Vývoj programů, styl, praktické tipy

IB111 Základy programování
Radek Pelánek

2018

Dnešní témata

- dokumentace
- testování
- dělení projektu do souborů
- volání programu z příkazové řádky
- styl, PEP8
- případová studie – simulace sázení

vsuvka: regulární výrazy

Návrh a vývoj softwaru

„softwarové inženýrství“

vývoj rozsáhlého softwaru nejen o zvládnutí „programování“:

- specifikace, ujasnění požadavků
- návrh
- dokumentace
- testování
- integrace, údržba

mnoho metodik celého procesu: vodopád, spirála, agilní přístupy, ...

Dokumentace

pro koho:

- pro sebe (při vývoji i později)
- pro ostatní

jak:

- názvy (modulů, funkcí, proměnných)
- dokumentace funkcí, tříd, rozhraní
- komentáře v kódu

Dokumentace: obecné postřehy

- neaktuální dokumentace je často horší než žádná dokumentace
- sebe-dokumentující se kód
  Nejlepší kód je takový, který se dokumentuje sám.
  $\Rightarrow$ názvy funkcí, parametrů, dodržování konvencí, ...
- u rozsáhlých projektů dokumentace nezbytnost
- psaní *dobré* dokumentace – trochu umění, nezbytnost empatie

Dokumentace v Pythonu

- dokumentační řetězec (*docstring*)
- první řetězec funkce (třídy, metody, modulu)
- konvenčně zapisován pomocí „trojitých uvozovek“ (povolují víceřádkové řetězce)

```
def add(a, b):
    """Add two numbers and return the result."""
    return a + b
```

Dokumentační řetězec víceřádkový

```
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)
    """
    if imag == 0.0 and real == 0.0:
        return complex_zero
    ...
```

Dokumentační řetězce: ukázky

```
"""
This string, being the first statement in the file,
will become the module's docstring when the file is
imported.
"""

class MyClass(object):
    """The class's docstring"""

    def my_method(self):
        """The method's docstring"""

def my_function():
    """The function's docstring"""
```

Dokumentace vs. komentáře

- dokumentační řetězec
  - **co** kód dělá, jak se používá (volá), ...
  - brán v potaz při zpracování, dá se s ním dále pracovat: `__doc__` atribut, nástroje pro automatické zpracování, ...
- komentáře (#)
  - **jak** kód funguje, jak se udržuje, ...
  - při zpracování ignorovány

Testování: obecný postřeh

Přístupy k testování:

- nevhodný: „přesvědčit se, že program je v pořádku“
- vhodný: „najít chyby v kódu“

obecný kontext: „konfirmační zkreslení“ (confirmation bias)

Testování

rozsáhlé téma:

- testování vs formální verifikace
- různé úrovně testování: unit, integration, component, system, ...
- různé styly testování: black box, white box, ...
- různé typy testování: regression, functional, usability, ...
- metodiky (test-driven development), automatizované nástroje

unit testing (jednotkové testování) – testování samostatných „jednotek“ (např. funkce, metoda, třída)

Testování: dílčí tipy

- cíleně testujte okrajové podmínky, netypické příklady
  - prázdný seznam (řetězec)
  - záporná čísla, desetinná čísla
  - pravoúhlý trojúhelník, rovnoběžné přímky
  - přechodný rok
- „pokrytí kódu testem“ (code coverage)
  - Jsou pomocí testu „vyzkoušeny“ všechny části kódu (funkce, příkazy, podmínky)?

Assert

```
assert expression
```

- pokud `expression` není splněn, program „spadne“ (přesněji: je vyvolána výjimka, kterou lze odchytit)
- pomůcka pro ladění, testování
- explicitní kontrola implicitních předpokladů

Assert: příklad

```
def factorial(n):
    assert n == int(n) and n >= 0
    f = 1
    for i in range(1, n+1):
        f = f * i
    return f

print(factorial(-2))
```

```
Traceback (most recent call last):
  File "/home/xpelanek/temp/test.py", line 8, in <module>
    print(factorial(-2))
  File "/home/xpelanek/temp/test.py", line 2, in factorial
    assert n == int(n) and n >= 0
AssertionError
```

Refaktorování

- složitější kód nikdy nenapíšeme ideálně na poprvé
- refaktorování (code refactoring) – úprava kódu bez změny funkčnosti
- dobře napsané testy usnadňují refaktorování

Dělení projektu do souborů

- univerzální princip: velký projekt nechceme mít v jednom souboru
- důvody: jako dělení programu na funkce, o úroveň abstrakce výš

Dělení projektu do souborů

- jazykově specifické
- programovací jazyky se liší technickými požadavky i konvencemi
  - hlavičkové soubory v C
  - „hodně malých souborů“ v Javě
- terminologie: knihovny, moduly, balíčky, frameworky, ...

Terminologie v kontextu Pythonu

pojmy s přesným významem:

- **modul** (module): soubor s příponou .py, funkce/třídy s příbuznou funkcionalitou
- **balík** (package): kolekce příbuzných modulů, společná inicializace, ...

související pojmy používané volněji:

- **knihovna** (library)
- **framework** (framework)

Moduly v Pythonu

- modul poskytuje rozšiřující funkcionalitu
- zdroje modulů:
  - standardní distribuce (např. math, turtle)
  - separátní instalace (např. numpy, Image)
  - vlastní implementace (základ: „.py soubor ve stejném adresáři“)
- použití:
  - `import module` – následná volání `module.function()`
  - `import module as m`
  - `from module import function`
  - `from module import *` (nedoporučeno)

Jmenné prostory

jmenný prostor (*namespace*) $\sim$ mapování jmen na objekty

- jmenné prostory umožňují použití stejného jména v různých kontextech bez toho, aby to způsobilo problémy
- jmenné prostory mají funkce, moduly, třídy...
- moduly – tečková notace (podobně jako objekty)
  - `random.randint`
  - `math.log`

Proč nepoužívat „import *“

```
from X import *
```

V malém programu nemusí vadit, ale ve větších projektech považováno za velmi špatnou praxi.

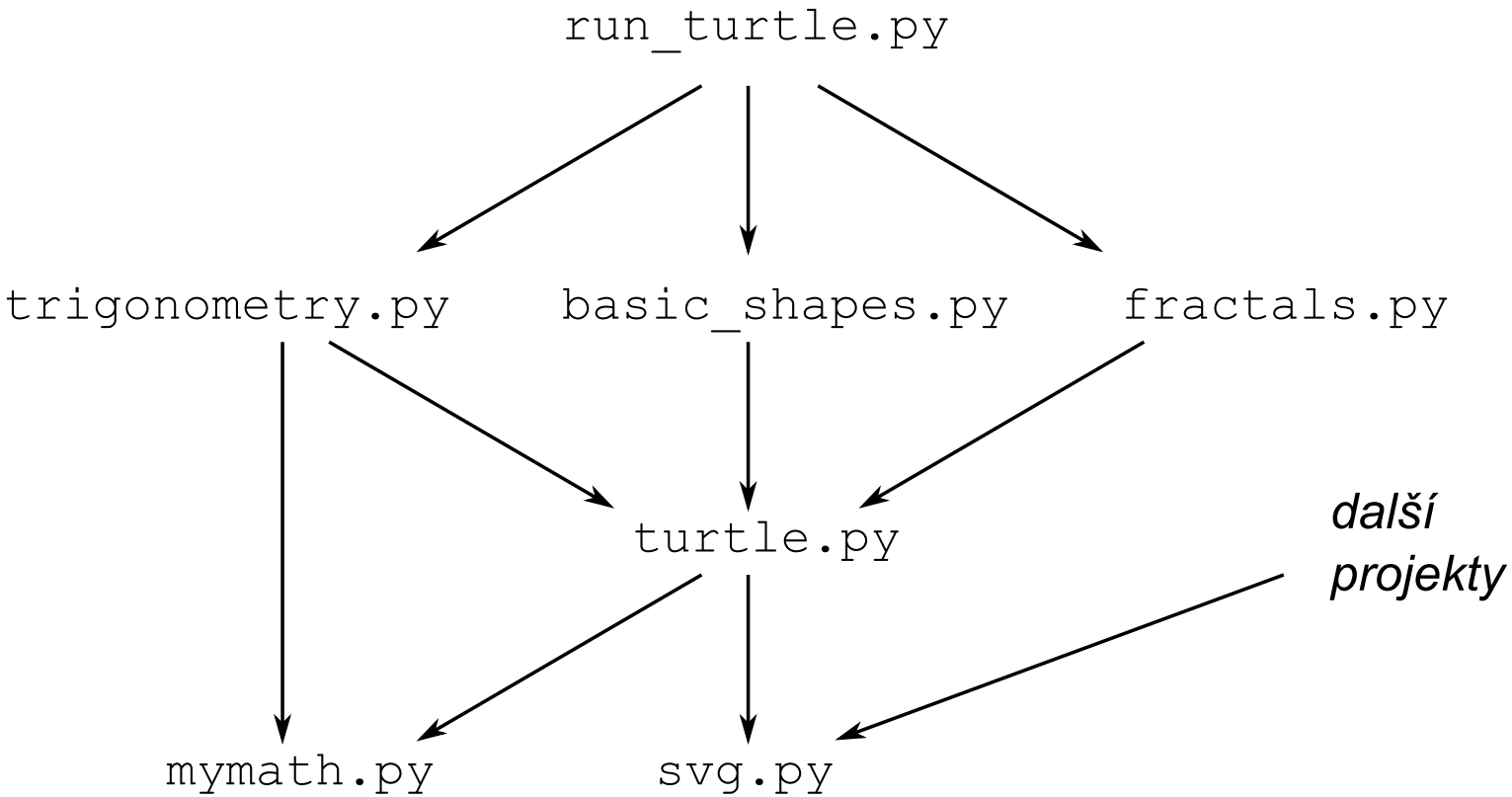
- „nepořádek“ v jmenném prostoru
- kolize, přepis
- překlepy se mohou chovat magicky
- složitější interpretace chybových hlášek
- Python Zen: Explicit is better than implicit. (import this)

Moduly – praktický příklad

želví grafika, vykreslování obrázků do SVG (pro následnou manipulaci)

- `mymath.py` – trigonometrické funkce počítající ve stupních
- `svg.py` – generování SVG kódu, funkce typu:
  - `svg_header()`, `svg_line()`, `svg_circle()`
  - manipulace s celkovým obrázkem (posun, rámeček), uložení do souboru
- `turtle.py` – třída reprezentující želvu, podpora pro více želv

Moduly – praktický příklad



Regulární výrazy

nástroj pro hledání „vzorů“ v textu

- programování
- textové editory
- příkazová řádka: např. `grep`
- teorie: formální jazyky, konečné automaty

Regulární výrazy

- obecně používaný nástroj
- syntax velmi podobná ve většině jazyků, prostředí
- bohatá syntax
- následuje „ochutnávka“, ukázky základního využití v Pythonu

http://xkcd.com/208/

Znaky a speciální znaky

- základní znak „vyhoví“ právě sám sobě
- speciální znaky: `.` `^` `$` `*` `+` `?` `{` `}` `[` `]` `\` `|` `(` `)`
  - umožňují konstrukci složitějších výrazů
  - chceme, aby odpovídaly příslušnému symbolu $\Rightarrow$ prefix \

Skupiny znaků

`[abc]` – jeden ze znaků `a, b, c`
`[^abc]` cokoliv jiného než `a, b, c`

`\d`	Čísla: `[0-9]`
`\D`	Cokoliv kromě čísel: `[^0-9]`
`\s`	Bílé znaky: `[ \t\n\r\f\v]`
`\S`	Cokoliv kromě bílých znaků: `[^ \t\n\r\f\v]`
`\w`	Alfanumerické znaky: `[a-zA-Z0-9_]`
`\W`	Nealfanumerické znaky: `[^a-zA-Z0-9_]`

Speciální symboly

.	libovolný znak
^	začátek řetězce
$	konec řetězce
|	alternativa – výběr jedné ze dvou možností

Opakování

`*`	nula a více opakování
`+`	jedno a více opakování
`?`	nula nebo jeden výskyt
`{m,n}`	m až n opakování

Pozn. `*`, `+` jsou „hladové“, pro co nejmenší počet opakování `*?`, `+?`

Příklady

Jaký je význam následujících výrazů?

- `\d[A-Z]\d \d\d\d\d`
- `\d{3}\s?\d{3}\s?\d{3}`
- `[a-z]+@[a-z]+\.cz`
- `^To:\s*(fi|kit)(-int)?@fi\.muni\.cz`

Regulární výrazy v Pythonu

- knihovna `re` (`import re`)
- `re.match` – hledá shodu na začátku řetězce
- `re.search` – hledá shodu kdekoliv v řetězci
- (`re.compile` – pro větší efektivitu)
- `re.sub` – nahrazení
- „raw string“ – r'vyraz' – nedochází k interpretaci speciálních znaků jako u běžných řetězců v Pythonu

Regulární výrazy v Pythonu: práce s výsledkem

- match/search vrací „MatchObject“ pomocí kterého můžeme s výsledkem pracovat
- pomocí kulatých závorek () označíme, co nás zajímá

Regulární výrazy v Pythonu: práce s výsledkem

```
>>> m = re.match(r'(\w+) (\w+)', \
                 'Isaac Newton, fyzik')
>>> m.group(0)
'Isaac Newton'
>>> m.group(1)
'Isaac'
>>> m.group(2)
'Newton'
```

Regulární výrazy v Pythonu: nahrazení

```
import re

text = 'Petr Novak, 329714, FI B-AP BcAP'
print(re.sub(r'(\w*) (\w*), (\d*)',
             r'\2 \1, UCO=\3',
             text))

# 'Novak Petr, UCO=329714, FI B-AP BcAP'
```

Regulární výrazy: xkcd

http://xkcd.com/1313/
http://www.explainxkcd.com/wiki/index.php/1313:_Regex_Golf
https://regex.alf.nu/

Kahoot

Kahoot: program 1

```
import re

def filter_list(regexp, alist):
    output = []
    for word in alist:
        if re.search(regexp, word):
            output.append(word)
    return output

fruits = ['apple', 'banana', 'pear', 'blackberry',
          'fig', 'peach']

print(filter_list(r'a.*e', fruits))
```

Kahoot: program 2

```
import re

def filter_list(regexp, alist):
    output = []
    for word in alist:
        if re.search(regexp, word):
            output.append(word)
    return output

nums = ['1.2', '3', '5.832', '428', '1.55']

print(len(filter_list(r'\d.\d', nums)))
```

Kahoot: program 3

```
import re

def filter_list(regexp, alist):
    output = []
    for word in alist:
        if re.search(regexp, word):
            output.append(word)
    return output

names = ['Kremilek a Vochomurka', 'Maxipes Fik',
         'Rakosnicek', 'Mala carodejnice',
         'Tom a Jerry']
print(len(filter_list(r'^\w*\s\w*$', names)))
```

Kahoot: program 4

```
import re

def decide(regex, string):
    if re.search(regex, string):
        return 'Y'
    return 'N'

for regex in [r'A.I', r'A.*I', r'A.*I$']:
    print(decide(regex, 'PARDUBICE'), end="")
```

Kahoot: program 5

```
import re

text = 'FI MU, Botanicka 68a, 602 00 Brno'

m = re.match(r'.*, (\w*).*,[\s\d]*(\w*)$', text)

print(m.group(1)+ ' ' + m.group(2))
```

Kahoot: program 6

```
import re

text = 'Petr /vykopal#sloveso/ diru.'

text = re.sub(r'/(.*)#(.*)/',
              r'<span cat="\2">\1</span>',
              text)

print(text)
```

Kahoot: program 7

```
text = 'one two  three'

text.split('e')

print(len(text))
```

Kahoot: program 8

```
def process(alist):
    output = []
    for x in alist:
        if isinstance(x, str):
            output.append(x)
    return output

alist = [3, '2.4', '1', 8.2, '2']

print("+".join(process(alist)))
```

Kahoot: program 9

```
def magic(alist):
    for i in range(len(alist)):
        alist[i] = alist[i].count('a')

fruits = ['apple', 'banana', 'fig']
magic(fruits)
print(fruits)
```

Volání programu z příkazové řádky

- specifické pro programovací jazyky, příp. i operační systémy
- Python, Linuxové systémy:
  - `#!/usr/bin/python3` na prvním řádku
  - `chmod u+x filename.py`

Volání programu z příkazové řádky

pro rozlišení „importování“ a „volání“:

- magická proměnná `__name__`
- funkce `main` (čistě konvence)

```
def main():
    # your code

if __name__ == "__main__":
    main()
```

Volání programu z příkazové řádky

- argumenty z příkazové řádky: `sys.argv`
- `./myprogram.py test 4`
- knihovna `argparse` – sofistikovanější zpracování

```
import sys
print("Name of the program:", sys.argv[0])
print("Number of arguments:", len(sys.argv))
if len(sys.argv) > 1:
    print("The first argument:", sys.argv[1])

# Name of the program: myprogram.py
# Number of arguments: 3
# The first argument: test
```

Styl psaní programů

Při programování jde nejen o korektnost a efektivitu, ale i čitelnost a čistotu kódu:

- snadnost vývoje, testování
- údržba kódu
- spolupráce s ostatními (sám se sebou po půl roce)

Styl psaní programů

obecná doporučení:

- nepoužívat copy&paste kód
- dekompozice na funkce, „funkce dělá jednu věc“
- rozumná jména proměnných, funkcí

specifická doporučení (závisí na programovacím jazyce, příp. společnosti) – důležitá hlavně konzistence:

- odsazování
- bílá místa (mezery v rámci řádku)
- styl psaní víceslovných názvů

PEP8 – doporučení pro Python

`https://www.python.org/dev/peps/pep-0008/`

- výběr vybraných bodů
- obecný „duch“ doporučení celkem univerzální
- částečně však specifické pro Python (pojmenování, bílá místa, ...)

PEP8: Styl a konzistence

konzistence (rostoucí důležitost)

- s doporučeními
- v rámci projektu
- v rámci modulu či funkce

PEP8: Odsazování a délka řádků

- standardní odsazení: 4 mezery
- nepoužívat tabulátor
- maximální délka řádku 79 znaků
- rady k zalomení dlouhých řádků

PEP8: Prázdné řádky

- oddělení funkcí a tříd: 2 prázdné řádky
- oddělení metod: 1 prázdný řádek
- uvnitř funkce: 1 prázdný řádek pro oddělení logických celků (výjimečně)

PEP8: Bílé znaky ve výrazech

- mezera za čárkou
- mezera kolem přiřazení a binárních operátorů
  - zachování čitelnosti celkového výrazu
  - ne okolo = v definici defaultní hodnoty argumentu
- nepoužívat přebytečné mezery uvnitř závorek

PEP8: Bílé znaky – příklady

```
Ano: spam(ham[1], {eggs: 2})
Ne:  spam( ham[ 1 ], { eggs: 2 } )

Ano: if x == 4: print x, y; x, y = y, x
Ne:  if x == 4 : print x , y ; x , y = y , x

Yes: spam(1)
No:  spam (1)

Yes: dct['key'] = lst[index]
No:  dct ['key'] = lst [index]
```

PEP8: Bílé znaky – příklady

Ano:

```
x = 1
y = 2
long_variable = 3
```

Ne:

```
x             = 1
y             = 2
long_variable = 3
```

PEP8: Bílé znaky – příklady

Ano:

```
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

Ne:

```
i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

PEP8: Pojmenování – přehled stylů

```
lowercase
lower_case_with_underscores (snake_case)
UPPERCASE
UPPER_CASE_WITH_UNDERSCORES
CapitalizedWords (CapWords, CamelCase, StudlyCaps)
mixedCase
Capitalized_Words_With_Underscores
_single_leading_underscore
single_trailing_underscore_
__double_leading_underscore
__double_leading_and_trailing_underscore__
```

PEP8: Pojmenování – základní doporučení

- proměnné, funkce, moduly: `lowercase`, příp. `lower_case_with_underscores`
- konstanty: `UPPERCASE`
- třídy: `CapitalizedWords`

Pojmenování: další rady

- jednopísmenné proměnné – jen lokální pomocné proměnné, nejlépe s konvenčním významem:
  - `n` – počet prvků (např. délka seznamu)
  - `i`, `j` – index v cyklu
  - `x`, `y` – souřadnice
- nikdy nepoužívat: `l`, `O`, `I` (snadná záměna s jinými znaky)
- angličtina, ASCII kompatibilita

PEP8: Komentáře

Komentář, který protiřečí kódu, je horší než žádný komentář.

- „inline“ komentáře používat výjimečně, nekomentovat zřejmé věci
  - Ne: `x = x + 1 # Increment x`
- doporučení ke stylu psaní komentářů (# následované jednou mezerou)

PEP8: Import

- vždy na začátku souboru
- doporučené pořadí: standardní moduly, third-party, lokální
- absolutní raději než relativní

Komentované ukázky kódů

- řešení z programátorských cvičení z `umimeprogramovat.cz`
- automatické hodnocení, hodnotí pouze korektnost
- ilustrované stylistické problémy ale časté i jinde

```
forward(100)
back(100)
right(18)
forward(100)
back(100)
right(18)
forward(100)
back(100)
right(18)
forward(100)
back(100)
right(18)
forward(100)
back(100)
right(18)
forward(100)
back(100)
right(18)
forward(100)
back(100)
right(18)
forward(100)
```

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

```
def digit_sum(n):
    counter=0
    string=str(n)
    length=len(string)
    for i in range(length):
        number=string[i]
        real_number=int(number)
        counter=counter+real_number

    return counter
```

```python
def palindrom(text):
    return text[::-1] == text

def palindrom(text):
    delka = len(text)
    pravaCast = text[::-1]
    levaCast = text[::]
    if (levaCast == pravaCast):
        return True
    else:
        return False
```

```
def palindrom(text):
    length = len(text)
    for i in range(length):
        if text[i] != text[length-i-1]:
            return False
    return True

def palindrom(text):
    pole = list(text)
    pole_rev = pole[::-1]
    sedi = True
    for i in range(len(pole)):
        if(pole[i] != pole_rev[i]):
            sedi = False
    if(sedi == False):
        return False
    else:
        return True
```

```
def factorize(n):
    oldn = n
    moj = []
    x = 2
    while(x < 1000):
        if n % x == 0:
            moj.append(x)
            n /= x
            x = 2
        else:
            x += 1
    print(oldn, "= ", end="")
    for i in range(len(moj)):
        if i == 0:
            print(moj[i], end=" ")
        else:
            print("*",moj[i], end = " ")
    print()
```

```
def bigN(n):
    n1=n
    n2=1
    n3=1
    for x in range(n):
        print("|",end="")
        for z in range(n-n1):
            print("",end=" ")
        for e in range(n2):
            print("\\",end="")
        for a in range(n-n3):
            print("",end=" ")
        print("|",end="")
        n1-=1
        n3+=1
        print()
```

Vývoj programu: případová studie

- jednoduchý, ale ne triviální problém – simulace sázení
- ilustrace různých přístupů k programování
- návrh, postupný vývoj, prototypování, testování

Simulace sázení: problém, motivace

Základní sázky 1:1 (panna/orel, férová mince)

Strategie Martingale:

- základní sázka 1
- po výhře dát základní sázku
- po prohře zdvojnásobit sázku

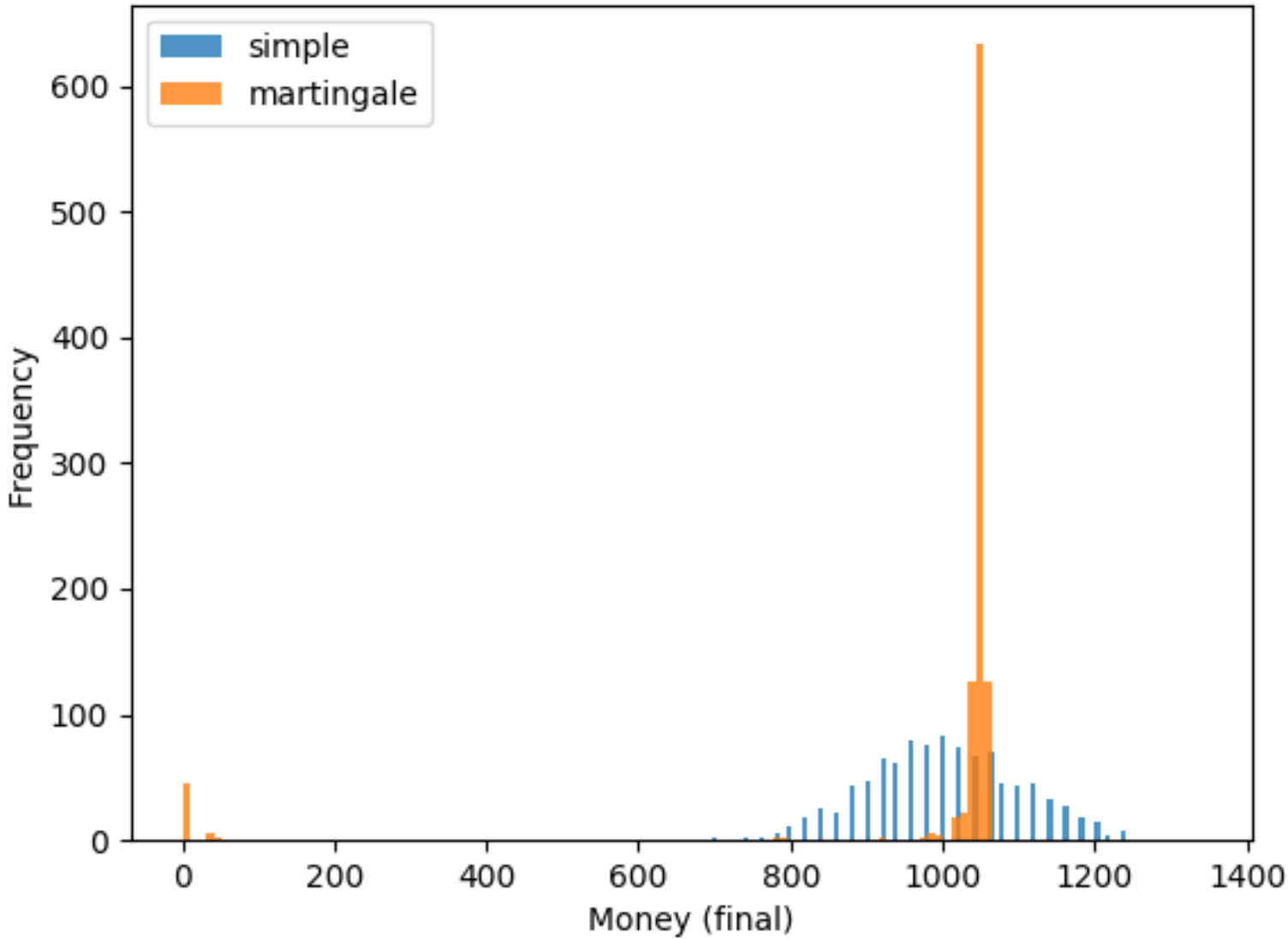
Martingale

Reklama na Martingale

Martingale je výborná strategie, vedoucí k zaručenému zisku!

Tedy pokud jste nekonečně bohatí. . .

Jak to dopadne reálně?


simple
martingale
600
500
400
300
200
100
0
Frequency
0
200
400
600
800
1000
1200
1400
Money (final)

```
import random
import pylab as plt
fm = []
for j in range(100):
    b = 1000
    s = 1
    for i in range(100):
        if random.randint(0, 1) == 1:
            b += s
            s = min(1, b)
        else:
            b -= s
            s = min(2*s, b)
    fm.append(b)
plt.hist(fm)
plt.show()
```

Simulace sázení: škaredé řešení

ukázka řešení s typickými chybami:

- nevhodné názvy proměnných
- copy&paste kód (při porovnání s jinou strategií)
- konstanty s nejasným významem
- absence funkcí

proč špatné:

- nečitelné
- nevhodné k testování
- nejde snadno experimentovat
- nejde snadno zobecnit

Simulace sázení: základní návrh programu

- `class Strategy` – reprezentace strategie sázení
  - atributy `budget`, `history`
  - metoda `make_bet`
- `play` – odehrání jedné série sázek
- `get_final_budget_stats` – opakovaná simulace jedné strategie
- `print_stats`, `compare_strategies` – zobrazení výsledků

Shrnutí

Vývoj kvalitního softwaru je nejen o dobrém zvládnutí samotného programování.

$\Downarrow$

navazující předměty

příště: praktické tipy, přehled programovacích jazyků, opakování, „zkouškový Kahoot“