špatnou paměť.

18.2 Složené datové typy

18.2.1 Úvod

Již jsme viděli, že při použití doktríny strukturovaného programování dokážeme s použitím překvapivě málo příkazů (=, if, while, for, ...) a vhodným rozdělením kódu na funkce řešit i komplikované problémy. Stejná doktrína předpokládá, že složitější datové struktury se mohou skládat z jednodušších a programovací jazyk to vhodným způsobem podporuje.

Pokud bychom v přednášce používali jazyk určený pro výuku programování, jako je například Pascal, potkali bychom toto již dávno. V Pythonu jsme aktivně použili zatím jedině strukturování dat: pole (a jim podobné seznamy, tuple, řetězce). Pokud jsme v jedné proměnné chtěli uložit více druhů informace, rozlišovali jsme je hodnotou indexu.

Následující kód demonstruje na příkladě sady barevných kruhů komplikace související s tím, že pokud chceme používat pole z numpy, musíme mít všechny položky stejného typu. Kolečko je přitom zadáno čtyřmi parametry:

- x-ovou souřadnicí středu
- y-ovou souřadnicí středu
- poloměrem
- barvou

```

[ ]: import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import CSS4_COLORS # https://matplotlib.org/stable/gallery/color/
↳named_colors.html

def rgbstr_to_int(s):
    "Převéde barvu '#ff00ff' na celé číslo"
    assert s[0] == '#' and len(s) == 7
    return int(s[1:],16)

def disk(x, y, size, color):
    if color[0] != '#':
        color = CSS4_COLORS[color] # 'red' -> '#ff0000' atp.

    return np.array([x, y, size, rgbstr_to_int(color)])

```

(continues on next page)

```

def pappus_disk(n):
    """viz. https://en.wikipedia.org/wiki/Pappus\_chain"""
    r = 0.6

    if n == 'inner':
        return disk(r/2, 0, r/2, 'slategray' )
    if n == 'outer':
        return disk(1/2, 0, 1/2, 'lightgray' )

    w = n*n*(1-r)**2+r
    x = r*(1+r)/(2*w)
    y = n*r*(1-r)/w
    s = (1-r)*r/(2*w)

    return disk(x, y, s, 'lightcoral' if n%2 else 'steelblue' )

def disk_plot(disks, imgsize=6):
    "Namaluje sadu koleček. Parametr imgsize určuje velikost obrázku"

    xmin, xmax = min(disks[:,0]-disks[:,2]), max(disks[:,0]+disks[:,2])
    ymin, ymax = min(disks[:,1]-disks[:,2]), max(disks[:,1]+disks[:,2])
    xcenter, ycenter = (xmax+xmin)/2, (ymax+ymin)/2

    halfwidth = 0.55 * max(ymax - ymin, xmax - xmin)

    factor = imgsize*111 # plt.scatter používá svérázné jednotky velikosti
    sizes = (factor/(2*halfwidth) * disks[:,2])**2 # a velikost udává plochou
    colors= [f'#{int(c):06x}' for c in disks[:,3]] # konverze na #ff8800 atp.

    plt.figure(figsize=(imgsize, imgsize))
    plt.xlim(xcenter - halfwidth, xcenter + halfwidth)
    plt.ylim(ycenter - halfwidth, ycenter + halfwidth)
    #plt.axis(False)

    plt.scatter(disks[:,0], disks[:,1], s=sizes, c=colors )

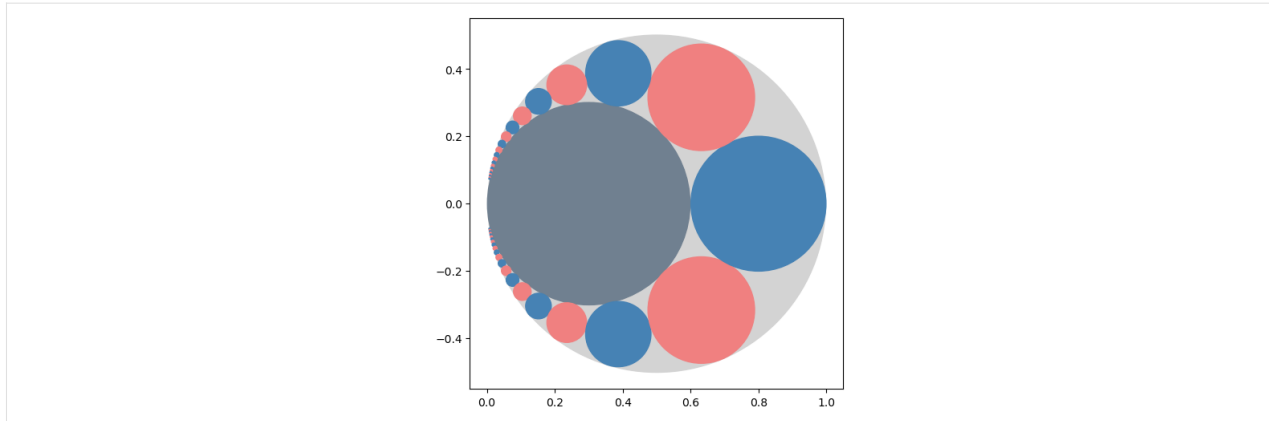
    plt.show()

# hlavní program
# - vytvoří sadu koleček
# - uloží je do proměnné disks
# - a namaluje je

disks = np.array( [pappus_disk('outer'), pappus_disk('inner')] +
                  [pappus_disk(n) for n in range(-20,21)] )

disk_plot(disks)

```



Barva ale není přirozeně dána reálným číslem a tak v kódu postupně konvertujeme barvu zadanou jako řetězec do podoby celého čísla a následně reálného čísla. Jako reálné číslo ji uložíme v proměnné `disk`, ale při vykreslování číslo barvy zase převádíme zpět do podoby řetězce. To vše proto, aby veškerá informace byla uložena v jediné proměnné. To věci natolik komplikuje, že obvykle

- použijeme místo polí `numpy.ndarray` typ `list`, jehož položky mohou být různého typu, ale ztrácíme komfort řezů ve více indexech, jako je např. `disks[:, 0]` vybírající x-ové souřadnice všech bodů;
- pro podobnou sadu koleček použijeme tři proměnné: jedna v podobě pole dimenze $n \times 2$ reálných čísel obsahuje souřadnice středů, další je pole n poloměrů a poslední pak pole n barev;
- zavedeme nový typ (viz dále).

Navic je kód psán nebezpečně, kdybychom přidali ke každému kolečku další atribut, např. hmotnost, mohly by se některé hodnoty indexů změnit a my bychom museli kód procházet a všechny `[:, 3]` měnit na `[:, 4]`. Tomu lze zabránit použitím konstant ve významu hodnot indexů příslušejících jednotlivým položkám. Správně by se takové konstanty měly v Pythonu psát velkými písmeny, takže bychom pak měli

```
DISK_COORD_X, COORD_Y, DISK_RADIUS, DISK_COLOR = 0, 1, 2, 3
...
xmin, xmax = min(disks[:,DISK_COORD_X]-disks[:,DISK_RADIUS]), max(disks[:,DISK_COORD_
→X]+disks[:,DISK_RADIUS])
```

Takto napsaný kód připraven na zavedení další položky, např. `DISK_MASS`. Tlak na 'rychlé' psaní kódu ale často vede k opomenutí podobných rad a pak dlouho hledáme, kde nám zbyl neopravený index `[3]`.

18.2.2 Datový typ záznam/struktura

Ve fyzice většinou vystačíme (či budeme nuceni vystačit) s postupem uvedeným výše. Přesto je rozumné vědět, že již dlouho nabízejí programovací jazyky možnost vytvořit nový datový typ sdružující do jedné entity všechny potřebné položky, ať už jsou jakéhokoli typu a při použití takové proměnné lze pak

- uvést jen identifikátor proměnné a tím zacházet s uvažovanými daty jako celkem, např. funkce `overlap(d1, d2)` zjistí, zda se dva disky překrývají.
- uvést identifikátor proměnné následovaný tečkou a specifikací atributu, např. podmínka `d1.radius > abs(d1.y)` testuje, zda diskem `d1` prochází osa `x`.

V Pythonu je tohle bohužel komplikováno principem, že přiřazení znamená odkaz na existující data a tak zavedení nového typu oproti běžným jazykům neznamená popis uložení položek v novém typu, ale rovnou návod jak data takového typu vytvořit.

Minimální kód vytvářející proměnnou popisující disk podle minulého příkladu je:

```
[ ]: class ColorDisk:
    x = 0
    y = 0
    radius = 0
    color = '#000000'

def disk(x, y, size, color):
    d = ColorDisk()
    d.x = x
    d.y = y
    d.radius = size
    d.color = color

    return d

d1 = disk(0,0,1,'blue')
```

Povšimněte si příkazu `d = ColorDisk()`. Ten ukazuje, že zavedením `class ColorDisk` jsme také založili stejnojmennou funkci, která se použije, když potřebujeme vytvořit data daného typu. To že jde o nový typ indikuje výsledek výrazu `type(d1)`.

Vidíme, že v tomto příkladě se zbytečně nejprve nastaví `x` na hodnotu nula a teprve poté na požadovanou hodnotu. To proto, že jsme nepoužili podstatu objektového programování a spolu s novou třídou jsme nezavedli její konstruktor/inicializátor `__init__`. Podobně, abychom mohli k výpisu obsahu proměnné používat `print(d1)`, potřebovali bychom zavést metodu `__str__`. S tím vším se seznámíte v nějakém kurzu jazyka Python, do našich 'základů' se *objektové programování* již nevejde.

Je zajímavé, že i zběhlí uživatelé jazyka Python toto považovali za natolik komplikované (otravné), že pro zjednodušení zavedli automatizaci těchto kroků za použití knihovny `dataclasses`. Protože její použití činí zavedení nového typu velmi podobným jazykům jako je C nebo Pascal, použijeme tuto knihovnu a ukážeme si na ní práci s nově zavedeným typem zavádějícím strukturu/záznam.

```
[ ]: from dataclasses import dataclass

@dataclass      # proved' výše zmíněnou automatizaci
class ColorDisk:
    x : float
    y : float
    radius : float
    color : str = 'black'

d1 = ColorDisk(0, 0, 1, 'blue')
d2 = ColorDisk(1, 0, 0.5)

print(d2)

ColorDisk(x=1, y=0, radius=0.5, color='black')
```

Zde vidíme, že nový typ je složený z položek, které

- mají své jméno (uvádí se za tečkou, tedy např. `d1.x`);
- mají svůj typ;
- se inicializují v rámci volání `ColorDisk(...)` hodnotami argumentů v pořadí, v jakém jsou definovány příkazem `class`;
- mohou podobně jako v případě argumentů funkcí mít výchozí hodnotu, takže si ve druhém případě ušetříme uvedení barvy kolečka.

Konverze na řetězec, kterou automaticky volá příkaz `print` pak vrací srozumitelnou informaci o dané proměnné.

18.3 Příklad na použití vlastní datové struktury

```
[ ]: """
Program řeší srážení volně se pohybujících kruhových disků.
"""

import math
from dataclasses import dataclass
import matplotlib.pyplot as plt
from matplotlib import animation
from IPython.display import HTML

# každý z disků má 2D polohu a rychlost, hmotnost, poloměr a barvu
@dataclass
class ColorDisk:
    x : float
    y : float
    vx : float
    vy : float
    r : float
    m : float
    color : str = 'blue'

DISK_FIXED = 1e40 # dostatečně těžký disk se při srážce nepohne

def bounce(a, b):
    """
    Modifikuje rychlosti v důsledku plně elastického odrazu
    viz https://en.wikipedia.org/wiki/Elastic\_collision#Two-dimensional\_collision\_with\_
    ↪two\_moving\_objects
    """

    delta_r2 = (a.x-b.x)**2 + (a.y-b.y)**2

    assert abs( delta_r2 - (a.r+b.r)**2 ) < 1E-10, 'Disky se nedotýkají!'

    deltax_dot_deltar = (a.vx-b.vx)*(a.x-b.x) + (a.vy-b.vy)*(a.y-b.y)
    factor = 2*deltax_dot_deltar/(delta_r2 * (a.m+b.m))

    a.vx = a.vx - b.m*(a.x-b.x)*factor
```

(continues on next page)

```

a.vy = a.vy - b.m*(a.y-b.y)*factor

b.vx = b.vx + a.m*(a.x-b.x)*factor
b.vy = b.vy + a.m*(a.y-b.y)*factor

def time_to_bounce(a, b):
    """
    Výpočet času zbývajícího do srážky dvou disků.
    Pokud se v budoucnu nepotkají, vrací funkce záporné číslo.
    """

    d2 = (a.r+b.r)**2
    x2 = (a.x-b.x)**2 + (a.y-b.y)**2

    if x2-d2 < -1E-7:
        # povolujeme protínající se těžké disky tvořící arénu
        assert a.m == b.m == DISK_FIXED, f'Disky se protínají! ({x2-d2=} {a=} {b=})'
        return -1

    v2 = (a.vx-b.vx)**2 + (a.vy-b.vy)**2
    if v2 == 0:
        return -1 # disky se vůči sobě nepohybují

    deltav_dot_deltar = (a.vx-b.vx)*(a.x-b.x)+(a.vy-b.vy)*(a.y-b.y)
    discriminant = deltav_dot_deltar**2 - v2*(x2-(a.r+b.r)**2)
    if discriminant <= 0:
        return -1 # disky se míjejí

    return (-deltav_dot_deltar-math.sqrt(discriminant))/v2

#####
###   Vykreslení disků
#####

def plot_frame():
    """Vykresli disky. Pro jednoduchost založeno na funkci pyplot.scatter."""
    plt.gca().clear() # musíme smazat minulý obrázek

    plt.xlim(-1, 1)
    plt.ylim(-1, 1)
    plt.axis(False)

    factor = 111*imgsize/2 # plt.scatter používá svérázné jednotky velikosti

    displayed_disks = disks[4:]

    x = [d.x for d in displayed_disks]
    y = [d.y for d in displayed_disks]
    colors = [d.color for d in displayed_disks]

```

(continues on next page)

(pokračujte na předchozí stránce)

```

sizes = [(factor * d.r)**2 for d in displayed_disks]

plt.scatter(x, y, s=sizes, c=colors )

#####
###   Evoluce
#####

def advance_time(new_time):
    """Posune disky podle jejich rychlosti a upraví čas.
    Předpokládá, že za dobu dt nedojde ke srážce."""
    global time

    dt = new_time - time
    for p in disks:
        p.x = p.x + p.vx*dt
        p.y = p.y + p.vy*dt

    time = new_time

#####
###   Animace
#####

fps = 16.0
time = 0
next_bounce_time = -1 # signalizuj, že je třeba jej spočítat
m_next, n_next = -1, -1

def draw_frame(i):
    """Vykreslení i-tého snímku animace."""

    # Bohužel plt.scatter nefunguje s ArtistAnimation a tak se musí používat metoda
    # FuncAnimation, která vyžaduje callback pro každý snímek.
    # funkce draw_frame proto vždy posune čas do okamžiku dalšího snímku

    global time, next_bounce_time, m_next, n_next

    frame_time = i * dt_frame
    assert frame_time >= time, "neumím nechat plynout čas nazpátek"

    # 1. vyřeš srážky předcházející vykreslení snímku
    while next_bounce_time < frame_time:
        # 1.1 proved' pohyb disků a následný odraz
        dt = next_bounce_time - time
        if dt >= 0: # ignoruj signalizaci neznalosti času srážky
            advance_time(next_bounce_time)
            bounce(disks[m_next], disks[n_next])

        # 1.2 spočti čas příští srážky
        dt_min = 1e100 # nikdy se nesrazí

```

(continues on next page)

```

for m in range(len(disks)):
    for n in range(m):          # stačí testovat n < m
        dt = time_to_bounce(disks[m], disks[n])
        if dt >= 0 and dt < dt_min: # najdi nejmenší nezáporné dt
            dt_min, m_next, n_next = dt, m, n

    next_bounce_time = time + dt_min

advance_time(frame_time) # 2. posuň čas do okamžiku snímku
plot_frame()             # 3. vykresli polohu disků

def some_color(i,j):
    return f'#{11247*i+13697*j & 0xf8f8f8:06x}'

#####
###   Seznam disků
#####
b0 = 100
r0 = b0 - 1
disks = [ColorDisk(-b0,0,0,0,r0,DISK_FIXED),
         ColorDisk(+b0,0,0,0,r0,DISK_FIXED),
         ColorDisk(0,-b0,0,0,r0,DISK_FIXED),
         ColorDisk(0,+b0,0,0,r0,DISK_FIXED)]

h = 1/16
for i in range(-6,7,3):
    for j in range(-6,7,3):
        disks.append( ColorDisk(i*h, j*h, 0, 0, h, 1, some_color(i,j)) )

disks[4].vx = 5
disks[4].vy = 2

#####
###   Vytvoření animace
#####
imgsize = 6          # proměnnou používá i plot_frame()
fig = plt.figure(figsize=(imgsize, imgsize))

time_factor = 0.2    # jak rychle plyne čas v simulaci
dt_frame = (1/fps) * time_factor

ani = animation.FuncAnimation(fig, draw_frame, frames=400, interval=50)

tmp = ani.to_html5_video()
plt.close()
HTML(tmp)
<IPython.core.display.HTML object>

```

Poznámky:

- Vzpomeňte si, proč uvnitř funkce `advance_time` je `global time` ale nikoli `global disks`, když stejně jako

čas, funkce mění i polohy disků.

- Funkce `some_color` generuje řetězec nějaké barvy v podobě `"#rrggbb"`. Vhodně zvolená konstanta v logickém součinu zařídí, že formátovaný výstup celého čísla závislého na `i` a `j` má šest hexadecimálních cifer a zároveň disk nebude úplně bílý, aby se ztratil na pozadí.
- Aby nebyl kód dlouhý používáme knihovnu `matplotlib`. O vykreslování se stará `scatter`, animaci přenecháváme funkci `FuncAnimation`. Bohužel,

18.4 Rozbalování polí představujících struktury

Ještě se vraťme k situaci, kdy nás okolnosti nutí používat pro uložení podobných dat pole `numpy.ndarray`. V takovém případě můžeme použít rozbalení (*unpacking*) do sady vhodně pojmenovaných proměnných a psát:

```
def bounce(a, b):
    # založíme a_x, ...
    a_x, a_y, a_vx, a_vy, a_r, a_m, a_color = a
    b_x, b_y, b_vx, b_vy, b_r, b_m, b_color = b
    # používáme je místo a[0], ...
    delta_r2 = (a_x-b_x)**2 + (a_y-b_y)**2
    ...
    b_vy = b_vy + a_m*(a_y-b_y)*factor
    # změněné hodnoty nutno uložit zpět do odkazem předaných polí
    a[2:4] = a_vx, a_vy
    b[2:4] = b_vx, b_vy
```

tedy ve výpočtech, používat srozumitelný zápis `b_vy` místo nebezpečného `b[2]` nebo nepohodlného `b[VX]`.

Podobný trik jsme použili při řešení soustav obyčejných diferenciálních rovnic, kde numerické metody vyžadují, abychom pracovali s polem, ale jeho jednotlivé složky mají jednu význam poloh, jindy rychlostí, takže rozbalení / sbalení mělo podobu

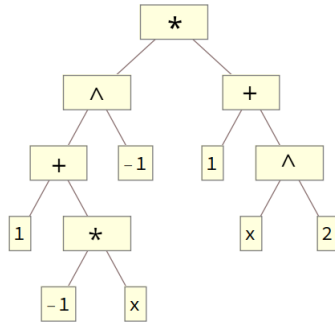
```
def pohybova_rovnice_planety(t, U):
    "Předpokládá U = [x,y,vx,vy]. Vrací časovou derivaci U dle Newtona."
    x, y, vx, vy = U
    ...
    return np.array([vx, vy, ax, ay])
```

18.5 Dynamické datové struktury

Již pole a seznamy umožňovaly měnit počet jejich prvků (ale u polí se to nedoporučuje). Pokud v závislosti na stavu výpočtu nabývá seznam různé délky, jde o nejjednodušší příklad dynamické datové struktury. V historii programování se nicméně objevily ještě složitější datové struktury. Důležitou podmnožinu tvoří stromy. Potkáme se s ním často, například výraz

$$\frac{1+x^2}{1-x}$$

lze (jak to dělá SW Mathematica) zapsat jako stromový graf



Program, který umí pracovat s takovými výrazy musí dokázat v nějaké proměnné udržet informaci o libovolném výrazu, takže není možné mít fixní datovou strukturu, jako by například bylo racionální číslo (vždy jeden číselník a jeden jmenovatel). Navíc, při úpravách mohou nové výrazy vznikat a nebo zanikat.

Zatímco v jazycích, jako je C, musíme věnovat hodně pozornosti již navrženému vhodnému typu a pak i práci s ním, jazyk Python nám ledačes skryje.

Zapomeneme na mocnění a zkusíme vystačit s operacemi +, * a operandy v podobě celých čísel, proměnných a funkcí.

```
[9]: # Ukázka reprezentace výrazu jako stromu. Implementuje jen malou podmnožinu výrazů:
#
# výraz je
#   celé číslo
#   'název_proměnné'
#   ['+', výraz, výraz (, výraz , ...)]
#   ['*', výraz, výraz (, výraz , ...)]
#   ['název_fce', výraz (, výraz , ...)]

def to_str(e):
    "Vrátí řetězec odpovídající výrazu e"

    if type(e) == int:
        return str(e) # 3

    if type(e) == str:
        return e # x

    assert type(e) == list

    if e[0] == '+': # x + 3*z
        return '+' + '.join([to_str(a) for a in e[1:]])

    if e[0] == '*': # x*y nebo x*(y+z)
        return '*'.join(['('+to_str(a)+')' if type(a)==list else to_str(a) for a in e[1:
        →]])

        # sin(x)
        return e[0]+'('+'.join([to_str(a) for a in e[1:]])+')'

def diff(e,x):
    "Vrátí parciální derivaci výrazu e podle x"

    if type(e) == int:
```

(continues on next page)

(pokračujte na předchozí stránce)

```

    return 0

if type(e) == str:
    return int(e == x)

assert type(e) == list

if e[0] == '+':
    de = [diff(a,x) for a in e[1:]]
    return ['+'] + de

if e[0] == '*':
    de = [diff(a,x) for a in e[1:]]
    s = ['+']
    for i in range(len(de)):
        s1 = e[1:] # řezý typu list jsou kopie
        s1[i] = de[i]
        s.append(['*'] + s1)
    return s

assert False, "Derivace funkcí nejsou implementovány"

def int_product(a):
    p = 1
    for x in a:
        p = p * x

    return p

def normal(e):
    "Provede částečnou normalizaci výrazu, např 1*(x-3)+4 -> x+1, chybí mj. x+x -> 2*x"
    if type(e) != list:
        return e

    f = e[1:]
    for i in range(len(f)):
        f[i] = normal(f[i])

    if e[0] != '+' and e[0] != '*':
        return [e[0]] + f

    if e[0] == '+':
        ff = []
        for a in f:
            if type(a)==list and a[0]=='+':
                for b in a[1:]: # x*(y*z) -> x*y*z
                    ff.append(b)
            else:
                ff.append(a)

        f1 = sorted([a for a in ff if type(a)==str])

```

(continues on next page)

```

f2 = [a for a in ff if type(a)==list]
f3 = [a for a in ff if type(a)==int]

assert(len(f1)+len(f2)+len(f3)==len(ff))

f3=[sum(f3)]
if f3[0] == 0 and len(f1)+len(f2)>0:
    f3 = []

f = f1+f2+f3
if len(f)==1:
    return f[0]
return ['+'] + f

assert e[0] == '*'

ff = []
for a in f:
    if type(a)==list and a[0]=='*':
        for b in a[1:]:
            ff.append(b)
    else:
        ff.append(a)

f1 = sorted([a for a in ff if type(a)==str])
f2 = [a for a in ff if type(a)==list]
f3 = [a for a in ff if type(a)==int]

assert(len(f1)+len(f2)+len(f3)==len(ff))

f3 = [int_product(f3)]
if f3[0] == 0:
    return 0          # 0*x -> 0

if f3[0] == 1 and len(f1)+len(f2)>0:
    f3 = []          # 1*x -> x

f = f3+f1+f2
if len(f)==1:
    return f[0]
return ['*'] + f

expr1 = ['*',
        ['+', 'x', -2],
        ['+', 'x', 2]
        ]
expr2 = diff(expr1, 'x')
expr3 = normal(expr2)

print( 'expr1 = ', to_str(expr1) )

```

(continues on next page)

(pokračujte na předchozí stránce)

```
print( 'expr2 = ', to_str(expr2) )
print( 'expr3 = ', to_str(expr3) )

expr1 = (x + -2)*(x + 2)
expr2 = (1 + 0)*(x + 2) + (x + -2)*(1 + 0)
expr3 = x + x
```

Rozmyslete si

- jak zařídit, aby výraz jako $x + y + x$ byl normalizován do podoby $2x + y$,
- jak by vypadala funkce `subs(e, d)`, která by do výrazu e za proměnné dosadila hodnoty (případně výrazy) podle tabulky d , např: `subs(expr1, {'x':1, 'y':2})`,
- co vše by se muselo doplnit, aby bylo možno spočítat derivace výrazů obsahujících funkce.

Nám zápis výrazu umožnil demonstrovat na konkrétním příkladě, co to je dynamická datová struktura. Počítačová manipulace s algebraickými výrazy je ovšem seriózní obor, který se vyvíjí již mnoho desetiletí. V některých oblastech fyziky *počítačová algebra* nabízí možnost zvládat vzorečky s miliony členů. Mezi knihovnamy Pythonu je balík `SymPy`, studenti MFF UK mají nárok na instalaci ještě mnohem rozsáhlejšího SW `Mathematica`.

 Jak zrychlit výpočet

- Pokročilejší matematika a fyzika:

Na příkladě výpočtu numerické hodnoty π uvidíme, že zrychlení může být ohromné:

- Pokud bychom počítali podle vztahu

$$\pi = 4 \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

tak ten má chybu $\approx \pm 1/n$, kde n udává horní mez u které ukončíme sčítání řady. Pro deset cifer bychom tak potřebovali deset miliard sčítanců.

- Ukazovali jsme si tzv. Richardsonovu extrapolaci. Ta zde funguje a polynomiální extrapolací hodnot sumy pro 30, 60, 90, 120, 150 sčítanců získáme správně 10 cifer π . (Viz též naše ukázka Rombergovy kvadratury)
- Ramanujan navrhl řadu

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{99^2} \sum_{k=0}^{\infty} \frac{(4k)!}{k!^4} \frac{26390k + 1103}{396^{4k}}$$

u níž stačí vzít první dva sčítance a máme 13 cifer π správně.

- Na úrovni programování je obdobou lepšího vzorečku lepší algoritmus. (Viz quicksort níže.)
- Výjimečně pomůže lepší překladač (tohle se týká Pythonu, viz numba)
- Poslední, ovšem velmi pracné řešení představuje paralelismus. Mirný (váš domácí počítač má možná i deset jader) nebo masivní (graf. karty a clustery počítačů).

19.1 Urychlení běhu programů Pythonu s použitím knihovny numba

Připomeňme si, že Python obětoval rychlost vykonávání kódu vyšším cílům. Proto zatímco procedura v jazyce C

```
int x;

void zvetsi_x(int n)
{
    x = x + n;
}
```

přeloží proceduru do strojového kódu o dvou instrukcích procesoru - jedné zařizující sčítání `add variable_x`, `register_edi` a jedné pro návrat z procedury `ret`, musí být Python připraven u kódu

```
def zvetsi_x(n):
    global x
    x = x + n
```

zvětšit libovolný typ globální proměnné `x` podporující sčítání:

[5]: `import numpy as np`

```
def zvetsi_x(n):
    global x
    x = x + n

x = 40
zvetsi_x(2)
print(x)

x = np.zeros(10)
zvetsi_x(2)
print(x)
```

```
42
[2. 2. 2. 2. 2. 2. 2. 2. 2. 2.]
```

Výchozí interpret jazyka Python tzv. CPython tuto přitěžující (zpomalující) okolnost bere jako danost s tím, že pro výkon programu klíčové funkce se nepiší v Pythonu, ale používají se jejich varianty z nějaké vhodné dobře optimalizované knihovny.

Existují ale nástroje, které se snaží vykonávání příkazů Pythonu přiblížit procesoru. My v této přednášce pro jednoduchost použití volíme funkci `njit` z knihovny `numba`. Ta pro daný typ argumentů vyrobí přeloženou variantu funkce. K času běhu prvního volání funkce se tak přičítá čas potřebný pro překlad, při dalších voláních se ale její rychlost za příznivých okolností může přiblížit ekvivalentnímu kódu v C. Použití `njit` má však mnoho omezení. U výše uvedeného příkladu například odmítne modifikovat globální proměnnou. Pokud překlad pomocí `njit` selže, dozvíme se o tom a máme šanci kód upravit.

To, že zrychlení pomocí `njit` je podstatné, demonstruje i následující kód pro výpočet polohy těžiště metodou Monte Carlo. I když generování náhodných čísel voláním funkce `random.random()` zabere hodně času, stále `njit` přinese 14x rychlejší výpočet.

19.2 Monte Carlo II (ukázka použití numba.njit)

Již jsme viděli, že náhodná čísla vytvářená v počítači ve velkém počtu umožní spočítat nenáhodný výsledek (například určitý integrál). Ještě jednou se k tomu vrátíme na příkladu výpočtu těžiště objektu. Objasníme si znovuy princip metody a navíc porovnáme dva styly, jak psát příslušný kód.

Budeme počítat polohu těžiště polokoule z homogenního materiálu. V principu jde o výpočet

$$\vec{x}_T = \frac{\int_V \vec{x} dm}{\int_V dm}, \quad dm = \rho dV.$$

Povšimněme si, že pokud by místo materiálu s pravidelným uspořádáním atomů šlo o ideazovaný plyn tvořený stejně hmotnými atomy, poloha těžiště by se nezměnila, ale těžiště by bylo dáno sumou místo integrálu

$$\vec{x}_T = \frac{\sum m_i \vec{x}_i}{\sum m_i} = \frac{1}{N} \sum \vec{x}_i.$$

Těžiště tělesa lze tedy určit jako průměr poloh atomů "plynu" tělesa **homogenně** vyplňujícího.

Rovnoměrné rozložení plynu budeme modelovat použitím (pseudo-)náhodných poloh. Klíčové je důležité pozorování: Homogenní rozložení dosáhneme obráběním jiného homogenního rozložení, nikoli však "kováním", kdy v místech kde nám tvar nevyhovujeme přebývající materiál vměstnáme do nějaké formy. U náhodných čísel to bude znamenat **zahazovat** body (představující atomy) ležící mimo polokouli.

```
[10]: from random import random
      from numba import njit

      @njit
      def nahodny_bod_polokoule():
          while True:
              x = 2*random() - 1
              y = 2*random() - 1
              z = random()

              if x*x + y*y + z*z < 1:
                  return x, y, z

      @njit
      def teziste_polokoule(n):

          xT,yT,zT = 0.0, 0.0, 0.0
          for i in range(n):
              x,y,z = nahodny_bod_polokoule()
              xT += x
              yT += y
              zT += z

          return xT/n, yT/n, zT/n

      teziste_polokoule(10**2) # zavoláme funkci, aby proběhl její překlad a ten
      ↪ nezapočítával do měření času %time
      %time teziste_polokoule(10**6)
```

```
CPU times: user 72.1 ms, sys: 0 ns, total: 72.1 ms
Wall time: 72.1 ms
```

```
[10]: (-0.0006740258738324543, 8.31535722820731e-06, 0.3751882573340752)
```

Můžete experimentováním zkoumat jak dobře se pro rostoucí počet bodů spočtená poloha těžiště blíží hodnotě 3/8.

19.3 QuickSort (ukázka chytrého/rychlého algoritmu třídění)

Vnitřní třídění seznamu znamená seřazení seznamu, které se odehrává uvnitř počítající části počítače a ne třeba na magnetické pásce, proto vnitřní. Historicky totiž první počítače neměly dost paměti, aby se tam tříděná data vešla najednou, takže existovala i disciplína vnějšího třídění.

Pro třídění potřebujeme mít definovanou funkci \leq dvou parametrů typu položky pole k setřídění. Část položky, která obsahuje informaci potřebnou pro porovnání se nazývá klíč. Obvykle z klíče nevyplývá přímo poloha v setříděném seznamu a má význam jen při porovnání s jiným klíčem.

Seznam považujeme za setříděný, platí-li

$$A_1 \leq A_2 \leq A_3 \leq \dots \leq A_N$$

Viděli jsme, že *bubble sort* obsahuje dva cykly a je jasným příkladem $O(N^2)$ algoritmu. V začátcích informatiky se hrály třídící algoritmy důležitou roli a jejich rozbor je v tomto oboru stále důležitý. Ačkoli jako fyzikové budeme používat funkce poskytující implementaci nějakého efektivního algoritmu, je zážitek porovnání neefektivního (např. bubblesort) a efektivního (např. quicksort) algoritmu důležitý pro každého, kdo programuje. Nejprve si tedy zopakujme tedy třídění probubláváním.

```
[11]: import numpy as np
from numba import njit

@njit
def bubble_sort_inplace(a):
    """procedura setřídí pole, které dostane jako parametr. Nic nevrací. Učíme se na ní.
    ↪ práci s poli, jinak je bubble sort na nic."""

    n = len(a)

    for i in range(n-1,0,-1):
        for j in range(i):
            if a[j]>a[j+1]:
                a[j], a[j+1] = a[j+1], a[j]

arr = np.random.randint(0,100, 20)
bubble_sort_inplace(arr)
print("Krátký test setřídění:")
print(arr)

arr = np.random.randint(0,100, 10**5)

%time bubble_sort_inplace(arr)

Krátký test setřídění:
[ 4 12 16 20 22 25 29 32 39 47 48 50 63 75 76 77 77 81 94 98]
CPU times: user 12 s, sys: 8.02 ms, total: 12 s
Wall time: 12 s
```

Quicksort je název algoritmu (Hoare cca 1960), který většinou dokáže seřadit seznam v čase $O(N \log N)$. Využívá toho, že do přesné polohy položky mají nejvíce co mluvit ty sousední. Proto:

- Zvolí některou hodnotu ze seznamu jako tzv. *pivot*.
- Rozdělí celý seznam bez pivotu na dvě skupiny: tu, kde jsou položky menší než pivot a zbytek.
- Poté za použití rekurze předá obě části sám sobě k seřazení. Pokud nám minulý krok rozdělí pole na zhruba dvě stejně velké části, dostáváme, podobně jako metody u půlení intervalu, jen logaritmický počet kroků, kterých je zapotřebí, abychom došli k seznamu délky 1, který již není třeba třídit.

Potíž spočívá v tom, že položku, podle které rozdělujeme seznam na dvě části, vezmeme aniž víme jestli je blízko "prostředku". Proto se může stát, že pro nevhodně uspořádaný vstup vezmeme vždy tu nejmenší/největší a skočíme u časové náročnosti $O(N^2)$.

Rozdíl mezi $O(N^2)$ a $O(N \log N)$ je ukázán [zde](#).

Pedagogická varianta kódu, jakou v mnoha podobách najdeme v učebnicích Pythonu vytváří seznam menších a větších prvků, ty nechá rekurzivním voláním seřadit a vrátí spojený seznam. Za pivot bere první prvek. To je názorné, ale ne moc rychlé. Navíc intenzivní užití seznamů brání použití akcelerace `@njit`.

[7]: `import numpy as np`

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        left = [x for x in arr[1:] if x < pivot]
        right = [x for x in arr[1:] if x >= pivot]
        return quicksort(left) + [pivot] + quicksort(right)
```

Example usage

```
arr = list( np.random.randint(0,100, 20) )
sorted_arr = quicksort(arr)
print("Krátký test třídění:")
print(sorted_arr)
print("Dlouhý test třídění:")
arr = np.random.randint(0,100, 10**5)
%time sorted_arr = quicksort(arr)
```

Krátký test třídění:

```
[0, 3, 3, 9, 19, 39, 58, 62, 62, 66, 66, 69, 70, 71, 71, 78, 85, 86, 97, 97]
```

Dlouhý test třídění:

```
CPU times: user 5.68 s, sys: 0 ns, total: 5.68 s
```

```
Wall time: 5.68 s
```

Rychlejší rozdělení probíhá přehazováním položek, které do příslušných částí nepatří a předáváním odkazu na pole a informace jako jeho část je potřeba seřazovat. Zde již provádíme samé "jednoduché" operace a lze použít akceleraci `@njit`.

[13]: `import numpy as np`
`from numba import njit`

```
@njit
def quicksort(arr, l=0, r=-1):
    if r<0:
```

(continues on next page)

```

    r = len(arr)-1

    pivot = arr[(l + r) // 2]

    i, j = l, r
    while i<j:
        while arr[i] < pivot:
            i = i + 1
        while pivot < arr[j]:
            j = j - 1
        if i <= j:
            arr[i], arr[j] = arr[j], arr[i]
            i = i + 1
            j = j - 1
    if l < j:
        quicksort(arr, l, j)
    if i < r:
        quicksort(arr, i, r)

# Použití
arr = np.random.randint(0,100, 20)
quicksort(arr)
print("Krátký test třídění:")
print(arr)
print("Dlouhý test třídění:")
arr = np.random.randint(0,100, 10**5)
%time quicksort(arr)

Krátký test třídění:
[ 7 21 26 29 29 32 36 43 50 52 55 73 75 75 78 83 93 98 98 99]
Dlouhý test třídění:
CPU times: user 5.91 ms, sys: 0 ns, total: 5.91 ms
Wall time: 5.91 ms

```

To, že kód je 700x rychlejší má dva důvody. Faktorem 90 přispěje *numba*, zbytek dá vynechání konstrukce nových seznamů uvnitř každého kroku tím, že spoléháme na *in place* přehazování prvků ležících v nesprávné "polovině" pole.

Podobně efektivní nástroj ke třídění je k dispozici v podobě standardní funkce `sorted`. Následující kód na příkladu náhodných komplexních čísel ukazuje, jakým způsobem můžeme určit, podle jakého klíče se mají data třídít. Složitější případy vyžadují konzultovat [dokumentaci](#). Další zajímavost: Všimněte si strukturované varianty rozbalování (*unpacking*) výsledku funkce `subplots`.

```

[ ]: import numpy as np
import matplotlib.pyplot as plt

n = 20
data1 = 2*(np.random.random(n) + 1j*np.random.random(n))-(1+1j)
data2 = sorted(data1, key=lambda z: z.real)
data3 = sorted(data1, key=lambda z: z.imag)
data4 = sorted(data1, key=lambda z: abs(z))

```

(continues on next page)

(pokračujte na předchozí stránce)

```

# protože sorted vrací seznam, pro malování konverujeme na numpy.ndarray
data2 = np.array(data2)
data3 = np.array(data3)
data4 = np.array(data4)

# čtyři obrázky v jednom
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(10,10))

ax1.plot(data1.real, data1.imag, '-. ')
ax2.plot(data2.real, data2.imag, '-. -r')
ax3.plot(data3.real, data3.imag, '-. -g')
ax4.plot(data4.real, data4.imag, '-. -b')
plt.show()

```

