PROGRAMOVÉ STRUKTURY: PYTHON PRO POKROČILÉ

Skripty, třídy a objekty, událostmi řízené programování, modul Tkinter

Skripty

- Skupina po sobě jdoucích příkazů
  - uložená do souboru
- Původně krátký program
  - pro příkazový interpret operačního systému
- Z důvodu strukturování
  - je lépe vytvořit v souboru řídící funkci
  - a tu v tom souboru hned vyvolat
- Obvykle se spouští z příkazového řádku
  - častěji v Linuxu (Unixu) než Windows

Skript skript1.py


```
# Spuštění skriptu z konzole:

# python.exe skript1.py
# skript1.py

# Definice funkce main(), může se ale jmenovat i jinak než main
def main():
    print('Běží skript 1')
    jmeno = input('Jak se jmenuješ? : ')
    print('Ahoj {0}'.format(jmeno))

# zavolání funkce main()
main()
input('Skonči stiskem ENTER')
```

Argumenty skriptu

- Z příkazového řádku (konzole) lze předávat argumenty
  - ty jsou uloženy v podobě seznamu
  - v systémové proměnné `sys.argv`
    - vhodné použít `import sys`
- Spouští se výhradně z konzole
  - analogický způsob v IDLE nelze udělat
- Spuštění `skript2.py arg1 arg2` naplní seznam
  - `argv[0]` obsahuje `'skript1.py'`
  - `argv[1]` obsahuje `'arg1'`
  - `argv[2]` obsahuje `'arg2'`
  - `argv = ['skript1.py', 'arg1', 'arg2']`

Skript skript2.py


```
# Spuštění skriptu výhradně z konzole

# Pouze název skriptu: python.exe skript2.py
# Včetně libovolných argumentů:
# skript2.py arg1 arg2

import sys

def main():
    print("Při spuštění přidány argumenty, uloží se do sys.argv")
    print(sys.argv)

main()
```

Přepínače skriptu

- Skript může z konzole akceptovat i přepínače (volby)
  - stejně jako argumenty
- K tomu je vhodné použít modul getopt
  - obsahuje podporu pro
    - syntaktickou analýzu řetězce přepínačů
      - které chce skript rozpoznávat
    - a argumentů
- Funkce getopt() vrací dva seznamy
  1. seznam nalezených přepínačů
  2. seznam argumentů

Přepínače skriptu (2)

- Podoba přepínačů – známo z Unixu
  - zkrácený, např. `-x50 -v`
  - dlouhý, např. `--file=soubor.out --help`
  - lze je kombinovat
- Jaké přepínače máme definujeme maskou ve funkci `getopt()`
  - nezáleží na pořadí
  - např. `'f:h'`
    - symbol : říká, že přepínač vyžaduje hodnotu
  - např. `['file=', 'help']`
    - přepínače jsou uvedeny v seznamu
    - symbol = říká, že přepínač vyžaduje hodnotu

Skript skript3.py


```
# Spuštění skriptu výhradně z konzole

# Dva přepínače dostupné v krátké i dlouhé verzi
# -f nebo --file
# -h nebo --help

import getopt, sys

def main():
    (volby, argumenty) = getopt.getopt(sys.argv[1:], 'f:h', ['file=',
'help'])
    print('volby: {0}'.format(volby))
    print('argumenty: {0}'.format(argumenty))

main()
```

Pouštění skriptu skript3.py

- Přepínač v krátké verzi

```
skript3.py -h
volby: [('-h', '')]
argumenty: []
```

- Přepínač v dlouhé verzi

```
skript3.py –help
volby: [('--help', '')]
argumenty: []
```

Pouštění skriptu skript3.py (2)

- Přepínač v krátké verzi s argumenty

```
skript3.py -f soubor arg1, arg2
volby: [('-f', 'soubor')]
argumenty: ['arg1,', 'arg2']
```

- Přepínač v dlouhé verzi s argumenty

```
skript3.py -file=soubor arg1, arg2
volby: [('--file', 'soubor')]
argumenty: ['arg1,', 'arg2']
```

Modul fileinput

- Podpora pro zpracování řádků ze vstupních souborů
- Z proměnné `sys.argv` načte argumenty
  - použije je jako jména vstupních souborů
    - které po řádcích zkracuje
- Poskytuje metody (výběr)
  - `fileinput.input()`
    - čte řádky souborů (které jsou dány jako argumenty)
  - `fileinput.isfirstline()`
    - test prvního řádku souboru
  - `fileinput.filename()`
    - vrací název souboru
  - `fileinput.lineno()`
    - vrací počet řádek souboru

Skript skript4.py


```
# Spuštění skriptu výhradně z konzole

# Vypíše zdrojový kód bez komentáře a spočítá jeho počet řádků

# python4.py python4.py

import fileinput

def main():
    for radek in fileinput.input():
        if fileinput.isfirstline():
            print('Soubor {0:s}'.format(fileinput.filename()))
        if radek[:1] != '#':
            print(radek)
    print('Pocet radek {0}'.format(fileinput.lineno()))

main()
```

Přesměrování vstupů a výstupů

- Je realizováno při volání skriptu z konzole
  - přesměrování vstupu, tj. klávesnice
    - ze souboru: `<soubor.in`
  - přesměrování výstupu, tj. obrazovky
    - do souboru: `>soubor.out`
- Ukázka spuštění skriptu
  - vyžaduje existenci souboru `jmeno.txt`
  - za jménem musí být 2x odřádkováno

```
skript1.py <jmeno.txt >vystup.txt
```

Skript skript5.py


```
# Spuštění skriptu výhradně z konzole

# Ve vstupním souboru nahradí všechny výskyty
# hledaného řetězce novou hodnotou

# Přejmenování metody main() na hlavni()
# skript5.py main hlavni <skript1.py >novy_skript.py

import string
import sys

def main():
    sys.stdout.write(sys.stdin.read().replace(sys.argv[1],
                                              sys.argv[2]))

main()
```

Rozsáhlé skripty

- Krátkým skriptům stačí jedna funkce
- Rozsáhlé skripty je vhodné rozdělit
  - na několik funkcí
  - a jednu hlavní funkci `main()`
- Skript `skript6.py` provádí překlad dvouciferných čísel do slovního tvaru
  - řídící (hlavní) funkce `main()`
    - volá funkci `prekladDo99()`
    - se zadaným argumentem
  - ukázka spuštění skriptu

    `skript6.py 23`

Skript skript6.py


```
import sys

do9 =      {'0':'', '1':'one', '2':'two', '3':'three', '4':'four' }# atd.
od10do19 = {'0':'ten', '1':'eleven', '2':'twelve', '3':'thirteen'} # atd.
od20do90 = {'2':'twenty', '3':'thirty', '4':'fourty', '5':'fifty'} # atd.

def prekladDo99(cifernyTvar):
    if cifernyTvar == '0': return('zero')
    if len(cifernyTvar) > 2: return "více než dvouciferné číslo"
    cifernyTvar = '0' + cifernyTvar
    decades, units = cifernyTvar[-2], cifernyTvar[-1]
    if decades == '0': return do9[units]
    elif decades =='1': return od10do19[units]
    else: return od20do90[decades] + ' ' + do9[units]

def main():
    print(prekladDo99(sys.argv[1]))

main()
```

Skript jako modul a obráceně

- Skript lze použít jako modul
- Modul lze použít jako skript
- V obou případech, bude-li spuštěn v jiném
  - skriptu
  - či modulu
- Obojí zařídí hodnota proměnné `__name__`
  - `__main__` – pouštíme jako skript
  - název souboru – pouštíme jako modul
- Zařídíme to podmíněným voláním řídící funkce `main()`

```
if __name__ == '__main__' : main( )
else:  # případný inicializační kód modulu
```

Skripty skript7.py a skriptJakoModul.py

- Skript `skript7.py`
- stejný jako skript `skript6.py`
- jen volání funkce `main()` je podmíněno

```
if __name__ == '__main__':
    main()
else:
    print('{0} je zaveden jako modul'.format(__name__))
```

- Skript `skriptJakoModul.py`

```
import skript7
c = '12'
print(skript7.prekladDo99(c))
```

- Odpověď

```
skript7 je zaveden jako modul
twelve
```

Třídy a objekty

- Název třídy začíná velkým písmenem
- Metody třídy mají první parametr `self`
  - slouží pro přístup k samotnému objektu
  - slouží k identifikaci atributů třídy
- Třída může mít definován konstruktor
  - jeho název je `__init__`
  - uvnitř definuje atributy třídy svojí inicializací
- Privátní atributy začínají dvěma podtržítky
  - jsou použitelné pouze uvnitř třídy
- Je podporována vícenásobná dědičnost

Třídy a objekty (2)

- Volání konstruktoru rodiče
  - v těle konstruktoru potomka
    - tvar: `rodic.__init__(self, …)`
- Probíhá-li výpočet uvnitř třídy metody, mají přístup do
  - lokálního prostoru jmen
    - argumenty a proměnné deklarované v metodě
  - globálního prostoru jmen
    - funkce a proměnné deklarované na úrovni modulu
  - vestavěného prostoru jmen
    - vestavěné funkce a výjimky

Program oop1.py


```
class Ctverec:
    def __init__(self, strana): # Konstruktor
        self.strana = strana    # Explicitně uváděné this = self

    def vypocitejPlochu(self):
        return self.strana**2


class Kruh:
    def __init__(self, polomer):
        self.polomer = polomer

    def vypocitejPlochu(self):
        import math
        return math.pi*(self.polomer**2)
```

Program oop1.py (2)


```
class Obdelnik2x3:  # Když nemá konstruktor žádný
                    # parametr, nemusí se uvést
    def vypocitejPlochu(self):
        return 2*3


seznam = [Kruh(8), Ctverec(2.5), Kruh(3), Obdelnik2x3()]

for tvar in seznam:
# Tady se self již neuvádí, není zde definované
    print("Plocha je: {0}".format(tvar.vypocitejPlochu()))
```

Dědičnost obecně

- Dědit lze i od rodiče z jiného modulu

```
class Potomek(modul.Rodic):
    <příkaz1>
    …
    <příkazN>
```

- Příkazy jsou nejčastěji definicemi metod
- Lze definovat vnořené třídy
- Metody jsou implicitně dynamické (virtuální)
  - mohou překrýt metodu rodiče

Metody třídy

- Statickou metodu odekorujeme jako @staticmethod

```
@staticmethod
def nazevMetody(…): ...
```

  - nemá parametr `self`

- Podobné jsou metody třídy bez parametru `self`
  - dekorovány jako `@classmethod`
  - první parametr je třída

```
@classmethod
def nazevMetody(Trida, …): …
```

Destruktor

- Python je vybaven *garbage collectorem*
  - dynamicky spravuje paměť objektů
  - paměť objektu uvolní, jakmile na něj nic neodkazuje
- Destruktor
  - má pevně stanovený název `__del__()`
  - provede se, jakmile je objekt rušen
  - zpravidla obsahuje různé úklidové práce

Vícenásobná dědičnost

- Definice násobné dědičnosti

```
class Potomek(Rodic1, Rodic2, …, RodicM):
    <příkaz1>
    …
    <příkazN>
```

- Řešení problému násobného dědění
  - není-li atribut v `Potomkovi`
    - hledá se v `Rodiči1`, pak v rodiči `Rodiče1`, …
  - v `Rodiči2`, …
  - tj. do hloubky a pak z leva do prava

Program oop2.py


```
class Otec:
    def __init__(self):
        self.oci = 'zelené'
        self.usi = 'velké'
        self.ruce = 'šikovné'

    def popis(self):
        print('Oči {0}, uši {1}, ruce {2}.'.format(self.oci, self.usi, self.ruce))

class Matka:
    def __init__(self):
        self.oci = 'modré'
        self.nos = 'malý'
        self.nohy = 'dlouhé'

    def popis(self):
        print('Oči {0}, nos {1}, nohy {2}.'.format(self.oci, self.nos, self.nohy))
```

Program oop2.py (2)


```
class Potomek(Otec, Matka):
    def __init__(self):
        Otec.__init__(self)  # 1. rodič
        Matka.__init__(self) # 2. rodič

    def popis(self):
        print('Oči {0}, uši {1}, ruce {2}, nos {3}, nohy {4}.'.format(self.oci,
self.usi, self.ruce, self.nos, self.nohy))

petr = Potomek()
petr.popis()
```

Výsledek programu oop2.py

- Pořadí rodičů Otec, Matka

```
Oči modré, uši velké, ruce šikovné, nos malý, nohy dlouhé.
```

- Pořadí rodičů Matka, Otec, tj.

```
class Potomek(Otec, Matka):
    def __init__(self):
        Matka.__init__(self)
        Otec.__init__(self)
```

```
Oči zelené, uši velké, ruce šikovné, nos malý, nohy dlouhé.
```

Program oop3.py


```
class A:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def save(self, fn):
        f = open(fn, "w")
        f.write(str(self.x) + '\n') # preved na retezec
        f.write(str(self.y) + '\n')
        return f    # do stejneho souboru budou sve hodnoty
                    # pripisovat objekty odvozenych trid

    def restore(self, fn):
        f = open(fn)
        self.x = int(f.readline()) # preved zpet na puvodni typ
        self.y = int(f.readline())
        return f
```

Program oop3.py (2)


```
a = A(1, 2)                  # Vytvorime instance.
a.save('a.txt').close() # Ulozime instance.
newA = A(5, 6)               # Obnovime instance.
newA.restore('a.txt').close()
print("A: {0} {1}".format(newA.x, newA.y))
```

- Ukazuje zcela obecnou možnost vytváření perzistentních objektů
  - uložení objektů do souboru
  - na disku vytvoří soubor `a.txt` s obsahem 1, 2

Program oop4.py


```
class UkazkaDestruktoru:
    def __init__(self, soubor):
        self.file = open(soubor, 'w')
        self.file.write('tohle zapisuji do souboru\n')

    def write(self, retezec):
        self.file.write(retezec)

    def __del__(self):  # Destruktor
        self.write( "__del__ se provedlo")

def zkus():
    logickeJmenoSouboru = UkazkaDestruktoru('pomocnySoubor')
    logickeJmenoSouboru.write('tohle take zapisuji do souboru\n')
    # zde objekt logickeJmenoSouboru prestane existovat

zkus()
```

Prázdná třída

- Poslouží jako typ záznam

```
class Record:
    pass
```

- Vytvoření prázdného záznamu o Petrovi

```
petr = Record( )
```

- Atributy (položky) instance není nutné deklarovat předem

```
petr.jmeno = 'Petr Veliky'
petr.vek = 39
petr.plat= 40000
```

Výjimky jsou také třídy


```
try:
        #ošetřované příkazy
except TypVyjimky:
        #zpracování výjimky
except TypVyjimky:
#při neuvedení TypVyjimky zachytí se všechny dosud nechycené
        #zpracování výjimky
        ...
else:
        #činnost, když nenastane výjimka (nepovinné)
finally:
        #činnost prováděná ať výjimka nastane či nenastane
…
raise JmenoVyjimky  #vyhození výjimky, bráno jako instance výjimky
```

Program oop5.py


```
print('Start programu.')

try:
            data = open('data.txt', 'r')
            print('Soubor s daty byl otevren.')

            try:
                        hodnota = int(data.readline().split()[2])  #  radek tvaru slovo slovo cislo …
                        print('Hodnota je {0:d}.'.format(hodnota/(122 - hodnota)))

            except ZeroDivisionError:
                        print('Byla nactena hodnota 122.')

            except: print("Stalo se neco neocekavaneho.")

finally:
            data.close()
            print('Soubor s daty byl uzavren.')

print('Konec programu.')
```

Program oop6.py

- Uživatelem definované výjimky jsou instance tříd
  - odvozený z třídy `Exception`
- Hierarchie zpracování výjimek je shodná jako v Javě

```
class MujError(Exception):
    pass

try:
    raise MujError('Informace co se děje \n')

except MujError:
    print('Narazili jsme na chybu při zpracování.')
```

Program oop7.py


```
class ChybaZustatku(Exception):
    hodnota = 'Nelze vybrat. Na vašem účtu je jen {0:5.2f} korun.'

class Ucet:
    def __init__(self, pocatecniVklad):
        self.stav = pocatecniVklad
        print('Máte založen účet s vkladem {0:5.2f} korun.'.format(self.stav))

    def vlozit(self, castka):
            self.stav = self.stav + castka

    def vybrat(self, castka):
        if self.stav >= castka:
            self.stav = self.stav - castka
        else:
            raise ChybaZustatku(ChybaZustatku.hodnota.format(self.stav))
```

Program oop7.py (2)


```
    def zustatek(self):
        return self.stav

    def prevod(self, castka, ucet):
        try:
            self.vybrat(castka)
            ucet.vlozit(castka)
        except ChybaZustatku as e:
            print(str(e))

mujUcet = Ucet(300)
mujUcet.vybrat(200)
mujUcet.vybrat(200) # lze otestovat různé částky
```

Událostmi řízené programování a GUI

- Vlastnosti
  - program po spuštění čeká v nekonečné smyčce na výskyty událostí
  - při jejím výskytu provede odpovídající akci a nadále čeká
  - program skončí, až nastane událost indikující konec
- Události může generovat
  - OS (obvyklé u programů s GUI)
  - vnější čidla
- Předvedeme na freewarovém multiplatformním systému Tk
  - Tkinter pro Python

Program gui1.py

- Program zachycuje události stisknutí klávesy
  - dokud nenastane ukončující událost (mezerník)
- Na stisk klávesy reaguje výpisem kódu klávesy

1. Vytvoření třídy pro naši aplikaci
2. Třída obsahuje metody pro zpracování událostí
3. Součástí konstruktoru je vytvoření GUI okna
4. Vytvoříme instanci této třídy
5. Instanci zašleme zprávu *mainloop*

- POZOR! Spouštet pouze z konzole, ne z IDLE
  - IDLE samo využívá Tkinter

Základní prvky GUI Tkinter


```
>>> from tkinter import *
```

- naimportuje jména ovládacích prvků

```
>>> top = Tk()
```

- vytvoří widget (ovládací prvek) na nejvyšší úrovni
  - ostatní budou jeho potomky
  - je to okno, do něj budeme přidávat další prvky

```
>>> dir(top)
```

- vypíše všechna jména (atributy) objektu top třídy Tk

```
>>> F = Frame(top)
```

- vytvoří widget rámec (frame), který je potomkem top
- do něj budeme umisťovat ostatní prvky

```
>>> F.pack()
```

- aktivuje packer, rámec je prázdný, zmenší ho na lištu

Základní prvky GUI Tkinter (2)


```
>>> lPozdrav = Label(F, text = 'everybody out')
```

- vytvoří objekt třídy `Label` jako potomka `F`
- atribut má iniciovanou hodnotu
- lze definovat barvu, typ písma, …

```
>>> lPozdrav.pack()
```

- popisek se objeví v okénku

```
>>> lPozdrav.configure(text = 'vsichni ven')
```

- metoda `configure()` dovolí měnit vlastnosti objektu

```
>>> lPozdrav['text'] = 'everybody out'
```

- stejná akce jako metodou `configure()`
- v případě změny jednoho atributu se jedná o kratší zápis

Základní prvky GUI Tkinter (3)


```
>>> F.master.title('Ahoj')
```

- dovoluje nastavit titulek okna metodou title pro widget na vrcholu hierarchie
  - objekt top

```
>>> bQuit = Button(F, text = 'Konec', command = F.quit)
```

- vytvoří tlačítko s nápisem Konec
  - je spojeno s příkazem `F.quit`
  - předáváme jméno metody `quit()`

```
>>> bQuit.pack()
```

- zajistí zviditelnění tlačítka

```
>>> top.mainloop()
```

- odstartuje provádění smyčky
- činnost teď řídí Tkinter

Program gui2.py


```
from tkinter import *

# Vytvorime okno.
top = Tk()
F = Frame(top)
F.pack()

# Pridame ovladaci prvky.
lPozdrav = Label(F, text="everybody out")
lPozdrav.pack()

F.master.title('Nazev')
bQuit = Button(F, text="Konec", fg='white',bg='blue',command=F.quit)
bQuit.pack()

# Spustime smycku udalosti.
top.mainloop()
quit()
```

Program gui3.py


```
from tkinter import *

vrchol = Tk()

def odezva(e):
   print('klik na {0}, {1}'.format(e.x, e.y))

def klik():
    print("Stiskl jsi tlačítko!")

f = Frame(vrchol, width = 200, height = 300)
f.bind('<Button-1>', odezva) # <Button-1> je leve, <Button-2> je prave tlacitko
f.pack()
for i in range(4):
    tlacitko=Button(text = "Já jsem tlačítko", command = klik)
    tlacitko.pack()

vrchol.mainloop()
quit()
```

Program gui4.py


```
from tkinter import *

# Nejdříve vytvoříme funkci pro ošetření události.
def vymazat():
  eTxt.delete(0, END) # Bude mazat text

def eHotKey(u):
  vymazat() # Zavedeme pro mazání i hot key

# Vytvoříme hierarchicky nejvyšší okno a rámeček.
hlavni = Tk()
F = Frame(hlavni)
F.pack()
```

Program gui4.py (2)


```
# Nyní vytvoříme rámeček s polem pro vstup textu.
fVstup = Frame(F, border = 20)     # Velikost ohraničení je 20
eTxt = Entry(fVstup)               # Prvek pro zadávání jednořádkového textu
eTxt.bind('<Control-m>', eHotKey) # Navázání CTRL+M na mazání
fVstup.pack()
eTxt.pack()

# Nakonec vytvoříme rámečky s tlačítky.
# Pro zviditelnění je vmáčknutý = SUNKEN
fTlacitka = Frame(F, relief = SUNKEN, border = 1)
bVymazat = Button(fTlacitka, text = "Vymaz text", command = vymazat)
bVymazat.pack(side = RIGHT, padx = 5, pady = 2)
bKonec = Button(fTlacitka, text = "Konec", command = F.quit)
bKonec.pack(side = LEFT, padx = 5, pady = 2)
fTlacitka.pack(side = TOP, expand = True)

# Nyní spustíme čekací smyčku
F.mainloop()
quit()
```

OO přístup ke GUI aplikacím

- Celá aplikace se zapouzdří do třídy buď tak, že
1. odvodíme třídu aplikace od Tkinter třídy `Frame`
   - užívá dědičnost
2. uložíme referenci na hierarchicky nejvyšší okno do členské proměnné
   - užívá kompozici
- Použijeme variantu 2 pro konstrukci
   - textového pole `Entry`
   - tlačítky `Vymaz` a `Konec`

Program gui5.py

- Do konstruktoru aplikace dáme jednotlivé části GUI
- Referenci na prvek typu `Frame` přiřadíme do `self.hlavniOkno`
  - zajistíme metodám třídy přístup k prvku typu `Frame`
- Ostatním prvkům, ke kterým přistupujeme, přiřadíme členským proměnným instance z `Frame`
- Funkce pro zpracování událostí se stanou metodami třídy aplikace
  - mohou přistupovat k datovým členům aplikace pomocí reference `self`

Program gui5.py


```
from tkinter import *

class AplikaceVymazat:
    def __init__(self, rodic = 0):
        self.hlavniOkno = Frame(rodic, width = 200, height = 100)
        self.hlavniOkno.pack_propagate(0)

        # Vytvoříme widget třídy Entry
        self.vstup = Entry(self.hlavniOkno)
        self.vstup.insert(0, "Pocatecni text")
        self.vstup.pack(fill = X)

        # Nyní přidáme dvě tlačítka a použijeme efekt drážky.
        fTlacitka = Frame(self.hlavniOkno, border = 2, relief = GROOVE)
        bVymazat = Button(fTlacitka, text = "Vymazat", width = 8, height = 1, command = self.vymazatText)
        bKonec = Button(fTlacitka, text = "Konec", width = 8, height = 1, command = self.hlavniOkno.quit)
        bVymazat.pack(side = LEFT, padx = 15, pady = 1)
        bKonec.pack(side = RIGHT, padx = 15, pady = 1)
        fTlacitka.pack(fill = X)
        self.hlavniOkno.pack()

        # Nastavíme nadpis okna.
        self.hlavniOkno.master.title("Vymazat")
```

Program gui5.py (2)


```
    def vymazatText(self):
        self.vstup.delete(0, END)

aplikace = AplikaceVymazat()
aplikace.hlavniOkno.mainloop()
quit()
```

Běžné Tk prvky

Button	Tlačítko
Canvas	Plátno
Checkbutton	Zaškrtávací tlačítko
Entry	Jednořádkový vstup
Frame	Rámeček
Label	Popiska, nálepka
Listbox	Více řádek textu
Menu	Rolovací menu
Menubutton	Tlačítko výběru
Radiobutton	Přepínací tlačítko
Scale	Posuvný prvek, stupnice
Scrollbar	Posuvník
Text	Textový editor