

Programy a algoritmy pracující s čísly

IB113

Radek Pelánek

2021

Rozcvička 1

0 1 1 2 1 2 2 3 1 2 2 3 2 3 3 4 1 2 2 3 2 3 3 4 2
3 3 4 3 4 ...

Rozcvička 2

$$1^2 + 2^2 + 3^2 + \cdots + 99^2 + 100^2$$

Ukázka nevhodného programu

```
def ctverec(n):  
    for i in range(1, n):  
        print("*" * i)  
        if i == n-1:  
            p = 1  
            while n+1 > p:  
                print("*" * n)  
                n = n - 1
```

- práce s čísly v Pythonu
- ukázky programů, ilustrace použití základních konstrukcí
- upozornění na záludnosti, ladění
- ukázky jednoduchých algoritmů, ilustrace rozdílu v efektivitě

- `int` – „integer“, celá čísla
- `float`
 - „floating-point number“
 - čísla s plovoucí desetinnou čárkou
 - reprezentace: mantisa \times báze^{exponent}
 - nepřesnosti, zaokrouhlování
- (`complex` – komplexní čísla)

Nepřesnosti

Přesná matematika:

$$\left(\left(1 + \frac{1}{x}\right) - 1\right) \cdot x = 1$$

Nepřesné počítače:

```
>>> x = 2**50
>>> ((1 + 1 / x) - 1) * x
1.0
>>> x = 2**100
>>> ((1 + 1 / x) - 1) * x
0.0
```

Nepřesné výpočty – praktický případ

záměr



chyba



```
step = 0.1
while value <= bound:
    # draw line ...
    value += step
```


Číselné typy – poznámky

- explicitní přetypování: `int(x)`, `float(x)`
- automatické „nafukování“ typu `int`:
 - viz např. `2**100`
 - pomalejší, ale korektní
 - rozdíl od většiny jiných prog. jazyků (běžné je „přetečení“)

v Python2.7 dělení: rozdíl `3/2` a `3/2.0`

v Python3 dělení intuitivní

Kvízové otázky

Co udělá program?

```
n = 1
while n > 0:
    print(n)
    n = n / 10
print("done")
```

Co když použijeme „ $n = n * 10$ “?

A co „ $n = n * 10.0$ “?

Co udělají analogické programy v jiných programovacích jazycích?

Pokročilejší operace s čísly

Některé operace v knihovně `math`:

- použití knihovny: `import math`
- zaokrouhlování: `round`, `math.ceil`, `math.floor`
- absolutní hodnota: `abs`
- `math.exp`, `math.log`, `math.sqrt`
- goniometrické funkce: `math.sin`, `math.cos`, ...
- konstanty: `math.pi`, `math.e`

Záludné detaily

```
>>> round(2.5)
```

```
2
```

```
>>> round(3.5)
```

```
4
```

```
>>> round(2.675, 2)
```

```
2.67
```

<https://docs.python.org/3/library/functions.html#round>

The behavior of `round()` for floats can be surprising: for example, `round(2.675, 2)` gives 2.67 instead of the expected 2.68. This is not a bug: it's a result of the fact that most decimal fractions can't be represented exactly as a float.

Ciferný součet

- vstup: číslo x
- výstup: ciferný součet čísla x
- příklady:
 - $8 \rightarrow 8$
 - $15 \rightarrow 6$
 - $297 \rightarrow 18$
 - $11211 \rightarrow 6$

Ciferný součet: základní princip

opakovaně provádíme:

- dělení 10 se zbytkem – hodnota poslední cifry
- celočíselné dělení – „okrajování“ čísla

Ciferný součet – nevhodná pasáž

```
if n % 10 == 0:
    f = 0 + f
elif n % 10 == 1:
    f = 1 + f
elif n % 10 == 2:
    f = 2 + f
elif n % 10 == 3:
    f = 3 + f
elif n % 10 == 4:
    f = 4 + f
...
```

Ciferný součet – řešení

```
def digit_sum(n):  
    result = 0  
    while n > 0:  
        result += n % 10  
        n = n // 10  
    return result
```


Ciferný součet: jednořádkové řešení

Čistě pro zajímavost:

```
digit_sum = lambda n: sum(map(int, str(n)))
```

(skrze řetězec vyrobíme seznam jednotlivých cifer, sečteme vestavěnou funkcí sum, funkci definujeme lambda abstrakcí)

Return vs. print

připomenutí:

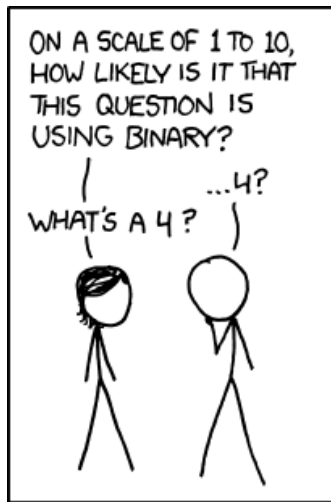
- `print` = výpis
- `return` = návratová hodnota, se kterou můžeme dále pracovat
 - blízký vztah k matematickým funkcím

příklad:

výpis všech čísel menších jak 1000 s ciferným součtem 13

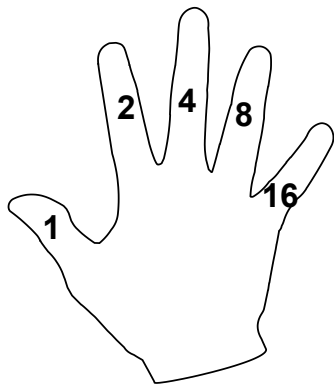
Volání funkce, parametry funkce

- definice funkce
 - obecný zápis, jak se funkce počítá
 - nezpůsobí žádný konkrétní výpočet
- volání funkce
 - výpočet funkce pro konkrétní zadané hodnoty parametrů
- hodnoty parametrů funkce
 - zadáváme při volání, nikoliv při definici
 - (až na defaultní parametry, ale ty můžete zatím ignorovat)



<https://xkcd.com/953/>

Binární soustava: počítání na prstech



<https://www.khanacademy.org/math/vi-hart/v/binary-hand-dance>

Příklad: převod na binární zápis

Problém: převodník z desítkové na binární soustavu

vstup: číslo v desítkové soustavě

výstup: číslo v binární soustavě

- Jak převedeme „22“ na binární zápis?
- Jak převedeme obecné číslo na binární zápis?

Převod na binární zápis

```
def to_binary(n):  
    output = ""  
    while n > 0:  
        if n % 2 == 0:  
            output = "0" + output  
        else:  
            output = "1" + output  
        n = n // 2  
    return output  
  
print(to_binary(22))
```

Převod na binární zápis – průběh výpočtu

n = 22	output =
n = 11	output = 0
n = 5	output = 10
n = 2	output = 110
n = 1	output = 0110
n = 0	output = 10110

Tajná posloupnost

0 1 1 2 1 2 2 3 1 2 2 3 2 3 3 4 1 2 2 3 2 3 3 4 2
3 3 4 3 4 ...

Tajná posloupnost

0 1 1 2 1 2 2 3 1 2 2 3 2 3 3 4 1 2 2 3 2 3 3 4 2
3 3 4 3 4 ...

Počet jedniček v binárním zápisu.

Jak vypsát programem?

Tajná posloupnost

```
def count_ones(n):  
    count = 0  
    while n > 0:  
        if n % 2 == 1:  
            count += 1  
        n = n // 2  
    return count  
  
def sequence(n):  
    for i in range(n):  
        print(count_ones(i), end=" ")
```

Collatzova posloupnost

- vezmi přirozené číslo:
 - pokud je sudé, vyděl jej dvěma
 - pokud je liché, vynásob jej třemi a přičti jedničku
- tento postup opakuj, dokud nedostaneš číslo jedna

Collatzova posloupnost: výpis

```
def collatz_sequence(n):  
    while n != 1:  
        print(n, end=" ", " ")  
        if n % 2 == 0:  
            n = n // 2  
        else:  
            n = 3*n + 1  
    print(1)
```


Bonus: Vykreslení grafu v Pythonu

Využívá seznamy a knihovnu pylab

```
import pylab

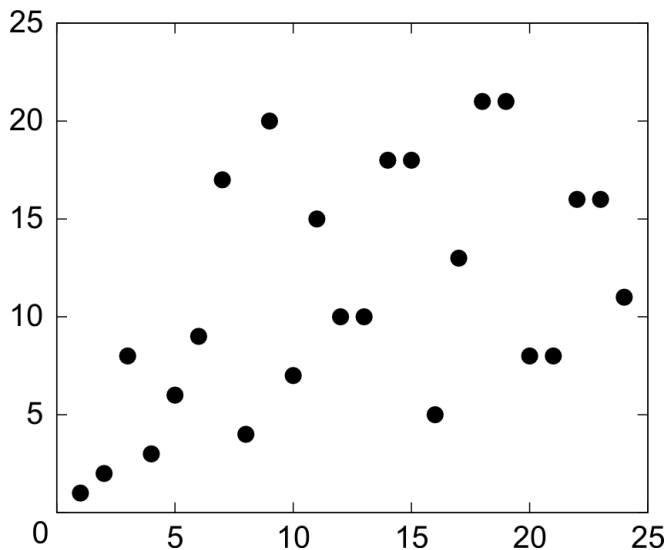
def collatz(n):
    sequence = []
    while n != 1:
        sequence.append(n)
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3*n + 1
    sequence.append(1)
    return sequence

pylab.plot(collatz(27))
pylab.show()
```

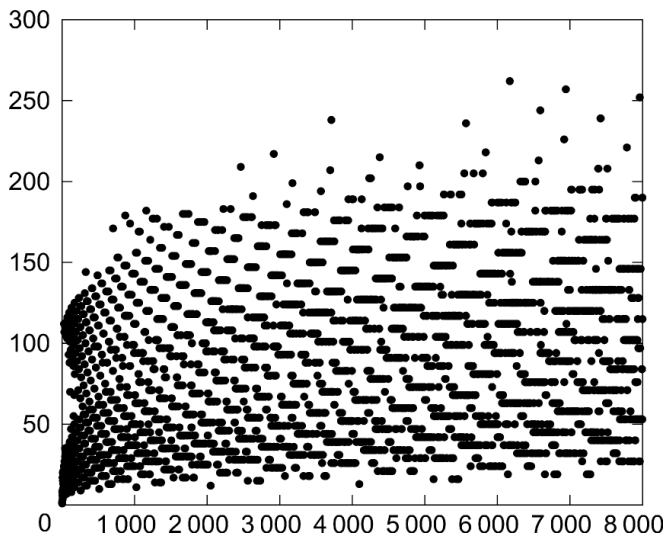

Collatzova posloupnost: délka posloupnosti

```
def collatz_length(n):  
    length = 1  
    while n != 1:  
        if n % 2 == 0:  
            n = n // 2  
        else:  
            n = 3*n + 1  
        length += 1  
    return length  
  
def collatz_table(count):  
    for i in range(1, count+1):  
        print(i, collatz_length(i))
```

Collatzova posloupnost: délka posloupnosti I



Collatzova posloupnost: délka posloupnosti II



Collatzova hypotéza

- Hypotéza: Pro každé počáteční číslo n , posloupnost narazí na číslo 1.
- experimentálně ověřeno pro velká n ($\sim 10^{18}$)
- důkaz není znám

Největší společný dělitel

- vstup: přirozená čísla a, b
- výstup: největší společný dělitel a, b
- příklad: 180, 504

Jak na to?

Naivní algoritmus I

- projít všechny čísla od 1 do $\min(a, b)$
- pro každé vyzkoušet, zda dělí a i b
- vzít největší

Naivní algoritmus II

- „školní“ algoritmus
- najít všechny dělitele čísel a, b
- projít dělitele, vybrat společné, vynásobit
- příklad:
 - $180 = 2^2 \cdot 3^2 \cdot 5$
 - $504 = 2^3 \cdot 3^2 \cdot 7$
 - $NSD = 2^2 \cdot 3^2 = 36$

Euklidův algoritmus: základ

základní myšlenka: pokud $a > b$, pak:

$$NSD(a, b) = NSD(a - b, b)$$

příklad:

<i>krok</i>	<i>a</i>	<i>b</i>
1	504	180
2	324	180
3	180	144
4	144	36
5	108	36
6	72	36
7	36	36
8	36	0

Neefektivita

- uvedená verze je neefektivní
- může být pomalejší než naivní algoritmus I
- kdy?

Euklidův algoritmus: vylepšení

vylepšená základní myšlenka: pokud $a > b$, pak:

$$NSD(a, b) = NSD(a \bmod b, b)$$

<i>krok</i>	<i>a</i>	<i>b</i>
1	504	180
2	180	144
3	144	36
4	36	0

Euklidův algoritmus: program

bez rekurze

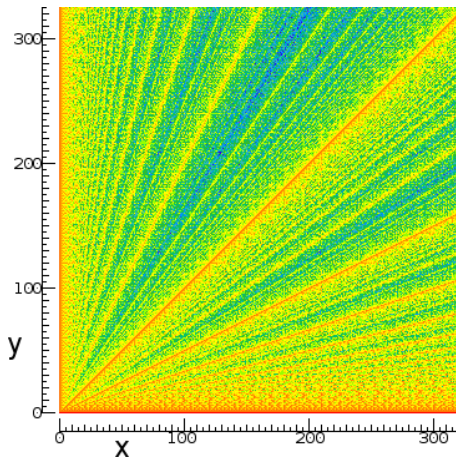
```
def gcd_euclid(a, b):  
    if a == 0:  
        return b  
    while a != 0 and b != 0:  
        if a > b:  
            a = a % b  
        else:  
            b = b % a  
    return max(a, b)
```

Euklidův algoritmus: program

modulo varianta, rekurzivně

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

Euklidův algoritmus – vizualizace



http://en.wikipedia.org/wiki/Euclidean_algorithm

Efektivita algoritmů

- proč byly první dva algoritmy označeny jako „naivní“?
- časová náročnost algoritmu:
 - naivní: exponenciální vůči počtu cifer
 - Euklidův: lineární vůči počtu cifer
- různé algoritmy se mohou **výrazně** lišit svou efektivností
- často rozdíl použitelné vs nepoužitelné
- více později (a v dalších předmětech)

Výpočet odmocniny

- vstup: číslo x
- výstup: přibližná hodnota \sqrt{x}

Jak na to?

Výpočet odmocniny

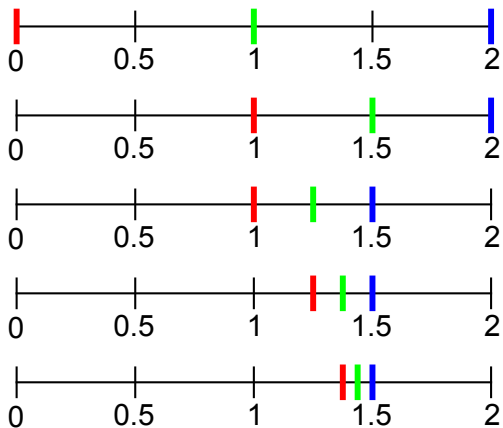
- vstup: číslo x
- výstup: přibližná hodnota \sqrt{x}

Jak na to?

Mnoho metod, ukázka jedné z nich (rozhodně ne nejvíce efektivní)

Výpočet odmocniny: binární půlení

spodní odhad střed horní odhad



Výpočet odmocniny: binární půlení

```
def square_root(x, precision=0.01):  
    upper = x  
    lower = 0  
    middle = (upper + lower) / 2  
    while abs(middle**2 - x) > precision:  
        print(lower, upper, sep="\t")  
        if middle**2 > x:  
            upper = middle  
        if middle**2 < x:  
            lower = middle  
        middle = (upper + lower) / 2  
    return middle
```

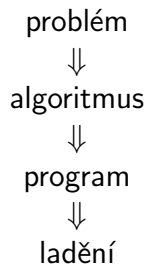
Výpočet odmocniny – chyba

Drobný problém: Program není korektní.

Kde je chyba?

- Funguje korektně jen pro čísla ≥ 1 .
- Co program udělá pro čísla < 1 ?
- Proč?
- Jak to opravit?

Vsuvka: Obecný kontext



Poznámka o ladění

- laděním se nebudeme (na přednáškách) příliš zabývat
- to ale neznamená, že není důležité...

Ladění je dvakrát tak náročné, jak psaní vlastního kódu. Takže pokud napíšete program tak chytře, jak jen umíte, nebudete schopni jej odladit. (Brian W. Kernighan)

Postřeh k ladění

Do průšvihů nás nikdy nedostane to, co nevíme. Dostane nás tam to, co víme příliš jistě a ono to tak prostě není. (Y. Berry)

- ladící výpisy
 - např. v každé iteraci cyklu vypisujeme stav proměnných
 - doporučeno vyzkoušet na ukázkových programech ze slidů
- použití debuggeru
 - dostupný přímo v IDLE
 - sledování hodnot proměnných, spuštěných příkazů, breakpointy, ...

- dobrá dekompozice na funkce usnadňuje ladění
- „hledání chyby v celém programu“ vs. „hledání chyby v dílčí funkci“
- „unit testing“, „test driven development“

Čtení chybových hlášek

```
Traceback (most recent call last):  
  File "sorting.py", line 63, in <module>  
    test_sorts()  
  File "sorting.py", line 59, in test_sorts  
    sort(a)  
  File "sorting.py", line 52, in insert_sort  
    a[j] = curent  
NameError: name 'curent' is not defined
```

- kde je problém? (identifikace funkce, číslo řádku)
- co je za problém (typ chyby)

Základní typy chyb

- **SyntaxError**
 - invalid syntax: zapomenutá dvojtečka či závorka, záměna = a ==, ...
 - EOL while scanning string literal: zapomenutá uvozovka
- **NameError** – špatné jméno proměnné (překlep v názvu, chybějící inicializace)
- **IndentationError** – špatné odsazení
- **TypeError** – nepovolená operace (sčítání čísla a řetězce, přiřazení do řetězce, ...)
- **IndexError** – chyba při indexování řetězce, seznamu a podobně („out of range“)

projeví se „rychle“ (program spadne hned):

- zapomenutá dvojtečka, závorka, uvozovka
- překlepy
- použití = tam, kde mělo být ==
- špatný počet argumentů při volání funkce
- zapomenuté len v “for i in range(alist)”

nemusí se projevit rychle / vždy:

- použití `==` tam, kde mělo být `=`
- `"True"` místo `True`
- záměna `print` a `return`
- dělení nulou
- chybné indexování (řetězce, seznamy)

Součet druhých mocnin

- Lze zapsat zadané číslo jako součet druhých mocnin?
- Příklad: $13 = 2^2 + 3^2$
- Která čísla lze zapsat jako součet druhých mocnin?

Součet druhých mocnin: řešení I

```
def sum_of_squares_test(n):  
    for i in range(n+1):  
        for j in range(n+1):  
            if i**2 + j**2 == n:  
                print(n, "=", i**2, "+", j**2)
```

- Program je zbytečně neefektivní. Proč?
- Vypis čísel, která lze zapsat jako součet čtverců

Testování druhé mocniny: nevhodný if

```
def is_square(n):  
    square_root = int(n**0.5)  
    if square_root**2 == n:  
        return True  
    else:  
        return False
```


Součet druhých mocnin: řešení II

```
def is_square(n):  
    square_root = int(n**0.5)  
    return square_root**2 == n  
  
def is_sum_of_squares(n):  
    for i in range(int(n**0.5) + 1):  
        rest = n - i**2  
        if is_square(rest):  
            return True  
    return False  
  
def print_sums_of_squares(count):  
    for i in range(count):  
        if is_sum_of_squares(i):  
            print(i, end=" ",)
```

Podobné náměty

- variace: součet tří druhých mocnin, součet dvou třetích mocnin, ...
- další náměty na posloupnosti: *The On-Line Encyclopedia of Integer Sequences*, <http://oeis.org/>

Kontrolní otázky

- Jaký je rozdíl mezi číselnými typy `int` a `float`?
- Jak zapisujeme operace „celočíselné dělení“ a „dělení se zbytkem“?
- Co dělají funkce `round`, `math.floor`, `math.ceil`?
- Jaká je základní myšlenka algoritmu pro výpočet ciferného součtu?
- Jaká je základní myšlenka algoritmu pro výpočet odmocniny?
- Jaká je základní myšlenka algoritmu pro převod na binární zápis?
- Jaký je význam příkazu `return`? Jaký je rozdíl mezi tím, když na konci funkce zavoláme `return` a `print`?

Doporučené procvičování

`https://www.umimeprogramovat.cz/porozumeni`

- Výpočty s čísly

`https:`

`//www.umimeprogramovat.cz/programovani-v-pythonu`

- Proměnné a číselné výrazy
- Logické výrazy

- operace s čísly, náhoda
- ukázky programů
- ukázky algoritmů, efektivita

Příště: náhodná čísla a simulace, řetězce a šifry