Výjimky.

Výjimky. Hiearchie výjimek. Propagace výjimky. Chráněné bloky.

Tomáš Bayer | bayertom@natur.cuni.cz

Katedra aplikované geoinformatiky a kartografie, Přírodovědecká fakulta UK.

Obsah přednášky

1. Typy chyb v SW

V SW 3 základní typy chyb:

A) Syntax Errors:

Porušení gramatických chyb, chybná syntaxe.

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

Zabrání kompilaci (a spuštění) programu.

Zobrazeny v průběhu kódování (Syntax Checking) nebo při překladu

B) Runtime Errors:

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

Projevem pád nebo zatuhnutí programu.

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

Zpravidla nejdou identifikovat v průběhu kódování, komplikované ošetření, použití Debuggeru.

Z pohledu matematiky: hodnoty mimo D_f nebo H_f.

C) Logic Errors:

Chyba v modelu, postupu řešení, matematickém aparátu.

Konceptuální chyby, ale i přehlédnutí.

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

Nevede k Runtime Error.

Zpravidla obtížné odstranění, použití Debuggeru.

Obtížně identifikovatelné v průběhu kódování.

2. Ošetření Runtime Errors

Neošetřený Runtime Error:

pád aplikace $\Rightarrow$ ztráta dat $\Rightarrow$ finanční ztráta $\Rightarrow$ nespokojený uživatel.

Nežádoucí stav, nutno mu předcházet nebo být připraven.

Varianty reakce na “chybu”:

1. *Ukončení běhu programu*
   Nepředvídatelnost vzhledem ke vstupu, ztráta dat.
2. *Ošetření všech singularit*
   Snaha ošetřit všechny možné chybové stavy.
   Lze provést u jednodušších problémů.
   Prakticky nemožné ve komplexních situacích, nepřehledný kód.
3. *Použití Exceptions (Výjimek)*
   Odchycení a ošetření skupin chyb jednotně.

Často se kombinují metody 2), 3).

Mechanismus práce s výjimkami podobný pro většinu jazyků.

Neřeší chyby A) C).

3. Syntax vs. Runtime vs. Logic Error

Syntax Error:

```
>>> primt("Hello world")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'primt' 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)               # c = (a**2 + b**2)**0.5
```

4. Exception (Výjimka)

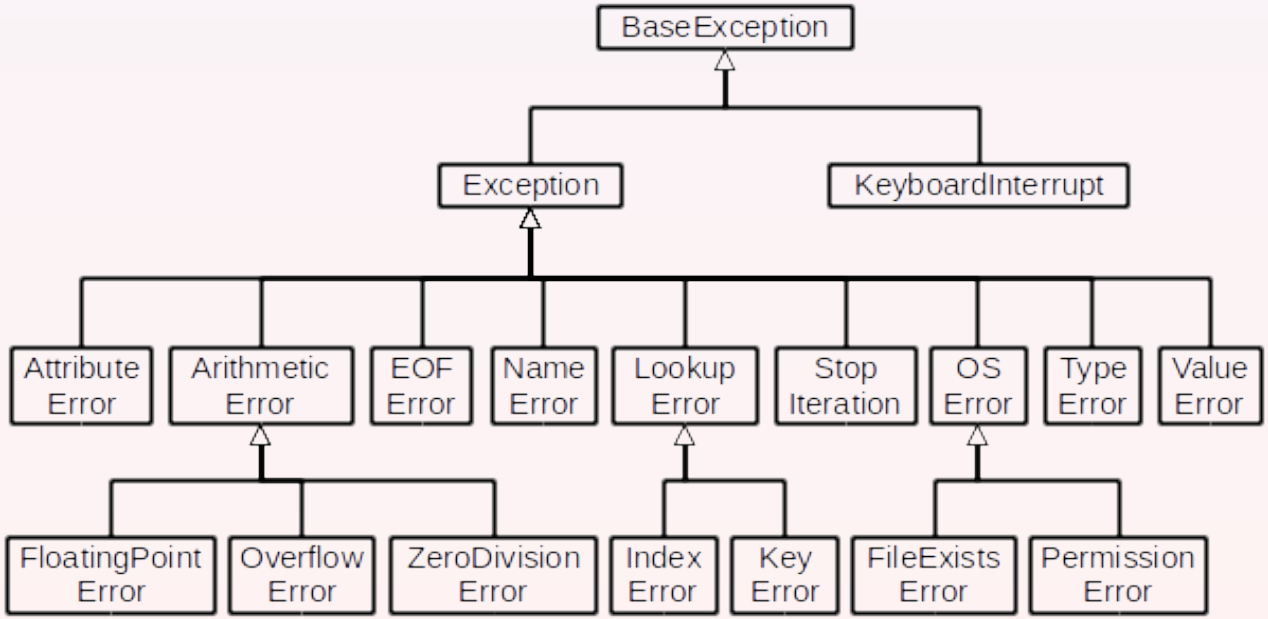
Stav programu v okamžiku, kdy dojde při jeho vykonávání k chybě.

- 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 použití výjimek:

- řešení příčiny,
- řešení následku.

Řešení příčiny:

Testování hodnot vstupních parametrů.

Cokoliv mimo D_f, vyhození výjimky.

Často poměrně komplikovaný postup, zejména u složitých problémů.

Obtížně lze detekovat všechny singularity před výpočtem.

Řešení následku:

Primárně nezjišťujeme, co chybové stavy způsobilo.

Pokud hodnota mimo H_f, vyhození výjimky.

Často jediné řešení u složitých problémů.

Varianta příčiny preferována.

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)     #Vyjimka s argumenty
raise exception           #Konkretni vyjimka
raise                     #Genericky typ
```

Nepoužívat generický typ, ale co nejkonkrétnější.
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 getR(per):
    if per > 0:                                          #Kladne perim
        return per / (2 * pi)
    else                                                #Zaporna hodnot
        raise ValueError("Perimeter < 0: ", per)  #Co se stalo
```

Volání:

```
>>> print(getR(1))
0.1591549430918953 5
```

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:                #Dojde-li k vyjimce typu 1, osetri
   pass
except (Ex_2, Ex_3):        #Dojde-li k 2 vyjimkam, osetri
   pass
except:                     #Dojde-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.

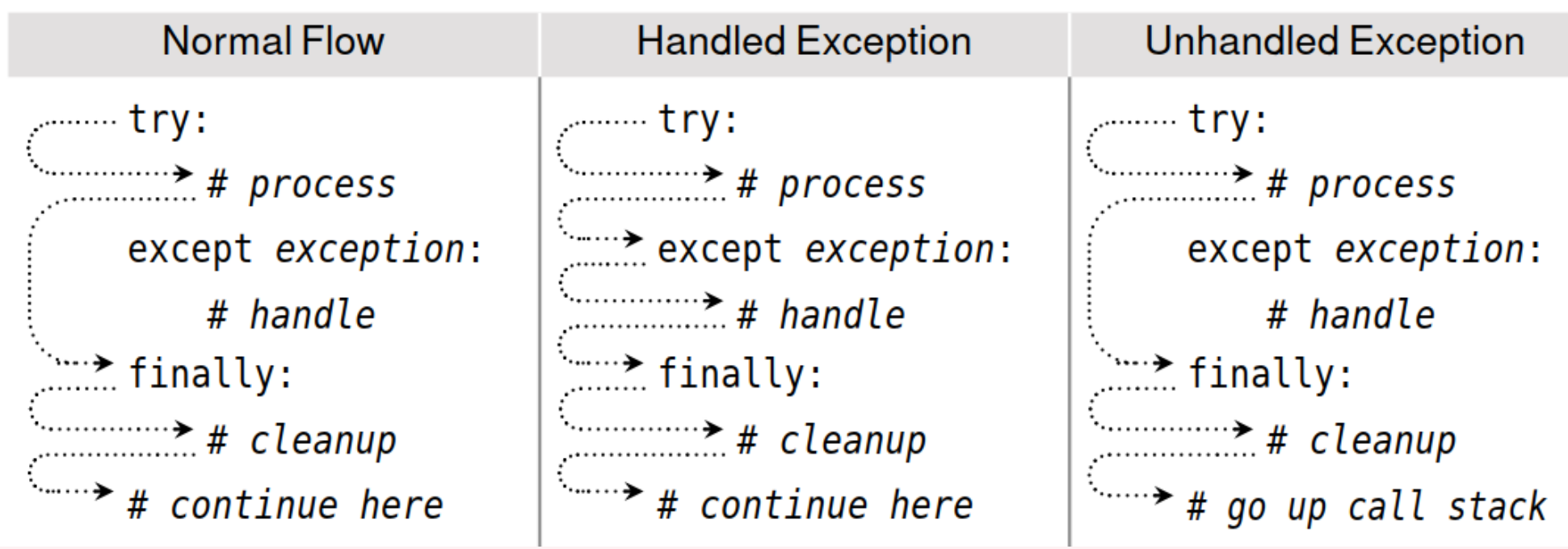
Blok `finally()`, volitelný, provede se vždy.

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.



Nezachycená výjimka může vést k pádu programu.

11. Řešení příčiny

Výpočet hodnoty

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

Nemusíme použít výjimky, vracíme info o chybovém stavu:

```
def f2 (x):
    if len(x) < 2:          #Chybi jmenovatel
        return -1
    if x[1] == 0:           #Jmenovatel nulovy
        return -2
    if x[0] > 0 and x[1] < 0 or x[1] < 0 and x[1] > 0: #Mimo definicni obor
        return -3


    return (x[0]/x[1])**0.5
```

Volání funkce:

```
>> x = [1, 2, 7]                #3 prvky, OK
>> r = f(x)                     #Vrati 1/2
1/2
>> x = [1, -2, 7]               #Vyjimka, vrati -3
-1
```

Nutnost definice chybových stavů v dokumentaci.

Výsledkem hodnota mimo H_f funkce.

12. Řešení následku

Řešení následku:

```
def f (x):
    try:
        return sqrt(x[0]/x[1])
    except IndexError:
        return -1        #Chybovy kod, spatny index
    except ZeroDivisionError:
        return -2        #Chybovy kod, deleni nulou
    except ValueError:
        return -3        #Chybovy kod, mimo definicni obor
    except Exception:
        return -4        #Nespecifikovana chyba
    else:

        print("OK results")
```

Volání funkce:

```
>> x = [1, 2, 7]              #3 prvky, OK
>> r = f(x)                   #Vrati 1/2
>> x = [1]                    #1 prvky, OK
>> r = f(x)                   #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 `Exception`.
- *Odchycení více výjimek*
  Řazení dle dědické hierarchie, od konkrétního k obecnému.

```
except Predek:                      #Konkretni
    ...
except Nejaky_rodic:                #Obecnejsi
    ...
except Nejaky_rodic_rodice:         #Jeste obecnejsi
    ...
```

  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 identifikace 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, ošetření příčiny:

```
def f (x):
    if len(x) < 2:                                      #Chybi jmenovatel
       raise IndexError('Size x < 2', x)
    if x[1] == 0:                                       #Jmenovatel nulovy
        raise ZeroDivisionError('Division by zero', x)
    if x[0] > 0 and x[1] < 0 or x[1] < 0 or x[1] > 0: #Mimo def. obor
        raise ValueError('Negative fraction', x)

    return (x[0]/x[1])**0.5
```

Varianta 2, ošetření následku:

```
def f (x):
    try:
       return (x[0]/x[1])**0.5
    except IndexError as e:              #Spatny index?
        raise e
    except ZeroDivisionError as e:       #Delime nulou?
        raise e
    except ValueError as e:              #Mimo definicni obor?
        raise e
    except Exception as e:               #Nejaka jina chyba?
```

16. Chráněné bloky, agregace

Agregace aritmetických chyb:

```
def main():
    try:
        res = f(x)
    except ArithmeticError:                          #Vsechny aritmeticke ch
        print('Bad data', x)                         #Co se stalo, zdroj chy
    except IndexError:                               #Index pole
        print('List contains <2 elements', x)        #Co se stalo, zdroj chy
```

Agregace všech chyb:

```
def main():
    try:
        res = f(x)
    except Error 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 `Exception`.

Inicializátor `__init__()`:
Předání informace, co se stalo, a zdroj chyby.

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