

# Výjimky.

Výjimky. Hierarchie výjimek. Propagace výjimky. Chráněné bloky.

Tomáš Bayer | bayertom@natur.cuni.cz

Katedra aplikované geoinformatiky a kartografie, Přírodovědecká fakulta UK.

# Obsah přednášky

- 1 Výjimky
- 2 Propagace výjimky
- 3 Metoda chráněných bloků
- 4 Kombinace obou metod

# 1. Typy chyb v SW

V SW 3 základní typy chyb:

## A) Syntax Errors:

Porušení syntaktických pravidel.

Např. překlep, opomenutí závorek, :, nevhodné použití příkazu...

Zabrání komplikaci (a spuštění) programu.

Zobrazeny v průběhu kódování (Syntax Checking) nebo při překladu.

## B) Runtime Errors:

Chyby, ke kterým dochází za běhu programu (nepovolená operace) ⇒ výjimky (Exceptions).

Projevem pád nebo zatuhnutí programu.

Příklady: chyby vstupu, výstupu, chyby aritmetických operací (dělení nulou).

Zpravidla nejdou identifikovat v průběhu kódování, ošetření, použití debuggeru.

## C) Logic Errors:

Chyba v modelu, postupu řešení, matematickém aparátu.

Konceptuální chyby, ale i přehlédnutí.

SW zdánlivě pracuje, ale dává jiné výsledky, než čekáme.

Obtížně identifikovatelné v průběhu kódování ⇒ Unit Testing.

## 2. Ošetření Runtime Errors

Neošetřený Runtime Error ⇒ Exception:

pád aplikace ⇒ ztráta dat ⇒ finanční ztráta ⇒ nespokojený uživatel.

Nežádoucí stav, nutno mu předcházet nebo být připraven.

Varianty reakce na "chybu":

① *Ošetření všech singularit*

Snaha ošetřit všechny možné chybové stavy.

Lze provést u jednodušších problémů.

Prakticky nemožné v komplexních situacích ⇒ nepřehledný kód.

② *Ukončení běhu programu*

Nepředvídatelnost vzhledem ke vstupu, ztráta dat.

③ *Použití Exceptions (Výjimek)*

Odchycení a ošetření skupin chyb jednotně.

Často se kombinují metody 1), 3).

Výjimky: řeší pouze B).

Mechanismus práce s výjimkami podobný pro většinu jazyků.



### 3. Syntax vs. Runtime vs. Logic Error

#### Syntax Error:

```
>>> print("Hello world")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'print' is not defined
```

#### Runtime Error:

Odmocnina ze záporného čísla

```
>>> sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
```

Dělení nulou

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

#### Logic Error:

Chyba v zápisu:

```
>> def mean (a, b, c)
>>     return a + b + c / 3;           # (a + b + c) / 3
```

Chybný matematický aparát:

```
>> c = (a**2 + b**2)**0.5
```

# c = (a\*\*2 + b\*\*2)\*\*0.5

## 4. Exception (Výjimka)

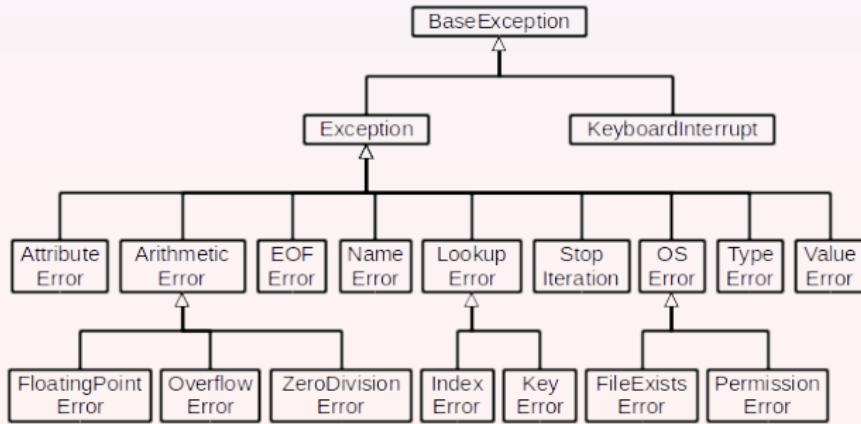
Stav programu v okamžiku, kdy dojde při jeho vykonávání k chybě.

- Vytvořen objekt, který nese informaci o typu chyby.
- Upozornění na výjimku chybovým hlášením.

Hierarchické uspořádání, všechny výjimky potomkem třídy `BaseExceptions`.

Specifické výjimky pro různé typy chyb.

Nejčastější použití: aritmetické chyby, překročení indexu, práce se soubory, konverze.



## 5. Metody ošetření výjimek

Výjimky lze ošetřit 3 způsoby:

**1 Propagace výjimky:**

Výjimka není ošetřena v metodě, ve které vznikla.

Je předána do nadřazené úrovně.

**2 Chráněné bloky:**

Výjimka je ošetřena v metodě, ve které vznikla.

Na vyšších úrovních neřešeno.

**3 Kombinace 1) a 2):**

Výjimka (částečně) ošetřena v místě, kde vznikla.

Dále předána do nadřazené úrovně.

Metoda 3 používána nejčastěji.

Umožňuje komplexní obsloužení výjimky.

Náročnější na implementaci.

## 6. Příčina vs. následek

U komplikovanějších problémů 2 strategie:

- řešení příčiny,
- řešení následku.

### Řešení příčiny:

Testování hodnot vstupních parametrů.

Cokoliv mimo  $D_f$ , vyhození výjimky.

Často poměrně komplikované, zejména u složitých problémů.

Obtížně lze detektovat všechny singularities před výpočtem.

### Řešení následku:

Primárně nezjišťujeme, co chybové stavu způsobilo.

Nekorektní funkcionality (např. hodnota mimo  $H_f$ )  $\Rightarrow$  vyhození výjimky.

Často jediné řešení u složitých problémů.

Varianta příčiny preferována, ne vždy možná (komplikované problémy).

Nutnost definice chybových stavů v dokumentaci.

## 7. Propagace výjimky

Výjimka není ošetřena v místě, kde vznikla.

Její zpracování předáváme do nadřazené úrovně.

Hierarchie předávání:

```
metoda() -> main() -> Python
```

### Typické použití:

Cyklus, nechceme opakovaně chybu řešit v těle.

Při prvním výskytu výjimky ji předáme výše ⇒ dořešení ⇒ zastavení výpočtu.

**Pozor:** Není -li dořešena v `main()`, ukončení programu !

Propagace (vyhození, throw) výjimky příkazem `raise`

```
raise exception(args)      #Výjimka s argumenty  
raise exception            #Konkretní výjimka  
raise                      #Genericky typ
```

Nepoužívat generický typ, ale co nejkonkrétnější informace o příčině.

Sdělit co se stalo a kdo je viníkem stavu (zdroj chyb).

```
raise ValueError("What happened", variable)
```

## 8. Ukázka propagace výjimky

Výpočet poloměru kružnice při známém obvodu.

Propagace z lokální funkce do main()

```
def getRadius(per):
    if perimeter > 0:                                #Kladne per, OK
        return per / (2 * pi)
    else
        raise ValueError("Perimeter < 0: ", per)   #Co se stalo
```

Volání:

```
>>> print(getR(1))
0.15915494309189535
```

Pokud v main() neošetřeno, pád programu:

```
>>> print(getR(-1)) #Neosetrena vyjimka
File "...exceptions.py", line 20, in <module>
    main()
File "...exceptions.py", line 15, in main
    print(getR(-1))
```

# 9. Ošetření metodou chráněných bloků

Ošetření výjimky v místě, kde vznikla.

Ve většině programovacích jazyků konstrukce try-catch, v Pythonu try-except.

Chráněný blok try() obsahuje "nebezpečný" kód.

```
try:                                #Potencialne nebezpecny kod
    pass
except Ex_1:                         #Daje-li k vyjimce typu 1, osetri
    pass
except (Ex_2, Ex_3):                 #Daje-li k 2 vyjimkam, osetri
    pass
except:                             #Daje-li k libovolne jine vyjimce, osetri
    pass
else:                               #Pokud try bylo uspesne, pokracuj zde
    pass
finally:                            #At bylo ci nebylo uspesne, delej toto
    pass
```

Blok except(), ošetření konkrétního typu/typů nebo libovolné výjimky.

Blok else(), volitelný, provede se, pokud try() úspěšný.

Blok finally(), volitelný, provede se vždy.

Informace o výjimce lze uložit do proměnné (objekt)

```
except ZeroDivisionError as var:    #Uloz do promenne var
    print(var)                      #Vytiskni
```

# 10. Ukázka funkcionality try-except

Normální běh, zachycená, nezachycená výjimka.

Normal Flow	Handled Exception	Unhandled Exception
<pre>try:     ...     # process  except exception:     # handle  finally:     ...     # cleanup     ...     # continue here</pre>	<pre>try:     ...     # process  except exception:     # handle  finally:     ...     # cleanup     ...     # continue here</pre>	<pre>try:     ...     # process  except exception:     # handle  finally:     ...     # cleanup     ...     # go up call stack</pre>

Nezachycená výjimka může vést k pádu programu.

# 11. Řešení příčiny, příklad

Výpočet hodnoty

$$c = \sqrt{\frac{a}{b}}, \quad a \geq 0 \wedge b > 0 \vee a \leq 0 \wedge b < 0.$$

Využití podmíněného příkazu, vracíme kód chybového stavu:

```
def f1 (a, b):
    if b == 0:           #Jmenovatel nulovy
        return -1
    if a / b < 0:        #Mimo definicni obor
        return -2

    return (a/b)**0.5
```

Volání funkce:

>> a = 1; b = 2; f1(a,b)	#Vrati 1/2
>> b = 0; f1(a,b)	#Vyjimka, vrati -1
>> b = -1; f1(a,b)	#Vyjimka, vrati -2
>> b = [1]; f2(a,b)	#Neosetreno, TypeError

Často není možné predikovat všechny chybové stavky.

Chybový kód může ležet v  $H_f$ , nerozlišíme, zda je výsledek OK.

Nutnost definice chybových stavů v dokumentaci.

## 12. Využití try-except

Řešení následku

Využití try-except + chybového kódu.

```
def f2 (a, b):
    try:
        return sqrt(a/b)
    except ZeroDivisionError:
        return -1          #Chybovy kod, deleni nulou
    except ValueError:
        return -2          #Chybovy kod, mimo definicni obor
    except BaseException:
        return -3          #Nespecifikovana chyba
    else:
        print("OK results")
```

Volání funkce:

>> a = 1; b = 2; f2(a,b)	#Vrati 1/2
>> b = 0; f2(a,b)	#Vyjimka, vrati -1
>> b = -1; f2(a,b)	#Vyjimka, vrati -2
>> b = [1]; f2(a,b)	#Vyjimka, vrati -3

Nutnost definice chybových stavů v dokumentaci.

Stejná nevýhoda jako v předchozím případě.

# 13. Zásady práce s výjimkami

Přehled doporučení při práci s výjimkami:

- Nepoužívat prázdné bloky `except ()`

Nutná nějaká reakce na chybu (žádná reakce, vše je OK, ale není).

- Odchycení obecné chyby

Pokud nevíme, co ji způsobuje, použít co nejobecnější typ.

Společný předchůdce, třída `BaseException`.

- Odchycení více výjimek

Řazení dle dědičné hierarchie, od konkrétního k obecnému.

```
except Predek:                      #Konkretni  
...  
except Nejaky_rodic:                 #Obecnějsi  
...  
except Nejaky_rodic_rodice:          #Jste obecnějsi  
...
```

Čím obecnější, tím níže v `except()` bloku!

- Odchycení informací o výjimce do proměnné

Uchování informace o stavu a zdroji chyb)

```
except ValueError as e:
```

- Vytištění informace o výjimce

Při odlaďování programu, pomoc při identifikaci zdroje chyb

```
except ValueError as e:  
    print(e)
```

## 14. Kombinace chráněných bloků a propagace

Kombinace předchozích postupů.

Umožňuje výjimku ošetřit v místě, kde vznikla a ji předat výše.

Princip metody:

- ① Definice chráněných bloků.
- ② Propagace výjimky uvnitř chráněného bloku.
- ③ Ošetření výjimky na vyšší úrovni.

Na výjimku lze reagovat opakovaně na různých úrovních programu.

V praxi nejpoužívanější metoda, zvláště u komplikovaných SW.

Lze použít agregaci výjimek.

### Agregace výjimek:

Seskupení bloků výjimek odvozených tříd do 1 bloku třídy nadřazené.

Umožňuje jednotnou reakci na tyto výjimky.

### Použití:

Stejná reakce na špatná vstupní data.

Časté použití u "matematických" chyb, input/output chyb, atd.

# 15. Propagace výjimky, dvě varianty

Místo chybových kódů propagovány specifické výjimky.

Varianta 1, využití podmínek (test  $D_f$ ) a propagace:

```
def f3(a, b):
    if type(a) != float or type(b) != float:                      #Spatny typ
        raise TypeError('Incorrect Type', type(a), type(b))
    if b == 0:                                                       #Jmenovatel rovn
        raise ZeroDivisionError('Divide by zero', b)
    if a * b < 0 :                                                 #Mimo definicni
        raise ValueError('Result is negative', a, b)
    return (a/b)**0.5
```

Varianta 2, kombinace chráněných bloků a propagace::

```
def f4 (a, b):
    try:
        return (a/b)**0.5
    except TypeError as e:                                         #Chybna data index?
        raise e
    except ZeroDivisionError as e:                                    #Jmenovatel roven nule
        raise e
    except ValueError as e:                                         #Mimo definicni obor
        raise e
    except BaseException as e:                                     #Nejaka jina chyba
        raise e
```

# 16. Chráněné bloky, agregace

Agregace aritmetických chyb:

```
def main():
    try:
        res = f3(a, b)
    except TypeError:                      #Chyba vstupu
        print(e)
    except ArithmeticError:                #Co se stalo, zdroj chyb
        print(e)
    except BaseException:                 #Aritmeticka chyba
        print(e)
    except BaseException:                 #Co se stalo, zdroj chyb
        print(e)
    except BaseException:                 #Jiny problem
        print(e)
    except BaseException:                 #Co se stalo, zdroj chyb
```

Agregace všech chyb:

```
def main():
    try:
        res = f3(a, b)
    except BaseException as e:            #Libovolna chyba
        print(e)
```

Pokud nepotřebujeme specifickou reakci, zpravidla preferována 2. varianta.

# 17. Vlastní výjimky

Při řešení problémů nutná reakce na jejich singularity.

Široké množství problémů, několik tříd výjimek nepokryje vše.

Možnost ošetření stavů, které nelze odchytit běžnými výjimkami.

Lze vytvářet **vlastní výjimky**, formou tříd.

Časté u rozsáhlých SW či specifických problémů.

Princip OOP, vlastní výjimka odpovídá třídě, která je potomkem třídy `BaseException`.

Inicializátor `__init__()`:

Předání informace, co se stalo, a zdroj chyby.

```
class FactorialError(BaseException):
    def __init__(self, value, message):      #Inicializator, zdoj a informace
        self.value = value                   #Initialize polozky rodicovske trid
        self.message = message               #Initialize polozky rodicovske trid
```

Volání:

```
def f(n):
    if abs(n) > 100:
        raise FactorialError(n, "Factorial, n>100")
    if n > 1:
        return n * f(n-1)
    else:
        return 1
```



# 18. Pareto Rule

Vilfredo Pareto (1848-1923):

*"About 80% of Italy's land belonged to 20% of the country's population."*

Tento princip lze přenést i do SW vývoje.

*"80% of the task is completed by 20% coding."*

Pozor, zvýšení efektivity o 20% spotřebuje 80% nákladů.

