IB113 Úvod do programování a algoritmizace
Přednáška 3

Programy a algoritmy pracující s čísly
Ladění programu

Nikola Beneš

2. říjen 2017

Co bude dnes?

Práce s čísly v Pythonu

- ilustrace základních konstrukcí
- ukázky jednoduchých algoritmů, zamyšlení nad efektivitou
- úvod do generování náhodných čísel

Ladění programu

- jak hledat chyby
- použití debuggeru ve vývojovém prostředí PyCharm

Připomenutí: číselné typy

- celá čísla: typ `int`
  - v Pythonu neomezený rozsah
  - celočíselné dělení `//`
- „reálná“ čísla: typ `float`
  - ve skutečnosti tzv. čísla s plovoucí desetinnou čárkou (floating-point numbers)
  - reprezentace: báze, exponent
  - nepřesnosti, zaokrouhlování
- (pro zajímavost) komplexní čísla: typ `complex`
  - reálná a imaginární složka typu `float`
  - zápis `1j` je imaginární jednotka i
  - `3 + 5j` je komplexní číslo $3 + 5i$

Nepřesnost datového typu `float`

Přesná matematika:

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

Nepřesné počítače:

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

Pro které nejmenší číslo x *platí* `1 / x + 1 == 1`*?*

Knihovna pro matematické funkce

Knihovna `math`

- použití knihovny: `import math`
  - typicky na začátku programu
- zaokrouhlování: `round`, `math.ceil`, `math.floor`
- absolutní hodnota: `abs`
- logaritmus `math.log`, e^x `math.exp`
- odmocnina `math.sqrt`
- goniometrické funkce: `math.sin`, `math.cos`, ...
- konstanty: `math.pi`, `math.e`

- import jen některé funkce z knihovny math:
  `from math import sqrt`
  - v programu pak možno použít `sqrt` místo `math.sqrt`

Příklad: Ciferný součet

- *vstup:* přirozené číslo n
- *výstup:* ciferný součet čísla n
- příklady konkrétního vstupu a výstupu:
  - $0 \to 0$
  - $7 \to 7$
  - $17 \to 8$
  - $42 \to 6$
  - $999 \to 27$
  - $72525 \to 21$
- jak na to?
  - potřebujeme umět „odebrat jednu číslici z čísla“

Příklad: Ciferný součet

Myšlenka řešení

- poslední číslice z čísla: zbytek po dělení deseti
- odebrání poslední číslice: celočíselné dělení deseti
- opakovat tak dlouho, dokud číslo není 0

```
def digit_sum(n):
    result = 0
    while n > 0:
        # co bude tady?
        n = n // 10
    return result
```

Příklad: Ciferný součet

Nevhodné řešení

```
if n % 10 == 1:
    result = result + 1
elif n % 10 == 2:
    result = result + 2
elif n % 10 == 3:
    result = result + 3
elif n % 10 == 4:
    result = result + 4
# atd.
```

Příklad: Ciferný součet

Vhodné řešení

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

DRY vs. WET programování

DRY

- „Don't Repeat Yourself“

WET

- „Write Everything Twice“
- „We Enjoy Typing“
- „Waste Everyone's Time“

Největší společný dělitel

- *vstup:* přirozená čísla a, b
- *výstup:* největší číslo takové, že dělí a i b
- příklad:
  - 504, 540 $\rightarrow$ 36

K čemu chceme počítat největšího společného dělitele dvou přirozených čísel?

Největší společný dělitel

Naivní algoritmus

- projdeme všechna čísla od 1 do menšího z a, b
- pro každé vyzkoušíme, zda dělí a i b
  - jak to zjistíme?
- vezmeme největší z dělitelů

```
def gcd_naive(a, b):
    best = 0
    for i in range(1, min(a, b) + 1):
        if a % i == 0 and b % i == 0:
            best = i
    return best
```

- nějaký nápad na vylepšení?
  - procházet kandidáty od největšího
  - funkci můžu ukončit, jakmile najdu společného dělitele

Největší společný dělitel

„Školní“ algoritmus

- rozložit a, b na součin prvočísel
- vybrat, co je společné, vynásobit

- příklad:
  - $504 = 2^3 \cdot 3^2 \cdot 7$
  - $540 = 2^2 \cdot 3^3 \cdot 7$
  - $nsd(504, 540) = 2^2 \cdot 3^2 = 36$

Největší společný dělitel

Euklidův algoritmus

- základní myšlenka: pokud $a > b$, pak $nsd(a, b) = nsd(a - b, b)$
- příklad:
  1. 165, 120
  2. 120, 45
  3. 75, 45
  4. 45, 30
  5. 30, 15
  6. 15, 15
  7. 15, 0
- příklad s 540 a 504: 16 kroků

Největší společný dělitel

```
def gcd(a, b):
    if a == 0:
        return b
    while b != 0:
        if a > b:
            a -= b
        else:
            b -= a
    return a
```

- proč stačí v cyklu `while` podmínka `b != 0`? proč netestujeme i `a`?

Největší společný dělitel

Euklidův algoritmus

- vylepšená myšlenka: pokud $a > b$, pak $nsd(a, b) = nsd(a \bmod b, b)$
- příklad:
  1. 165, 120
  2. 120, 45
  3. 45, 30
  4. 30, 15
  5. 15, 0
- příklad:
  1. 540, 504
  2. 504, 36
  3. 36, 0

Největší společný dělitel

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

- proč i zde stačí testovat v podmínce cyklu jen `b`?

Poznámka: s použitím ntic (*tuples*) by se tělo cyklu dalo napsat takto:
`a, b = b, a % b`
(o *tuples* se dozvíme více později)

Největší společný dělitel

- rekurzivní varianta (pro zajímavost)

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

    return gcd(b, a % b)
```

Efektivita algoritmů

- časová náročnost algoritmů pro nsd:
  - naivní, „školní“: exponenciální vůči počtu cifer
  - Euklidův: lineární vůči počtu cifer
- různé algoritmy mohou řešit tentýž problém různě rychle
  - často rozdíl použitelné vs. nepoužitelné

Výpočet odmocniny

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

- co to znamená *přibližná*?
- jak na to?
  - mnoho různých metod, ukážeme si jednu z nich (zdaleka ne tu nejefektivnější)

Výpočet odmocniny

Půlení intervalu

- chceme vypočítat $\sqrt{2}$
- řešení musí být v intervalu $\langle 0, 2 \rangle$
- zvolíme střed intervalu, tj. 1
- $1^2 = 1$, což je méně než 2
- řešení tedy musí být v intervalu $\langle 1, 2 \rangle$
- zvolíme střed intervalu, tj. 1,5
- $1.5^2 = 2.25$, což je více než 2
- řešení tedy musí být v intervalu $\langle 1, 1.5 \rangle$
- atd.

- kdy skončíme?

Výpočet odmocniny

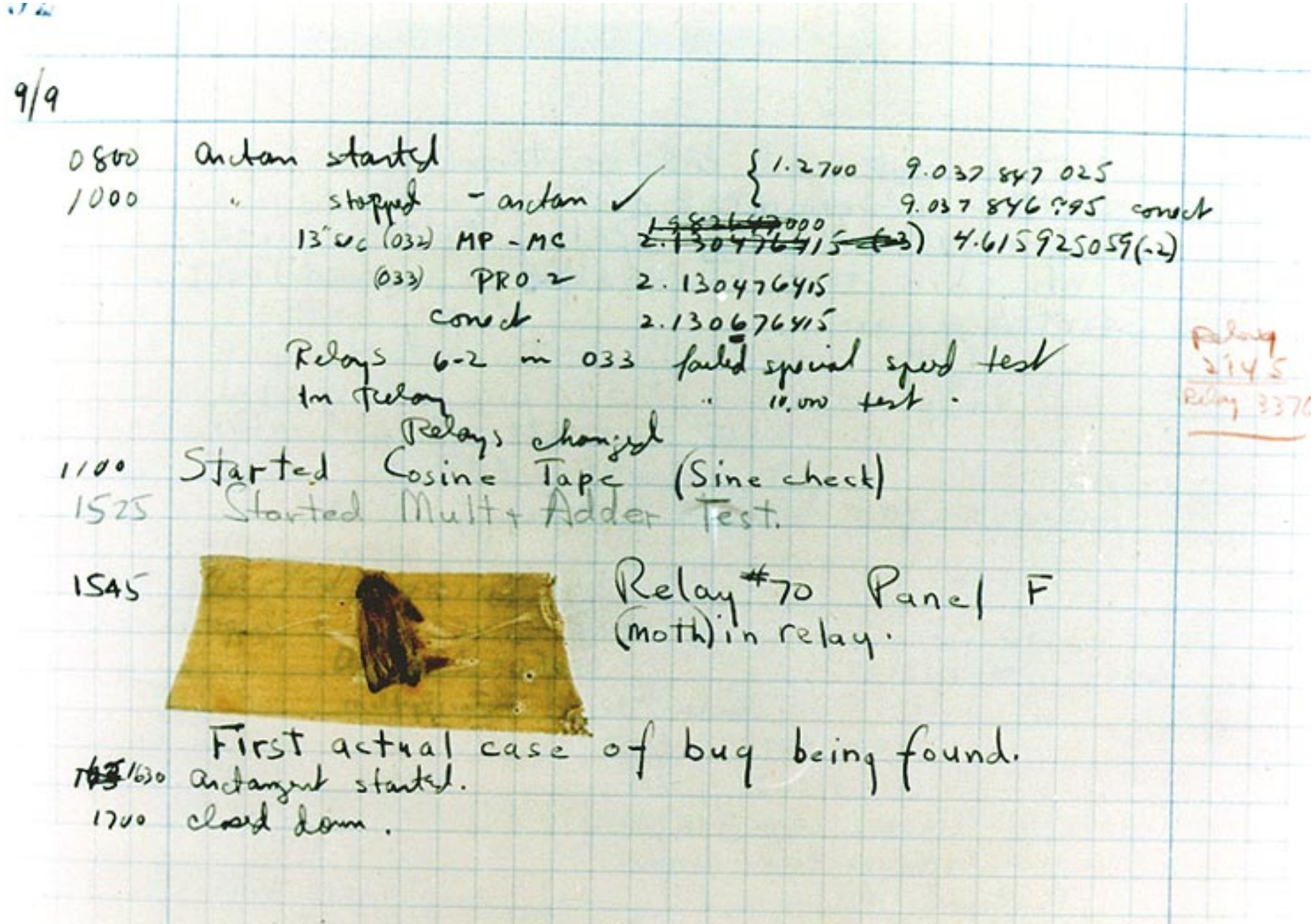
```
def square_root(x, precision=0.01):
    lower = 0
    upper = x
    middle = (lower + upper) / 2
    while abs(middle**2 - x) > precision:
        if middle**2 > x:
            upper = middle
        elif middle**2 < x:
            lower = middle
        middle = (lower + upper) / 2
    return middle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

Výpočet odmocniny

- předchozí program má drobný problém: není korektní
- kde je chyba?

- korektně funguje jen pro čísla ≥ 1
- co se stane, když bude x na vstupu < 1?
- proč?
- jak to opravit?

Ladění (debugging)



Ladění (debugging)

„Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.“

– Brian Kernighan

Ladění (debugging)

Ladicí výpisy

- pomocné výpisy, které nám říkají, v jakém stavu se program nachází
- např. v každé iteraci cyklu vypisujeme obsah proměnných

Debugger

- nástroj, pomocí něhož je možné postupně provádět jednotlivé kroky programu, sledovat hodnoty proměnných apod.
- typicky součástí vývojového prostředí
- ukázka: použití debuggeru v PyCharmu
- více na cvičeních

Ladění (debugging)

Debugger v PyCharmu

- nastavení řádků, kde se má výpočet zastavit: breakpoints
  - stačí kliknout vpravo vedle čísla řádku
- spuštění programu s debuggerem
  - místo Run použít Debug
- hodnoty proměnných v okně Variables
  - přidání nového výrazu, tlačítka + a −
- krokování
  - Step Over – krok na další řádek
  - Step Into – vstoupí dovnitř volání funkce
  - Step Out – vynoří se z volání funkce
  - Resume Program – běží až po další breakpoint

Náhodná čísla

- (ve skutečnosti *pseudo-náhodná* čísla)
- využití v programování: výpočty, simulace, hry, ...

Náhodná čísla v Pythonu – knihovna `random`

- `import random`
- `random.random()` – float mezi 0 a 1
- `random.randint(a, b)` – celé číslo mezi `a` a `b` (včetně)
- mnoho dalších funkcí

Závěr

Co jsme se dnes dozvěděli?

- existence knihovny matematických funkcí
- příklady jednoduchých algoritmů s čísly
- algoritmy mohou mít různou (časovou i jinou) složitost
- jak hledat chyby v programech
- jak použít debugger v PyCharmu

Co bude příště?

- náhodná čísla podrobněji, simulace
- datový typ seznam
- práce s řetězci