Python 3

úvod a jednoduché matematické výpočty

K. Tribonacci & Zdeněk Halas

KDM MFF UK

provizorní verze

2. května 2026

Obsah

Úvod

Proč programování pro budoucí učitele

Tento text vzniká jako podpora pro předmět *IT pro učitele* na Matematicko-fyzikální fakultě UK, kde se vyučují základy programování v jazyce Python 3 (a také základy GeoGebry a TEXu).

Vycházíme ze zkušenosti, že pro učitele matematiky je výhodné mít alespoň minimální základy programování, neboť se mu tím otevírají nové možnosti, například:

- generování úloh podobného typu podle přesně požadovaných pravidel (procvičování, pracovní listy, testy),
- generování matematicky zajímavých údajů (rozklady na součin prvočísel, pýthagorejské trojice),
- porozumění (a tedy možnost kontroly a úprav) kódu generovaného AI a schopnost tento kód spustit.

Učitel matematiky si tak může podstatně usnadnit práci. Navíc může i v rámci hodin matematiky ukázat na vhodných místech kratičké prográmky, které něco užitečného dělají, případně které formalizují nějaký algoritmus.

Výhodou je, že ke spuštění Pythonu nepotřebujeme počítač (PC či notebook), stačí tablet či prakticky libovolný mobilní telefon (pod Androidem např. aplikace Pydroid 3). A samotný programovací jazyk včetně jednoduchého editoru je zcela zdarma (viz oficiální stránky python.org).

Programy v učebnicích matematiky

V 80. letech se počítače začaly šířit i mezi „běžné“ lidi. A lidé byli nadšení. Softwaru mnoho nebylo, programování bylo samozřejmou součástí, předpokladem pro jejich používání. Možnost napsat si aspoň krátký prográmek, který skutečně „něco dělá“, případně je užitečný, byla velmi přitažlivá. Krátké ukázky kódu a vývojové diagramy se objevovaly v učebnicích matematiky i ve sbírkách úloh. Tehdy používaným jazykem byl BASIC. V základních verzích byl snadno naučitelný, kód v něm napsaný byl však velmi nepřehledný zejména kvůli absenci podpory strukturovaného programování. Příkaz goto nahrazoval dnes běžné příkazy cyklu (for, while), vznikal tak „spaghetti code“.

V prosinci roku 1989 začal vývoj jazyka Python, který je také snadno naučitelný, jeho návrh však odpovídá současným požadavkům kladeným na přehlednost kódu. Je to jazyk v současné době hodně rozšířený, používaný v základních kurzech programování. Jednoduché prográmky v něm zapsané vypadají velmi esteticky. Myslím, že by se opět mohla vrátit tradice zařazovat jednoduché programy do učebnic matematiky; možnost si „něco vyzkoušet“ a zažít, že „ono to dělá to, co potřebuji“ je stále velmi atraktivní.

Část I

Přehled vybraných příkazů a konstrukcí

Kapitola 1

Nejdůležitější příkazy a konstrukce

1.1 Tisk a práce s proměnnou

```
print("Ahoj!")
```

Uložení hodnoty do proměnné, tisk hodnoty proměnné

```
i = 3
print(i)
print(i*i + 2)
print()     # prazdny radek
```

Výpis hodnoty proměnné a komentáře

1. jednoduchý postup, na výstupu vše oddělováno mezerami

   ```
   print("Hodnota je", i, ", to není mnoho.")
   ```

2. moderní způsob formátovaného výstupu

   ```
   print("Hodnota je {}, to není mnoho.".format(i) )
   ```

Vstup z klávesnice

```
n = int( input('Zadejte n: ') )     # input vrací retezec, proto je treba
    pretypovat
x = float( input('Zadejte x: ') )
```

Výstup do souboru

```
muj_soubor = open('Jméno mého souboru.txt', 'w')        # w - write, a -
   append, r - read
i = 12
muj_soubor.write(str(i) + "\t" + "Ahoj, zapisuji do souboru." + "\n")
muj_soubor.write("A jeste neco na dalsí rádek: write zapisuje pouze
   retezce.")
muj_soubor.close()
```

1.2 Cyklus for

Známe počet průchodů cyklem (nebo procházíme daný seznam).

```
# pro i od 0 do 5 (nikoli vcetne 5) tiskneme i
for i in range(5):
    print(i)

# cyklus for: pro k od 5 do 19 s krokem 3, tj. k = 5, 8, 11, 14, 17
for k in range(5, 20, 3):
    print("cislo:", k)

for k in [5, 8, 11, 14, 17]:        # totéz jako predchozí
    print("cislo:", k)
```

1.3 Cyklus while

Je obecnější než cyklus for. Při typickém použití neznáme předem počet průchodů cyklem, provádí se, *dokud* je splněna v něm obsažená podmínka.

Pro které $n \in \mathbb{N}$ součet $1+2+3+4+5+\cdots+n$ přeroste číslo 25? Vypisujme postupně částečné součty, *dokud* nepřerostou číslo 25.

```
n = 0
soucet = 0

while soucet < 25:
    n = n + 1
    soucet = soucet + n
    print(n, soucet)
```

Výstup programu:

```
1 1
2 3
3 6
4 10
5 15
6 21
7 28
```

1.4 Podmíněný příkaz (if)

Je-li podmínka uvedená v hlavičce jednoduchého podmíněného příkazu za klíčovým slovem if splněna, provede se blok příkazů, které tvoří tělo podmíněného příkazu (příkazy uvedené za dvojtečkou).

```
a = 7

if a > 0:
    print(a, "je kladné")
```

Úplný podmíněný příkaz obsahuje navíc větev else; příkazy z této větve se provedou v případě, že podmínka není splněna.

```
a, b, c = 1, 2, 5
D = b*b - 4*a*c

if D >= 0:
    print("kvadraticka rovnice ma reseni v R")
else:
    print("kvadraticka rovnice nema reseni v R")
```

Je-li třeba, můžeme přidat libovolný počet rozšiřujících podmíněných větví.

```
if znamka == 1:
    print('vyborne')
elif znamka == 2:
    print('velmi dobre')
elif znamka == 3:
    print('dobre')
elif znamka == 4:
    print('neprospel')
else:
    print('takovou znamku nemame')
```

1.5 Seznamy a řetězce

```
print(s)        # vypíse retezec s
print(s[:3])        # tisk prvních 3 znaku retezce (od 0 do 2), indexuje
   se od nuly
print(s[1:3])       # tisk 2. az 3. znaku
print(s[8:])        # tisk od 9. znaku az do konce

# spojování retezcu
a = "matematická"
b = "analýza"
```

```
print(a + b)
print(a + " " + b)

c = a + " " + b
print(len(c))       # délka retezce

# seznam 5 nul
a = [0 for x in range(5)]
print(a)
```

1.6 Definice funkce

```
def moje_funkce(x):
    return x*x - 2**x

print(moje_funkce(3))

for x in range(-5, 6):
    print(x, moje_funkce(x))

c = 2 * moje_funkce(2) + 3 * moje_funkce(3)


def delitele(n):
    D = []
    for d in range(1, n+1):
        if n % d == 0:      #je-li zbytek po delení nulový, tak je d
   delitel a tiskneme jej
            D.append(d)
    return D

for n in [97, 98, 99, 101, 102, 998, 999, 1001, 1002, 1003]:    # výpis
   delitelu vybraných císel
    print(n, delitele(n))
```

Kapitola 2

Další užitečné příkazy a konstrukce

2.1 Modul decimal

```
import decimal
decimal.getcontext().prec = 100     # vypocty s presnosti na 100 mist

print("1/97 =", decimal.Decimal(1) / decimal.Decimal(97))

print("sqrt(5) =", decimal.Decimal(5).sqrt())   # druha odmocnina

decimal.Decimal("0.1")      # spravne: desetinna cisla pomoci retezce
decimal.Decimal(0.1)        # zatizeno prevodem z dvojkove soustavy
```

Část II

Úvod do programování

Kapitola 3

Jak psát přehledný kód

- pěkně napsaný kód je přehledný, srozumitelný, snadno upravitelný (i po čase) a rozšiřitelný
- nižší riziko chyb věcných i čistě syntaktických
- vyšší efektivita práce

3.1 Odsazování

- správné odsazování (ideální jsou 4 mezery; ne tabulátor – má proměnnou délku)
- mezi logickými částmi kódu (např.: inicializace dat, výpočet, tisk výsledků) vynecháváme řádek
- mezi funkcemi vynecháváme rozumné množství řádků (u menších funkcí většinou 2)
- řádky nejsou příliš dlouhé, případně je lze vhodně rozdělit
- neužíváme přemíru závorek a interpunkce (středníky apod.)

3.2 Názvy

- názvy proměnných i funkcí odpovídají tomu, co obsahují či dělají
- názvy nejsou přehnaně dlouhé
- většinou nepoužíváme jednopísmenné názvy kromě proměnné cyklu for (i, j, k, n) či argumentů matematických funkcí (x, n)
- v textu nenecháváme záhadné číselné hodnoty (raději definujeme konstanty s výstižným názvem)

3.3 Komentáře

- volba výstižných názvů eliminuje mnohé přebytečné komentáře omezí se tak možnost chyby (při změně kódu se občas zapomene upravit příslušný komentář)
- program začíná stručným popisem toho, co dělá

3.4 Funkce

- funkce by měla být relativně malá a průhledná – dělat jednu věc a pořádně
- neměla by dělat „další změny na pozadí“ (činnosti a změny v datech, které k vykonání jejího úkolu nejsou nutné)
- funkce nemá mít příliš mnoho parametrů (zpravidla 0 až 3)
- problém by měl být na funkce rozložen přirozeně – logicky

3.5 Knihovny

- pokud je soubor příliš dlouhý, můžeme z funkcí vytvářet knihovny
- přehledný soubor zpravidla nemá více než 100 – 200 řádků
- funkce použitelné i v jiném programu sdružujeme do nových knihoven, neprogramujeme je stále znovu
- neměla by vznikat potřeba kopírovat nějakou část kódu na více míst (kandidát na zapouzdření do funkce)

3.6 Postup

- přehledný kód často (zejména u větších projektů) nevzniká hned
- *ihned a automaticky: správně odsazujeme a volíme vhodné názvy*
- většinou už při návrhu: úlohu rozdělíme na menší části; některé funkce však vzniknou až při psaní kódu
- přehlednosti kódu věnujeme přiměřenou pozornost (na začátku se nejvíce soustředíme zejména na základní funkčnost, poté můžeme kód refaktorovat (zpřehledňovat), někdy dojde i k podstatnému zjednodušení)
- nepotřebný kód uvážlivě mažeme (máme-li už lepší) či odkládáme do jiného souboru (může-li se hodit k něčemu jinému)

Literatura

- M. Fowler: Refactoring
- R. C. Martin: Čistý kód
- S. McConnell: Dokonalý kód

3.7 Obfuskátor – přehlednost kódu je důležitá

Python podporuje psaní hezkého kódu. Pokud dodržujeme základní pravidla, typicky bude náš kód přehledný.

V některých případech se kód záměrně zatemňuje, aby se ztížilo případné přebírání kódu někým jiným, například konkurencí. Existují dokonce programy – obfuskátory – které kód záměrně znepřehledňují. Nahrazují názvy proměnných matoucími slovy, vynechávají mezery a vynechané řádky, ...

Pro porovnání si můžeme předvést krátký program, který je záměrně znepřehledněný, a program napsaný „normálně“.

Nepřehledný kód

```
a=0;b=1;
for c in range(1,10):a=a+c;print(c,a,end="\t");b*=c;print(b);
```

Přehledná verze téhož programu

```
soucet = 0
faktorial = 1

for i in range(1, 10):
    soucet += i
    faktorial *= i
    print(i, soucet, "\t", faktorial)
```

Kapitola 4

Instalace Pythonu 3 a spouštění programů

Pokud je třeba něco naprogramovat, doporučuji použít jednoduchý, univerzální a hojně používaný programovací jazyk Python 3. Má jednoduchou a elegantní syntaxi, není třeba v něm deklarovat proměnné, dá se velmi snadno naučit, umožňuje programovat nejen klasicky imperativně, ale podporuje také programování generické, funkcionální a objektově orientované.

- Programovací jazyk Python 3 je open source, lze jej zdarma stáhnout z oficiálních stránek projektu https://www.python.org/ (verze 3.14 nebo vyšší).
- Oficiální tutoriál pro zájemce: https://docs.python.org/3/tutorial/.

4.1 Instalace – Windows

- Na stránce https://www.python.org na záložce Downloads zvolit pod nadpiskem *Download for Windows* tlačítko Python 3.14.0 (případně vyšší aktuální verze).
- Stažený soubor python-3.14.0-amd64.exe (nebo podobný název dle typu hardwaru) spustit a proklikat se k instalování. Po proběhnutí instalace je připraven k používání Python 3 i IDLE. Programy pak otevíráme v editoru IDLE.
- Programy mají příponu `.py`, otevírají se v editoru IDLE, která je součástí instalace Pythonu: jeden klik pravým tlačítkem myši na soubor s příponou `.py` a zvolit *Edit with IDLE*).
- Spuštění programu v IDLE: F5.

4.2 Instalace – Mac

- Na stránce https://www.python.org na záložce Downloads zvolit pod nadpiskem *Download for macOS* tlačítko Python 3.14.0 (případně vyšší aktuální verze).
- Stažený soubor python-3.14.0-macos11.pkg spustit a proklikat se k instalování. Po proběhnutí instalace je připraven k používání Python 3 i IDLE. Programy pak otevíráme v editoru IDLE.
- Spuštění programu v IDLE: F5.

4.3 Instalace – Linux

Většina linuxových systémů již obsahuje Python 3, je tedy třeba jen nainstalovat editor, doporučuji standardní a jednoduchý IDLE. Případné novější verze Pythonu a příslušného editoru IDLE lze jednoduše nainstalovat ve správci balíčků (většinou balíčky python3 a idle3, nebo python3.14 a idle-python3.14) nebo pomocí standardních příkazů zadaných v terminálu.

Například v operačních systémech založených na Debianu (např. populární Ubuntu (verze 25.10) a jeho různé deriváty) jsou příkazy následující:

```
sudo apt install python3.14
sudo apt install idle-python3.14
```

Spuštění programu v IDLE: F5.

4.4 Instalace – Android

- V Obchod Play lze zdarma stáhnout např. aplikaci Pydroid 3 – IDE for Python 3.

Kapitola 5

Základní příkazy a konstrukce

5.1 Proměnná, tisk

Tisk řetězce

Řetězec si můžeme představit jako posloupnost znaků v uvozovkách. Lze použít uvozovky jednoduché (`"řetězec"`) i dvojité (`'řetězec'`).

`print()` je funkce, proto u sebe musí mít závorky, které obsahují případné parametry. Zvýrazněna je fialově, neboť se jedná o vestavěnou funkci.

```
print("Ahoj!")
```

Uložení hodnoty do proměnné, tisk hodnoty proměnné

Založení nové proměnné s názvem `i` a její inicializace, tj. uložení celého čísla 3 do proměnné `i`.

```
i = 3
```

Tisk hodnoty proměnné a tisk hodnoty výrazu:

```
print(i)
print(i*i + 2)
print( (3*i*i - 1) / 2**4 )
print(10**i)
```

Umocňování zapisujeme pomocí dvou hvězdiček, např. `2**4` znamená 2^4.

V proměnné `i` je stále uloženo číslo 3, použití proměnné `i` ve výrazech totiž nemění její hodnotu. Zmenšení hodnoty proměnné `i` o 1 a její tisk (vytiskne se tedy číslo 2):

```
i = i - 1
print(i)
```

Tisk prázdného řádku – netiskne se nic, jen se zalomí řádek (dle defaultního nastavení, které lze změnit užitím parametru `end`):

```
print()
```

Formátovaný výstup

Neboli výpis hodnoty proměnné s nějakým komentářem.

1. Primitivní přístup – na výstupu se vytiskne vše vedle sebe a oddělené mezerami:

   ```
   print("Hodnota je", i, "- to není mnoho.")
   ```

2. **Moderní způsob formátovaného výstupu** (nejlepší):

   ```
   print("Hodnota je {} - to není mnoho.".format(i) )
   ```

3. Tisk spojení řetězců, spojování řetězců zajišťuje operátor +. Celé číslo `i` je převedeno na řetězec vestavěnou funkcí `str()`.

   ```
   print("Hodnota je " + str(i) + " - to není mnoho.")
   ```

4. Spojení řetězců uloženo do proměnné, ta se pak tiskne:

   ```
   retezec = "Hodnota je " + str(i) + " - to není mnoho."
   print(retezec)
   ```

5. Starší způsob formátovaného výstupu (nedoporučuji):

   ```
   print("Hodnota je %d - to není mnoho." % i)
   ```

5.2 Cyklus for

Chceme-li provést nějaké příkazy vícekrát bezprostředně za sebou, je vhodné použít cyklus. Známe-li přesný počet průchodů cyklem, použijeme zpravidla cyklus `for`. Blok příkazů, který je uvnitř cyklu `for`, je odsazen.

V následujícím příkladu provedeme pro `i` od 2 do 5 (avšak nikoli včetně 5) tisk hodnoty proměnné `i`. Generátor `range(2, 5)` totiž postupně generuje celá čísla počínaje dvojkou, končí pak celým číslem ostře menším než 5 (tj. čtyřkou).

```
for i in range(2, 5):
    print(i)
```

Výstup programu:

```
2
3
4
```

Pokud bychom nezadali počáteční hodnotu 2, generoval by generátor `range(5)` celá čísla od nuly počínaje, tj. čísla 0, 1, 2, 3, 4. Těchto čísel je právě pět, což je počet průchodů cyklem, který je tak ze zápisu `range(5)` dobře patrný. Postupuje-li se tedy od nuly, generování končí číslem o 1 menším, než je hodnota parametru generátoru, aby existoval snadný způsob zadání daného počtu průchodu cyklem: `range(5)` zajistí právě pět průchodů.

Porovnejme:

- pro `k` od 0 do 2 včetně (cyklus proběhne třikrát)

  ```
  for k in range(3):
      print("cislo:", k)
  ```

  Výstup programu:

  ```
  cislo: 0
  cislo: 1
  cislo: 2
  ```

- pro `k` od 3 do 7 včetně

  ```
  for k in range(3, 8):
      print("cislo:", k)
  ```

  Výstup programu:

  ```
  cislo: 3
  cislo: 4
  cislo: 5
  cislo: 6
  cislo: 7
  ```

- pro `k` od 3 do 7 včetně s krokem 2

  ```
  for k in range(3, 8, 2):
      print("cislo:", k)
  ```

  Výstup programu:

  ```
  cislo: 3
  cislo: 5
  cislo: 7
  ```

Tiskneme-li mnoho hodnot, nebývá účelné, aby každá z nich byla na samostatném řádku. To lze změnit nastavením parametru `end`.

Zde je parametr `end` nastaven na čárku a mezeru, za každým číslem se tedy bude tisknout čárka a mezera.

```
for i in range(20):
    print(i, end=', ')
```

Výstup programu:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
```

5.2.1 Tabulka druhých a třetích mocnin

Pro `i` od 1 do 5 (nikoli 6) tiskneme hodnotu proměnné `i` a jeho druhou a třetí mocninu.

```
for i in range(1, 6):
    print(i, i*i, i**3)
```

Výstup programu:

```
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
```

5.2.2 Malá násobilka – dvojitý cyklus for

Výpis malé násobilky je zajímavý tím, že potřebujeme dvojitý cyklus `for`. Všimněme si odsazení závěrečné funkce `print()`, která vypíše prázdný řádek po každém bloku deseti řádků násobilky.

```
print("Malá násobilka")

for i in range(1, 11):
    for j in range(1, 11):
        print(i, "·", j, "=", i*j)
    print()
```

Výstup programu:

```
Malá násobilka
1 · 1 = 1
1 · 2 = 2
1 · 3 = 3
```

```
...

10 · 8 = 80
10 · 9 = 90
10 · 10 = 100
```

5.3 if – podmíněný příkaz

(pozor na odsazení)

```
cislo = 2

if cislo > 0:
    print("Zadané číslo {} je kladné.".format(cislo))
else:
    print("Zadané číslo {} není kladné.".format(cislo))
```

Výstup programu:

```
Zadané číslo 2 je kladné.
```

Kdybychom na začátku zadali:

```
cislo = -3
```

obdrželi bychom výstup:

```
Zadané číslo -3 není kladné.
```

Pokud bychom měli vypsat všechna kladná čísla ze zadaného seznamu [2, -3, 5, 0, -7, 12], použili bychom cyklus for.

```
for n in [2, -3, 5, 0, -7, 12]:
    if n > 0:
       print(n, end=", ")
```

Výstup programu:

```
2, 5, 12,
```

5.3.1 Slovní hodnocení na základě známky

```
známka = int( input("Zadejte známku: ") )

print("Slovní hodnocení:", end='\t')


if známka == 1:
    print('výborný')
elif známka == 2:
    print('chvalitebný')
elif známka == 3:
    print('dobrý')
elif známka == 4:
    print('dostatecný')
elif známka == 5:
    print('nedostatecný')
else:
    print('takovou známku nemáme')
```

Pozor, je třeba odlišovat:

`=` přiřazovací rovná se (např. `z = 1`: do proměnné `z` se uloží číslo 1),

`==` porovnávací rovná se (např. `z == 1`: tento výraz má hodnotu `True`, pokud je $z = 1$, pokud je $z \neq 1$, nabývá tento výraz hodnoty `False`).

V řetězcích můžeme používat také např.:
`\t` – tabulátor, `\n` – newline (nový řádek).

5.4 Definice funkcí

- funkce může (ale nemusí) vracet nějakou hodnotu či více hodnot
- vrácení hodnoty zajišťuje klíčové slovo `return`, po vrácení hodnoty se činnost funkce ukončí
- funkce může (ale nemusí) mít nějaké parametry
- ale vždy musí mít za svým názvem závorky

V následující ukázce si definujeme vlastní funkci. V definici funkce se jen definuje, co se bude dělat při volání funkce (tj. při jejím použití); má-li parametry, je funkce volána s konkrétními hodnotami. Definice funkce musí vždy předcházet jejímu volání. Program obsahující pouze definici funkce z pohledu uživatele nedělá nic.

```
def moje_funkce(x):
    return x*x - 2**x
```

Až na následujícím řádku probíhá volání funkce, zde se příkazy z definice funkce skutečně provádějí pro $x = 3$. Vytiskne se tedy funkční hodnota pro $x = 3$, tj. $3 \cdot 3 - 2^3 = 1$. Tuto hodnotu vrátí moje_funkce(3) a vestavěná funkce print() ji vytiskne.

```
print(moje_funkce(3))
```

Výstup programu:

```
1
```

Tisk zadaného b a funkční hodnoty v bodě b:

```
b = 5
print(b, moje_funkce(b))
```

Výstup programu:

```
5 -7
```

```
# Tabulka hodnot funkce moje_funkce() od -4 do 4
for u in range(-4, 5):
    print(u, MojeFunkce(u))
```

Výstup programu:

```
-4 15.9375
-3 8.875
-2 3.75
-1 0.5
0 -1
1 -1
2 0
3 1
4 0
```

5.4.1 Funkce signum

Napišme nyní funkci signum. Ta je definována po částech: pro kladná čísla nabývá hodnoty 1, $\operatorname{sgn} 0 = 0$ a signum záporného čísla je -1.

```
def sgn(x):
    if x > 0:
        return 1
    elif x < 0:
        return -1
    else:
        return 0

for c in [2, 3.21, 7, 0, -2.81, -1, -.02]:
    print(c, sgn(c))
```

Výstup programu:

```
2       1
3.21    1
7       1
0       0
-2.81   -1
-1      -1
-0.02   -1
```

5.4.2 Funkce hodnotu netiskne, není-li k tomu vážný důvod

Následující funkce nevrací žádnou hodnotu, výsledná hodnocení tiskne. To je v mnoha případech velmi nevýhodné, s výsledným řetězcem nemůžeme dále pracovat.

```
def hodnoceni_print(n):
    if n == 1:
        print('výborný')
    elif n == 2:
        print('chvalitebný')
    elif n == 3:
        print('dobrý')
    elif n == 4:
        print('dostatecný')
    elif n == 5:
        print('nedostatecný')
    else:
        print('takovou známku nemáme')


známka = int( input('Zadejte známku: ') )
print(známka, end='\t')
hodnoceni_print(známka)
```

Výstup programu:

```
Zadejte známku: 1
1   výborný
```

5.4.3 Funkce hodnotu vrací

Modifikace – funkce, která výsledný řetězec vrací (return místo print()). S tímto řetězcem je pak možno dále pracovat, případně jej vytisknout dle potřeby.

```
def hodnoceni_return(n):
    if n == 1:
        return 'výborný'
    elif n == 2:
        return 'chvalitebný'
    elif n == 3:
        return 'dobrý'
    elif n == 4:
        return 'dostatecný'
    elif n == 5:
        return 'nedostatecný'
    else:
        return 'takovou známku nemáme'


známka = int( input('Zadejte známku: ') )
print( známka,  hodnoceni_return(známka) )
```

Výstup programu:

```
Zadejte známku: 1
1 výborný
```

5.4.4 Knihovny funkcí

V oficiální dokumentaci: https://docs.python.org/3/ je seznam běžně používaných knihoven pod odkazem Library Reference.

Nejčastěji budeme nejspíše používat knihovnu matematických funkcí math. Knihovnu musíme před použitím funkcí v ní obsažených importovat (např. import math). K volání funkcí z knihovny použijeme tečkovou notaci (např. math.sin(x)). Vestavěná funkce sinus očekává vstup v obloukové míře. Budeme-li tedy chtít vypsat tabulku hodnot funkce sinus s krokem jednoho stupně, bude potřeba převést vstup na obloukovou míru: 1 stupeň odpovídá $\frac{\pi}{180}$, takže n stupňů odpovídá $\frac{n\pi}{180}$.

```
import math

for n in range(0, 91):
    x = n * math.pi / 180
    print(n, math.sin(x))
```

Výstup programu:

```
0 0.0
1 0.01745240643728351
2 0.03489949670250097
3 0.05233595624294383
...
89 0.9998476951563913
90 1.0
```

Pokud bychom chtěli použít pouze jednu či několik málo funkcí z dané knihovny, není potřeba ji importovat celou, stačí importovat jen tyto funkce. Voláme je pak bez math.:

```
from math import sin, pi

for n in range(0, 91):
    x = n * pi / 180
    print(n, sin(x))
```

5.5 while – zatímco

Cyklus while probíhá, dokud je splněna podmínka (uvedená za klíčovým slovem sf while). Většinou se používá v případech, kdy není znám počet průchodů cyklem (tehdy by se totiž použil cyklus for). Typická použití cyklu while tedy zahrnují např.:

- výpočet aproximace nějaké hodnoty, dokud není dosaženo zadané přesnosti (tj. dokud není chyba menší, než je zadáno),
- hledání nějakého prvku (provádí se výpočet, dokud není výsledkem číslo s požadovanými parametry),
- ...

Následuje jednoduchá (i když málo užitečná) ukázka cyklu while. Dokud bude $a < 5$, budou se příkazy provádět. Pro $a = 5$ už nebude splněna podmínka, tak se příkazy v těle cyklu neprovedou, cyklus se opustí.

```
a = 1
while a < 5:
    print(a)
```

```
    a = a + 1
```

Výstup programu:

```
1
2
3
4
```

Všimněme si, že cyklus while je obecnější než cyklus for, neboť je schopen jej namodelovat. Předchozí příklad by bylo možno zapsat pomocí cyklu for takto:

```
for a in range(1, 5):
    print(a)
```

5.5.1 Hra – hádání čísla

Tipujeme číslo, dokud (while) jej neuhodneme. V následujícím programu je pevně zvoleno tipované číslo 2.

```
n = int( input("Tipnete si cislo od 1 do 5:   ") )

while n != 2:           #   !=  nerovná se
    print("Neuhodli jste...")
    n = int( input("Tipnete si cislo od 1 do 5:   ") )

print("Ano, je to cislo 2.")
```

Výstup programu:

```
Tipnete si cislo od 1 do 5:   3
Neuhodli jste...
Tipnete si cislo od 1 do 5:   1
Neuhodli jste...
Tipnete si cislo od 1 do 5:   4
Neuhodli jste...
Tipnete si cislo od 1 do 5:   2
Ano, je to cislo 2.
```

Volbu čísla, které máme uhodnout, může provádět počítač. Tato modifikace je obsažena v kapitole věnované hrám.

5.5.2 Nekonečná smyčka

Pokud podmínku v cyklu while nastavíme na True, bude vždy splněna a cyklus se nikdy neukončí. Vznikne tak nekonečná smyčka.

Následující program tiskne postupně čísla od nuly do nekonečna.

```
k = 0
while True:
    print(k, end=" ")
    k = k + 1
```

5.6 Základy práce s řetězci

```
s = "Toto je muj první řetezec v Pythonu."
print(s)     # vypíse řetezec s



# indexuje se od nuly

# tisk prvních 3 znaku řetezce (od 0 do 2)   Tot
print(s[:3])

# tisk 2. az 3. znaku                        ot
print(s[1:3])

# tisk od 9. znaku az do konce               muj první řetezec v Pythonu.
print(s[8:])

# tisk posledních 3 znaku                    nu.
print(s[-3:])



# spojování řetezcu
a = "matematická"
b = "analýza"
print(a + b)            # matematickáanalýza
print(a + " " + b)      # matematická analýza

c = a + " " + b
print(c)        # matematická analýza
```

```
print(len(c))   # délka řetezce: 19
```

Řetězce nemusíme jen sčítat, můžeme je i násobit.

```
chichotani = "hi " * 3
print(chichotani)       # hihihi
```

Sčítání i násobení lze samozřejmě kombinovat:

```
smich = "hi " * 3  +  "ha " * 3
print(smich)
```

Výstupem je:

```
hi hi hi ha ha ha
```

5.7 Seznamy

Seznamy jsou vždy indexovány od 0.

Stručný přehled

```
S = [2, 4, 5, 1]
print("celý seznam:", S)

print("nultý a první prvek:", S[0], S[1])
print("poslední a predposlední prvek:", S[-1], S[-2])

print("od zacátku po prvek s indexem 2 (bez):", S[:2])
print("od prvku s indexem 2 do konce:", S[2:])

S[0] = 15
print("nultý prvek nahrazen:", S)

S.append(23)
print("pripojit další prvek:", S)

prazdny = []
print("prázdný seznam:",  prazdny)
```

Výstup:

```
celý seznam: [2, 4, 5, 1]
nultý a první prvek: 2 4
poslední a predposlední prvek: 1 5
od zacátku po prvek s indexem 2 (bez): [2, 4]
od prvku s indexem 2 do konce: [5, 1]
nultý prvek nahrazen: [15, 4, 5, 1]
pripojit další prvek: [15, 4, 5, 1, 23]
prázdný seznam: []
```

Nejzákladnější operace s řetězci s komentářem

- Seznam [2, 4, 7] uložíme do proměnné S:

```
S = [2, 4, 7]
print(S)        # [2, 4, 7]
```

- Do seznamu S připojíme na konec další prvek: řetězec "ahoj":

```
S.append("ahoj")
print(S)        # [2, 4, 7, 'ahoj']
```

- Kterýkoli prvek můžeme kdykoli změnit, např.:

```
S[1] = 132
print(S)           # [2, 132, 7, 'ahoj']
```

- Kterýkoli prvek můžeme kdykoli vypsat, případně s ním dále pracovat, např.:

```
print(S[1])           # 132
```

Odstranění položky ze seznamu

```
S.pop(3)
print(S)
```

Odstranili jsme poslední položku (v 4-prvkovém seznamu je to položka s indexem 3, indexujeme totiž od nuly). Jelikož se indexy chovají cyklicky, můžeme místo indexu 3 použít index -1, čímž bezpečně odstraníme poslední prvek (ať už je počet prvků seznamu jakýkoli).

```
S.pop(-1)
print(S)
```

Výsledek bude v obou případech stejný, místo seznamu [2, 132, 7, 'ahoj'] dostaneme seznam [2, 132, 7]. Odstraňovaný prvek můžeme uložit i do nějaké proměnné.

```
a = S.pop(-1)
```

Ze seznamu lze odstranit prvek i na základě jeho hodnoty (místo indexu):

```
D = [1, 2, 8, 7, 8]
D.remove(8)
print(D)
```

Výstupem je seznam [1, 2, 7, 8]. Odstraní se první prvek s hodnotou rovnou argumentu metody remove().

Uložení více prvků do seznamu

Kdybychom chtěli vytvořit například seznam pěti nul, máme několik možností:

```
# postupné pridávání prvku do seznamu
v = []
for i in range(5):
    v.append(0)
print(v)


v = [0] * 5
print(v)


# nejlepsi varianta:
v = [0 for n in range(5)]
print(v)
```

Můžeme vytvořit i seznam obsahující různé prvky.

```
v = []
for i in range(1, 7):
    v.append(i)
print(v)           # [1, 2, 3, 4, 5, 6]
```

Ideální je použít stručnější zápis:

```
nulovy_vektor = [i for i in range(1, 7)]
print(nulovy_vektor)
```

Výstup programu je v obou případech stejný:

```
[1, 2, 3, 4, 5, 6]
```

5.8 Zápis do souboru

```
# otevreme soubor
muj_soubor = open("Muj název souboru.txt", "w")

i = 12

# zapisujeme do souboru
muj_soubor.write(str(i) + '\t' + "Ahoj, zapisuji do souboru." + '\n')
muj_soubor.write("A jeste neco na dalsí rádek.")

# zavreme soubor
muj_soubor.close()
```

.write()

- umí zapisovat do souboru pouze řetězec, tj. čísla je třeba konvertovat na řetězce pomocí funkce str()
- nepřidává na rozdíl od print() odřádkování (proto je třeba přidávat \n)
- \t - tabulátor, \n – nový řádek

Funkce open() otevře soubor s názvem uvedeným jako první parametr. Druhý parametr funkce open() může být:

- "w" – otevření souboru pro zápis (write)
- "r" – otevření souboru pro čtení (read)
- "a" – otevření souboru pro přidávání dalších dat (append)

```
# zápis do souboru pomocí spojení retezcu (primitivní postup)
import math # importování knihovny matematických funkcí, v níz je funkce
   sqrt - odmocnina

f = open('NaDruhou.txt', 'w')

for i in range(1, 11):
    f.write(str(i) + '\t' + str(i*i) + '\t' + str(math.sqrt(i)) + '\n')

f.close()
```

```
# tentýz program, vyuzití formátovaného retezce (lepsí postup)
import math
```

```
f = open('NaDruhou1.txt', 'w')

for i in range(1, 11):
    radek = "{}\t{}\t{}\n".format(i, i*i, math.sqrt(i) )
    f.write(radek)
f.close()
```

5.9 Výpočty s libovolnou přesností – modul decimal

Při výpočtech můžeme narazit na následující problémy:

- Čísla v desítkové soustavě jsou konvertována do dvojkové soustavy, v níž se provádějí všechny výpočty. Výsledek je pak převeden zpět do desítkové soustavy. Při konverzi mohou vznikat chyby zaokrouhlování. Tak je tomu například v následujícím jednoduchém výpočtu.

  ```
  print(0.2 + 0.1)
  ```

  ```
  0.30000000000000004
  ```

  Porovnáme-li to se zadáním

  ```
  print( 0.3 )
  ```

  ```
  0.3
  ```

  vidíme, že nedostaneme stejný výsledek, přestože $0,2 + 0,1 = 0,3$.

- Někdy potřebujeme pracovat s vyšší přesností než nabízí vestavěný datový typ (float).

Oba problémy lze odstranit použitím modulu decimal. Při jeho používání všechny výpočty probíhají přímo v desítkové soustavě, pracuje se přímo s ciframi desetinných rozvojů, odpadá tak převádějí mezi desítkovou a dvojkovou soustavou.

V modulu decimal je přesnost přednastavena na 28 desetinných míst. Přesnost lze libovolně upravit nastavením konstanty decimal.getcontext().prec.

```
import decimal
decimal.getcontext().prec = 100     # nastaveni presnosti na 100 mist

print("1/97 =")
print("decimal:", decimal.Decimal(1) / decimal.Decimal(97))
print("float:  ", 1/97)
```

Výstup programu:

```
1/97 =
decimal: 0.010309278350515463917525773195876288659793814432989690
           721649484536082474226804123711340206185567010 31
float:   0.010309278350515464
```

Podporovány jsou i nejdůležitější elementární funkce. Počítat tak můžeme například hodnotu odmocniny či exponenciály s libovolnou přesností.

```
print("odmocnina z 5 =", decimal.Decimal(5).sqrt())

print("e na 1 =", decimal.Decimal(1).exp())
```

```
# spravne (pomoci retezce) a nespravne (prevodem z float) zadani
   desetinneho cisla decimal
a = decimal.Decimal("0.1")
print("cislo 0,1 presne (decimal vyzaduje retezec):", a)

# float ulozeno ve dvojkove soustave, presnost max. na 16 des. mist
b = decimal.Decimal(0.1)
print("cislo 0,1 nepresne (float prevedeno na decimal):", b)
```

Výstup programu:

```
cislo 0,1 presne (decimal vyzaduje retezec): 0.1

cislo 0,1 nepresne (float prevedeno na decimal):
   0.1000000000000000055511151231257827021181583404541015625
```

Výpočet e

Chceme-li pozorovat konvergenci posloupnosti

$$\left(1+\frac{1}{n}\right)^n,$$

musíme použít modul decimal, jinak bychom místo konvergence pozorovali spíše chyby zaokrouhlení. Porovnání obou variant můžeme vidět na výpisu pro $k = 10^n$.

```
import decimal

for n in [1, 3, 5, 7, 10, 15, 20, 25, 27]:
    # pozor: bez decimal chyba zaokrouhleni:
    k = 10 ** n
    print(n, (1+1/k)**k, "bez decimal" )
    # s decimal skutecne konverguje k e:
    k = decimal.Decimal(1).shift(n)  # posun o n des. mist, vytvareni
   mocnin deseti pouhym pridanim nul
    print(n, (1+1/k)**k )
    print()
```

Výstup programu:

```
1 2.5937424601000023 bez decimal
1 2.5937424601

3 2.7169239322355936 bez decimal
3 2.7169239322358924573830881 22
```

```
5 2.7182682371922975 bez decimal
5 2.7182682371744896680350648 24

7 2.7182816941320818 bez decimal
7 2.7182816925449662711985502 26

10 2.7182820532347876 bez decimal
10 2.7182818283231311439497940 01

15 3.035035206549262 bez decimal
15 2.7182818284590438762193732 42

20 1.0 bez decimal
20 2.7182818284590452353466960 62

25 1.0 bez decimal
25 2.7182818284590452353602873 35

27 1.0 bez decimal
27 2.7182818284590452353602874 70
```

Vidíme, že posloupnost konverguje k Eulerovu číslu velmi pomalu, např. při $k = 1\,000$ dosáhneme přesnosti pouze na dvě desetinná čísla.

Část III

Programování v rámci různých (nejen) matematických témat

Kapitola 6

Součty

6.1 Úloha malého Gausse: součet čísel od 1 do 100

```
soucet = 0

for i in range(1, 101):
    soucet = soucet + i

print("soucet cisel od 1 do 100: ", soucet)
```

Modifikujte tento program na součet prvních n členů libovolné aritmetické posloupnosti (se zadanou diferencí d, prvním členem a_1).

Napište program, který vypočte součet prvních n členů geometrické posloupnosti (se zadaným kvocientem q, prvním členem a_1).

6.2 Výpočet součtu prvních n členů dané řady

```
# e = exp(1) = 1 + 1/1! + 1/2! + 1/3! + 1/4! + ...

print("Vypocet e pomoci Taylorova rozvoje funkce exp x")

soucet = 1
clen = 1
pocet_clenu = 18

for i in range(1, pocet_clenu + 1):
    clen =  clen / i
    soucet = soucet + clen
    print(i, soucet)
```

Výstup programu:

```
Pocítáme e pomocí Taylorova rozvoje funkce exp x
1 2.0
2 2.5
3 2.6666666666666665
4 2.708333333333333
5 2.7166666666666663
6 2.7180555555555554
7 2.7182539682539684
8 2.71827876984127
9 2.7182815255731922
10 2.7182818011463845
11 2.718281826198493
12 2.7182818282861687
13 2.7182818284467594
14 2.71828182845823
15 2.718281828458995
16 2.718281828459043
17 2.7182818284590455
18 2.7182818284590455
```

```python
# ln 2 = 1 - 1/2 + 1/3 - 1/4 + 1/5 - 1/6 + ...
s = 0
z = -1   # zajistí strídání znaménka

for n in range(1, 10**6 + 1):
    z *= -1
    s += z / n   # s += (-1)**(n+1) / n   by bylo mnohem špomalejí

print("soucet clenu rady:   ", s)


from math import log   # jen pro kontrolu
print("pro kontrolu:  ln 2 =", log(2))
```

Výstup programu:

```
soucet clenu rady:    0.6931476805552527
pro kontrolu:  ln 2 = 0.6931471805599453
```

6.3 Výpočet hodnot funkce sinus s libovolnou přesností pomocí Taylorova rozvoje

Funkce vracející sinus x, kde x je v radiánech.

Sinus se počítá pomocí Taylorova rozvoje se středem 0:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

Taylorův rozvoj funguje nejlépe pro x blízká nule.

V případě x od 0 vzdálenějších než 2π si lze snadno pomoci volbou x z intervalu $\langle 0, 2\pi)$, případně z intervalu $\langle 0, \frac{\pi}{2}\rangle$.

Užití:

```
alfa = decimal.Decimal('0.3')
print(sin(alfa))


import decimal

def sin(x, presnost=28):
    decimal.getcontext().prec = presnost + 2
    i, zbytek, soucet  =  1, 0, x
    faktorial, citatel, znamenko  =  1, x, 1
    while soucet != zbytek:
        zbytek = soucet
        i += 2
        faktorial *= i * (i-1)
        citatel *= x * x
        znamenko *= -1
        soucet += citatel / faktorial * znamenko
    decimal.getcontext().prec -= 2
    return +soucet     # unární plus aktivuje ěnov nastavenou řpesnost


alfa = decimal.Decimal('1')
print(sin(alfa))

print(sin(alfa, 50))


# pro kontrolu - vestavená hodnota sinu:
from math import sin
print(sin(1))
```

Výstup programu:

```
0.84147098480789650665250232163029899962256306079837
0.8414709848078965
```

Kapitola 7

Opakování základních konstrukcí

7.1 While – jídelna

Jídelna dopoledne uvařila jistý počet jídel (jidel_k_dispozici). Namodelujme si situaci, kdy:

- Na tabuli svítí vždy aktuální počet jídel (aktualizovaná hodnota proměnné jidel_k_dispozici), která má jídelna ještě k dispozici.
- Zákazníci postupně přicházejí a každý z nich si odebírá porce dle své potřeby, ovšem ze zbývajícího počtu porcí (svítí na tabuli), které má jídelna ještě k dispozici. Každý zákazník tedy odebírá nejvýše tolik jídel, kolik jich jídelně ještě zbývá.
- Jídelna vaří skvěle, a tak o zákazníky nemá nouzi; každý den se prodají všechna připravená jídla. Zákazníci tedy odebírají uvařená jídla, dokud (while) je počet jídel k dispozici kladný.
- Po prodání všech jídel se odpoledne provede závěrečné shrnutí: vedoucí jídelny pošle majiteli celkový počet obsloužených zákazníků (pocet_obslouzenych_zakazniku).

```
import random

jidel_k_dispozici = 50
print("celkem uvareno jídel: ", jidel_k_dispozici)

pocet_obslouzenych_zakazniku = 0

while jidel_k_dispozici > 0:
    objednavka = random.randint(1, jidel_k_dispozici)
    if jidel_k_dispozici - objednavka >= 0:
        jidel_k_dispozici  =  jidel_k_dispozici - objednavka
        print("objednávka:", objednavka, " zbývá jídel:",
   jidel_k_dispozici)
    pocet_obslouzenych_zakazniku += 1

print("pocet obslouzených zákazníku: ", pocet_obslouzenych_zakazniku)
```

Výstup programu:

```
celkem uvareno jídel:  50
objednávka: 8  zbývá jídel: 42
objednávka: 12  zbývá jídel: 30
objednávka: 8  zbývá jídel: 22
objednávka: 7  zbývá jídel: 15
objednávka: 4  zbývá jídel: 11
objednávka: 10  zbývá jídel: 1
objednávka: 1  zbývá jídel: 0
pocet obslouzených zákazníku:  7
```

Pokud by každý den jídelna připravila stejný počet jídel, můžeme se ptát, jaký by byl průměrný počet obsloužených zákazníků za jeden den.

```
import random

def zakazniku_za_den( jidel_k_dispozici ):
    pocet_obslouzenych_zakazniku = 0
    while jidel_k_dispozici > 0:
        objednavka = random.randint(1, jidel_k_dispozici)
        if jidel_k_dispozici - objednavka >= 0:
            jidel_k_dispozici  =  jidel_k_dispozici - objednavka
        pocet_obslouzenych_zakazniku += 1
    return pocet_obslouzenych_zakazniku


jidel = 100
pocet_dnu = 365000

for i in range(5):
    soucet = 0
    for n in range(pocet_dnu):
        soucet += zakazniku_za_den(jidel)
    print("prumerny pocet obslouzenych zakazniku: ", soucet / pocet_dnu)
```

Pokud by se zákazníci chovali, co se počtů odebraných jídel týče, zcela náhodně, zjistili bychom, že počet obsloužených zákazníků by byl za daných podmínek průměrně:

- $2,929$ zákazníků při počtu 10 jídel za den,
- $5,1875$ zákazníků při počtu 100 jídel za den,
- $7,485$ zákazníků při počtu 1 000 jídel za den.

Omezení počtu jídel odebraných jedním zákazníkem

Pozorujeme, že počet obsloužených zákazníků s počtem připravených jídel příliš neroste. Mohli bychom tedy zkusit omezit počet jídel odebraných jedním zákazníkem.

Například: pokud bychom zvolili omezení, že jeden zákazník může odebrat nejvýše 10 porcí (pokud by jich žádal více, dostal by jich právě 10), průměrné počty obsloužených zákazníků by se zvýšily:

- 13,013 zákazníků při počtu 100 jídel za den a omezení nejvýše 10 porcí na 1 zákazníka,
- 104,055 zákazníků při počtu 1 000 jídel za den a omezení nejvýše 10 porcí na 1 zákazníka.

Omezení přidáme snadno, stačí do funkce zakazniku_za_den(jidel_k_dispozici, omezeni = 10) připojit omezující podmínku (přidat dva řádky). Omezení počtu odebraných porcí jedním zákazníkem můžeme defaultně nastavit například na 10 porcí.

```
def zakazniku_za_den( jidel_k_dispozici,  omezeni = 10 ):
    pocet_obslouzenych_zakazniku = 0
    while jidel_k_dispozici > 0:
        objednavka = random.randint(1, jidel_k_dispozici)
        if  objednavka > omezeni:
            objednavka = omezeni
        if jidel_k_dispozici - objednavka >= 0:
            jidel_k_dispozici  =  jidel_k_dispozici - objednavka
        pocet_obslouzenych_zakazniku += 1
    return pocet_obslouzenych_zakazniku
```

7.2 Collatzova hypotéza

Roku 1937 předložil německý matematik Lothar Collatz hypotézu, která nese jeho jméno, viz například https://en.wikipedia.org/wiki/Collatz_conjecture.

Uvažujme jednoduchý algoritmus někdy označovaný jako HOTPO (half or triple plus one), který každému přirozenému číslu n přiřadí přirozené číslo tímto způsobem:

$$\mathrm{HOTPO}(n) = \begin{cases} \frac{n}{2} & \text{pro } n \text{ sudé}, \\ 3n+1 & \text{pro } n \text{ liché}. \end{cases}$$

Pokud tento algoritmus opakovaně aplikujeme na dané číslo $n \in \mathbb{N}$, dostaneme se k číslu 1 vždy po konečně mnoha iteracích?

Například pro $n = 3$ dostaneme posloupnost sedmi iterací: $3, 10, 5, 16, 8, 4, 2, 1$, neboť

$$\mathrm{HOTPO}(3) = 10\,, \quad \mathrm{HOTPO}(10) = 5\,, \quad \mathrm{HOTPO}(5) = 16\,, \quad \mathrm{HOTPO}(16) = 8\,,$$

$$\mathrm{HOTPO}(8) = 4\,, \quad \mathrm{HOTPO}(4) = 2\,, \quad \mathrm{HOTPO}(2) = 1\,.$$

Po sedmi (tedy po konečně mnoha) krocích jsme došli k číslu jedna.

Nyní můžeme řešit různé problémy, například:

- kolik iterací je třeba, abychom pro dané n došli k jedné,

- vytvořit tabulku: ke každému n od 1 do 100 vypsat seznam iterací,
- vytvořit tabulku: ke každému n od 1 do 100 vypsat počet iterací,
- ověřit, že se dostaneme k číslu 1 po konečně mnoha iteracích pro všechna přirozená čísla $n < 10\,000$,
- najít k danému počtu iterací číslo n, které po tomto počtu iterací dojde k jedné,
- najít mezi čísly od 1 do 10^k číslo s největším počtem iterací.

Samotný algoritmus HOTPO lze naprogramovat snadno:

```
def HOTPO(n):
    if n % 2 == 0:
        return n // 2
    else:
        return 3*n + 1
```

Případně stručnější a rychlejší verze je:

```
def HOTPO(n):
    return  n // 2  if  not n % 2  else 3*n + 1
```

Výpis všech iterací pro dané n, dokud nedosáhneme jedné, je pěknou aplikací cyklu while.

```
n = 3
print(n)
while n != 1:
    n = HOTPO(n)
    print(n)
```

Z tohoto algoritmu můžeme vytvořit funkci, která bude pro dané n vracet seznam Collatzových iterací.

```
def Collatzovy_iterace(n):
    Collatz = [n]
    while n != 1:
        n = HOTPO(n)
        Collatz.append(n)
    return Collatz
```

Otestovat ji můžeme snadno:

```
for i in range(2, 8):
    print( i,  Collatzovy_iterace(i) )
```

Výstup programu:

```
2 [2, 1]
3 [3, 10, 5, 16, 8, 4, 2, 1]
4 [4, 2, 1]
5 [5, 16, 8, 4, 2, 1]
6 [6, 3, 10, 5, 16, 8, 4, 2, 1]
7 [7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

Podobně můžeme napsat funkci vracející pro dané n počet Collatzových iterací.

```
def pocet_Collatzovych_iteraci(n):
    iteraci = 0
    while n != 1:
        n = n // 2 if not n % 2 else 3*n+1   # HOTPO
        iteraci = iteraci + 1
    return iteraci
```

V cyklu while jsme místo n = HOTPO(n) použili stručný zápis bez volání funkce HOTPO(n), který je rychlejší.

```
n = n // 2  if  not n % 2  else 3*n + 1
```

7.3 Flaviův problém

Problém je inspirován situací, kterou vylíčil jako přímý účastník židovský velitel Flavius Iosephus (1. stol.) v jeho *Válce židovské*.

Josef neztratil v těchto nesnázích rozvahu. V důvěře v ochranu boží vsadil záchranu na jeden los a řekl: „Je-li tedy rozhodnuto, že zemřeme, nuže svěřme losu, kdo má komu zasadit smrtelnou ránu. Vylosovaný ať padne rukou toho, kdo bude vylosován po něm, a takto bude osud putovat od jednoho ke druhému a nikdo ať nepadne vlastní rukou. Bylo by nespravedlivé, kdyby někdo po smrti ostatních změnil své smýšlení a zachránil se.“ Když toto řekl, uznali jeho věrnost. Přesvědčiv je, losoval s nimi. Vylosovaný ochotně poskytoval dalšímu po něm příležitost ke smrtelné ráně, protože i vojevůdce měl být zabit. Smrt s Josefem považovali za příjemnější nežli život. Josef však zbyl nakonec ještě s jedním, ať již je třeba mluvit o náhodě, či o boží prozřetelnosti. Usiloval o to, ani aby nebyl losem odsouzen k smrti, ani aby si pravici neposkvrnil vraždou soukmenovce, kdyby měl zůstat poslední, a proto i toho druhého přemluvil, aby zůstali naživu na danou záruku.

Flavius Iosephus, Židovská válka, III. kniha, kap. 8, odst. 387–391

Flavius a jeho 40 vojáků uvízli v jeskyni obklopené římskými vojáky. Zvolili smrt před zajetím a rozhodli se, že to provedou losováním. Iosephus uvádí, že zůstal on a jeden další muž jako poslední a vzdali se Římanům.

Pokud losování probíhalo rozpočítáváním, tak byl další na řadě vždy ten, kdo stál o n pozic dále. Vzniká tak otázka, kolikátý v kole stál Flavius, případně kolikátý v kole byl druhý bojovník, který s Flaviem zůstal.

Tuto situaci můžeme zobecnit na následující úlohu:

N lidí stojí v kruhu (modelujeme seznamem $(1, ..., N)$*), stanoví se, kolikátý vždy jde z kola ven (proměnná* `posun`*). Kolikátý člověk zůstane jako poslední?*

Tato úloha připomíná hru „... ten musí jít z kola ven“, kde je posun dán počtem slabik říkanky („En ten týky...“).

Následující funkce vrací údaj, kolikátý člověk zůstane jako poslední.

```
def Flavius(N, posun):
    C = [i for i in range(N)]
    k = 0
    for i in range(N-1):
        k = k + (posun-1)
        L = len(C)
        if k > L - 1:
            k = k % L
        C.pop(k)
    return C[0] + 1 # +1 kvuli indexování od 0
```

Výpis, jak dopadne tato hra

```
for i in range(1, 42):
    for posun in range(1, i):
        print( i, posun, Flavius(i, posun) )
    print()
```

7.4 Posloupnost zadaná rekurentně – druhé odmocniny

Výpočet odmocniny čísla x mezopotámským postupem, tj. pomocí rekurentně zadané posloupnosti

$$a_{n+1} = \frac{x + a_n^2}{2 \cdot a_n}$$

```
x = 5       # císlo, jehoz odmocninu pocítáme
N = 15  # pocet iterací pri výpoctu odmocniny (pro vetsí presnost je
   potreba více iterací)

a = x       # první clen posloupnosti

for i in range(1, N+1):
    a = (x + a*a) / (2*a)       # výpocet n-tého clenu posloupnosti

print("Odmocnina čísla", x, "je", a)
```

Výstup programu:

```
Odmocnina čísla 5 je 2.23606797749979
```

Zabalme nyní tento kód do funkce.

N – počet iterací při výpočtu odmocniny (pro větší přesnost je potřeba více iterací). Jelikož je N inicializováno, tak je to nepovinný parametr. Nebude-li N při volání funkce zadáno, tak se použije defaultní hodnota 15.

```
def Mezop_odmoc(x, N=15):
    a = x       # prvni clen posloupnosti

    for i in range(1, N+1):
        a = (x + a*a) / (2*a)       # vypocet n-teho clenu posloupnosti

    return a


print("Cislo  Odmocnina")

for x in range(1, 6):
    print("{}\t{}".format(x, Mezop_odmoc(x)) )
```

Výstup programu:

```
Cislo  Odmocnina
1   1.0
2   1.414213562373095
3   1.7320508075688772
4   2.0
5   2.23606797749979
```

```
# testování vlivu poctu iterací na presnost
print("Odmoc(5) pocítaná pomocí n iterací:")

for n in range(1, 9):
    print("{}\t{}".format(n, Mezop_odmoc(5,n)) )
```

Výstup programu:

```
Odmoc(5) pocítaná pomocí n iterací:
1   3.0
```

```
2   2.3333333333333335
3   2.2380952380952386
4   2.2360688956433634
5   2.236067977499978
6   2.2360679774997894
7   2.23606797749979
8   2.23606797749979
```

7.5 Do sebe vnořené odmocniny a zlatý řez

Snadno lze dokázat, že

$$\sqrt{1+\sqrt{1+\sqrt{1+\sqrt{1+\sqrt{1+\ldots}}}}} = \frac{1+\sqrt{5}}{2}\,.$$

Toto číslo se nazývá zlatý řez a značí se zpravidla φ. Jeho přibližná hodnota je $\varphi = 1,618\,033\,988\,...$

Důkaz uvedené rovnosti lze založit na tom, že celou levou stranu s do sebe vnořenými druhými odmocninami označíme φ (jež prozatím považujeme za neznámé) a všimneme si, že výraz pod další druhou odmocninou je opět φ. Lze tedy psát

$$\sqrt{1+\varphi} = \varphi\,,$$

neboli

$$1+\varphi = \varphi^2\,.$$

Jediným kladným řešením této kvadratické rovnice je právě číslo

$$\varphi = \frac{1+\sqrt{5}}{2}\,.$$

Pokusme se napsat jednoduchý program, který bude modelovat konvergenci výrazu na levé straně.

```
x = 1

for i in range(35):
    x = x+1
    x = x ** .5
    print(i, x)
```

Výstupem budou členy postupně konvergující posloupnosti, ustálení se dosáhne pro $i = 30$:
30 1.618033988749895.

Kapitola 8

Rekurze

Pomocí rekurze lze snadno naprogramovat některé úlohy. Uvedeme si několik jednoduchých příkladů.

8.1 Fibonacciho posloupnost

Tato posloupnost je definována rekurentně:

$$F_n = F_{n-1} + F_{n-2}, \quad F_0 = 0, F_1 = 1\,.$$

```
def fib1(n):
    if n > 1:
        return fib1(n-1) + fib1(n-2)
    else:
        return 1
```

Stručnější verze (nemá vliv na rychlost):

```
def fib2(n):
    return fib2(n-1) + fib2(n-2) if n > 1 else 1
```

8.2 Geometrická posloupnost

Na příkladu geometrické posloupnosti můžeme pozorovat, že rekurzi lze použít i u funkcí, které mají více než jeden parametr. Zde je kvocient q druhým parametrem.

```
def geom(n, q):
    if n == 0:
        return 1
    return q * geom(n-1, q)

for k in range(5):
    print(k, geom(k, 2))
```

8.3 Faktoriály

```
def faktorial_rekurze(n):
    if n < 2:
        return 1
    else:
        return n * faktorial_rekurze(n - 1)
```

Podrobněji se budeme výpočtu faktoriálu věnovat v následující kapitole.

8.4 Kombinační čísla

```
def kombin_rekurze(n, k):
    if n < k:
        return 0
    if k == 0:
        return 1
    else:
        return kombin_rekurze(n-1, k-1) + kombin_rekurze(n-1, k)
```

Je možné pro kontrolu vypsat několik prvních řádků Pascalova trojúhelníku.

```
print("ůPascalv trojúhelník - rekurze")
for n in range(7):
    for k in range(n+1):
        print(kombin_rekurze(n, k), end=" ")
    print()
```

Výpočet pomocí rekurze brzy narazí na omezení hloubky rekurze:

```
n = 1500
s = 0
try:
    for k in range(n+1):
        s += kombin_rekurze(n, k)
    print("čsouet řůádk Pascalova trojúhelníku:")
    print(s)
    print()
except:
    print("Chyba: čvyerpán limit omezující hloubku rekurze...", k)
```

Výpis programu:

```
Chyba: vycerpán limit omezující hloubku rekurze... 1
```

8.5 Mocniny s celočíselným exponentem

Chceme-li umocnit nějaké číslo x např, na dvanáctou, můžeme jen prostě jedenáctkrát násobit:

$$x \quad \to \quad x \cdot x = x^2 \quad \to \quad x^2 \cdot x = x^3 \quad \to \quad x^3 \cdot x = x^4 \quad \to \quad \ldots \quad \to \quad x^{11} \cdot x = x^{12}\,.$$

Mnohem efektivnější je použít již vypočtené součiny. Základem může být jednoduché pozorování:

$$x^{12} = (x \cdot x)^6\,.$$

Místo 11 násobení zde budeme počítat součinů pouze 6: jeden $x \cdot x$, zbylých pět při výpočtu šesté mocniny.

Nabízí se použít tuto myšlenku i dále, tj. při výpočtu šesté mocniny a dále. Minimalizujeme tím počet násobení použitých při výpočtu n-té mocniny.

```
def umocneni(a, n):
    if n == 0:
        return 1
    else:
        cinitel = umocneni(a, n//2)
        if n % 2 == 0:  # n je sudé
            return cinitel * cinitel
        else:           #n je liché
            return cinitel * cinitel * a
```

Funkci můžeme aspoň zběžně otestovat:

```
for k in range(8):
    print(k, umocneni(2, k), umocneni(3, k), umocneni(5, k))
```

Výstup programu:

```
0 1 1 1
1 2 3 5
2 4 9 25
3 8 27 125
4 16 81 625
5 32 243 3125
6 64 729 15625
7 128 2187 78125
```

Můžeme vyzkoušet, zda je naše funkce `umocneni(a, n)` aspoň tak efektivní, jako umocnění pomocí vestavěného operátoru **. Porovnání provedeme na výpočtu hodnoty $2^{(2^{24})}$.

Jelikož budeme pracovat s příliš velkými celými čísly (více než 4 300 cifer), budeme muset zvýšit maximální počet cifer.

```
import time, sys
sys.set_int_max_str_digits(2**30)

n = 24

s = time.time()
print(len(str(2**(2**n))))
t = time.time()
print(t - s)


print()

s = time.time()
a = umocneni(2, n)
print(len(str(umocneni(2, a))))
t = time.time()
print(t - s)
```

Výstup programu (netiskneme celý výsledek, ale jen počet cifer):

```
5050446
1.1403958797454834

5050446
1.12882399559021
```

Vidíme, že je naše funkce prakticky stejně rychlá jako vestavěný operátor umocňování.

Kapitola 9

Faktoriály a kombinační čísla

9.1 Faktoriály

9.2 Faktoriál – různé postupy

Funkce vracející $n!$ naprogramovaná různými způsoby, porovnáváme efektivitu.

```
# faktoriál klasicky - for
def faktorial_for(n):
    f = 1
    for i in range(1, n+1):
        f = f * i
    return f
```

```
# faktoriál - while
def faktorial_while(n):
    f = 1
    i = 0
    while i < n:
        i = i + 1
        f = f * i
    return f
```

Výpočet faktoriálu pomocí rekurze je založen na posloupnosti zadané rekurentně:

$$n! = \begin{cases} 1 & \text{pro } n = 0, \\ n \cdot (n-1)! & \text{pro } n > 1. \end{cases}$$

```
# faktoriál pomocí rekurze - funkce volá sama sebe
def faktorial_rekurze(n):
    if n == 0:
        return 1
    else:
        return n * faktorial_rekurze(n - 1)



# faktoriál pomocí rekurze - varianta
def faktorial_rekurze_var(n):
    if n < 2:
        return 1
    else:
        return n * faktorial_rekurze_var(n - 1)



# strucnejsi zapis, je i rychlejsi
def faktorial_rekurze_strucna(n):
    return 1 if n < 2 else n * faktorial_rekurze_strucna(n - 1)
```

Ještě zajistíme výpisy. Budeme přitom měřit čas, abychom alespoň přibližně porovnali efektivitu jednotlivých funkcí. Vypočteme $n!$ pro $n = 21\,000$. Rekurze by selhala, poslední n, pro které funguje při standardním nastavení je $n = 993$. Nastavíme tedy větší hloubku rekurze na 10^5.

```
import time
from sys import setrecursionlimit
setrecursionlimit(10**5)

n = 21000

cas1_for = time.time()
f = faktorial_for(n)
cas2_for = time.time()
print("for:\t", cas2_for - cas1_for)

cas1_while = time.time()
f = faktorial_while(n)
cas2_while = time.time()
print("while:\t", cas2_while - cas1_while)

cas1_rekurze = time.time()
```

```
f = faktorial_rekurze(n)
cas2_rekurze = time.time()
print("rekurze:\t", cas2_rekurze - cas1_rekurze)

cas1_rekurze_strucna = time.time()
f = faktorial_rekurze_strucna(n)
cas2_rekurze_strucna = time.time()
print("rekurze-strucna:\t", cas2_rekurze_strustrucna -
   cas1_rekurze_strucna)

from math import factorial
cas1_vestaveny = time.time()
f = factorial(n)
cas2_vestaveny = time.time()
print("vestaveny:\t", cas2_vestaveny - cas1_vestaveny)
```

Výstup programu obsahuje dobu výpočtu v sekundách.

```
for:     0.08006596565246582
while:   0.07624602317810059
rekurze:     0.07684326171875
rekurze-strucna:     0.07609891891479492
vestaveny:   0.011753082275390625
```

9.3 Kombinační čísla

Následující funkce vracejí hodnoty kombinačních čísel $\binom{n}{k}$. Při výpočtu nemusíme používat funkce faktoriál:

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!},$$

potřebovali bychom zbytečně mnoho násobení. Výhodnější je použít této definice:

$$\binom{n}{k} = \frac{n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot (n-k+1)}{1 \cdot 2 \cdot 3 \cdot \cdots \cdot k},$$

kde v čitateli i ve jmenovateli je právě k činitelů. Výpočet tedy můžeme realizovat takto:

$$\binom{n}{k} = \frac{n}{1} \cdot \frac{n-1}{2} \cdot \frac{n-2}{3} \cdot \cdots \cdot \frac{n-k+1}{k}.$$

Navíc lze ukázat, že všechny zde uvedené zlomky mají celočíselnou hodnotu, proto můžeme použít celočíselného dělení.

Výbornou variantou pro výpočet kombinačních čísel je následující spolehlivá funkce vracející přímo celá čísla.

```
# nejlepsi varianta
def kombin(n, k):
    komb = 1
    for i in range(1, k+1):
        komb = komb * (n-i+1) // i
    return komb

print(kombin(1020, 510))
```

Kdybychom uvnitř cyklu for nepoužili celočíselné dělení, float by přinášel pro velká čísla chyby zaokrouhlení, vycházely by výsledky ve tvaru 3.0 místo 3, navíc by taková funkce podléhala omezení na velikost použitých čísel; mohli bychom vypočítat $\binom{1020}{510}$, ale $\binom{1021}{510}$ už by vrátilo inf. Funkce by sice byla rychlá (rychlejší než varianta s celočíselným dělením), ale nespolehlivá.

```
# nespolehlivé, trpí mnohými problémy
def kombin_float(n, k):
    komb = 1
    for i in range(1, k+1):
        komb = komb * (n-i+1) / i
    return komb    # konverze int(komb) by problémy s chybami prohloubila
```

Kdybychom si nebyli jisti, že všechny zlomky $\frac{n-i+1}{i}$ pro $i = 0, 1, \ldots, k$ mají celočíselnou hodnotu, mohli bychom výpočet rozdělit: nejprve bychom počítali čitatele, poté jmenovatele. Zbytečně bychom však navýšili počet cyklů for na dva a pracovali bychom se zbytečně velkými čísly. Například čas potřebný pro výpočet $\binom{1020}{510}$ by se zvýšil na dvojnásobek oproti funkci kombin(n, k).

```
# spolehlivé, ale pomalé
def kombin_dva_for(n, k):
    komb = 1
    for i in range(1, k+1):
        komb = komb * (n-i+1)
    for i in range(1, k+1):
        komb = komb // i
    return komb
```

Kombinační čísla bychom mohli počítat i pomocí zlomků, eliminovali bychom tak chyby zaokrouhlení při dělení. Následující funkce tedy funguje spolehlivě i pro obrovská čísla, je však velmi pomalá: asi dvacetkrát pomalejší než funkce kombin(n, k).

```
from fractions import Fraction

def kombin_frac(n, k):
    komb = Fraction(1, 1)
    for i in range(1, k+1):
        komb = komb * Fraction(n-i+1, i)
    return komb
```

Kombinační čísla – výpočet pomocí rekurze

Kombinační čísla lze naprogramovat i pomocí rekurze. Využije se přitom vlastnosti Pascalova trojúhelníku:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

```
def kombin_rekurze(n, k):
    if k > n:
        return 0
    elif k == 0:
        return 1
    else:
        return kombin_rekurze(n-1, k-1) + kombin_rekurze(n-1, k)
```

Můžeme vypsat prvních několik řádků Pascalova trojúhelníku.

```
print("Pascaluv trojuhelnik - rekurze")
for n in range(7):
    for k in range(n+1):
        print(kombin_rekurze(n, k), end=" ")
    print()
```

Výpočet pomocí rekurze je však omezen hloubkou rekurze. Můžeme to snadno otestovat.

```
n = 1026
s = 0
try:
    for k in range(n+1):
        s = s + kombin_rekurze(n, k)
    print("soucet radku Pascalova trojuhelniku:")
    print(s)
    print()
except:
    print("Chyba: vycerpán limit omezující hloubku rekurze...", k)
```

Výstup programu:

```
Chyba: vycerpán limit omezující hloubku rekurze... 1
```

Kombinační čísla – testy spolehlivosti

Otestujme nyní spolehlivost funkce kombin_float(n, k). Porovnáme hodnoty, které vrací, s hodnotami funkce kombin(n, k).

```
N = 60

for n in range(N+1):
    for k in range(n+1):
        if kombin_float(n, k) != kombin(n, k):
            print("pozor", n, k)
print("hotovo")
```

Výpis programu je dlouhý, uvedeme jen začátek:

```
pozor 55 26
pozor 55 27
pozor 55 28
pozor 55 29
pozor 55 30
pozor 55 31
...
```

Vidíme, že hodnoty obou porovnávaných funkcí se pro $n \geq 55$ liší v mnoha případech, takže funkce kombin_float(n, k) je pro $n \geq 55$ nepoužitelná.

```
print("testování spolehlivosti - soucet rádku Pascalova trojúhelníku")
print()
N = 200

print("float: ")
for n in range(N+1):
    s = 0
    for k in range(n+1):
        s += kombin_float(n, k)
    if s != 2**n:
        print(n, end=" ")
print()

print("int: ")
for n in range(N+1):
    s = 0
    for k in range(n+1):
        s += kombin_int(n, k)
    if s != 2**n:
        print(n, end=" ")
print()

print("velka: ")
for n in range(N+1):
    s = 0
```

```
    for k in range(n+1):
        s += kombin_velka(n, k)
    if s != 2**n:
        print(n, end=" ")
print()

print("frac: ")
for n in range(N+1):
    s = 0
    for k in range(n+1):
        s += kombin_frac(n, k)
    if s != 2**n:
        print(n, end=" ")
print()


# ěřmení času
print("ěřmení času")
print()

from time import time
N = 500

print("problémy: kombin_float(n, n) == 0 pro n:")
s = time()
for n in range(N+1):
    for k in range(n+1):
        if kombin_float(n, k) == 0:
            print(n, end=" ")
            #print("pozor", n, k)
t = time()
print()
print("float:", t - s)

s = time()
for n in range(N+1):
    for k in range(n+1):
        if kombin_int(n, k) == 0:
            print("pozor", n, k)
t = time()
print("int:", t - s)

s = time()
for n in range(N+1):
    for k in range(n+1):
        if kombin_velka(n, k) == 0:
            print("pozor", n, k)
t = time()
```

```
print("velka:", t - s)

"""
s = time()
for n in range(N+1):
    for k in range(n+1):
        if kombin_frac(n, k) == 0:
            print("pozor", n, k)
t = time()

print("frac:", t - s)
"""
print("frac pro 500: 42 s")
```

9.3.1 Další testy spolehlivosti

```
n = 1020
k = 510

print("čkombinaní číslo {} nad {}:".format(n, k) )

print("float:  ", kombin_float(n, k)) # pouze do 1020 nad 510
print()
print("zlomky:", kombin_frac(n, k))


print()

K = kombin_velka(n, k)
print("velká: ", K)
print("cifer: ", len(str(K)))
# čstruný výpis
Ks = str(K)
print("{},{} * 10na{}".format(Ks[0], Ks[1:15], len(str(K)) - 1 ) )



print("\n")



# pro kontrolu - čsouet řádku Pascalova trojúhelníku je 2**n

print("Pro kontrolu - čsouet komb. čísel v 1 řádku Pascalova trojúh. =
```

```
   2**n:")
print("2**n:")
print(2**n)
print()

# funkce kombin_velka() funguje ěspolehliv
print("Bez konverze z float je čvýpoet velkých čkombinaních čísel
   spolehlivý:")
s = 0
for k in range(n+1):
    s += kombin_velka(n, k)
print("čsouet řůádk Pascalova trojúhelníku:")
print(s)
print()


# kombin_float() - řpi konverzi z float kontrola neprojde
print("S konverzí z float čvýpoet velkých čkombinaních čísel funguje jen
   pro prvních 15 cifer:")
s = 0
for k in range(n+1):
    s += kombin_float(n, k)
print("čsouet řůádk Pascalova trojúhelníku:")
print(s)
print()


# kombin_frac()
print("S modulem frac je čvýpoet velkých čkombinaních čísel spolehlivý:")
s = 0
for k in range(n+1):
    s += kombin_frac(n, k)
print("čsouet řůádk Pascalova trojúhelníku:")
print(s)
```

Výstup programu:

```
č
kombinaní číslo 1020 nad 510:
float:
   28062677682996256679590600220084400157205040967445890310926897706447302337711


zlomky:
   28062677682996227103941430788273532516895081724250422585218474338430346745905
```

```
velká:
   280626776829962271039414307882735325168950817242504225852184743384303467459055

cifer:  306
2,80626776829962 * 10na305


Pro kontrolu - čsouet komb. čísel v 1 řádku Pascalova trojúh. = 2**n:
2**n:
112355820928894744233081574424314045851123561183894160795893800723582922378438102


Bez konverze z float je čvýpoet velkých čkombinaních čísel spolehlivý:č
souet řůádk Pascalova trojúhelníku:
112355820928894744233081574424314045851123561183894160795893800723582922378438102


S konverzí z float čvýpoet velkých čkombinaních čísel funguje jen pro
   prvních 15 cifer:č
souet řůádk Pascalova trojúhelníku:
112355820928894851261986371813226107229788433867025358577331574978040501499206360


S modulem frac je čvýpoet velkých čkombinaních čísel spolehlivý:č
souet řůádk Pascalova trojúhelníku:
112355820928894744233081574424314045851123561183894160795893800723582922378438102
```

Kapitola 10

Fibonacciho posloupnost – opakování různých konstrukcí

Fibonacciho posloupnost je ideální téma pro ukázku různých přístupů a konstrukcí – od těch nejjednodušších po pokročilejší. Existuje mnoho různých způsobů, jak v Pythonu napsat program, který vypíše prvních 10 členů Fibonacciho posloupnosti. Zde si ukážeme několik možností. Jelikož se bude jednat o programy se stále stejnou funkcionalitou, budeme v jejich záhlaví vynechávat komentář informující o jejich účelu.

Jednotlivé kapitoly budeme nazývat podle charakteristického prvku, na němž je program založen.

10.1 Pomocná proměnná

Potřebujeme namodelovat posun – uložit do původních proměnných a, b hodnoty b, $a+b$:

$$b \to a \qquad \text{a} \qquad a+b \to b\,,$$

neboli

$$a = b \qquad \text{a} \qquad b = a+b\,.$$

Problém je, že uložením $b \to a$ ztratíme původní hodnotu a, proto si ji předem zálohujeme do pomocné proměnné c. Tuto zálohovanou hodnotu použijeme na posledním řádku při výpočtu nové hodnoty proměnné b, tj. místo $b = a + b$ píšeme $b = c + b$.

```
a = 0  # prvni dva cleny posloupnosti
b = 1

for i in range(10):
    print(a)
    c = a
    a = b
    b = c + b
```

Jinou možností je zálohovat si hodnotu proměnné b:

```
a = 0
b = 1
```

```
for i in range(10):
    print(a)
    c = b
    b = a + b
    a = c
```

10.2 Současná inicializace

Python umožňuje provést posun

$$b \to a \qquad \text{a} \qquad a + b \to b$$

elegantně, bez pomocné proměnné c, stačí použít současnou inicializaci proměnných a, b.

```
a, b = 0, 1

for i in range(10):
    print(a)
    a, b = b, a + b
```

Tato verze je asi nejjednodušší, zároveň je velmi přehledná.

Pokud bychom chtěli mít přehledně parametr udávající počet členů, můžeme založit novou proměnnou. To nám umožní zpřehlednit program, nebude se v něm objevovat „záhadná“ konstanta. Kvůli stručnosti budeme tento prvek v ostatních programech vynechávat. Při běžném programování bychom však jednoznačně preferovali verzi s konstantou uloženou v řádně nazvané pomocné proměnné.

```
pocet_clenu = 10

a, b = 0, 1

for i in range(pocet_clenu):
    print(a)
    a, b = b, a + b
```

10.3 Cyklus while

Cyklus `while` je obecnější, než cyklus `for`. Cyklus `for` lze tedy vždy namodelovat pomocí cyklu `while`. Tento postup však nepovažujeme za vhodný, neboť zastírá, že se jedná o cyklus s předem známým počtem průchodů (přesně 10). Proměnná i zde plní stejnou pomocnou funkci, jako v předchozích cyklech `for`.

```
a, b = 0, 1
i = 0

while i < 10:
```

```
    print(a)
    a, b = b, a + b
    i += 1
```

10.4 Seznam

Následující verze je mimořádně přehledná: členy posloupnosti, které jsou vždy součtem předchozích dvou členů, se postupně připojují (`append()`) do seznamu.

```
F = [0, 1]
for i in range(2, 10):
    F.append(F[i-1] + F[i-2])

print(F)
```

Užití pomocné proměnné není nutné, Python umožňuje pohodlně odkazovat na poslední a předposlední prvek seznamu.

```
F = [0, 1]
for i in range(10-2):
    F.append(F[-1] + F[-2])

print(F)
```

Jinou možností je použít cyklus `while`, v němž se bude hlídat délka seznamu: dokud bude menší než 10, budou se do seznamu doplňovat jednotlivé členy.

```
F = [0, 1]
while len(F) < 10:
    F.append(F[-1] + F[-2])

print(F)
```

10.5 Seznam – list comprehension

```
F = [0, 1]
[ F.append(F[-1] + F[-2]) for i in range(10 - 2) ]
print(F)
```

Následující kód nedělá, co bychom si přáli: výsledkem bude seznam jedniček, neboť se stále jen sčítají původní $0 + 1$, nikoli poslední dvě hodnoty postupně rozšiřovaného seznamu.

```
chyba = [0, 1]
chyba = [ chyba[-2]+chyba[-1] for i in range(10 - 2) ]
print(chyba)
```

10.6 Funkce

Definujeme funkci, která vypíše prvních n členů Fibonacciho posloupnosti. Následně tuto funkci zavoláme s parametrem 10.

```
def fibonacci(n):
    a, b = 0, 1
    for i in range(n):
        print(a)
        a, b = b, a + b

fibonacci(10)
```

Mohli bychom také napsat funkci, která místo vypisování vypočtené hodnoty vrátí. Výhodou je, že vrácenou hodnotu můžeme dále používat, případně se můžeme rozhodnout ji vytisknout.

Zde je funkce, která vrací n-tý člen Fibonacciho posloupnosti. Použití takové funkce na výpis mnoha po sobě jdoucích členů je velmi neefektivní, při každém zavolání se totiž opakují výpočty provedené při výpočtu předchozího členu.

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

for k in range(10):
    print(k, fibonacci(k))
```

Závěrečný cyklus `for` zajišťuje výpis jednotlivých členů.

10.7 Rekurze

```
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

for i in range(10):
    print(fibonacci(i))
```

Rozhodně lze doporučit porovnání s verzí napsanou s použitím seznamu.

Tělo funkce lze napsat i na pouhém jednom řádku:

```
def fibonacci(n):
    return n if n<=1 else fibonacci(n-1) + fibonacci(n-2)
```

10.8 Rekurze s ukládáním vypočtených hodnot

```
def fibonacci(n, cache={}):
    if n in cache: # overi se, zda byla hodnota pro dane n jiz vypoctena
        return cache[n] # vrati jiz vypoctenou hodnotu
    elif n > 1:
        return cache.setdefault(n, fibonacci(n-1) + fibonacci(n-2))
    return n

for i in range(10):
    print(fibonacci(i), end=', ')
```

10.9 Generátor (yield)

Generátor umožňuje elegantní a paměťově úsporný způsob generování posloupnosti. Neukládáme všechny vypočtené členy do jednoho seznamu, ale tyto členy postupně generujeme až v případě potřeby.

```
def fibonacci_gen():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

generator_fibon = fibonacci_gen()
for i in range(10):
    print(next(generator_fibon))
```

Generátor můžeme vytvořit i s omezením do n, které je parametrem. Takový generátor pak funguje pouze při generování prvních n členů, zde konkrétně prvních 10 členů.

```
def fibonacci_gen_n(n):
        a, b = 0, 1
        for i in range(n):
            yield a
            a, b = b, a + b
n = 10
generator_fib_n = fibonacci_gen_n(n)
```

```
for i in range(n):
    print(next(generator_fib_n))
```

Kapitola 11

Rovnice

11.1 Řešení rovnice $f(x) = 0$ metodou půlení intervalu

Uvažujme například rovnici

$$2^x - x^2 = 0\,.$$

Ta má právě tři kořeny. Kromě kladných kořenů 2 a 4 má také jeden kořen záporný ($-0.766\,664\,695\,962\,123\ldots$). Z grafu je patrné, že je tento záporný kořen větší než -1, leží tedy v intervalu $(-1, 0)$.

Ověřit, že rovnice $f(x) = 0$ má v intervalu (a, b) kořen, je snadné. Je-li funkce na $\langle a, b\rangle$ spojitá, stačí ověřit, že znaménka funkčních hodnot $f(a)$ a $f(b)$ jsou různá (tj. $f(a) \cdot f(b) < 0$). Následně můžeme interval rozpůlit ($x = \frac{a+b}{2}$) a prošetřit pomocí if, zda kořen leží v levé (tj. $\langle a, x\rangle$), nebo pravé (tj. $\langle x, b\rangle$) polovině intervalu $\langle a, b\rangle$, tj. ověříme, zda

$$f(a) \cdot f(x) < 0 \qquad \text{nebo} \qquad f(x) \cdot f(b) < 0\,.$$

Půlení intervalu se provádí, dokud je jeho délka $b - a$ větší než zadaná přesnost. Jedná se tedy o typický příklad použití cyklu while.

Program očekává zadání a, b, mezi kterými leží alespoň jeden kořen rovnice $f(x) = 0$.

```
def f(x):
    return x*x - 2**x

a = -1
b = 0
presnost = 1e-15

while b - a > presnost:
    x = (a+b) / 2
    if f(a) * f(x) < 0:
        b = x
    elif f(x) * f(b) < 0:
        a = x
    else:
        print("chyba")
        break
```

```
print(a)
print(b)
```

Výstup programu:

```
-0.7666646959621235
-0.766664695962123
```

11.2 Hledání kořene postupným procházením od daného x_0

Hledat kořen lze velmi jednoduše také tak, že budeme od zvoleného x_0 postupně procházet čísla osy x s daným krokem h. Dokud se nebudou lišit znaménka $f(x_0)$ a $f(x_0 + h)$, nenarazili jsme na kořen. Bude se tedy opět jednat o jednoduchou aplikaci cyklu while.

```
def f(x):
    return 2**x - x*x

x = -1
h =  0.0000001

while f(x) * f(x+h) > 0:
    x = x + h

print(x, f(x))
print(x+h, f(x+h))
```

Výstup programu ukazuje porovnání hodnot pro nalezené x a $x+h$. Je tak zřejmá přesnost aproximace, kořen leží mezi -0.7666647 a -0.7666646.

```
-0.7666647001228174 -8.07484168419137e-09
-0.7666646001228175 1.8599953099940336e-07
```

V každém kroku jsme počítali dvě funkční hodnoty: f(x) a f(x+h). To však není potřeba, stačí čekat, než funkční hodnota změní znaménko.

```
def f(x):
    return 2**x - x*x

x = -1
h = 0.000001

def sgn(x):
    if x > 0:
        return 1
```

```
    if x < 0:
        return -1
    else:
        return 0

znamenko = sgn(f(x))

if znamenko == 0:
    print("koren: ", x)
else:
    while sgn(f(x)) == znamenko:
        x = x + h
    print(x, "hodnota v x:  ", f(x))
    print(x+h, "hodnota v x+h:", f(x+h))
```

11.3 Hledání kořene iterační metodou

Z rovnice $f(x) = 0$ se můžeme pokusit sestavit rekurentní předpis tak, že vyjádříme nějakým vhodným způsobem x, tj.

$$f(x) = 0 \qquad \Rightarrow \qquad x = g(x) \qquad \Rightarrow \qquad x_{n+1} = g(x_n)\,.$$

Pokud posloupnost $\{x_n\}$ konverguje při vhodně zvoleném x_0 ke kořenu rovnice $f(x) = 0$, máme vyhráno. Iterovat budeme, dokud nebude rozdíl $|x_n - g(x_n)|$ menší než zadaná přesnost. Bude se tedy opět jednat o jednoduchou aplikaci cyklu while.

11.3.1 Rovnice $x = \cos x$

Řešme například rovnici $x = \cos x$. Rekurentní předpis můžeme například zvolit takto:

$$x_{n+1} = \cos x_n\,, \qquad x_0 = 0\,.$$

```
import math

def g(x):
    return math.cos(x)

x = 0
presnost = 1e-15

while abs(x - g(x)) > presnost:
    x = g(x)

print(x)
print(g(x))
```

Výstup programu:

```
0.7390851332151611
0.7390851332151603
```

11.3.2 Al-Kášího kubická rovnice pro $\sin 1°$

V 15. století použil al-Káší v Samarkandu způsob, jak najít aproximaci hodnoty $\sin 1°$ s libovolnou přesností. Musel při tom řešit kubickou rovnici. Jeho postup podstatně upravíme, aby byl jednodušší, a modernizujeme.

Už z doby antiky (Klaudios Ptolemaios, kol. roku 150 po Kr.) byla známa hodnota $\sin 3°$, kterou bylo možno poměrně pohodlně aproximovat s libovolnou přesností. Připomeňme si vztah pro $\sin 3\alpha$:

$$\sin 3\alpha = 3\sin\alpha - 4\sin^3\alpha\,.$$

Píšeme-li místo neznámého $\sin\alpha$ pro $\alpha = 1°$ neznámou x, tj. $x = \sin 1°$, dostaneme kubickou rovnici:

$$\sin 3° = 3x - 4x^3\,,$$

neboli $3x = 4x^3 + \sin 3°$, odkud vydělením třemi získáme vztah

$$x = \frac{4x^3 + \sin 3°}{3}$$

vhodný pro iterační předpis

$$x_{n+1} = \frac{4x_n^3 + \sin 3°}{3}\,.$$

Jelikož je $\sin x \approx x$ pro x blízká nule, můžeme jako jednoduchý odhad kořene $(\sin 1°)$ použít $x_0 = \frac{1}{60}$, neboť

$$\sin 1° = \sin\frac{\pi}{180} \approx \frac{\pi}{180} \approx \frac{3}{180} = \frac{1}{60}\,.$$

Potřebujeme ještě znát hodnotu $\sin 3°$ s postačující přesností:

$$\sin 3° = 0,052\,335\,956\,242\,943\,832\,722\,118\,629\,609\,07\ldots$$

```
x = 1 / 60
sin3 = 0.052335956242943832722118629609078

for i in range(8):          # 8 iterací
    print(i, '\t', x)
    x = (sin3 + 4*x*x*x) / 3
```

Výstupem programu je:

```
0    0.016666666666666666
1    0.01745149158715412
2    0.01745240532273798
3    0.017452406435925612
```

```
4    0.01745240643728186
5    0.01745240643728351
6    0.017452406437283515
7    0.017452406437283515
```

Porovnáme-li výstup programu s přesnou hodnotou

$$\sin 1^\circ = 0,017\,452\,406\,437\,283\,512\,819\,418\,978\,516\,31\ldots,$$

vidíme, že jsme úspěšně (a navíc poměrně efektivně) získali aproximaci hodnoty $\sin 1^\circ$ pomocí řešení kubické rovnice iterační metodou.

11.4 Hornerovo schéma – hodnoty polynomu

Při výpočtu hodnoty polynomu přesně podle předpisu

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0\,,$$

brzy zjistíme, že provádíme zbytečně mnoho umocňování a násobení, což jsou poměrně náročné operace. Řešením je neumocňovat x na prvou, druhou, ..., n-tou, ale umocnění provést jen jednou postupným násobením a využíváním mezivýsledků:

$$P(x) = \Big(\big((a_n x + a_{n-1})x + a_{n-2}\big)x + \cdots + a_1\Big)x + a_0\,.$$

```
def horner(P, x):
    y = 0
    for a_i in P[::-1]:
        y = y * x + a_i
    return y
```

Například polynom $P(x) = 3x^2 + 5x + 7$ je reprezentován uspořádanou trojicí `P = [7, 5, 3]`. Hodnota tohoto polynomu v bodě 2 je 29, jak snadno zjistíme:

```
print( horner([7, 5, 3], 2) )
```

`P[::-1]` nám umožní procházet seznam koeficientů polynomu „odzadu“, tj. počínaje a_n. Jednotlivé koeficienty polynomu jsou postupně v proměnné `a_i`.

Kapitola 12

Dělitelnost

12.1 Seznam všech dělitelů

Vypisují se dělitelé zadaného přirozeného čísla n. Očekává se tedy zadání přirozeného čísla $n > 1$. Testování, zda je k dělitelem čísla n, probíhá jednoduše: je-li zbytek po dělení n % k roven nule, je k dělitelem čísla n a vytiskneme k.

```
n = int( input("Zadejte cislo, jehoz delitele vypiseme: ") )

for k in range(1, n+1):
    if n % k == 0:      # test, zda k deli n
        print(k, end=" ")
```

Při potřebě prošetřit několik konkrétních hodnot vnoříme for do tohoto cyklu for (pozor na odsazení):

```
for n in [98, 99, 101, 102, 998, 999, 1001, 1002]:
    print(n, end="\t")
    for k in range(1, n+1):
        if n % k == 0:    # test, zda k deli n
            print(k, end=" ")
    print()     # vynechat řádek po vypsaných delitelích
```

Výstup programu:

```
98   1 2 7 14 49 98
99   1 3 9 11 33 99
101  1 101
102  1 2 3 6 17 34 51 102
998  1 2 499 998
999  1 3 9 27 37 111 333 999
1001     1 7 11 13 77 91 143 1001
1002     1 2 3 6 167 334 501 1002
```

Pokud několik řádků kódu:

- je třeba opakovaně využívat,
- má jasný smysl i samostatně,

tak by se tato část kódu měla stát funkcí. Kód se tím výrazně zpřehlední a snáze se udržuje.

Funkce, která vypisuje všechny dělitele čísla `n`:

```
def vypis_delitelu(n):
    print(n, end="\t")
    for k in range(1, n+1):
        if n % k == 0:
            print(k, end=" ")
    print()


for číslo in [12, 360, 20, 17]:
    vypis_deliteluč(íslo)
```

Výstup programu:

```
12  1 2 3 4 6 12
360 1 2 3 4 5 6 8 9 10 12 15 18 20 24 30 36 40 45 60 72 90 120 180 360
20  1 2 4 5 10 20
17  1 17
```

Funkce vracející seznam dělitelů

Nevýhodou funkce `vypis_delitelu(n)` je, že nalezené dělitele tiskne. Nemůžeme je pak používat při dalších výpočtech. Řešením je, že nalezené dělitele nebudeme postupně tisknout, ale ukládat do seznamu `D`.

```
def delitele(n):
    D = []
    for k in range(1, n+1):
        if n % k == 0:
            D.append(k)
    return D

print(delitele(60))
```

12.2 Test prvočíselnosti a počet prvočísel (funkce $\pi(n)$)

Pro testování prvočíselnosti zadaného přirozeného čísla $n > 1$ nepotřebujeme všechny dělitele. Stačí testovat, zda postupně některé z čísel mezi 2 a n je dělitelem čísla n. Pokud najdeme dělitele, ukončíme testování a funkce zda_je_prvocislo(n) vrátí False. Pokud jej nenajdeme, je zadané n prvočíslem a funkce zda_je_prvocislo(n) vrátí True.

```
def zda_je_prvocislo(n):
    if n < 2:
        return False
    for d in range(2, n):
        if n % d == 0:
            return False
    return True
```

Tuto funkci můžeme použít například při výpisu všech prvočísel od 1 po zadané číslo n.

```
n = 100

for k in range(n+1):
    if zda_je_prvocislo(k):   # vrátí-li funkce True
        print(k, end=", ")
```

Efektivnější je použít Eratosthenovo síto. Jeho naprogramováním se zabýváme dále v této kapitole.

Funkci zda_je_prvocislo(n) můžeme také použít při programování prvočíselné funkce $\pi(n)$, která vrací počet všech prvočísel menších nebo rovných n. V matematice je funkce $\pi(x)$ definována na celém $\mathbb{R}$; v našem programu by stačilo vzít místo n dolní celou část x.

```
def pi(n):
    pocet = 0
    for k in range(1, n+1):
        if zda_je_prvocislo(k):   # vrátí-li funkce True
            pocet = pocet + 1
    return pocet
```

12.3 Test prvočíselnosti – testování pouze do odmocniny n

Základní pozorování, které podstatně zvýší efektivitu testování prvočíselnosti zadaného n, je založeno na tom, že nepotřebujeme hledat dělitele až do zadaného n, ale stačí do druhé odmocniny z n. Nenajdeme-li dělitele $d \leq \sqrt{n}$, tak už jej nenajdeme. Má-li totiž například číslo 30 dělitele 3, pak má také dělitele 10 (neboť $30 = 3 \cdot 10$)... Dělitelé daného čísla jsou tedy vždy v páru, součin obou prvků takového páru je roven právě n. Pozorujme například všechny dělitele čísla 30:

$$\mathbf{1},\ \textit{2},\ \mathbf{3},\ 5,\qquad 6,\ \mathbf{10},\ \textit{15},\ \mathbf{30}\ .$$

Mezníkem mezi oběma skupinami čísel je $\sqrt{n}$, neboť $\sqrt{n} \cdot \sqrt{n} = n$. A pozorujme páry:

$$30 = 1 \cdot 30 = 2 \cdot 15 = 3 \cdot 10 = 5 \cdot 6\,.$$

Stačí tedy hledat dělitele v první části, tj. mezi čísly $\{2, 3, 4, 5, 6, ..., \lfloor\sqrt{n}\rfloor\}$. Modifikace výše uvedené funkce zda_je_prvocislo(n) je snadná, stačí upravit horní mez v cyklu for.

```
def zda_je_prvocislo_odmoc(n):
    if n < 2:
        return False
    odmocnina = int(n**(1/2)) + 1
    for d in range(2, odmocnina):
        if n % d == 0:
            return False
    return True
```

Případně můžeme zkusit procházet v cyklu for pouze lichá čísla. Je však třeba ošetřit prvočíslo 2 a sudá čísla.

```
def zda_je_prvocislo_odmoc_licha(n):
    if n == 2:
        return True
    elif n % 2 == 0:
        return False
    elif n < 2:
        return False
    odmocnina = int(n**(1/2)) + 1
    for d in range(3, odmocnina, 2):
        if n % d == 0:
            return False
    return True
```

Zkusme nyní porovnat časovou náročnost obou funkcí. Funkce pi(n), pi_odmoc(n) a pi_odmoc_licha(n) se liší pouze tím, že ve svých definicích používají funkce zda_je_prvocislo(n), zda_je_prvocislo_odmoc(n), resp. zda_je_prvocislo_odmoc_licha(n).

```
import time

# bez odmocniny:
s = time.time()

for k in range(1, 6):
    print( "10^{}\t{}". format(k, pi(10**k)) )

t = time.time()
print("čas bez odmocniny:", t - s)
```

```
# s odmocninou:
s = time.time()

for k in range(1, 6):
    print( "10^{}\t{}". format(k, pi_odmoc(10**k)) )

t = time.time()
print("čas s odmocninou: ", t - s)


# s odmocninou a testování pouze lichých čísel:
s = time.time()

for k in range(1, 6):
    print( "10^{}\t{}". format(k, pi_odmoc_licha(10**k)) )

t = time.time()
print("čas s odmocninou: ", t - s)
```

Výpis bude následující:

```
10^1    4
10^2    25
10^3    168
10^4    1229
10^5    9592
cas bez odmocniny: 14.165167331695557

10^1    4
10^2    25
10^3    168
10^4    1229
10^5    9592
cas s odmocninou:  0.11808967590332031

10^1    4
10^2    25
10^3    168
10^4    1229
10^5    9592
cas s odmocninou a lichá: 0.11945843696594238
```

Vidíme, že testování pouze lichých čísel nepřináší zrychlení. Komplikované testování na začátku přináší zdržení, které by jinak nastalo v cyklu for.

12.4 Sudá dokonalá čísla

Součet všech dělitelů $< n$ (včetně 1):

```
def soucet_delitelu(n):
    soucet_delitelu = 1
    for j in range(2, n):
        if not n % j:     # o šestinu šrychlejí žne n % j == 0
            soucet_delitelu += j
    return soucet_delitelu
```

Funkce vracející vektor dokonalých čísel, která hledáme hrubou silou:

```
def dokonala_cisla(n):
    N = 2**n - 1
    dokonala_cisla = []
    for k in range(2, N+1):
        if soucet_delitelu(k) == k:
            dokonala_cisla.append(k)
    return dokonala_cisla

n = 14
D = dokonala_cisla(n)

print("Dokonalá čísla hrubou silou ({}): ".format(n) )
print(D)
```

Výstup programu:

```
Dokonalá čísla hrubou silou (14):
[6, 28, 496, 8128]
```

12.5 Největší společný dělitel – Eukleidův algoritmus

Pomocí Eukleidova algoritmu lze hledat největšího společného dělitele dvou čísel a, b. Ten je v Eukleidově algoritmu roven poslednímu nenulovému zbytku.

Klasickým způsobem, jak Eukleidův algoritmus naprogramovat, je například ten, v němž posun mezi jednotlivými řádky v Eukleidově algoritmu realizujeme výměnou proměnných `a`, `b`, kterou provedeme užitím pomocné proměnné `c`.

```
def NSD(a, b):
    while b != 0:
        c = b
```

```
        b = a % b
        a = c
    return a
```

Použití pomocné proměnné se lze vyhnout pomocí současné inicializace.

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

Velmi stručně lze naprogramovat Eukleidův algoritmus pomocí rekurze.

```
def NSDr(a, b):
    if b == 0:
        return a
    return NSDr(b, a % b)
```

Rekurzi lze provést i jinak, chceme-li zdůraznit postupné odečítání.

```
def NSD_minus2(a, b):
    rozdil = a - b
    if rozdil == 0:
        return a
    elif rozdil < 0:
        rozdil = -rozdil
    return NSD_minus(b, rozdil)
```

Předchozí postup lze zestručnit.

```
def NSD_minus(a, b):
    rozdil = a - b
    if rozdil == 0:
        return a
    return NSD_minus(b, rozdil) if rozdil > 0 else NSD_minus(b, -rozdil)
```

12.6 Rozklad na součin prvočísel

Následující funkce tiskne prvočísla z rozkladu n na prvočísla. Očekává se zadání přirozeného čísla $n > 1$.

```
def prvociselny_rozklad(n):
    print("{} - rozklad na prvocísla:\t".format(n), end="")
```

```
    i = 2
    while n > 1:
        if n % i == 0:      # % zbytek po delení
            print(i, end=" ")
            n = n // i      # // celocíselný podíl
        else:
            i = i + 1
    print()       # vynechat řádek po vypsaných prvocíslech


# výpis prvocísel z rozkladu k na prvocísla
k = int(input("Zadejte číslo, jez rozlozíme na prvocísla: "))

prvociselny_rozklad(k)
```

Výstup programu:

```
Zadejte číslo, žje žrozloíme na prvocísla: 360
360 - rozklad na prvocísla: 2 2 2 3 3 5
```

```
# pri potrebe prosetrit nekolik konkrétních hodnot:
for k in [98, 99, 101, 102, 998, 999, 1001, 1002]:
    prvociselny_rozklad(k)
```

Výstup programu:

```
98 - rozklad na prvocísla:  2 7 7
99 - rozklad na prvocísla:  3 3 11
101 - rozklad na prvocísla: 101
102 - rozklad na prvocísla: 2 3 17
998 - rozklad na prvocísla: 2 499
999 - rozklad na prvocísla: 3 3 3 37
1001 - rozklad na prvocísla:    7 11 13
1002 - rozklad na prvocísla:    2 3 167
```

Možnost vylepšení:

funkce by vrátila pouze hodnoty a je pak na uživateli, v jaké podobě tyto hodnoty vytiskne, případně použije v dalších výpočtech.

12.7 Rozklad na součin prvočísel – seznam

očekává se zadání přirozeného čísla $n > 1$

nová verze s použitím funkce vracející seznam

funkce prvočísla netiskne, ale vrací hodnoty

je pak na uživateli, v jaké podobě tyto hodnoty vytiskne, případně je použije v dalších výpočtech

```
# funkce vrací seznam čprvoísel z rozkladu n na čsouin čprvoísel
def prvociselny_rozklad(n):
    P = []
    d = 2
    while n > 1:
        if n % d == 0:      # % zbytek po ědlení
            P.append(d)
            n = n // d      # // čceloíselný podíl
        else:
            d = d + 1
    return P

# výpis čprvoísel z rozkladu k na čprvoísla
for k in [49, 499, 4999, 49999, 499999, 4999999, 49999999]:
    print( k, prvociselny_rozklad(k) )
```

Výstup programu:

```
49 [7, 7]
499 [499]
4999 [4999]
49999 [49999]
499999 [31, 127, 127]
4999999 [4999999]
49999999 [7, 23, 310559]
```

```
# aplikace čvypoteného rozkladu -- rozklad Fermatova čísla F5
n = 2**2**5 + 1
prv = PrvociselnyRozklad(n)    # volání funkce, prv bude vektor
print("Rozklad F5:  ", prv)
```

Výstup programu:

```
Rozklad F5:   [641, 6700417]
```

```
# jiné žvyuití čvypoteného rozkladu na čprvoísla:
print("šVechna čprvoísla < 100: ")

for n in range(2, 100):
    rozklad = PrvociselnyRozklad(n)
    if len(rozklad) == 1:       # je-li délka seznamu == 1 (tj. obsahuje-
   li rozklad pouze samotné n)
        print(n, end=" ")
```

Výstup programu:

```
š
Vechna čprvoísla < 100:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

12.8 Eratosthenovo síto

Potřebujeme-li seznam prvočísel, která jsou menší nebo rovna zadanému přirozenému číslu $n > 2$, je vhodné použít efektivní postup známý již v době antiky: Eratosthenovo síto.

```
def Eratosthenovo_sito(n):
    n = n + 1
    prvocislo = [True] * n
    for i in range(2, n):
        if prvocislo[i]:
            j = 2 * i
            while j < n:
                prvocislo[j] = False
                j = j + i
    # prvocisla jsou indexy s True
    P = []
    for k in range(2, n):
        if prvocislo[k]:
           P.append(k)
    return P
```

Už tak mimořádně vysokou efektivitu můžeme ještě zvýšit proškrtáváním seznamu čísly volenými pouze do odmocniny n.

```
def Eratosthenovo_sito_odmoc(n):
    n = n + 1
    prvocislo = [True] * n
    odmoc = int(n ** .5)
    for i in range(2, odmoc):
```

```
        if prvocislo[i]:
            j = 2 * i
            while j < n:
                prvocislo[j] = False
                j = j + i
    # prvocisla jsou indexy s True
    P = []
    for k in range(2, n):
        if prvocislo[k]:
            P.append(k)
    return P
```

Porovnejme efektivitu obou funkcí. V prvním případě použijeme funkci Eratosthenovo_sito(n), ve druhém pak její modifikaci s odmocninou. Můžeme pozorovat, že modifikace s odmocninou je rychlejší.

```
import time

n = 10**7

print("Prvocisla do", n)

start = time.time()
P = Eratosthenovo_sito(n)
konec = time.time()
#print(P)
print("cas: ", round(konec - start, 3), "sek.")
print("pocet prvocisel:", len(P))

print()

start = time.time()
P = Eratosthenovo_sito_odmoc(n)
konec = time.time()
#print(P)
print("cas: ", round(konec - start, 3), "sek.")
print("pocet prvocisel:", len(P))
```

Výstup programu pro $n = 10^7$:

```
Prvocisla do 10000000

cas:  2.048 sek.
pocet prvocisel: 664579

cas:  1.256 sek.
pocet prvocisel: 664579
```

Výstup programu pro $n = 10^8$:

```
Prvocisla do 100000000

cas:  23.444 sek.
pocet prvocisel: 5761455

cas:  14.704 sek.
pocet prvocisel: 5761455
```

Zajímavostí je, že pro velká n lze poměrně dobře prvočíselnou funkci $\pi(x)$ aproximovat:

$$\pi(x) \approx \frac{x}{\ln x}\,.$$

```
import math
print("n / ln n: ", int( n / math.log(n) ) )
```

Pro $n = 10^7$ bychom dostali aproximaci 620 420, pro $n = 10^8$ pak 5 428 681.

Kapitola 13

Pýthagorejské trojice

13.1 Generování pýthagorejských trojic

Vypisujeme do tabulky pýthagorejské trojice, tj. uspořádané trojice $[a, b, c]$ splňující podmínku

$$a^2 + b^2 = c^2 \,.$$

Nejprve uvedeme jednoduchou verzi, v níž budou pýthagorejské trojice generované vzorci:

$$a = u^2 - v^2, \quad b = 2uv, \quad c = u^2 + v^2,$$

přičemž $0 < v < u \leq N$.

Počet cyklů for odpovídá počtu parametrů, které se volí.

```
N = 5

for u in range(1, N+1):
    for v in range(1, u):
        print("{}\t{}\t{}".format(u*u - v*v,  2*u*v,  u*u + v*v))
```

Výstup programu:

```
3   4   5
8   6   10
5   12  13
15  8   17
12  16  20
7   24  25
24  10  26
21  20  29
16  30  34
9   40  41
```

13.2 Pýthagorejské trojice „bez násobků“

Všimněme si, že některé trojice obsahují čísla, která jsou pouze násobky čísel jiné trojice, např. $[3, 4, 5]$ a $[8, 6, 10]$. Pokud bychom chtěli vypsat pouze základní pýthagorejské trojice bez násobků, mohli bychom použít Eukleidova algoritmu pro ověření, že jednotlivé prvky jsou čísla nesoudělná. Podmínku nesoudělnosti (tj. největší společný dělitel je roven jedné) stačí ověřit pro a a b.

```
print("Pýthagorejské trojice (bez násobku) do", N)

N = 30      # a se volí do N
            # b se volí do N*N   (má-li být seznam úplný)

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

for a in range(1, N + 1):
    for b in range(a,  N * N + 1):
        if NSD(a, b) == 1:  # pouze nesoudelná a, b
            c = (a * a + b * b) ** (0.5)
            if c == int(c):
                print("{}\t{}\t{}".format(a, b, int(c)))
```

V druhém cyklu (`for b in...`) začínáme od `a`. Pokud bychom začínali od 1, vypsaly by se i trojice se zaměněným a a b, tj. např.:

```
3   4   5
4   3   5
```

Kapitola 14

Obsahy, integrace

14.1 Numerická integrace

Následující program počítá aproximaci Riemannova integrálu obdélníkovou metodou. Používá se ekvidistantní dělení, interval $\langle a, b\rangle$ dělíme na N podintervalů stejné délky $\delta = \frac{b-a}{N}$.

Při výpočtu samotného integrálu se tedy sčítají obsahy jednotlivých obdélníků:

- jejich základna je délka dělicího intervalu delta, tj. $\delta = \frac{b-a}{N}$,
- jejich výšky jsou pak funkční hodnoty počítané ve středu dělicího intervalu, tj. postupně

$$f\left(a+\frac{\delta}{2}\right), \quad f\left(a+\frac{\delta}{2}+\delta\right), \quad f\left(a+\frac{\delta}{2}+2\delta\right), \quad f\left(a+\frac{\delta}{2}+3\delta\right), \ldots, \quad f\left(a+\frac{\delta}{2}+(N-1)\delta\right).$$

```
import math

print("Integrál sinu od a do b obdélníkovou metodou.")

def f(x):
    return math.sin(x)

a = 0
b = math.pi
N = 10**6     # interval <a, b> delíme na N stejných dílu

delta = (b - a) / N
integ = 0

for k in range(N):
    integ += delta * f( delta/2 + k*delta )

print("integrál od", a, "do", b, "je roven: ", integ)
```

Výstup programu:

```
Integrál sinu od a do b obdélníkovou metodou.
integrál od 0 do 3.141592653589793 je roven:  2.0000000000008424
```

Klíčovou část kódu z předchozího programu můžeme zabalit do funkce. Délku základny delta, která je stejná pro všechny obdélníčky, můžeme vytknout.

```
import math

def f(x):
    return (1 - x*x)**(1/2)

def g(x):
    return math.exp(math.sin(x)) * math.cos(x)

def h(x):
    return x * x


def integral(f, a, b, N=10**6):
    zakladna = (b - a) / N
    vyska = 0
    for k in range(N):
        vyska += f(zakladna/2 + k*zakladna)
    return zakladna * vyska


# 4 * obsah jednotkového čtvrtkruhu
print(4 * integral(f, 0, 1))

print(integral(g, math.pi, 2*math.pi))

print(integral(h, 0, 1))
```

Výstup programu:

```
3.1415926539343633
-8.512342306726817e-17
0.3333333333332426
```

14.2 Inspirace Archimédovou aproximací π

Aproximujeme pí pomocí obvodů vepsaných pravidelných n-úhelníků:

$$o_n = 2rn \sin \frac{180°}{n}$$

porovnáním s obvodem kruhu: $o = 2r\pi$ dostáváme aproximaci pí:

$$\pi_n = n \sin \frac{180°}{n}$$

```
import math

print("Aproximace pí pomocí ůobvod vepsaných pravidelných n-ůúhelník:")


for k in range(1, 6):
    n = 10 ** k
    VnitrniUhel = math.pi / n
    pi_n = n * math.sin(VnitrniUhel)
    print(n, "\t", pi_n)
```

Výstup programu:

```
Aproximace pí pomocí ůobvod vepsaných pravidelných n-ůúhelník:
10      3.090169943749474
100         3.141075907812829
1000        3.1415874858795636
10000   3.141592601912665
100000  3.1415926530730216
```

Kapitola 15

Kalendář

15.1 Určení dne v týdnu dle gregoriánského kalendáře

Funkce, která vrací číslo dne v týdnu k zadanému datu (pro neděli vrací nulu):

```
def den_tydne(d, m, r):
    dnu_v_mesici = [0, 0,3,3, 6,1,4,   6,2,5, 0,3,5]  # modulo 7

    if r % 4 == 0:     # prestupne roky
        if r % 100 != 0 or r % 400 == 0:
            for i in range(3, 13):
                dnu_v_mesici[i] = (dnu_v_mesici[i] + 1) % 7

    r = r - 2001
    i = r + r//4 - r//100 + r//400 + dnu_v_mesici[m] + d

    return i % 7
```

Tato funkce vychází poměrně názorně z pravidel gregoriánského kalendáře. Musíme také znát k jednomu datu, o jaký den v týdnu se jedná: volíme pondělí 1. 1. 2001 (pondělí – začátek týdne, navíc 1. ledna, tj. také začátek roku). Z tohoto údaje pak lze dopočítat den týdne pro libovolné datum.

```
dnu_v_mesici = [0, 0,3,3, 6,1,4,   6,2,5, 0,3,5]
```

Základní údaje:

- 2001 - 2001 + den(1.1.) = 1 = pondělí (1. 1. 2001 je pondělí)
- další rok bude + 1, protože 365 % 7 = 1 (350 + 14)
  v roce po přestupném roce bude + 2

gregoriánská korekce délky roku: místo 365,25 má být 365,2422:

$$1/4 - 1/100 + 1/400 = 0.2425$$

- každý 4. rok je přestupný
- kromě roků dělitelných 100
- ale roky dělitelné 400 přestupné jsou

Celkem (vše % 7):

- + (rok - 2000) (1. 1. 2001 je pondělí) také: (rok-5) % 7 či (rok+2) % 7
- + (rok - 2001) // 4 = počet přestupných roků po roce 2001 také: (rok–1) // 4
- - (rok - 2001) // 100 = počet přestupných roků po roce 2001 také: (rok–1) // 100
- + (rok - 2001) // 400 = počet roků po roce 2001 dělitelných 400: jsou přestupné
- + kolikátý den v roce – 1 (1.1.2001 už je 1, nic už tedy nepřičítáme)
  1 se u přestupných roků přičítá až k roku po přestupném, tj. k 2005, 2009, ...

Celkem (sečteno):

- r = r – 2001
- i = (r+1) + r//4 – r//100 + r//400 + dnu_v_mesici[m] + (d–1)

Poslední řádek ještě upravíme ($r + 1 + d - 1 = r + d$):

- i = r + r//4 - r//100 + r//400 + dnu_v_mesici[m] + d

15.2 Recyklace kalendáře

Filoména dostala od babičky zprávu, že má krásný starý kalendář z roku 1960 (všimněme si, že se jedná o přestupný rok). Pomozte jí napsat program (s využitím funkce den_tydne(d, m, r)), který by vypsal všechny roky od 2024 do 2080, kdy by bylo možno tento kalendář znovu použít.

Řešení je snadné:

- Zjistíme, který den v týdnu bylo 1. ledna 1960.
- Pomocí cyklu for projdeme všechny přestupné roky od 2024 do 2080 (přestupným rokem je tu každý 4. rok) a otestujeme, zda se den v týdnu 1. ledna v takovém roce shoduje s dnem v týdnu pro 1. leden 1960.

```
den1960 = den_tydne(1, 1, 1960)

for rok in range(2024, 2081, 4):
    if den_tydne(1, 1, rok) == den1960:
        print(rok)
```

Výstup programu:

```
2044
2072
```

15.3 Pátky třináctého

Ke každému roku chceme vypsat počet pátků třináctého.

```
def pocet_patku_13(rok):
    patku13 = 0
    for mesic in range(1, 13):
        if den_tydne(13, mesic, rok) == 5:
            patku13 = patku13 + 1
    return patku13

print("Pocet pátku 13. v daném roce:")
for rok in range(2020, 2031):
    print(rok, pocet_patku_13(rok), end=",\t")
```

Pokud bychom chtěli vypsat i měsíce, v nichž je obsažen pátek třináctého, tak by kód mohl vypadat např. takto.

```
def vypsat_patky_13(rok):
    patky13 = []
    for mesic in range(1, 13):
        if den_tydne(13, mesic, rok) == 5:
            patky13.append(mesic)
    return patky13

print("Mesíce, v nichz jsou v daném roce pátky 13.:")
for rok in range(2020, 2031):
    print(rok, vypsat_patky_13(rok), end="\t")
```

Výstup programu:

```
Mesíce, v nichz jsou v daném roce pátky 13.:
2020 [3, 11]    2021 [8]    2022 [5]    2023 [1, 10]    2024 [9, 12]
2025 [6]    2026 [2, 3, 11] 2027 [8]    2028 [10]   2029 [4, 7]
2030 [9, 12]
```

Zajímavé je pozorování, že v žádném roce nemůže být více než 3 pátky třináctého. Všechny roky s maximálním počtem pátků třináctého je možno si jednoduše vypsat:

```
def pocet_patku_13(rok):
    patku13 = 0
    for mesic in range(1, 13):
        if den_tydne(13, mesic, rok) == 5:
            patku13 = patku13 + 1
    return patku13

print("Roky, v nichz jsou 3 patky 13.:")
for rok in range(1600, 2102):
    if pocet_patku_13(rok) == 3:
        print(rok, end="\t")
```

Výstup programu:

```
Roky, v nichz jsou 3 patky 13.:
1609    1612    1615    1626    1637    1640    1643    1654    1665
1668    1671    1682    1693    1696    1699    1705    1708    1711
1722    1733    1736    1739    1750    1761    1764    1767    1778
1789    1792    1795    1801    1804    1807    1818    1829    1832
1835    1846    1857    1860    1863    1874    1885    1888    1891
1903    1914    1925    1928    1931    1942    1953    1956    1959
1970    1981    1984    1987    1998    2009    2012    2015    2026
2037    2040    2043    2054    2065    2068    2071    2082    2093
2096    2099
```

15.4 Kalendář na celý rok – funkce prcal()

V knihovně calendar je předdefinovaná poměrně netypická funkce prcal() (zkratka z *print calendar*).

```
import calendar

rok = int(input("Zadejte rok: "))
calendar.prcal(rok)
```

Výstup programu:

```
Zadejte rok: 2025
                                  2025

      January               February               March
Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
        1  2  3  4  5                  1  2                  1  2
 6  7  8  9 10 11 12   3  4  5  6  7  8  9   3  4  5  6  7  8  9
13 14 15 16 17 18 19  10 11 12 13 14 15 16  10 11 12 13 14 15 16
20 21 22 23 24 25 26  17 18 19 20 21 22 23  17 18 19 20 21 22 23
27 28 29 30 31        24 25 26 27 28        24 25 26 27 28 29 30
                                            31

      April                  May                   June
Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
    1  2  3  4  5  6               1  2  3  4                  1
 7  8  9 10 11 12 13   5  6  7  8  9 10 11   2  3  4  5  6  7  8
14 15 16 17 18 19 20  12 13 14 15 16 17 18   9 10 11 12 13 14 15
21 22 23 24 25 26 27  19 20 21 22 23 24 25  16 17 18 19 20 21 22
28 29 30              26 27 28 29 30 31     23 24 25 26 27 28 29
                                            30

       July                  August               September
Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
    1  2  3  4  5  6                  1  2  3   1  2  3  4  5  6  7
 7  8  9 10 11 12 13   4  5  6  7  8  9 10   8  9 10 11 12 13 14
14 15 16 17 18 19 20  11 12 13 14 15 16 17  15 16 17 18 19 20 21
21 22 23 24 25 26 27  18 19 20 21 22 23 24  22 23 24 25 26 27 28
28 29 30 31           25 26 27 28 29 30 31  29 30

      October               November              December
Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
        1  2  3  4  5                  1  2   1  2  3  4  5  6  7
 6  7  8  9 10 11 12   3  4  5  6  7  8  9   8  9 10 11 12 13 14
13 14 15 16 17 18 19  10 11 12 13 14 15 16  15 16 17 18 19 20 21
20 21 22 23 24 25 26  17 18 19 20 21 22 23  22 23 24 25 26 27 28
27 28 29 30 31        24 25 26 27 28 29 30  29 30 31
```

15.5 Datum Velikonoční neděle

Počítáme datum Velikonoc (po roce 1583). Teoreticky se jedná o první neděli po prvním jarním úplňku.

% je zbytek po dělení

// je celá část podílu

```
def velik(R):
 a = R % 19
 b = R // 100
 c = R % 100
 d = (19 * a + b - b // 4 - ((b - (b + 8) // 25 + 1) // 3) + 15) % 30
 e = (32 + 2 * (b % 4) + 2 * (c // 4) - d - (c % 4)) % 7
 f = d + e - 7 * ((a + 11 * d + 22 * e) // 451) + 114
 M = f // 31
 D = f % 31 + 1
 return [D, M, R]

for rok in range(2000, 2112):
    print( velik(rok) )
```

Výstup programu (upraveno do sloupců):

23.4.2000	20.4.2014	16.4.2028	6.4.2042	2.4.2056	30.3.2070	26.3.2084	20.4.2098
15.4.2001	5.4.2015	1.4.2029	29.3.2043	22.4.2057	19.4.2071	15.4.2085	12.4.2099
31.3.2002	27.3.2016	21.4.2030	17.4.2044	14.4.2058	10.4.2072	31.3.2086	28.3.2100
20.4.2003	16.4.2017	13.4.2031	9.4.2045	30.3.2059	26.3.2073	20.4.2087	17.4.2101
11.4.2004	1.4.2018	28.3.2032	25.3.2046	18.4.2060	15.4.2074	11.4.2088	9.4.2102
27.3.2005	21.4.2019	17.4.2033	14.4.2047	10.4.2061	7.4.2075	3.4.2089	25.3.2103
16.4.2006	12.4.2020	9.4.2034	5.4.2048	26.3.2062	19.4.2076	16.4.2090	13.4.2104
8.4.2007	4.4.2021	25.3.2035	18.4.2049	15.4.2063	11.4.2077	8.4.2091	5.4.2105
23.3.2008	17.4.2022	13.4.2036	10.4.2050	6.4.2064	3.4.2078	30.3.2092	18.4.2106
12.4.2009	9.4.2023	5.4.2037	2.4.2051	29.3.2065	23.4.2079	12.4.2093	10.4.2107
4.4.2010	31.3.2024	25.4.2038	21.4.2052	11.4.2066	7.4.2080	4.4.2094	1.4.2108
24.4.2011	20.4.2025	10.4.2039	6.4.2053	3.4.2067	30.3.2081	24.4.2095	21.4.2109
8.4.2012	5.4.2026	1.4.2040	29.3.2054	22.4.2068	19.4.2082	15.4.2096	6.4.2110
31.3.2013	28.3.2027	21.4.2041	18.4.2055	14.4.2069	4.4.2083	31.3.2097	29.3.2111

Modifikace:

1. Upravte program tak, aby data vypisoval ve formátu den. měsíc. rok
2. Upravte modifikaci 1 tak, aby se měsíc vypisoval slovně (března, dubna).

Kapitola 16

Náhodná čísla

16.1 Hra: hádání čísla (while)

```
import random

cislo_ktere_mam_uhodnout = random.randint(1, 10)

muj_odhad = int(input("Hádejte číslo od 1 do 10: "))

while muj_odhad != cislo_ktere_mam_uhodnout:
    print("neuhodli jste")
    muj_odhad = int(input("Hádejte jeste jednou: "))

print(muj_odhad, "je správné číslo, vyhráli jste!")
```

16.1.1 Modifikace s indikací, zda je náš odhad příliš malý či velký

```
import random

cislo_ktere_mam_uhodnout = random.randint(1, 10)

muj_odhad = int(input("Hadejte cislo od 1 do 10: "))

while muj_odhad != cislo_ktere_mam_uhodnout:
    if muj_odhad > cislo_ktere_mam_uhodnout:
        print(muj_odhad, "je prilis velke.")
    if muj_odhad < cislo_ktere_mam_uhodnout:
        print(muj_odhad, "je prilis male.")
    muj_odhad = int(input("Hadejte jeste jednou: "))

print(muj_odhad, "je spravne cislo, vyhrali jste!")
```

16.2 Hra: kámen, nůžky, papír

```
import random

kamen = "kámen"
nuzky = "ůžnky"
papir = "papír"

CoChcete = "Chcete {}, {}, nebo {} č(i konec - Enter)? ".format(kamen,
   nuzky, papir)

vyhra = "Vyhráli jste!"
prohra = "Vyhrál ččpoíta..."
nerozhodne = "ěNerozhodn..."

print("Kámen tupí ůžnky. ůžNky řstíhají papír. Papír balí kámen.\n")

moznosti = [kamen, nuzky, papir]

hrac = input(CoChcete)

while hrac != "":           # Hrajeme, dokud nestiskneme Enter
    hrac = hrac.lower()     # řpevod na malá písmena
    pocitac = random.choice(moznosti)    # volba jedné z žmoností
    print("Vybrali jste " + hrac + ", ččpoíta vybral " + pocitac + ".")

    if hrac == pocitac:
        print(nerozhodne)
    elif hrac == kamen:
        if pocitac == nuzky:
            print(vyhra)
        else:
            print(prohra)
    elif hrac == papir:
        if pocitac == kamen:
            print(vyhra)
        else:
            print(prohra)
    elif hrac == nuzky:
        if pocitac == papir:
            print(vyhra)
        else:
            print(prohra)
    else:
        print("ěNkde se asi stala chyba...")

    print()
```

```
    hrac = input(CoChcete)
```

16.3 Karty

16.3.1 Rozdáváme karty

```
import random

hodnoty = ["2","3","4","5","6","7","8","9","10","J","Q","K","A"]
barvy = ["kríze", "káry", "srdce", "piky"]

# rozdáme karty
print("Mám tyto karty:")
for i in range(7):
    moje_hodnota = random.choice(hodnoty)
    moje_barva = random.choice(barvy)
    print(moje_hodnota, moje_barva, end="   ")
```

Výstup programu:

```
Mam tyto karty:
6 piky   J piky   2 kríze   3 srdce   K káry   6 kríze   A kríze
```

16.3.2 Hra s kartami: přebíjená

```
import random

hodnoty = ["2", "3", "4", "5", "6", "7", "8", "9","10", "J", "Q", "K", "A
   "]
barvy = ["kríze", "káry", "srdce", "piky"] # na tomto hra nezávisí

pokracovat = True

while pokracovat:
    moje_hodnota = random.choice(hodnoty)
    moje_barva = random.choice(barvy)
    vase_hodnota = random.choice(hodnoty)
    vase_barva = random.choice(barvy)
    print("Mám: ", moje_hodnota, moje_barva)
    print("Máte:", vase_hodnota, vase_barva)

    if hodnoty.index(moje_hodnota) > hodnoty.index(vase_hodnota):
```

```
        print("Vyhrál jsem!")
    elif hodnoty.index(moje_hodnota) < hodnoty.index(vase_hodnota):
        print("Vyhráli jste!")
    else:
        print("Nerozhodne...")

    pokracovat = (input("Pokracovat (Enter) / konec (lib. klávesa): ") ==
    "")
    print()
```

Výstup programu:

```
Mám:  10 piky
Máte: Q srdce
Vyhráli jste!č
Pokraovat (Enter) / konec (lib. klávesa):

Mám:  3 kríze
Máte: 4 piky
Vyhráli jste!č
Pokraovat (Enter) / konec (lib. klávesa): d
```

16.4 Sportka

Jak hrát Sportku? Stačí si tipnout 6 čísel ze 49.

Ve hře Sportka se výhry dělí následovně:

- šesti čísel, získává sázející výhru 1. pořadí,
- pěti čísel a dodatkového čísla (5 + 1), získává sázející výhru 2. pořadí,
- pěti čísel, získává sázející výhru 3. pořadí,
- čtyř čísel, získává sázející výhru 4. pořadí,
- tří čísel, získává sázející výhru 5. pořadí.

```
import random

def tipovat():
    tipy = []
    print("Tipujte 6 úrzných čísel:")
    for i in range(6):
        cislo = int(input("č{}.íslo: ".format(i+1)))
        while cislo in tipy:
```

```
            cislo = int(input("Toto číslo ži bylo zvoleno, tipujte jiné:
    "))
        else:
            tipy.append( cislo )
    return tipy

def losovat():
    losovani = []
    for i in range(6):
        cislo = random.randint(1, 49)
        while cislo in losovani:
            cislo = random.randint(1, 49)
        else:
            losovani.append( cislo )
    return losovani

def spravnych_a_poradi(tipy, losovani):
    PocetSpravnychTipu = 0
    for i in tipy:
        if i in losovani:
            PocetSpravnychTipu += 1
    # řpoadí
    if PocetSpravnychTipu > 2: # výhra
        if PocetSpravnychTipu == 3:
            poradi = 5
        elif PocetSpravnychTipu == 4:
            poradi = 4
        elif PocetSpravnychTipu == 5 and tipy[5] == losovani[5]:
            poradi = 2
        elif PocetSpravnychTipu == 5:
            poradi = 3
        elif PocetSpravnychTipu == 6:
            poradi = 1
    else:
        poradi = 6              # prohra

    return [PocetSpravnychTipu, poradi]

def vyhodnotit(SpravnychPoradi):
    print("čPoet správných čísel: ", SpravnychPoradi[0])
    if SpravnychPoradi[1] < 6:
        print("Vyhráli jste {}. řpoadí".format(SpravnychPoradi[1]))
    elif SpravnychPoradi[1] == 6:
        print("Nevyhráli jste")
```

Pokud bychom chtěli dělat opakované pokusy s tipováním Sportky, mohli bychom připojit např. následující cyklus.

```
opakovat = ""
while opakovat == "":
    #tipy = tipovat()  # čruní tipování
    tipy = losovat()  # automatický čtipova (modelování ůtip u řžpepáky)
    print("šVae tipy:        ", tipy)

    losovani = losovat()
    print("Výsledek losování:", losovani)

    SpravnychPoradi = spravnych_a_poradi(tipy, losovani)

    vyhodnotit(SpravnychPoradi)
    opakovat = input("Opakovat? (Enter)\n")
```

Pokud bychom chtěli modelovat sázení každý týden po dobu produktivního života (např. 40 let: začneme sázet v 25 letech a skončíme v 65), tak bychom mohli připojit např. následující kód.

Funkce vypíše přehled výher za n let. Sázíme jeden sloupeček na slosování ve středu i v neděli, což činí přibližně 100 sázek za rok.

```
def PrehledVyherZa_n_Let(n):
    print("řPehled výher za {} let".format(n))
    PocetSazek = int( (2 * n * 365.25) //7)  # n let sázení ve stredu i
   nedeli
    KolikratVyhra = 0
    Kolikrat4poradi = 0
    KolikratLepsiNez4poradi = 0
    for i in range(PocetSazek):
        tipy = losovat()  # automatické tipování (modelování nasich ůtip)
        losovani = losovat()
        SpravnychPoradi = spravnych_a_poradi(tipy, losovani)

        if SpravnychPoradi[1] < 6:
            #print("šVae tipy:        ", tipy)
            #print("Výsledek losování:", losovani)
            KolikratVyhra += 1
            if SpravnychPoradi[1] == 4:
                Kolikrat4poradi += 1
            if SpravnychPoradi[1] < 4:
                print("- Vyhráli jste {}. poradí (na {} pokus).".format(
   SpravnychPoradi[1], i+1))
                KolikratLepsiNez4poradi += 1

    print("Vyhráli jste celkem {}krát, z toho: \n{}krát 4. poradí, \n{}
   krát lepsí poradí nez 4.\n".format(KolikratVyhra, Kolikrat4poradi,
   KolikratLepsiNez4poradi))
```

```
for i in range(4):
    PrehledVyherZa_n_Let(40)
```

Výstup programu vypadá typicky nějak takto:

```
Prehled výher za 40 let
Vyhráli jste celkem 70krát, z toho:
3krát 4. poradí,
0krát lepsí poradí nez 4.

Prehled výher za 40 let
Vyhráli jste celkem 68krát, z toho:
5krát 4. poradí,
0krát lepsí poradí nez 4.

Prehled výher za 40 let
Vyhráli jste celkem 85krát, z toho:
3krát 4. poradí,
0krát lepsí poradí nez 4.

Prehled výher za 40 let
Vyhráli jste celkem 69krát, z toho:
4krát 4. poradí,
0krát lepsí poradí nez 4.
```

Je vidět, že tento program může být výchovný, lze si nasimulovat celý život sázení a ušetřit tak peníze i čas.

Kapitola 17

Želví grafika

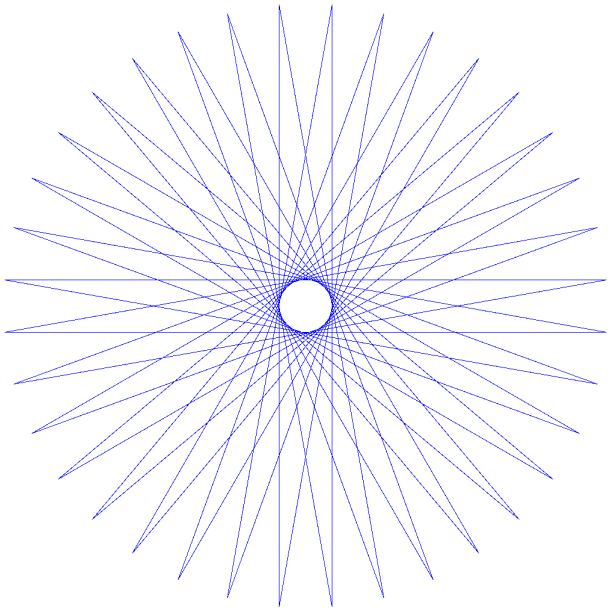
17.1 Hvězda

```
from turtle import *

color('blue')
pocatek = (-500, -40)  # stred vnitrní kruznice hvezdice
setposition(pocatek)
clear()

while True:
    forward(1000)  # posun vpred
    left(170)      # otocení o daný pocet stupnu
    if abs(pocatek[0] - pos()[0]) < 1 and abs(pocatek[1] - pos()[1]) < 1:
        break

hideturtle()
```



17.2 Barevná spirála

```
import turtle

barvy = ['red', 'purple', 'blue', 'green', 'yellow', 'orange']

t = turtle.Pen()

turtle.bgcolor('black')

for n in range(360):
    t.pencolor(barvy[n % 6])
    t.width(1  +  n / 100)
    t.forward(n)
    t.left(59)

turtle.hideturtle()
```