Python 3

Zdeněk Halas
KDM MFF UK

2023

Obsah

Část I

Úvod do programování

Kapitola 1

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

1.1 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.)

1.2 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)

1.3 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

1.4 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

1.5 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)

1.6 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

Kapitola 2

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 3.12 nebo vyšší).
- Oficiální tutoriál pro zájemce: https://docs.python.org/3/tutorial/.

2.1 Instalace – Windows

- Na stránce https://www.python.org na záložce Downloads zvolit pod nadpiskem *Download for Windows* tlačítko Python 3.12.0 (případně vyšší aktuální verze).
- Stažený soubor python-3.12.0-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.

2.2 Instalace – Mac

- Na stránce https://www.python.org na záložce Downloads zvolit pod nadpiskem *Download for macOS* tlačítko Python 3.12.0 (případně vyšší aktuální verze).
- Stažený soubor python-3.12.0-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.

2.3 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 python3.12 a idle-python3.12) nebo pomocí standardních příkazů zadaných v terminálu.

Například v operačních systémech založených na Debianu (např. populární Ubuntu (verze 23.10) a jeho různé deriváty) jsou příkazy následující:

```
sudo apt install python3.12
sudo apt install idle-python3.12
```

Spuštění programu v IDLE: F5.

2.4 Instalace – Android

- V Obchod Play doporučuji např. aplikaci Pydroid 3 – IDE for Python 3.

Kapitola 3

Základní příkazy a konstrukce

3.1 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)
   ```

3.2 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,
```

3.2.1 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
```

3.2.2 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
```

3.3 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,
```

3.3.1 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).

3.4 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
```

3.4.1 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))
```

3.5 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)
```

3.5.1 Hledání kořene metodou zkusmo

Funkce $f(x) = 2^x - x^2$ nabývá v bodě -1 zápornou hodnotu $f(-1) = -0,5$.

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

x = -1      # f(-1) = -0.5
krok = 0.001

while f(x) < 0:
    x = x + krok

print(x, f(x))
print(x-krok, f(x-krok))
```

3.5.2 Hra – hádání čísla

Tipujeme číslo, dokud (while) jej neuhodneme.

```
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.

3.5.3 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.

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

3.6 Základy práce s řetězci

```
s = "Toto je ůmj první řěetzec v Pythonu."
print(s)     # švypíe řěetzec s



# indexuje se od nuly

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

# tisk 2. ža 3. znaku                    ot
```

```
print(s[1:3])

# tisk od 9. znaku ža do konce           ůmj první řěetzec v Pythonu.
print(s[8:])

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



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

c = a + " " + b
print(c)        # matematická analýza
print(len(c))   # délka řěetzce: 19
```

3.7 Seznamy

```
# vektor 5 nul: [0, 0, 0, 0, 0]
a = [0 for n in range(5)]
print(a)


# vektor 5 nul: [0, 0, 0, 0, 0]
a = [0] * 5
print(a)

a[1] = 5
print(a)           # [0, 5, 0, 0, 0]


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

Výstup programu:

3.8 Zápis do souboru

```
# řoteveme souborů
mj_soubor = open("ůMj název souboru.txt", "w")

i = 12

# zapisujeme do souboruů
mj_soubor.write(str(i) + '\t' + "Ahoj, zapisuji do souboru." + '\n')ů
mj_soubor.write("A šějet ěnco na šdalí řádek.")

# řzaveme souborů
mj_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 řěůetzc (primitivní postup)
import math # importování knihovny matematických funkcí, v žní 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()
```

```
# žtentý program, žvyuití formátovaného řěetzce (šlepí postup)
import math

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

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

3.9 Výpočty s libovolnou přesností – modul decimal

```
import decimal
decimal.getcontext().prec = 100     # nastavení řpesnosti na 100 míst

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

Výstup programu:

```
1/97 = 0.01030927835051546391752577319587628865979381443298969
   072164948453608247422680412371134020618556701031
0.010309278350515464
```

```
# odmocnina s řpesností na 100 míst
print("sqrt(5) =", decimal.Decimal(5).sqrt())

# e s řpesností na 100 míst
print("exp(1) =", decimal.Decimal(1).exp())
```

```
# správné (pomocí řěetzce) a nesprávné (řpevodem z float) zadání
   desetinného čísla decimal
a = decimal.Decimal("0.1")
print("číslo 0,1 řěpesn (decimal žvyaduje řěetzec):", a)

b = decimal.Decimal(0.1)        # žzatíeno řpevodem z dvojkové soustavy,
   v žní je float v ččpoítai žuloeno
print("číslo 0,1 řěnepesn (float řpevedeno na decimal):", b)
```

Výstup programu:

```
č
íslo 0,1 řěpesn (decimal žvyaduje řěetzec): 0.1č

íslo 0,1 řěnepesn (float řpevedeno na decimal):
   0.1000000000000000055511151231257827021181583404541015625
```

Část II

Programování v rámci různých (nejen) matematických témat

Kapitola 4

Součty

4.1 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).

4.2 Výpočet součtu prvních n členů dané řady

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

print("čPoítáme e pomocí Taylorova rozvoje funkce exp x")

soucet = 1
clen = 1
pocet_scitanych_clenu = 18

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

Výstup programu:

```
č
Poítáme 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
```

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

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

print("čsouet čůlen řady:   ", s)


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

Výstup programu:

```
č
souet čůlen řady:    0.6931476805552527
pro kontrolu:  ln 2 = 0.6931471805599453
```

4.3 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 $\left\langle 0, \frac{\pi}{2}\right\rangle$.

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 - ěvestavná hodnota sinu:
from math import sin
print(sin(1))
```

Výstup programu:

0.8414709848078965066525023216
0.841470984807896506652502321630298999622563060798 37
0.8414709848078965

Kapitola 5

Opakování základních konstrukcí

5.1 if – automatické slovní hodnocení na základě známky

Následující funkce nevrací žádnou hodnotu.

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


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

Výstup programu:

```
Zadejte známku: 1
1   výborný
```

Modifikace – funkce vracející řetězec (return místo print):

```
def hodnoceni_return(n):
    if n == 1:
```

```
        return 'výborný'
    elif n == 2:
        return 'chvalitebný'
    elif n == 3:
        return 'dobrý'
    elif n == 4:
        return 'čdostatený'
    elif n == 5:
        return 'čnedostatený'
    else:
        return 'takovou známku nemáme'


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

Výstup programu:

```
Zadejte známku: 1
1 výborný
```

5.2 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 bylo uvězněno v jeskyni římskými vojáky. Zvolili sebevraždu před zajetím a rozhodli se pro sériovou metodu sebevraždy losováním. Iosephus uvádí, že štěstím nebo možná rukou Boží zůstali on a další muž až do konce a raději se vzdali Římanům, než aby se zabili.

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“. Posun je 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()
```

5.3 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       # číslo, žjeho odmocninu čpoítáme
N = 15  # čpoet iterací řpi čvýpotu odmocniny (pro ěšvtí řpesnost je
   řpoteba více iterací)

a = x       # první člen posloupnosti

for i in range(1, N+1):
    a = (x + a*a) / (2*a)       # čvýpoet n-tého členu 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       # první člen posloupnosti

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

    return a


print("Číslo  Odmocnina")

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

Výstup programu:

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

```
# testování vlivu čpotu iterací na řpesnost
print("Odmoc(5) čpoítaná pomocí n iterací:")

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

Výstup programu:

```
Odmoc(5) čpoítaná 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
```

Kapitola 6

Rekurze

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

6.1 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
```

6.2 Geometrická posloupnost

Na příkladu geometrické posloupnosti můžeme pozorovat, že rekurzi lze použít i u funkcí, které mají více než jeden parametr. Zde je druhý parametr kvocientem.

```
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))
```

6.3 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.

6.4 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()
```

Výpocet 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
```

Kapitola 7

Faktoriály a kombinační čísla

7.1 Faktoriály

7.2 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
```

```
# faktoriál pomocí rekurze - funkce volá sama sebe
def faktorial_rekurze(n):
    if n < 2:
        return 1
    else:
```

```
        return n * faktorial_rekurze(n - 1)



# šlo by také:
def faktorial_rekurze_nula(n):
    if n <= 0:
        return 1
    else:
        return n * faktorial_rekurze_nula(n - 1)



# čěšstrunjí zápis, je i šrychlejí
def čěfaktorial_rekurze_strun(n):
    return 1 if n < 2 else n * čěfaktorial_rekurze_strun(n - 1)
```

Ještě zajistíme výpisy. Budeme přitom měřit čas, abychom alespoň přibližně porovnali efektivitu jednotlivých funkcí.

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

n = 21000       # budeme čpoítat n!
# poslední n, pro které rekurze funguje: 993,
# řpi nastavení ěšvtí hloubky rekurze je limitč

as1_for = time.time()
f = faktorial_for(n)č
as2_for = time.time()
print("for:\t", čas2_for - čas1_for)č

as1_while = time.time()
f = faktorial_while(n)č
as2_while = time.time()
print("while:\t", čas2_while - čas1_while)č

as1_rekurze = time.time()
f = faktorial_rekurze(n)č
as2_rekurze = time.time()
print("rekurze:\t", čas2_rekurze - čas1_rekurze)ččě
```

```
as1_rekurze_strun = time.time()
f = čěfaktorial_rekurze_strun(n)ččě
as2_rekurze_strun = time.time()
print("rekurze-čěstrun:\t", ččěas2_rekurze_strun - ččěas1_rekurze_strun)

from math import factorialčě
as1_vestavný = time.time()
f = factorial(n)čě
as2_vestavný = time.time()
print("ěvestavný:\t", čěas2_vestavný - čěas1_vestavný)
```

Výstup programu obsahuje dobu výpočtu v sekundách.

```
for:     0.08006596565246582
while:   0.07624602317810059
rekurze:     0.07684326171875
rekurze-čěstrun:   0.07609891891479492ě
vestavný:    0.011753082275390625
```

7.3 Kombinační čísla

```
# POZOR: nespolehlivá od n = 55
# POZOR - zde je ěsprávn jen prvních 15 cifer, pak chyba zaokrouhlení
   float
# navíc ůkvli float:   ěmaximáln lze   kombin_mala(1020, 510)
# navíc kombin_float(n, k) == 0 pro n = 55 60 61 66 67 68 72 73 74 75 76
   82 89 90 92 94 95 96 100 102 103 104 105 107 110 111 113 120 123 124
   126 128 133 137 138 141 142 143 147 149 150 153 157 160 161 162 168
   169 171 176 177 181 182 183 187 191 194 195 196 199 200 207 213 217
   221 223 224 226 227 229 230 231 232 235 236 238 241 245 252 253 254
   255 258 259 261 262 263 265 266 267 279 280 281 284 286 289 290 294
   295 298 301 303 305 306 308 309 310 311 312 313 315 317 318 319 323
   324 325 327 329 331 335 338 339 341 343 345 346 349 350 351 353 356
   358 359 365 366 369 370 371 375 377 378 379 380 381 382 384 385 386
   388 389 390 391 392 393 395 396 398 399 400 402 406 408 409 411 414
   416 419 422 423 426 427 429 430 432 433 435 439 441 442 444 451 452
   453 454 455 462 464 465 466 469 470 472 473 474 475 477 478 479 480
   481 483 486 488 489 492 493 494 497 499 ...
# čkombinaní čísla n nad k
def kombin_float(n, k):
    komb = 1
    for i in range(1, k+1):
        komb = komb * (n-i+1) / i
```

```
    return int(komb)


# spolehlivé a šnejrychlejí (float je šrychlejí, ale nespolehlivý)
def kombin_int(n, k):
    komb = 1
    for i in range(1, k+1):
        komb = komb * (n-i+1) // i
    return komb


# čkombinaní čísla n nad k  - spolehlivá funkce i pro velké vstupní
   hodnoty
# velmi pomalé: 2x špomalejí žne kombin_int
def kombin_velka(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


# čkombinaní čísla pomocí ůzlomk - eliminace chyby zaokrouhlení
# funguje tedy ěspolehliv i pro obrovská čísla
# velmi pomalé: 20x špomalejí žne kombin_int
from fractions import Fraction

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

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

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


# kombin_rekurze() - omezení hloubkou rekurze
print("čVýpoet 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)


# testování relativní spolehlivosti
N = 100
for n in range(N+1):
    for k in range(n+1):
        if kombin_velka(n, k) != kombin_int(n, k):
            print("pozor", n, k)
print("porovnavani: hotovo")


print("testování spolehlivosti - čsouet řůádk 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")
```

7.3.1 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:
   2806267768299625667959060022008440015720504096744589031092689770644730233771
1


zlomky:
   2806267768299622710394143078827353251689508172425042258521847433843034674590
5


velká:
   2806267768299622710394143078827353251689508172425042258521847433843034674590
5

cifer:  306
2,80626776829962 * 10na305


Pro kontrolu - čsouet komb. čísel v 1 řádku Pascalova trojúh. = 2**n:
2**n:
1123558209288947442330815744243140458511235611838941607958938007235829223784381
0


Bez konverze z float je čvýpoet velkých čkombinaních čísel spolehlivý:č
souet řůádk Pascalova trojúhelníku:
1123558209288947442330815744243140458511235611838941607958938007235829223784381
0


S konverzí z float čvýpoet velkých čkombinaních čísel funguje jen pro
   prvních 15 cifer:č
souet řůádk Pascalova trojúhelníku:
1123558209288948512619863718132261072297884338670253585773315749780405014992063
6


S modulem frac je čvýpoet velkých čkombinaních čísel spolehlivý:č
souet řůádk Pascalova trojúhelníku:
1123558209288947442330815744243140458511235611838941607958938007235829223784381
0
```

Kapitola 8

Dělitelnost

8.1 Seznam všech dělitelů

```
# Vypisují se ědlitelé zadaného řpirozeného čísla.     čOekává se zadání
   řpirozeného čísla n > 1.
n = int( input("Zadejte číslo, žjeho ědlitelé mají být vypsáni: ") )

for k in range(1, n+1):
    if n % k == 0:      # je-li zbytek po ědlení číslem k nulový, tak je
   k ědlitel a tiskneme jej
        print(k, end=" ")
```

```
# řpi řěpoteb šřproetit ěnkolik konkrétních hodnot řvnoíme 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:    # je-li zbytek po ědlení číslem k nulový, tak
   je k ědlitel a tiskneme jej
            print(k, end=" ")
    print()     # vynechat řádek po vypsaných ědlitelích
```

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 švechny ědlitele 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
```

Výpis dělitelů zadaného čísla – další možnost použití definované funkce:

```
n = int(input("Zadejte šdalí číslo, žjeho ědlitelé mají být vypsáni: "))
Vypis_delitelu(n)    # volání funkce
```

8.2 Sudá dokonalá čísla

Součet všech dělitelů $<$ n (vč. 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]
```

8.3 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)
```

8.4 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.

8.5 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 Frematova 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
```

Kapitola 9

Hledání přirozených čísel dané vlastnosti

9.1 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
```

9.2 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 10

Obsahy, integrace

10.1 Numerická integrace

Následující program počítá aproximaci Riemannova integrálu obdélníkovou metodou. Používá se ekvidistantní dělení, sčítají se obsahy jednotlivých obdélníků: jejich základna je Delta, jejich výška pak funkční hodnota ve středu dělicího intervalu.

```
import math

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

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(0, N):
    integ += Delta * math.sin(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, 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
```

10.2 Inspirace Archimédovou aproximací π

Aproximujeme pí pomocí obvodů vepsaných pravidelných n-úhelníků:

$$o_n = 2rn \sin \frac{180^\circ}{n}$$

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

$$\pi_n = n \sin \frac{180^\circ}{n}$$

```
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 11

Rovnice

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

Předpokládá se zadání dvou čísel, mezi kterými má rovnice $f(x) = 0$ právě jeden kořen. Program ověří, zda mezi zadanými čísly kořen skutečně leží. Následně se provádí půlení intervalu, dokud není jeho délka menší, než zadaná přesnost. Jedná se tedy o typický příklad použití cyklu while.

```
# zadání intervalu, v nemz budeme hledat koren
a, b = 0, -1

presnost = 2 * 10**(-16)

import math

# definice funkce f
def f(x):
    return  x*x - 2**x       # x - math.cos(x)

# funkce, která zúzí interval
def zuzit_interval(v):
    a = v[0]
    b = v[1]
    c = (a + b) / 2
    if f(a) == 0:
        return [a]
    if f(b) == 0:
        return [b]
    if f(c) == 0:    # lze zvolit toleranci: abs(f(c)) < presnost
        return [c]
    if (f(a) < 0 and f(c) > 0) or (f(a) > 0 and f(c) < 0):
        return [a, c]
    if (f(b) < 0 and f(c) > 0) or (f(b) > 0 and f(c) < 0):
        return [c, b]
    else:
        return "koren nenalezen"
```

```
interval = [a, b]
pocet = 0

# hledání korene
while len(interval) == 2  and  abs(interval[1] - interval[0]) > presnost:
    interval = zuzit_interval(interval)
    pocet = pocet + 1

# pouzijeme výpis hodnot pod sebe - usnadnuje srovnání
if len(interval) == 2:
    for x in interval:
        print(x)
else:
    print(interval)

print("pocet pulení:", pocet)
```

Výstup programu:

```
-0.766664695962123
-0.7666646959621232
pocet pulení: 53
```

11.1.1 Stručná verze

Předchozí program můžeme napsat mnohem stručněji. Nebude sice tak explicitní zužování intervalu, přehlednost se však nesníží.

```
import math

def f(x):
    return x - math.cos(x)  # x * x - 2**x

a = 0
b = 1

presnost = 3e-16

if f(b) > f(a):
    a, b = b, a

pocet_puleni = 0
```

```
y = f(a)

while abs(y) > presnost:
    pocet_puleni = pocet_puleni + 1
    x = (a + b) / 2
    y = f(x)
    if y < 0:
        b = x
    elif y > 0:
        a = x

print("koren: ", x)
print("presnost: {},  pocet iterací: {}". format(presnost, pocet_puleni))
```

Výstup programu:

```
koren:  0.7390851332151607
presnost: 3e-16,  pocet iterací: 52
```

11.2 Hledání kořene postupným procházením od daného x

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

Uvažujme rovnici $2^x - x^2 = 0$. Ta má kromě kladných kořenů 2 a 4 také jeden kořen záporný ($-0.766664695962123\ldots$). Z grafu je patrné, že je tento kořen větší než $x = -1$.

```
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
```

Kapitola 12

Kalendář

12.1 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]

    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]
```

for i in range(3, 13): dnu_v_mesici[i] = (dnu_v_mesici[i] + 1) print(dnu_v_mesici)

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

test chování operátoru % pro záporná čísla přirozeně roluje, tj. chová se přesně tak, jak potřebujeme for i in range(-10, 10): print(i, i % 7)

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)

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

12.2 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
```

12.3 Kalendář na celý rok

Tentokrát nebudeme programovat sami, ale využijeme vestavěné funkce. Výsledkem je pěkný přehledný kalendář na celý rok.

```
import calendar

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

12.4 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(2020, 2041):
    print( velik(rok) )
```

Výstup programu:

```
[12, 4, 2020]
[4, 4, 2021]
[17, 4, 2022]
[9, 4, 2023]
[31, 3, 2024]
[20, 4, 2025]
[5, 4, 2026]
[28, 3, 2027]
[16, 4, 2028]
[1, 4, 2029]
[21, 4, 2030]
[13, 4, 2031]
[28, 3, 2032]
[17, 4, 2033]
[9, 4, 2034]
[25, 3, 2035]
[13, 4, 2036]
[5, 4, 2037]
[25, 4, 2038]
[10, 4, 2039]
[1, 4, 2040]
```

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 13

Náhodná čísla

13.1 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!")
```

13.1.1 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!")
```

13.2 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)
```

13.3 Karty

13.3.1 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
```

13.3.2 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
```

13.4 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 14

Želví grafika

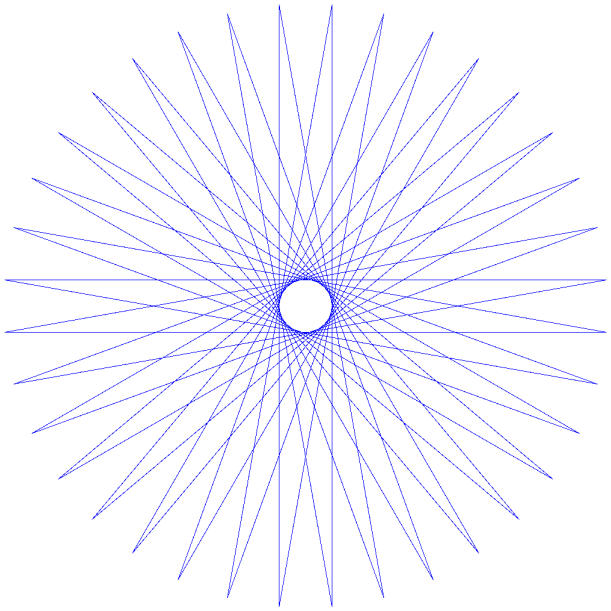
14.1 Hvězda

```
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()
```



14.2 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()
```