Oznam

Prvý test bude na cvičeniach v týždni 7, teda na cvičeniach:

- streda 30.10. 15:00, 17:00

- štvrtok 31.10. 10:00, 13:00, 15:00, 17:00

- náplňou testu bude všetko, čo preberieme na prednáškach a cvičeniach počas prvých 6 týždňov semestra

PROG1: Prednáška 6

While cykly

Cyklus

- Na prednáškach v druhom týždni sme si ukázali jeden typ cyklu (iterácií) v jazyku Python – **for**-cyklus, známy aj ako **cyklus s vopred známym počtom opakovaní.**

- Vo **for** cykle teda dopredu vieme povedať, koľkokrát sa vykoná telo cyklu.

Cyklus

- V programovaní sa často používa aj ďalší typ cyklu:
- **Cyklus s podmienkou na začiatku**, známy aj ako **while-**cyklus
- **While-**cyklus je typom iterácie, v ktorej sa telo cyklu vykonáva vtedy, **ak platí nejaká podmienka**, pričom táto podmienka sa vyhodnocuje **predtým**, než sa vykoná telo cyklu (preto „s podmienkou na začiatku“)

Cyklus

- Syntax while-cyklu v jazyku Python

while *podmienka* :
    príkaz
    príkaz
    …
    príkaz

Telo while-cyklu. Ide o príkazy, ktoré sa budú iterovane opakovať, **pokiaľ** bude *podmienka* pravdivá. Podmienka je teda nejaký booleovský výraz (má hodnotu True/False)

Cyklus

Priebeh **while**-cyklu v jazyku Python:

1) Určí sa, aká je pravdivostná hodnota *podmienky* (teda či je jej hodnota *True* alebo *False*)

2) Ak je *podmienka* **pravdivá** (True), tak sa **jedenkrát** vykoná telo cyklu, príkaz za príkazom a **znovu** sa otestuje pravdivostná hodnota podmienky.

3) Ak je *podmienka* **pravdivá**, tak sa **znovu** vykoná telo cyklu, príkaz za príkazom a **znovu** sa otestuje pravdivostná hodnota podmienky, atď.

4) Ak sa pri testovaní *podmienky* zistí, že **nie je pravdivá** (False), tak sa telo cyklu **preskočí** a pokračuje sa s príkazmi za telom while-cyklu.

Príklad z druhého týždňa

V druhom týždni sme definovali funkciu, ktorá vypíše čísla od 0 po *n-1*, kde *n* je parameter funkcie. Vľavo pôvodná verzia s for-cyklom, vpravo verzia s while-cyklom.

Pomocou for cyklu:

```
1 def vypis(n):
2     for i in range(n):
3         print(i)
```

Pomocou while cyklu:

```
1 def vypis(n):
2     i=0
3     while i!=n:
4         print(i)
5         i=i+1
```

Príklad z minulého týždňa

V minulom týždni sme pomocou rekurzie definovali funkciu, ktorá vypíše na obrazovku čísla od *n* po 1 a následne vypíše „Start!“

Pomocou rekurzie:

```
def odpocet(n):
    if n <= 0:
        print("Start!")
    else:
        print(n)
        odpocet(n-1)
```

Pomocou while cyklu:

```
def odpocet(n):
    while n > 0:
        print(n)
        n = n - 1
    print("Start!")
```

Cyklus

Cyklus **while** sa označuje ako **cyklus s podmienkou na začiatku**, pretože **predtým**, než sa začne vykonávať telo cyklu sa **najprv** skontroluje, či platí príslušná podmienka.

Ak podmienka neplatí, telo cyklu sa nevykoná a pokračuje sa s ďalšími príkazmi.

Ak podmienka platí, vykoná sa telo cyklu **jedenkrát** a znovu sa otestuje podmienka a postup sa zopakuje.

Do slovenčiny sa **while** prekladá aj ako **pokiaľ/kým**, teda **pokiaľ** je podmienka splnená, vykonávaj telo cyklu.

Dôležité upozornenie! Keďže cyklus sa opakovane vykonáva, pokiaľ je podmienka splnená, **malo by platiť**, že počas tela cyklu dôjde k zmene niektorých z hodnôt, na základe ktorých sa vyhodnocuje platnosť podmienky – v opačnom prípade existuje riziko, že sa program vo while-cykle **zacyklí**, teda telo cyklu by bežalo donekonečna.

Cyklus

Ak by sme v predošlom programe zabudli dekrementovať v každej iterácii hodnotu premennej *n,* cyklus by bežal donekonečna!

```
def odpocet(n):
    while n > 0:      # v cykle nedochadza k zmene premennej n
        print(n)      # ak bude parameter n > 0, program sa zacykli
    print("Start!")
```

Príklad – chcete byť slávny?

Je daná nasledovná funkcia – pre parameter *n* vypíše postupnosť čísiel a_i kde platí:
1) $a_0 = n$ (teda prvý člen postupnosti je *n*)
2.1) Ak a_i je párne číslo, potom $a_{i+1} = a_i / 2$
2.2) Ak a_i je nepárne číslo, potom $a_{i+1} = 3*a_i + 1$

V matematike existuje slávna, **stále nevyriešená domnienka (Collatzova domnienka)**, ktorá hovorí, že **každá takáto postupnosť** má tú vlastnosť, že postupne skonverguje do 1, t.j. že sa v nej postupne vyskytne číslo 1.

Napríklad ak *n* = 4, potom takáto postupnosť: 4, 2, **1, …**
Ak *n* = 3, potom takáto postupnosť: 3, 10, 5, 16, 8, 4, 2, **1**, ...
Ak *n* = *7*, potom takáto postupnosť: 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, **1, ...**

Príklad – chcete byť slávny?

```
def collatz(n):
    '''
    Funkcia vypise cisla v postupnosti podla Collatzovej hypotezy.
    Prvy clen postupnosti je parametrom funkcie.
    Funkcia ukonci vypis v momente, ked narazi na cislo 1.

    n: cele cislo, prvy clen postupnosti
    '''
    while n != 1:
        print(n)
        if n % 2 == 0:
            n = n//2
        else:
            n = 3*n+1
    print(n)
```

Off-topic, môžete preskočiť :)

V informatike je dokázaný slávny výsledok, že **neexistuje algoritmus**, ktorý by dokázal pre ľubovoľný program a jeho vstup rozhodnúť, či program skončí alebo nie.

Problém určenia, či daný počítačový program a jeho vstup v konečnom čase skončí alebo nie (neskončí = „zacyklí sa“ / beží do nekonečna) sa nazýva ako tzv. **problém zastavenia (halting problem)**.

To znamená, že **žiaľ**, ani počítače **nie sú všemocné** a napríklad ak by sme chceli napísať program v jazyku Python, ktorý by dokázal analyzovať iné programy tak, že by zobral:

a) nejaký iný program
b) vstup tohto iného programu

a chceli by sme, aby rozhodol, či daný iný program pre daný vstup skončí alebo nie, tak takýto program by sme **nevedeli napísať**, pretože proste **neexistuje**.

Hra-čkáčsky príklad

Definujme nasledovnú funkciu: *hadaj(maximum).* Funkcia vygeneruje celé číslo z rozsahu {1, …, *maximum}*. Následne vyzve používateľa, aby uhádol vygenerované číslo. Používateľ zadáva čísla z klávesnice, pokiaľ neuhádne vygenerované číslo, pričom po každom tipe mu program vypíše, či je tipované číslo menšie/väčšie, než to, ktoré zadal.

Keď používateľ uhádne vygenerované číslo, funkcia o tom vypíše hlášku a skončí.

Hra-čkáčsky príklad

Analýza:
1) Vidíme, že program bude opakovane vykonávať nejakú činnosť:
- používateľ zadá svoj tip
- program mu oznámi, či je hádané číslo menšie/väčšie než používateľov tip

2) Keďže používateľ bude hádať dovtedy, **pokiaľ netrafí**, na implementáciu použijeme **while**-cyklus, pretože potrebujeme **opakovane** vykonávať nejakú činnosť **pokiaľ** platí nejaká podmienka – tu bude podmienkou to, že používateľ **stále neuhádol**.

Činnosť programu je teda možné popísať ako:
3) **Pokiaľ** používateľ neuhádol:
    - **vypíš** či je hádané číslo menšie/väčšie ako jeho posledný tip
    - **načítaj nový tip používateľa**

4) Keďže podmienkou cyklu bude test, či používateľ neuhádol, **prvý tip** musí byť k dispozícii už pred **while**-cyklom! Preto načítanie prvého tipu vykonáme ešte pred while-cyklom.

Hra-čkáčsky príklad

```
import random  #pre nahodne generovanie pouzivame modul "random"

def hadaj(maximum):
    hadane_cislo = random.randint(1,maximum)   #funkcia random.randint(a,b)
                                               #vrati nahodne cele cislo z rozsahu {a, a+1, ..., b}
    tip = int(input("Zadaj svoj tip: "))       #nacitanie prveho tipu
    while (tip != hadane_cislo):               #tu testujeme, ci pouzivatel uhadol tipovane cislo
        if tip < hadane_cislo:                 #ak je tip mensi, vypiseme, ze hadane cislo je VACSIE
            print("Hadane cislo je VACSIE!")
        else:                                  #ak je tip vacsi, vypiseme, ze hadane cislo je MENSIE
            print("Hadane cislo je MENSIE!")
        tip = int(input("Zadaj svoj tip: "))   #nacitanie noveho tipu od pouzivatela
    print("UHADOL SI!")                        #ked cyklus skonci, pouzivatel uhadol
```

Predčasné ukončenie while-cyklu, **break**

While-cyklus tak, ako sme si ho predstavili teda **najprv** skontroluje *podmienku,* a ak je pravdivá, **jedenkrát** vykoná telo cyklu. Následne sa vráti na *podmienku* a znovu testuje jej platnosť a postup sa opakuje.

Teda vo vyššie uvedenom prípade by program ukončil vykonávanie **while** cyklu vtedy, ak by pri testovaní *podmienky* zistil, že nie je pravdivá.

Avšak veľa programovacích jazykov poskytuje možnosť **ukončiť** vykonávanie cyklu **predčasne**, vo vnútri tela cyklu pomocou špeciálneho príkazu **break**.

Vykonanie príkazu **break** má za následok, že sa vykonávané telo cyklu **ukončí,** príkazy v **tele cyklu za príkazom break sa nevykonajú** a program pokračuje s príkazmi, ktoré sú uvedené za while-cyklom.

Predčasné ukončenie while-cyklu, **break**

Upravme program zo slajdu č. 14 tak, aby používateľ nemal neobmedzený počet pokusov, ale pridajme do funkcie ďalší parameter *pocet_pokusov*, t.j. interfejs funkcie bude *hadaj(maximum, pocet_pokusov)*

V prípade, že používateľ neuhádne na daný *pocet_pokusov*, funkcia vypíše, že vyčerpal svoje pokusy a vráti *False*. V prípade že používateľ uhádol vygenerované číslo, vypíše o tom hlášku a vráti *True.*

Predčasné ukončenie while-cyklu, **break**

Analýza:
1) Podmienku cyklu ponecháme – cyklus sa vykonáva **pokiaľ** používateľ neuhádol.

2) **Ak** používateľ **vyčerpá** počet pokusov, potrebujeme cyklus **predčasne ukončiť.** Pre tieto účely použijeme príkaz **break** v prípade, že **vyčerpal počet pokusov**.

3) To, že používateľ vyčerpal počet pokusov zistíme tak, že si vytvoríme novú premennú *cislo_tipu*, ktorú nastavíme pri prvom tipe na 1 pri každom ďalšom tipe ju inkrementujeme. V momente, keď bude premenná *cislo_tipu* **rovnaká** ako *pocet_pokusov,* **pričom posledný tip bol stále nesprávny**, cyklus ukončíme (**break**) a zabezpečíme vrátenie hodnoty *False* a výpis príslušnej chybovej hlášky.

4) V programe sme vytvorili aj premennú *vycerpane_pokusy*, ktorá má hodnotu *True*, ak používateľ vyčerpal všetky pokusy a neuhádol. Inak má hodnotu *False.* Používame ju na to, aby sme po skončení cyklu vedeli, či používateľ uhádol hádané číslo (*vycerpane_pokusy == False)* alebo vyčerpal platné pokusy a neuhádol (*vycerpane_pokusy == True).*

Predčasné ukončenie while-cyklu, **break**

```
import random

def hadaj(maximum, pocet_pokusov):
    hadane_cislo = random.randint(1,maximum)

    cislo_tipu = 1 #pamatame si, kolko tipov pouzivatel urobil
    vycerpane_pokusy = False  #premenna, do ktorej ulozime True, ak vycerpa pocet pokusov
                              #na zaciatku nastavime na False, pretoze pocet pokusov nebol vycerpany
    tip = int(input("Zadaj svoj tip: "))
    while (tip != hadane_cislo):
        if tip < hadane_cislo:
            print("Hadane cislo je VACSIE!")
        else:
            print("Hadane cislo je MENSIE!")
        if cislo_tipu == pocet_pokusov:  #ak sa vycerpa pocet pokusov
            vycerpane_pokusy = True      #nastavime vycerpane_pokusy na True a
            break                        #predcasne ukoncime cyklus
        else:
            tip = int(input("Zadaj svoj tip: "))
            cislo_tipu += 1              #zaznacime si vykonanie dalsieho tipu
    if vycerpane_pokusy == True:
        print("Vycerpal si pocet pokusov :(")
        return False
    else:
        print("UHADOL SI! :)")
        return True
```

Nekonečný cyklus

Niekedy sa v programovaní využíva aj nasledovná konštrukcia, v ktorej sa ako podmienka **while** cyklu použije výraz, ktorý je **vždy pravdivý**, napr. priamo hodnota **True**

Samozrejme, v takom prípade **musí** niekde v tele cyklu dôjsť k príkazu **break**, inak by cyklus nikdy neskončil a program by bežal donekonečna.

while True:
    príkaz
    príkaz
    …
    príkaz

Niektorý z príkazov v tele cyklu musí obsahovať **break**

Príklad

Definujte funkciu *sucet_nenulovych()* bez parametrov, ktorá načítava čísla, kým používateľ nezadal nulu. Po zadaní nuly vráti súčet načítaných čísiel.

Porovnajte si ľavé a pravé riešenie: V ľavom riešení využívame konštrukciu **while True** a **break**, v pravom riešení je zase nutné prvé číslo načítať mimo cyklus, keďže premenná *cislo* sa používa priamo v podmienke:

```
def sucet_nenulovych():
    suma = 0
    while True:
        cislo = int(input())
        if cislo == 0:
            break
        suma += cislo
    return suma
```

```
def sucet_nenulovych():
    cislo = int(input())
    suma = cislo
    while cislo != 0:
        cislo = int(input())
        suma += cislo
    return suma
```

break

Príkaz **break** je možné používať aj vo **for**-cykle, s rovnakým účinkom ako vo **while**-cykle, teda že ukončí **ten cyklus**, v ktorom došlo k použitiu **break**.

POZOR!!!

Ak používate vnorené cykly, tak **break** ukončí vždy len ten cyklus, v rámci ktorého bol použitý, t.j. ten „najviac-vnorený“.

prednaska6.py - Z:\Documents\prednaska\prednaska6.py (3.12.6)

File Edit Format Run Options Window Help

```
for i in range(10):
    print("i = ",i,end=' ')
    for j in range(10):
        print("j = ",j, end=' ')
        if i == j:
            break
    print()
```

Príkaz break sa viaže len na ten cyklus, v ktorom bol použitý, t.j. na "najviac-vnorený"

```
i =  0 j =  0
i =  1 j =  0 j =  1
i =  2 j =  0 j =  1 j =  2
i =  3 j =  0 j =  1 j =  2 j =  3
i =  4 j =  0 j =  1 j =  2 j =  3 j =  4
i =  5 j =  0 j =  1 j =  2 j =  3 j =  4 j =  5
i =  6 j =  0 j =  1 j =  2 j =  3 j =  4 j =  5 j =  6
i =  7 j =  0 j =  1 j =  2 j =  3 j =  4 j =  5 j =  6 j =  7
i =  8 j =  0 j =  1 j =  2 j =  3 j =  4 j =  5 j =  6 j =  7 j =  8
i =  9 j =  0 j =  1 j =  2 j =  3 j =  4 j =  5 j =  6 j =  7 j =  8 j =  9
```

Aproximácie hodnôt

- Peknou ukážkou využitia cyklu **while** je naprogramovanie rôznych numerických algoritmov, ktoré počítajú matematické hodnoty pomocou aproximácie.
- Typickou ukážkou sú algoritmy, ktoré počítajú nejakú hodnotu tak, že postupne zlepšujú jej odhad, pričom konvergujú k požadovanej hodnote.

Aproximácie hodnôt

- Peknou ukážkou využitia cyklu **while** je naprogramovanie rôznych numerických algoritmov, ktoré počítajú matematické hodnoty pomocou aproximácie.
- Typickou ukážkou sú algoritmy, ktoré počítajú nejakú hodnotu tak, že postupne zlepšujú jej odhad, pričom konvergujú k požadovanej hodnote.

Odhad Eulerovho čísla

- Pre základ prirodzeného logaritmu, Eulerovo číslo *e*, platí, že k nemu konverguje súčet nasledovného radu:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots$$

Odhad Eulerovho čísla

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots$$

Čím viac členov sčítame, tým presnejší odhad Eulerovho čísla dostaneme:

Ak by sme sčítali len prvý člen: 1/0! = 1
Ak by sme sčítali 2 členy: 1/0! + 1/1! = 1+1 = 2
Ak by sme sčítali 3 členy: 1/0! + 1/1! + 1/2! = 1+1+1/2 = 2.5
Ak by sme sčítali 4 členy: 1/0! + 1/1! + 1/2! + 1/3! = 1+1+1/2+1/6 = 2.6666666
Ak by sme sčítali 5 členov: 1/0! + 1/1! + 1/2! + 1/3! + 1/4! = 1+1+1/2+1/6+1/24 = 2.708333

a tak ďalej.

Môžeme napríklad naprogramovať funkciu, ktorá bude odhadovať Eulerovo číslo, pričom počet sčítaných členov bude parameter funkcie, pozri ďalší slajd

Odhad Eulerovho čísla

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)

def hodnota_e_n(n):
    '''
    Funkcia aproximuje eulerovo cislo podla sumy

    n-1
    ---
    \\
    //   1/i!
    ---
    i = 0

    kde pocet scitancov n je parameter funkcie. '''
    e_vysledok = 0
    for i in range(n):
        e_vysledok += 1/factorial(i)
    return e_vysledok
```

Vytvorili sme si pomocnú funkciu na výpočet faktoriálu pomocou rekurzie.

Samotná funkcia sčíta potom *n* členov príslušnej postupnosti a vráti výsledok.

Odhad Eulerovho čísla

Čím viac sčítancov sčítame, tým je odhad Eulerovho čísla lepší!
Len pre istotu uvádzame, že hodnota Eulerovho čísla je približne:

$$e = 2,718281828459045235360287471352...$$

```
for i in range(1, 11):
    print('Pocet scitancov: ',i, ', e ~',hodnota_e_n(i))
```

```
Pocet scitancov:  1 , e ~ 1.0
Pocet scitancov:  2 , e ~ 2.0
Pocet scitancov:  3 , e ~ 2.5
Pocet scitancov:  4 , e ~ 2.6666666666666665
Pocet scitancov:  5 , e ~ 2.708333333333333
Pocet scitancov:  6 , e ~ 2.7166666666666663
Pocet scitancov:  7 , e ~ 2.7180555555555554
Pocet scitancov:  8 , e ~ 2.7182539682539684
Pocet scitancov:  9 , e ~ 2.718278769841270
Pocet scitancov:  10 , e ~ 2.7182815255731922
```

Odhad Eulerovho čísla

Keby sme teda chceli presný odhad Eulerovho čísla, môžeme alebo použiť uvedenú funkciu pre veľký počet sčítancov (napr. n = 100),

alebo funkciu upravíme tak, že budeme sledovať, aká je zmena výsledného súčtu.

Keďže súčet **konverguje**, čím viac sčítancov uvažujeme, tým menšia bude zmena súčtu po pripočítaní ďalšieho člena radu.

Spravíme teda verziu predošlej funkcie, kde budeme opakovať pripočítavanie ďalšieho člena radu, **pokiaľ** bude rozdiel 2 po sebe idúcich súčtov väčší ako nejaká nami definovaná hodnota – označíme ju *epsilon.*

Ak chceme sledovať, k akej zmene 2 po sebe idúcich hodnôt došlo, najjednoduchšie je vypočítať absolútnu hodnotu ich rozdielu.

Odhad Eulerovho čísla

```
def hodnota_e_epsilon(epsilon):
    '''
    Funkcia aproximuje eulerovo cislo podla sumy

    oo
    ---
    \\
    //   1/i!
    ---
    i = 0

    v ktorej je rozdiel 2 po sebe iducich suctov vacsi ako parameter funkcie epsilon.
    '''
    e_vysledok = 0
    i = 0
    while True:
        e_vysledok_novy = e_vysledok + 1/factorial(i)
        if abs(e_vysledok_novy-e_vysledok) < epsilon:
            break
        e_vysledok = e_vysledok_novy
        i += 1
    return e_vysledok
```

Odhad Eulerovho čísla

Ak teraz našu funkciu zavoláme s argumentom 0.001, tak vráti odhad Eulerovho čísla, kde sa 2 po sebe idúce odhady líšili o menej než 1 tisícinu – vidíme, že takýto odhad bol 2.71805555, čo zodpovedá súčtu 7 členov radu.

Ak chceme presnosť na miliónтiny, zavoláme funkciu s argumentom 0.000001, dostávame hodnotu 2.7182815… čo zodpovedá súčtu 10 členov radu.

```
presnost = 0.001 #presnost na tisiciny
print(hodnota_e_epsilon(presnost))
presnost = 0.000001 #presnost na miliontiny
print(hodnota_e_epsilon(presnost))
```

```
2.7180555555555554
2.7182815255731922
```

Odhad odmocniny

- Podobný postup si ukážeme na nasledovnej úlohe
- Tentokrát budeme odhadovať **druhú odmocninu**
- V matematike je známa tzv. Netwonova iteračná metóda ako spôsob hľadania riešení rovníc. Jednou z aplikácii metódy je algoritmus výpočtu odhadu druhej odmocniny.

Odhad odmocniny

- Myšlienka je nasledovná: Chceme vypočítať (odhadnúť) odmocninu z čísla *a*:
- Predpokladáme, že máme k dispozícii nejaký odhad tejto odmocniny, označme ho *x*
- Presnejší odhad odmocniny, označme ho *y* dostaneme ako:

$$y = \frac{x + a/x}{2}$$

Odhad odmocniny

- Nech napríklad chceme vypočítať odmocninu z 45, $a = 45$
- Nech náš odhad odmocniny je $x = 7$
- Podľa vzťahu: $y = \frac{x + a/x}{2}$ bude lepší odhad odmocniny

  $y = (7 + 45/7)/2 = 6.7143...$
- Ak by sme teraz **znovu** zopakovali toto „zlepšenie odhadu“, tak pre odhad $x = 6.7143$ by sme dostali:

  $y = (6.7143 + 45/6.7143)/2 = 6.7082...$
- Mimochodom, podľa kalkulačky je sqrt(45) = 6.7082...

Odhad odmocniny

Pre uvedený postup znovu platí, že odhad sa postupne spresňuje a po sebe idúce hodnoty odhadu sa blížia k správnej hodnote.

Znovu teda naprogramujeme výpočet tak, že budeme sledovať rozdiel po sebe idúcich hodnôt odhadu a keď klesne pod nejakú hranicu *epsilon*, výpočet zastavíme, pretože odhad bude spĺňať nejakú nami požadovanú presnosť.

Pre jednoduchosť nastavíme hodnotu prvého odhadu odmocniny ako polovicu hodnoty, z ktorej odmocninu hľadáme.

Odhad odmocniny

prednaska6.py - Z:\Documents\prednaska\prednaska6.py (3.12.6)

File Edit Format Run Options Window Help

```
def odhad_odmocniny(a, epsilon):
    x = a / 2
    while True:
        y = (x + a/x)/2
        if abs(y-x) < epsilon:
            break
        x = y
    return y

print(odhad_odmocniny(45,0.001))
print(odhad_odmocniny(45,0.000001))
```

```
6.708203970631391
6.708203932499369
```