

PYTHON

Honza Vrbata

vrbata@gopas.cz

Co je to PYTHON ?

- "Moderní" programovací jazyk (rok vzniku 1990)
- Autor Guido van Rossum, univerzita v Amsterdamu
- Vychází z jazyka ABC
- Je nezávislý na platformě (UNIX, Windows, MacOS X, OS/2, atd.)
- Je velice produktivní -> umožňuje rychlý návrh aplikací
- Má velice elegantní a čistý návrh
- Velmi dobrá integrace s jinými jazyky C, C++ (CPython), .NET (Iron Python) a JAVA (Jython) !!!
- Velké možnosti rozšíření externími moduly
- Je multiparadigmatický, umožňuje psaní programů pomocí více paradigmat (procedurální, objektové, částečně funkcionální)
- Dobře se učí :-)

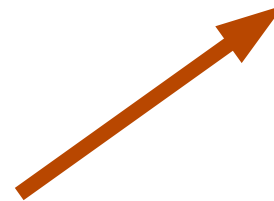
Struktura zápisu programu

```
int c;  
float a,b;  
a=2.584;  
c=1;  
while (c<100) {  
    ++c;  
    b=a*c;  
}
```

C

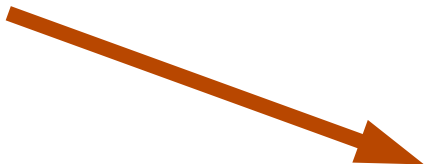


Pascal



```
var c: integer;  
    a,b: real;  
a:=2.584;  
c:=1;  
while (c<100) do begin  
    c:=c+1;  
    b:=a*c;  
end
```

PYTHON



```
a=2.584  
c=1  
while c<100:  
    c+=1  
    b=a*c  
print("Jede se dal")
```

Python zdroje

- Domácí stránka www.python.org
- V současné době se udržují dvě **stabilní větve** Pythonu, verze **2** a **3**.

Python shell = IDLE

Delete, Ctrl-d	smazání znaku vpravo od kurzoru
Backspace	smazání znaku vlevo od kurzoru
Ctrl-w	vyjmout
Ctrl-y	vložit
Alt-w	kopírovat
Alt-p	poslední příkaz z historie
Alt-n	následující příkaz v historii
Ctrl-z	zpět (undo)
Alt-z	obnovit (redo)
Alt-/	doplnění (expanze)

Komentáře

Můj první krásný program

a=2.584 # a je vstupní hodnota

c=1

while c<100:

c+=1

b=a*c

Datové typy (třídy)

- **integer** – celé číslo - **a=10**, **b=0xa**, **c=0o12**
- **long integer** – dlouhé celé číslo - **a=88888L**, **b=-777777L**
- **float** – čísla s pohyblivou desetinnou čárkou - **a=1.0**, **b=-1.5e3**
- **complex number** – komplexní číslo - **a=1+2j**, **b=1.5-2.58j**

(moduly *decimal* a *fractions*)

- **boolean** – **True**, **False**
- **None** - **NULL**, **nil**
- **string** (unicode string) – řetězec - **a="ahoj"**, **b='nazdar'**,
c=""zdar a silu""
- **tuple** – n-tice – **a=("jedna","dve",3,4)**
- **list** – seznam – **a=["jedna","dve",3,4]**
- **set** – množina – **a={"jedna","dve",3,4}**
- **dictionary** – slovník – **a={"jedna":1, "dve":2}**

Měnitelné a neměnitelné objekty

Neměnitelné (immutable) typy : **integer, *long integer*, float, complex number, string, tuple, frozenset, *bytes***

Měnitelné (mutable) typy : **list, set, dictionary, *bytearray***

"Aritmetické" operátory

- +** ... číselný součet, *spojování posloupností*
- ... číselný rozdíl
- *** ... číselný součin, *kopírování posloupností*
- **** ... číselné umocňování
- /** ... číselné dělení
- %** ... zbytek po celočíselném dělení, *operátor pro formátování řetězce*

Python akceptuje zkrácené operátory : **+=**, **-=**, ***=**, ...

Operátory porovnání

== ... rovno

!= ... nerovno

< ... menší než

> ... větší než

<= ... menší nebo rovno než

>= ... větší nebo rovno než

in ... je přítomno v posloupnosti (*not in*)

is ... je stejný objekt, ekvivalence (*is not*)

Logické operátory

and ... logický součin

or ... logický součet

not ... logická negace

Binární operátory

- &** ... logický součin, **AND**
- |** ... logický součet, **OR**
- ^** ... exkluzivní součet, nonekvivalence, **XOR**
- ~** ... negace, **NOT**
- >>** ... posun vpravo
- <<** ... posun vlevo

Čísla

1, -10, 458

celá čísla

0xff12, -0x14

celá čísla (hexadecimální zápis)

0o717, -0o1

celá čísla (oktalový zápis)

1.0, -5.5e3

čísla s pohyblivou desetinnou čárkou

9999999L

celá dlouhá čísla (Python 2)

1j, 5+4j, 2+5.4j

komplexní čísla

Vestavěné matematické funkce *abs*, *min*, *max*, *round*, ...

Další matematické funkce se nacházejí v modulu *math*.

Matematické funkce pro práci s komplexními čísly jsou v modulu *cmath*.

Řetězce

`r = 'Retezec se znakem noveho radku na konci \n'`

`r = "Retezec se znakem noveho radku na konci \n"`

`r = '''Retezec uvozeny tremi apostrofy
pres vice radku'''`

`r = """Retezec uvozeny tremi uvozovkami
pres vice radku"""`

`r = r"neupraveny, syrovy retezec si nevsima zadnych escape
sekvenci \n"`

`r = u"unicode řetězec \n"` (*Python 2*)

Přístup k řetězcům viz n-tice. Základní funkce *chr*, *ord*, *len*.
Další funkce se nacházejí v modulu *string*.

Řetězce – escape sekvence

\n ... nový řádek

\t ... tabulátor

**** ... zpětné lomítko

\' ... apostrof

\" ... uvozovky

\nnn ... osmičkový ASCII znak

\xnn ... šestnáctkový ASCII znak

Řetězce – *UNICODE*

`r = u"Řetězec UNICODE. Umožňuje používat unicode escape sekvence."`

`unicode('ščšć','iso8859-2')` ... funkce převede řetězec z osmibitového kódování do UNICODE
`unicode('ščšć','iso8859-1')`
`unicode('ščšć','ascii')`

`'ČŠČČŘ'.decode('iso8859-2')` ... metoda ***decode*** provede totéž co funkce ***unicode***

`'příšera'.encode('utf-8')`
`'příšera'.encode('base64')`

PYTHON 3 obsahuje pouze typ Unicode řetězce !!!

Kódování zdrojového kódu

Je nutné interpreteru sdělit v jakém kódování jsou řetězce umístěné ve zdrojovém kódu našeho programu.

Do zdrojového kódu umístíme **magický řádek** s touto informací.

```
#!/usr/bin/python
```

```
# -*- coding: <encoding name> -*-
```

například :

```
# -*- coding: utf-8 -*-
```


Řetězce – konverze datových typů

str (objekt) - řetězec

int (řetězec) - číslo typu *integer*

long (řetězec) - číslo typu *long integer*

float (řetězec) - číslo typu *float*

complex (řetězec) - číslo typu *complex*

Řetězce – formátování řetězců

formatovacíRetezec % objekt

formatovacíRetezec % (objekt1, objekt2, ...)

formatovacíRetezec % slovník

a=10

b=1.2

print ("Promenna a ma hodnotu %d" % a)

print ("Promenna a ma hodnotu %d, b hodnotu %f" % (a,b))

Základní formátovací operátory :

s ... řetězcové vyjádření objektu

d ... celé číslo

f ... číslo s desetinnou čárkou

Řetězce – formátování řetězců

Objekt **string** obsahuje metodu **format**, která také slouží k formátování řetězců.

```
age = 20  
name = 'Swaroop'
```

```
print ('{0} was {1} years old when he wrote this book'.format(name,  
age))  
print ('Why is {} playing with that python?'.format(name))
```

Řetězce – další operace

Základní metody objektu *string* :

find(s,substring)

join(seznam)

lower(s)

upper(s)

replace(s,substring,replace)

split(s,separator)

strip(s)

lstrip(s)

rstrip(s)

.....

hledání podřetězce

spojení posloupnosti do řetězce

změna velikosti písmen na malá

změna velikosti písmen na velká

záměna podřetězce

rozdělení řetězce na části

oříznutí řetězce z obou stran

oříznutí řetězce z levé strany

oříznutí řetězce z pravé strany

Seznam, list

x=[] ... vytvoří prázdný seznam

x=[1,2+3j,"dalsi",4] ... vytvoří a naplní seznam

[1,2] + ["tri","ctyri"] ... **[1,2,"tri","ctyri"]**

2 * [1,2] ... **[1,2,1,2]**

list ('gopas') ... **['g','o','p','a','s']** ... vytvoří seznam z posloupnosti

list ((1,85,96)) ... **[1,85,96]** ... vytvoří seznam z posloupnosti

len (['g','o','p','a','s']) = 5 ... počet prvků seznamu

Seznamy – přístup k seznamům, indexy, řezy

x=[1 , 2 , 3 , 4 , 5 , 6]

x[2] ... 3

x[-3] ... 4

x[1:4] ... [2,3,4]

x[:-2] ... [1,2,3,4]

x[3:] ... [4,5,6]

x[:] ... [1,2,3,4,5,6] (*vytváří mělkou kopii*)

Seznamy – modifikace seznamů

x=[1,2,3,4,5,6]

y=["dve","tri","ctyri"]

x[1]="dve" ... [1,"dve",3,4,5,6]

x[1] = y ... [1, ["dve", "tri", "ctyri"], 3, 4, 5, 6]

x[1:3] = y ... [1,"dve","tri","ctyri",4,5,6]

x.append(7) ... [1,2,3,4,5,6,7]

x.insert(0,"nula") ... ["nula",1,2,3,4,5,6]

x.remove(5) ... [1,2,3,4,6]

del x[1] ... [1,3,4,5,6]

del x[2:4] ... [1,2,5,6]

Seznamy – další operace

x=[2,4,1,5,6,3,3]

x.sort() ... [1,2,3,3,4,5,6]

3 in x ... True (pravda)

2 not in x ... False (nepravda)

min(x) ... 1

max(x) ... 6

x.index(1) ... 2 (index prvku v seznamu)

x.count(3) ... 2 (počet výskytů v seznamu)

n-tice (tuple)

x=() ... vytvoří prázdnou n-tici

x=(1,) ... vytvoří jednoprvkovou n-tici

x=(1,2+3j,"dalsi",4) ... vytvoří a naplní n-tici

(1,2) + ("tri","ctyri") ... (1,2,"tri","ctyri")

2 * (1,2) ... (1,2,1,2)

tuple ('gopas') ... ('g','o','p','a','s') ... vytvoří n-tici z posloupnosti

tuple ([1,85,96]) ... (1,85,96) ... vytvoří n-tici z posloupnosti

len (('g','o','p','a','s')) ... 5 ... počet prvků n-tice

N-tice – přístup k n-ticím

x=(1,2,3,4,5,6)

x[2] ... 3

x[-3] ... 4

x[1:4] ... (2,3,4)

x[:-2] ... (1,2,3,4)

x[3:] ... (4,5,6)

Množiny

x=set() ... vytvoří prázdnou množinu

x={1,2,3} ... vytvoří a naplní množinu

len(x) ... počet prvků v množině

x.add(4) ... přidá další prvek do množiny

x.remove(4) ... odstraní prvek z množiny (pokud prvek v množině není vytvoří výjimku)

x.discard(4) ... odstraní prvek z množiny

x.clear() ... vymaže všechny položky množiny

x.copy() ... mělká kopie množiny (*shallow copy*)

x1.union(x2) ... sjednocení množin

x1.intersection(x2) ... průnik množin

Slovníky

asociativní pole, transformační tabulka, hashovací tabulka

x={} ... vytvoří prázdný slovník

x={"cerveny":"red","zeleny":"green"} ... vytvoří a naplní slovník

x["bily"]="white" ... přiřadí položku do slovníku

len(x) ... 3 ... počet prvků ve slovníku

x.keys() ... ["cerveny","zeleny","bily"] ... seznam klíčů (na pořadí nezáleží)

x.values() ... ["red", "white", "green"] ... seznam hodnot (na pořadí nezáleží)

x.items() ... [('cerveny', 'red'), ('bily', 'white'), ('zeleny', 'green')]

x.clear() ... vymaže všechny položky slovníku

x.copy() ... mělká kopie slovníku (*shallow copy*)

Odkazy a kopie I.

```
a=[[0,1],2,3,4]
```

```
b=a
```

```
b[0][1]="jedna"
```

```
print (a,b)
```

Odkazy a kopie II.

```
a=[[0,1],2,3,4]
```

Mělká kopie (kontejnerové objekty, vytváří odkazy na položky v původním objektu) :

```
b=a[:]
b.append(5)
print a,b
b[0][1]="jedna"
print (a,b)
```

Hluboká kopie vytváří nový objekt a rekurzivně kopíruje všechny objekty, které obsahuje. Vytváří tak úplně nezávislou kopii :

```
import copy
b=copy.deepcopy(a)
b.append(5)
print a,b
b[0][1]="jedna"
print (a,b)
```

*Řízení běhu programu větvení pomocí **if-else***

if *logický_výraz*:
 blok příkazů 1
else:
 blok příkazů 2

*Řízení běhu programu větvení pomocí **if-elif-else***

```
if log_výraz1:  
    blok příkazů 1  
elif log_výraz2:  
    blok příkazů 2  
elif log_výraz3:  
    blok příkazů 3  
.....  
else:  
    blok příkazů
```


Řízení běhu programu cyklus while

```
while logický_výraz:  
    blok příkazů 1  
else:  
    blok příkazů 2
```

Řízení běhu programu cyklus for

```
for jméno in kolekce:  
    blok příkazů 1  
else:  
    blok příkazů 2
```

Příkazy : **range**, **break** a **continue**

Funkce a procedury

```
def název (parametr1, parametr2, ...):  
    """Dokumentální řetězec"""  
    tělo funkce  
    return hodnota
```

Funkce a procedury

```
def pozdrav ( ):  
    """Vytiskne pozdrav"""  
    print ("Dobrý den !")  
  
pozdrav ( )
```

```
def secti (a,b):  
    """Provede soucet dvou cisel"""  
    c=a+b  
    return c  
  
x=secti (3,2)  
print (x)
```

Lambda konstrukt

```
toUpper = lambda r : r.upper()
```

```
print (toUpper("ahoj, jak se mate ?"))
```

Funkce a procedury – implicitní parametry

```
def mocnina (z,e=2):  
    """Počítá celočíselnou mocninu"""  
    x=z  
    while e>1:  
        x=x*z  
        e=e-1  
    return x  
  
print (mocnina(2,3))  
print (mocnina(2))
```

Funkce a procedury – předání jménem parametru

```
def mocnina (z,e=2):  
    """Počítá celočíselnou mocninu"""  
    x=z  
    while e>1:  
        x=x*z  
        e=e-1  
    return x  
  
print (mocnina (2,3))  
print (mocnina (e=3,z=2))
```

Funkce a procedury – proměnný počet parametrů I

```
def maximum (*cisla):  
    """Počítá maximum z posloupnosti čísel"""  
    m=cisla[0]  
    for n in cisla[1:]:  
        if n>m:  
            m=n  
    return m  
  
print (maximum(1,5,4,2))  
print (maximum(1,5,4,2,23,23))
```


Funkce a procedury – proměnný počet parametrů II

```
def příklad (**parametry):  
    print (parametry)
```

```
příklad (a=1,b=2,c=3)
```

Funkce a procedury – proměnný počet parametrů III

```
def funkce(*args,**kwargs):  
    print (args)  
    print (kwargs)
```

```
funkce(1,2,3,a=4,b=5)
```

Funkce a procedury – měnitelné objekty jako argumenty

```
def priklad (n, seznam1, seznam2):  
    """Měnitelné objekty"""  
    seznam1.append('Jetel')  
    seznam2=[3,2,1]  
    n+=1
```

```
a=1  
b=['Franta']  
c=[1,2,3]  
priklad(a,b,c)  
print(a,b,c)
```

Funkce a procedury – jmenné prostory

```
def priklad ():  
    """Jmenné prostory"""  
    a=1  
    b=2
```

```
a=10  
b=20  
priklad()  
print(a,b)
```

Funkce a procedury – jmenné prostory

```
def priklad ():  
    """Lokální a globální proměnné"""  
    print(a)  
    b=2
```

```
a=10  
b=20  
priklad()  
print(a,b)
```

Python automaticky zpřístupňuje jména z globálního prostoru jmen uvnitř funkcí pouze pro čtení !!!

Funkce a procedury – práce se sekvencemi

```
def secti (a,b):  
    return (a+b)
```

```
prvni=[1,2,3]  
druhy=[4,5,6]
```

```
print (map(secti,prvni,druhy))
```

Funkce **map** aplikuje definovanou funkci na prvky posloupnosti. Návrátovou hodnotou je posloupnost výsledků.

Funkce a procedury – práce se sekvencemi

```
def sude (a):  
    return not (a%2)  
  
cisla=range(20)  
  
print (filter(sude,cisla))
```

Funkce *filter* aplikuje definovanou funkci na prvky posloupnosti, návratovou hodnotou této funkce je boolean hodnota. Výsledkem je posloupnost vstupních prvků, pro které je funkce pravdivá.

Generování seznamů – list comprehension

[<výraz> **for** <proměnná> **in** <kolekce> **if** <podmínka>]

Lze popsat takto :

```
L = []  
for proměnná in kolekce:  
    if podmínka:  
        L.append(výraz)
```

Příklad :

[n for n in range(10) if n%2==0]

Moduly

modul.py



```
"""Pokusný modul"""  
a=10  
def soucet (x,y):  
    """Součet čísel"""  
    s=x+y  
    return s
```

```
import modul
```

```
print (modul.a)  
vysledek=modul.soucet(2,3)  
print (vysledek)
```

Moduly

modul.py



```
"""Pokusný modul"""  
a=10  
def soucet (x,y):  
    """Součet čísel"""  
    s=x+y  
    return s
```

```
import modul as m
```

```
print (m.a)  
vysledek=m.soucet(2,3)  
print (vysledek)
```

Moduly

modul.py



```
"""Pokusný modul"""  
a=10  
def soucet (x,y):  
    """Součet čísel"""  
    s=x+y  
    return s
```

```
from modul import *
```

```
print (a)  
vysledek=soucet(2,3)  
print (vysledek)
```

Moduly (chráněná jména)

modul.py

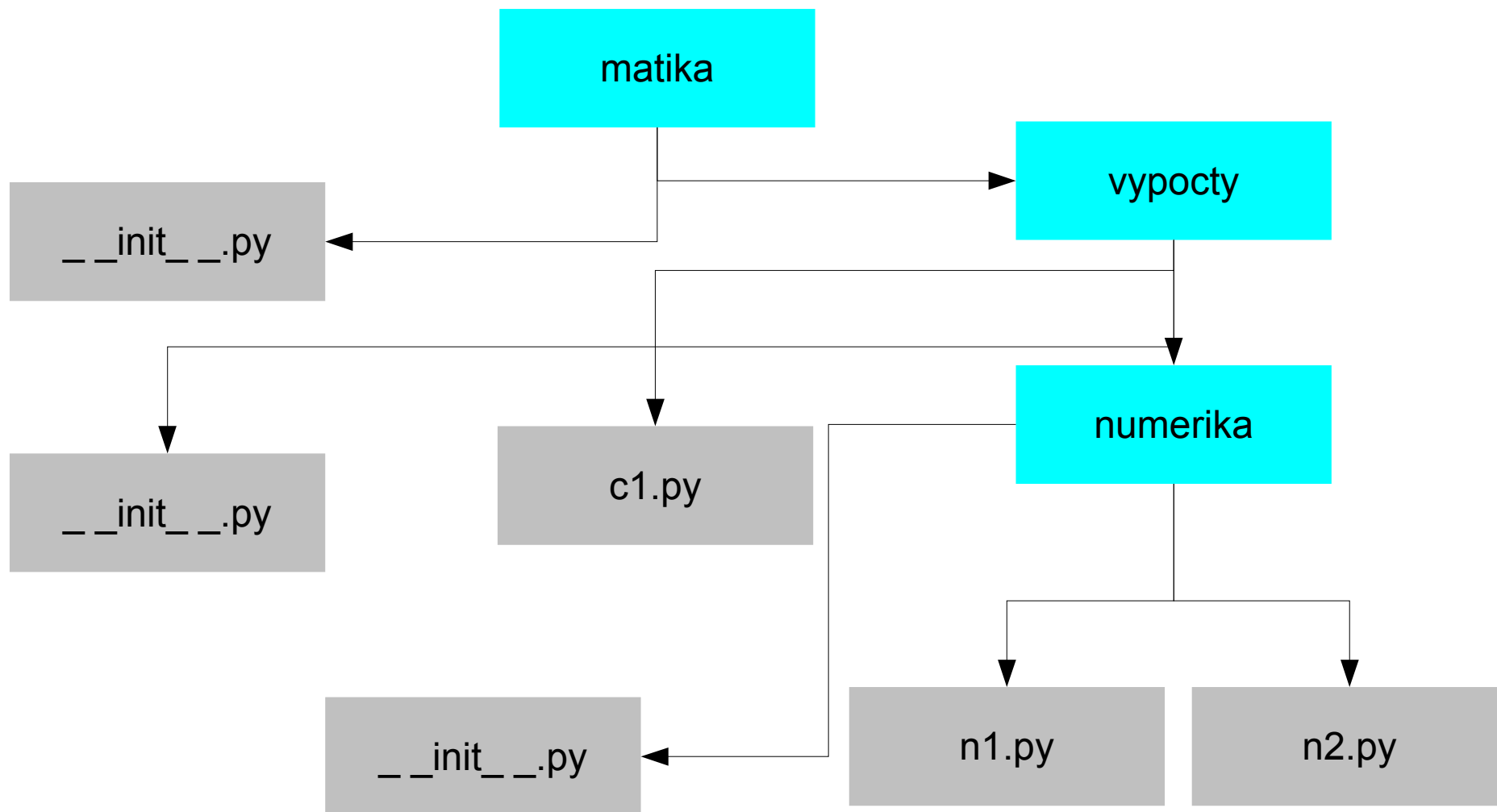


```
"""Pokusný modul"""  
a=10  
b=20  
def soucet (x,y):  
    """Součet čísel"""  
    s=x+y  
    return s
```

```
from modul import *
```

```
print (a)  
print (b)  
vysledek=soucet(2,3)  
print (vysledek)
```

Balíčky (packages)



Ošetření chyb

Typy chyb :

- **Syntaktické**
- **Logické** (statická analýza, debugging)
- **I/O** (vstupně výstupní)

Automatická kontrola kódu, statická analýza

- V případě dynamického jazyka, jako je Python, může docházet k logickým chybám, od kterých by nás staticky typovaný jazyk izoloval.
- K řešení nejen tohoto problému je možné používat analyzátory kódu jako jsou *pylint*, *pychecker* nebo *mypy*.

Odlad'ování, debugging

- Vložení debug kódu.
- Použití debuggeru. Python obsahuje debugger v modulu *pdb*. Existují nadstavby, například *pydb*. IDE jako *Wing* nebo *Pycharm* mají podporu pro debugging vestavěnou.

Ošetření chyb – mechanismus výjimek I.

- 1) Chyb si prostě nevšímáme**
- 2) Sledujeme návratové hodnoty ze všech I/O funkcí**
(používá se v klasických jazycích jako Pascal, C, ...)
- 3) Mechanismus VYJÍMEK**
(moderní jazyky jako JAVA, PYTHON, RUBY, ...)

Ošetření chyb – mechanismus výjimek II.

funkce ziskejZeServeru

try *zkus vykonat následující část programu*
otevriSitoveSpojeni....
posliHTTPpozadavek....
uzavriSitoveSpojeni....

except *pokud se během vykonávání vyskytla chyba*
obsloužení chyby

Ošetření chyb – mechanismus výjimek III.

try:

print (1/0)

except ZeroDivisionError:

print ("Pozor, chyba dělení nulou !!!")

Ošetření chyb – mechanismus výjimek IV.

try:

```
soubor = open("soubor.txt","r")
```

except IOError as vyjimka:

```
if (vyjimka.errno==2): # soubor neexistuje  
    print ("Pozor, tento soubor neexistuje")
```

else:

```
print ("Soubor existuje : %d : %s" % (vyjimka.errno,  
    vyjimka.strerror))
```

Ošetření chyb – mechanismus výjimek V.

```
def deleni (a,b):  
    if b==0:  
        raise ZeroDivisionError, "Chyba deleni nulou"  
    return 0  
v=a/b  
return v
```

```
try:  
    v=deleni(4,0)  
    print (v)  
except ZeroDivisionError as text:  
    print (text)
```

Ošetření chyb – mechanismus výjimek VI.

```
def deleni (a,b):  
    if b==0:  
        raise ZeroDivisionError, "Chyba deleni nulou"  
    v=a/b  
    return v
```

```
try:  
    v=deleni(4,0)  
    print (v)  
except ZeroDivisionError as vyjimka:  
    print (vyjimka)  
finally:  
    print ("Úklidová část")
```

Ošetření chyb – mechanismus výjimek VI.

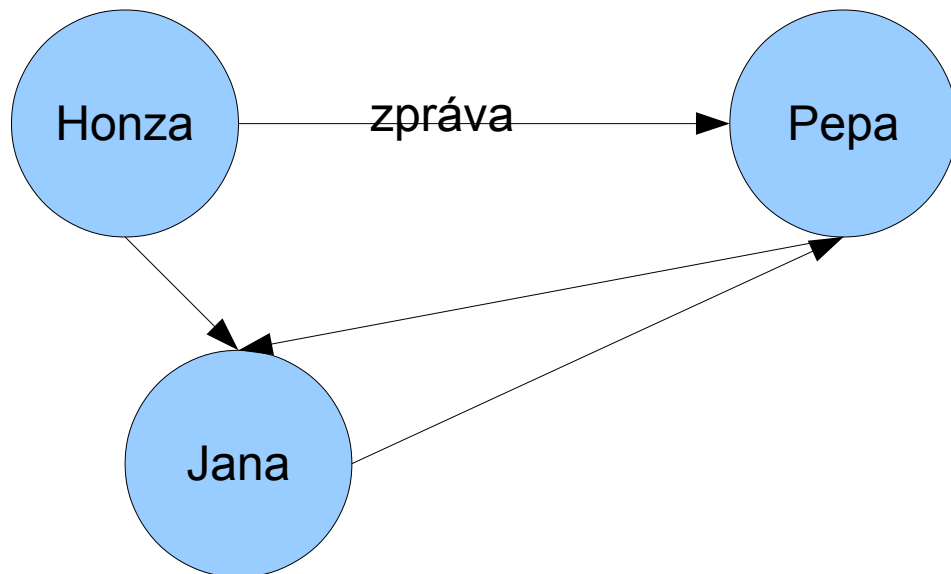
Exception

+-- SystemExit	+-- TypeError
+-- StopIteration	+-- AssertionError
+-- StandardError	+-- LookupError
+-- KeyboardInterrupt	+-- IndexError
+-- ImportError	+-- KeyError
+-- EnvironmentError	+-- ArithmeticError
+-- IOError	+-- OverflowError
+-- OSError	+-- ZeroDivisionError
+-- WindowsError	+-- FloatingPointError
+-- EOFError	+-- ValueError
+-- RuntimeError	+-- UnicodeError
+-- NotImplementedError	+-- UnicodeEncodeError
+-- NameError	+-- UnicodeDecodeError
+-- UnboundLocalError	+-- UnicodeTranslateError
+-- AttributeError	+-- ReferenceError
+-- SyntaxError	+-- SystemError
+-- IndentationError	+-- MemoryError
+-- TabError	+---Warning
	+-- UserWarning
	+-- DeprecationWarning
	+-- PendingDeprecationWarning
	+-- SyntaxWarning
	+-- RuntimeWarning
	+-- FutureWarning

Objektově orientované programování

Úkolem OOP je lépe přiblížit úlohu programování reálnému světu !!!

Objektově orientovaný program je libovolně strukturovaná síť objektů, které spolu navzájem komunikují. To je vše :-)



Objektově orientované programování

Objekty jsou **kompaktní, samostatné entity**, které nesou informace o **svém stavu** (uložené obvykle v tzv. **instančních proměnných**) a implementují svoji **funkcionalitu** (obvykle pomocí tzv. **metod**, což jsou v podstatě klasické procedury, funkce).

Python bohužel **neimplementuje plný objektový princip** s dynamickým **zasíláním zpráv** a **pozdní vazbou** jako **Smalltalk, Ruby, Objective-C**, a tak je jeho objektový model jaksi částečný, podobně jako **Java, C#**, atd.

V Pythonu je vše s čím pracujeme pojmenovaným objektem !!! Čísla, řetězce, různé kolekce, funkce, procedury, moduly, balíčky a další věci jsou objekty konkrétních typů (tříd).

Objektově orientované programování

Pojmy :

- **Třída**
- **Instance**
- **Instanční proměnná, metoda**
- **Atribut (property)**
- **Dědičnost**
- **Polymorfismus**
- **.....**

Objektově orientované programování

class Trida:

"Dokumentační řetězec"

tělo třídy

instance=Trida()

Objektově orientované programování

```
class Clovek:  
    pass
```

```
pepa=Clovek()  
pepa.jmeno="Josef"  
pepa.prijmeni="Novak"
```

```
lojza=Clovek()  
lojza.jmeno="Alois"  
lojza.prijmeni="Novy"
```

Objektově orientované programování

(proměnné instance, metody)

```
class Clovek:
    def tiskni(self):
        print ("Jmeno : %s, vek : %d" % (self.jmeno,self.vek))

pepa=Clovek()
pepa.jmeno="Josef"
pepa.vek=20
pepa.tiskni()
```

Objektově orientované programování

(magické metody)

Python obsahuje velkou množinu **speciálních metod**, které jsou automaticky provedeny, **pokud s objektem provádíme nějakou konkrétní činnost** :

- vytvoření, zánik objektu
- aritmetické operace
- logické operace (porovnávání)
- práce se sekvencemi
- práce s atributy
-

Objektově orientované programování

(magické metody)

```
class Clovek:
    def __init__(self,jmeno="",vek=0):
        self.jmeno=jmeno
        self.vek=vek
    def tiskni(self):
        print ("Jmeno : %s, vek : %d" % (self.jmeno,self.vek))

pepa=Clovek("Josef",20)
pepa.tiskni()
```

Objektově orientované programování

(magické metody)

```
class Clovek:
    def __init__(self,jmeno,vek):
        self.jmeno=jmeno
        self.vek=vek
    def __str__(self):
        return self.jmeno
    def tiskni(self):
        print ("Jmeno : %s, vek : %d" % (self.jmeno,self.vek))

pepa=Clovek("Josef",20)
print(pepa)
```


Objektově orientované programování

(magické metody)

```
class Clovek:
    def __init__(self,jmeno,vek):
        self.jmeno=jmeno
        self.vek=vek
    def __str__(self):
        return(self.jmeno)
    def __gt__(self,other):
        return self.vek>other.vek
    def tiskni(self):
        print ("Jmeno : %s, vek : %d" % (self.jmeno,self.vek))

pepa=Clovek("Josef",20)
lojza=Clovek("Alois",19)
print(pepa>lojza)
```

Objektově orientované programování

(magické metody)

```
class Clovek:
    def __init__(self,jmeno,vek):
        self.jmeno=jmeno
        self.vek=vek
    def __str__(self):
        return(self.jmeno)
    def __gt__(self,other):
        if (self.vek>other.vek):
            return True
        return False
    def __add__(self,other):
        return self.vek+other.vek
    def tiskni(self):
        print ("Jmeno : %s, vek : %d" % (self.jmeno,self.vek))

pepa=Clovek("Josef",20)
lojza=Clovek("Alois",19)
print(pepa+lojza)
```

Objektově orientované programování

(atributy, gettery, settery)

```
class Clovek(object):
    def __init__(self, jmeno, vek):
        self.__jmeno=jmeno
        self.__vek=vek
    def tiskni(self):
        print ("Jmeno : %s, vek : %d" % (self.__jmeno, self.__vek))
    @property
    def jmeno(self):
        return self.__jmeno
    @jmeno.setter
    def jmeno(self, jmeno):
        self.__jmeno=jmeno
```

```
pepa=Clovek("Josef", 20)
```

```
print(pepa.jmeno)
pepa.jmeno="Pepa"
```

Objektově orientované programování

(proměnné třídy)


```
class Clovek:
    Clovek_id=1

    def __init__(self,jmeno,vek):
        self.jmeno=jmeno
        self.vek=vek
        self.cid=Clovek.Clovek_id
        Clovek.Clovek_id+=1

    def tiskni(self):
        print ("Jmeno : %s, vek : %d, id : %d" % (self.jmeno,self.vek,self.cid))

pepa=Clovek("Josef",20)
lojza=Clovek("Alois",19)

pepa.tiskni()
lojza.tiskni()
```



The diagram illustrates how an instance of the `Clovek` class can access the class attribute `Clovek_id`. An arrow points from the text `self.__class__.Clovek_id` to the `Clovek_id` attribute access in the `__init__` method, showing that `self.__class__` refers to the `Clovek` class object.

Objektově orientované programování

(metody třídy)

```
class Clovek:
    Clovek_id=1

    def __init__(self,jmeno,vek):
        self.jmeno=jmeno
        self.vek=vek
        self.cid=Clovek.Clovek_id
        Clovek.Clovek_id+=1

    def resetClovek(cls):
        cls.Clovek_id=1

    resetClovek=classmethod(resetClovek)

    def tiskni(self):
        print ("Jmeno : %s, vek : %d, id : %d" % (self.jmeno,self.vek,self.cid))

pepa=Clovek("Josef",20)
pepa.resetClovek()
lojza=Clovek("Alois",19)

pepa.tiskni()
lojza.tiskni()
```

Objektově orientované programování

(metody třídy)

```
class Clovek:  
    Clovek_id=1
```

```
    def __init__(self,jmeno,vek):  
        self.jmeno=jmeno  
        self.vek=vek  
        self.cid=Clovek.Clovek_id  
        Clovek.Clovek_id+=1
```

@classmethod

```
def resetClovek(cls):  
    cls.Clovek_id=1
```

dekorátor (funkce, která má parametr funkci
a návratová hodnota je opět funkce)

```
    def tiskni(self):  
        print ("Jmeno : %s, vek : %d, id : %d" % (self.jmeno,self.vek,self.cid))
```

```
pepa=Clovek("Josef",20)  
pepa.resetClovek()  
lojza=Clovek("Alois",19)
```

```
pepa.tiskni()  
lojza.tiskni()
```

Objektově orientované programování

(dědičnost)

```
class Clovek:
    def __init__(self,jmeno,vek):
        self.jmeno=jmeno
        self.vek=vek
    def tiskni(self):
        print ("Jmeno : %s, vek : %d" % (self.jmeno,self.vek))
```

```
class Student(Clovek):
    def __init__(self,jmeno,vek,skola):
        Clovek.__init__(self,jmeno,vek)
        self.skola=skola
    def tiskni(self):
        Clovek.tiskni(self)
        print("Skola : %s" % self.skola)
```

```
pepa=Student("Josef",20,"ZS")
pepa.tiskni()
```

Objektově orientované programování

(soukromá/chráněná jména)

```
class trida:  
    def __init__(self):  
        self.x=1  
        self.__y=2  
    def tiskni(self):  
        print (self.x)  
        print (self.__y)
```

```
t=trida()  
t.tiskni()  
print (t.x)  
print (t.__y)
```


Objektově orientované programování ***(introspekce)***

Zjištění zda je konkrétní instance instance třídy :

isinstance(punta, Pes)

Zjištění vazby mezi rodičovskou třídou :

issubclass(Pes, Zvire)

Práce se soubory

soubor=open ("/tmp/soubor.txt","r") ... otevření souboru pro čtení
soubor.close() ... uzavření souboru

Režimy otevření souboru :

r ... čtení

w ... zápis

a ... připojení

Práce se soubory

Metody souborového objektu pro :

čtení : **read()**
readline()
readlines()
xreadlines()

zápis : **write(string)**
writelines(kolekce řetězců)

*Práce se soubory - ošetření IO chyb pomocí **try/finally***

```
try:  
    f = open('/etc/passwd','r')  
    try:  
        for r in f.readlines():  
            print(r)  
  
    finally:  
        f.close()  
  
except IOError:  
    print("Chyba")
```

*Práce se soubory - ošetření IO chyb pomocí **with***

```
try:
    with open('/etc/passwd','r') as f:
        for r in f.readlines():
            print(r.strip())

except IOError:
    print("chyba")
```

Práce se soubory

čtení ze souboru

```
soubor=open ("/etc/passwd","r")
pocet=0
while soubor.readline() != "":
    pocet+=1
soubor.close()

print ("V systému je %d uzivatelu" % pocet)
```

```
soubor=open ("/etc/passwd","r")
radky=soubor.readlines()
for radek in radky:
    print (radek.strip())
soubor.close()
```

Práce se soubory

zápis do souboru

```
soubor=open("/tmp/soubor.txt","w")  
soubor.write("Prvni radek\n")  
soubor.write("Druhy radek\n")  
soubor.close()
```

```
soubor1=open("/etc/passwd","r")  
radky=soubor1.readlines()  
soubor1.close()  
soubor2=open("/tmp/passwd.bak","w")  
soubor2.writelines(radky)  
soubor2.close()
```

Práce se soubory

operační systém

import os

print (os.name) nt, posix

print (os.getcwd()) /home/pepa

print (os.listdir('/tmp')) vrátí seznam souborů v daném adresáři

os.chdir('/tmp') přechod do adresáře

print (os.path.join('home','honza')) home/honza

print (os.path.exists('/tmp')) zjistí existenci souboru

print (os.path.isfile('/etc/passwd')) zjistí zda je soubor ***regular file***

print (os.path.isdir('/etc/passwd')) zjistí zda je soubor ***directory***

Získání uživatelského vstupu

```
a = input("Zadej cislo a : ")  
b = input("Zadej cislo b : ")  
print ("Soucet %d + %d = %d" % (a,b,a+b))
```

Funkce : ***raw_input()***

Standardní vstup, výstup, chybový výstup

V modulu *sys* existují tři speciální souborové objekty :

sys.stdin ... standardní vstup

sys.stdout ... standardní výstup

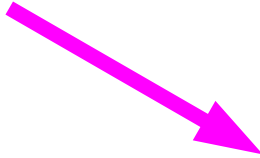
sys.stderr ... standardní chybový výstup

Vstup implementuje metody ***readline***, ***readlines*** a ***xreadlines***

Výstupy implementují metody ***write*** a ***writelines***

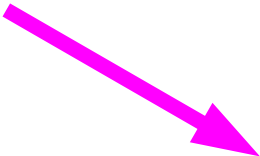
Nakládání objektů do souboru

Uložení do
souboru



```
import pickle
a="Toto je muj textovy retezec"
b=[1,2,4,5,6]
soubor=open("/tmp/stav","wb")
pickle.dump(a,soubor)
pickle.dump(b,soubor)
soubor.close()
```

Získání ze
souboru



```
import pickle
soubor=open("/tmp/stav","rb")
a=pickle.load(soubor)
b=pickle.load(soubor)
soubor.close()
print (a)
print (b)
```

Pozn.: Zcela stejně lze použít modul **JSON**, serializační formát je na rozdíl od **pickle** standardizován.

Modul shelve

```
import shelve
adresar=shelve.open("/tmp/adresy")
adresar["policie"]=["Statni policie","158"]
adresar["hasici"]=["Hasicky sbor","150"]
adresar.close()
```

```
import shelve
adresar=shelve.open("/tmp/adresy")
print (adresar["policie"])
print (adresar["hasici"])
adresar.close()
```

Skripty I.

```
#!/usr/bin/python
```

```
def main():  
    print ("Tak tohle je nas skript !!!")
```

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

Skripty II.

argumenty předávané z příkazového řádku

```
#!/usr/bin/python
```

```
import sys  
def main():  
    print (sys.argv)
```

```
main()
```

Skripty III.

modul getopt

```
#!/usr/bin/python
```

```
import sys, getopt
```

```
def main():
```

```
    (volby, argumenty)=getopt.gnu_getopt(sys.argv[1:], "a:b:c")
```

```
    print (volby)
```

```
    print (argumenty)
```

```
main()
```

```
./pokus.py -a 1 -b 2 -c arg1 arg2
```

Skripty - spuštění externího programu

modul subprocess

```
#!/usr/bin/python
```

```
import sys,getopt,subprocess
```

```
navrat = subprocess.call(["ls","-la","/etc"],shell=True)
```


Skripty - spuštění externího programu

modul subprocess

```
#!/usr/bin/python
```

```
import sys,getopt,subprocess
```

```
proces = subprocess.Popen(["ls","-la"],stdout=subprocess.PIPE,  
stderr=subprocess.STDOUT)
```

```
for a in proces.stdout.readlines():  
    print(a.strip())
```

Paralelní programování

vlákna - modul thread

```
import thread
import time

# Define a function for the thread
def print_time(threadName, delay):
    count = 0
    while count < 5:
        time.sleep(delay)
        count += 1
        print "%s: %s" % (threadName, time.ctime(time.time()))

# Create two threads as follows
try:
    thread.start_new_thread(print_time, ("Thread-1", 2,))
    thread.start_new_thread(print_time, ("Thread-2", 4,))
except:
    print "Error: unable to start thread"

while 1:
    pass
```

Paralelní programování

vlákna - modul threading

```
import threading
import time

exitFlag = 0

class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print "Starting " + self.name
        print_time(self.name, self.counter, 5)
        print "Exiting " + self.name

def print_time(threadName, delay, counter):
    while counter:
        if exitFlag:
            threadName.exit()
        time.sleep(delay)
        print "%s: %s" % (threadName, time.ctime(time.time()))
        counter -= 1

# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# Start new Threads
thread1.start()
thread2.start()

print "Exiting Main Thread"
```

Paralelní programování GIL

- Referenční implementace Pythonu, tedy **CPython**, obsahuje mechanismus **GIL** (Global Interpreter Lock).
- Díky tomuto zámku je virtuálním strojem vykonávaný bytecode prováděn vždy jen v jednom vlákně !!!
- Použití vláken na víceprocesorovém stroji nemá tudíž v CPythonu **žádný smysl**. Jiná situace je ale v **Jythonu**, **PyPy** nebo **IronPythonu**.
- Pro paralelní programování v CPythonu lze poměrně dobře využít modul **multiprocessing**, který nabízí téměř shodné API jako klasické "vláknové" programování.

Paralelní programování

vlákna - modul multiprocessing

```
import os
from multiprocessing import Process

def info(title):
    print(title)
    print('module name:', __name__)
    print('parent process:', os.getppid())
    print('process id:', os.getpid())
    print()

def f(name):
    info('> function f')
    print('hello', name)

if __name__ == '__main__':
    info('> main line')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

Paralelní programování

vlákna - modul multiprocessing

```
import os
from multiprocessing import Process
import time

def f():
    print(os.getpid(), ": zacatek...")
    time.sleep(os.getpid() % 7)
    print(os.getpid(), ": trvalo mi to", (os.getpid() % 7), "s.")

if __name__ == '__main__':
    for i in range(7):
        p = Process(target=f, args=())
        p.start()
```

Paralelní programování

vlákna - modul multiprocessing

```
import os
from multiprocessing import Process, Lock
import time

def f(l):
    print(os.getpid(), ": zacatek ...")
    l.acquire()
    time.sleep(os.getpid() % 7)
    l.release()
    print(os.getpid(), ": trvalo mi to", (os.getpid() % 7), "s.")

if __name__ == '__main__':
    lock = Lock()
    for i in range(7):
        Process(target=f, args=(lock,)).start()
```

Paralelní programování

vlákna - modul multiprocessing

```
from multiprocessing import Process, Lock, Value

def f(l, x):
    l.acquire()
    x.value += 1
    print(x.value)
    l.release()

if __name__ == '__main__':
    lock = Lock()
    n = Value('d', 0)
    sez = list()

    for i in range(10):
        sez.append(Process(target=f, args=(lock, n)))
        sez[i].start()

    for i in range(10):
        sez[i].join()

    print("Výsledná hodnota n:", n.value)
```


Sít'ová komunikace

Python obsahuje komplexní podporu pro sít'ování od socketů po klientské implementace řady běžných aplikačních protokolů ve vestavěných modulech :

- **socket**
- **http**lib
- **ftplib**
- **urllib**
- **smtp**lib
- **nntplib**
- **pop**lib
- **imap**lib
-

HTTP klient

```
import urllib
```

```
try:
```

```
    seznam=urllib.urlopen("http://www.seznam.cz/")
```

```
    try:
```

```
        for radek in seznam.readlines():
```

```
            print (radek.strip())
```

```
    finally:
```

```
        seznam.close()
```

```
except:
```

```
    print("Chyba !!!!")
```

SMTP klient

```
import smtplib
```

```
zprava="Subject: Dnesni zprava\r\n\r\nTelo zpravy."
```

```
server=smtplib.SMTP("10.2.1.6")
```

```
server.sendmail("vrbata@gopas.cz","honza@vrbata.cz",zprava)
```

```
server.quit()
```

Komunikace klient-server pomocí HTTP

serverová část

```
import BaseHTTPServer,datetime
```

```
class odpoved(BaseHTTPServer.BaseHTTPRequestHandler):
```

```
    def do_GET(self):
```

```
        self.send_response(200)
```

```
        self.send_header("Content-type","text/html")
```

```
        self.end_headers()
```

```
        html = """<html><body>
```

```
        <h1>Vita Vas Honzuv server !!!!</h1>
```

```
        <h3>Je prave : %s</h3>
```

```
        </body></html>""" % datetime.datetime.now()
```

```
        self.wfile.write(html)
```

```
server = BaseHTTPServer.HTTPServer(("",80),odpoved)
```

```
server.serve_forever()
```

Komunikace klient-server pomocí HTTP

klientská část

```
import httplib
```

```
def request ():  
    c=httplib.HTTP("localhost:8000")  
    c.putrequest("GET","/index.html")  
    c.putheader("Data","Tohle jsou vstupni data")  
    c.endheaders()  
    errcode,errmsg,headers = c.getreply()  
    telo=c.getfile()  
    print (headers['Franta'])  
    return (errcode)
```

```
request()
```

Vzdálené volání procedur pomocí XML-RPC

XML-RPC požadavek

```
import xmlrpclib  
xmlrpclib.ServerProxy('http://sortserver/RPC').searchsort.sortList([10, 2], True)
```

```
<?xml version='1.0'?>  
<methodCall>  
  <methodName>searchsort.sortList</methodName>  
  <params>  
    <param>  
      <value>  
        <array>  
          <data>  
            <value><i4>10</i4></value>  
            <value><i4>2</i4></value>  
          </data>  
        </array>  
      </param>  
      <param><value><boolean>1</boolean></value></param>  
    </params>  
  </methodCall>
```

Vzdálené volání procedur pomocí XML-RPC

XML-RPC odpověď

```
<?xml version='1.0'?>
<methodResponse>
  <params>
    <param>
      <value>
        <array>
          <data>
            <value><i4>2</i4></value>
            <value><i4>10</i4></value>
          </data>
        </array>
      </value>
    </param>
  </params>
</methodResponse>
```

Vzdálené volání procedur pomocí XML-RPC

serverová část

```
import SimpleXMLRPCServer
```

```
def soucet(a,b):  
    return a+b
```

```
def rozdil(a,b):  
    return a-b
```

```
server = SimpleXMLRPCServer.SimpleXMLRPCServer(("",8080))
```

```
server.register_function(soucet)  
server.register_function(rozdil)
```

```
server.serve_forever()
```


Vzdálené volání procedur pomocí XML-RPC

klientská část

```
import xmlrpclib
```

```
server = xmlrpclib.ServerProxy("http://10.2.20.143:8080/")
```

```
v = server.soucet(2,3)
```

```
print(v)
```

Práce s XML

- **Značkovací jazyk** pro popis **hierarchických strukturovaných dat**.
- XML dokument obsahuje jeden nebo více **elementů** ohraničených počátečními a koncovými značkami.
- XML dokument :

```
<dokument>  
</dokument>
```

Práce s XML

- Elementy lze **zanořovat** do libovolné hloubky.
- Prvnímu elementu se říká **kořenový element** (root element).
- Elementy mohou mít **atributy**, dvojice jméno-hodnota.

```
<dokument jazyk="cesky"></dokument>
```
- Uvnitř jednoho elementu se atributy **nesmí opakovat**.
- Hodnoty atributů musí být uzavřeny v uvozovkách nebo apostrofech.

Práce s XML

- Pokud je v jednom elementu více atributů, pak na jejich **pořadí nezáleží**.
- Počet atributů u elementů není nijak omezen.
- Elementy mohou obsahovat text :

<dokument>Toto je text</dokument>

- Prázdné elementy lze zapisovat zkráceně :

<dokument/>

Práce s XML

- Python nabízí několik možností jak zpracovat XML.
- Lze použít tradiční parseery **DOM** a **SAX**, nebo knihovnu **ElementTree** pro ještě jednodušší zpracování.

Práce s XML - zpracování RSS

```
import xml.etree.ElementTree as etree
```

```
tree = etree.parse("rss.xml")  
root = tree.getroot()
```

```
channel = root.find("channel")
```

```
print (channel.tag)  
print (channel.attrib)  
print (channel.text)
```

```
items = channel.findall("item")
```

```
for item in items:  
    title = item.find("title")  
    link = item.find("link")  
  
    print (title.text)  
    print (link.text+"\n")
```

Práce s XML - zpracování RSS

```
import xml.etree.ElementTree as etree
import urllib

rss = urllib.urlopen("http://servis.idnes.cz/rss.aspx?c=zpravodaj")
tree = etree.parse(rss)
root = tree.getroot()

channel = root.find("channel")

print (channel.tag)
print (channel.attrib)
print (channel.text)

items = channel.findall("item")

for item in items:
    title = item.find("title")
    link = item.find("link")

    print (title.text)
    print (link.text+"\n")
```

Grafické uživatelské rozhraní GUI

1) dialog, cdialog, xdialog

2) wxPython

3) Tkinter

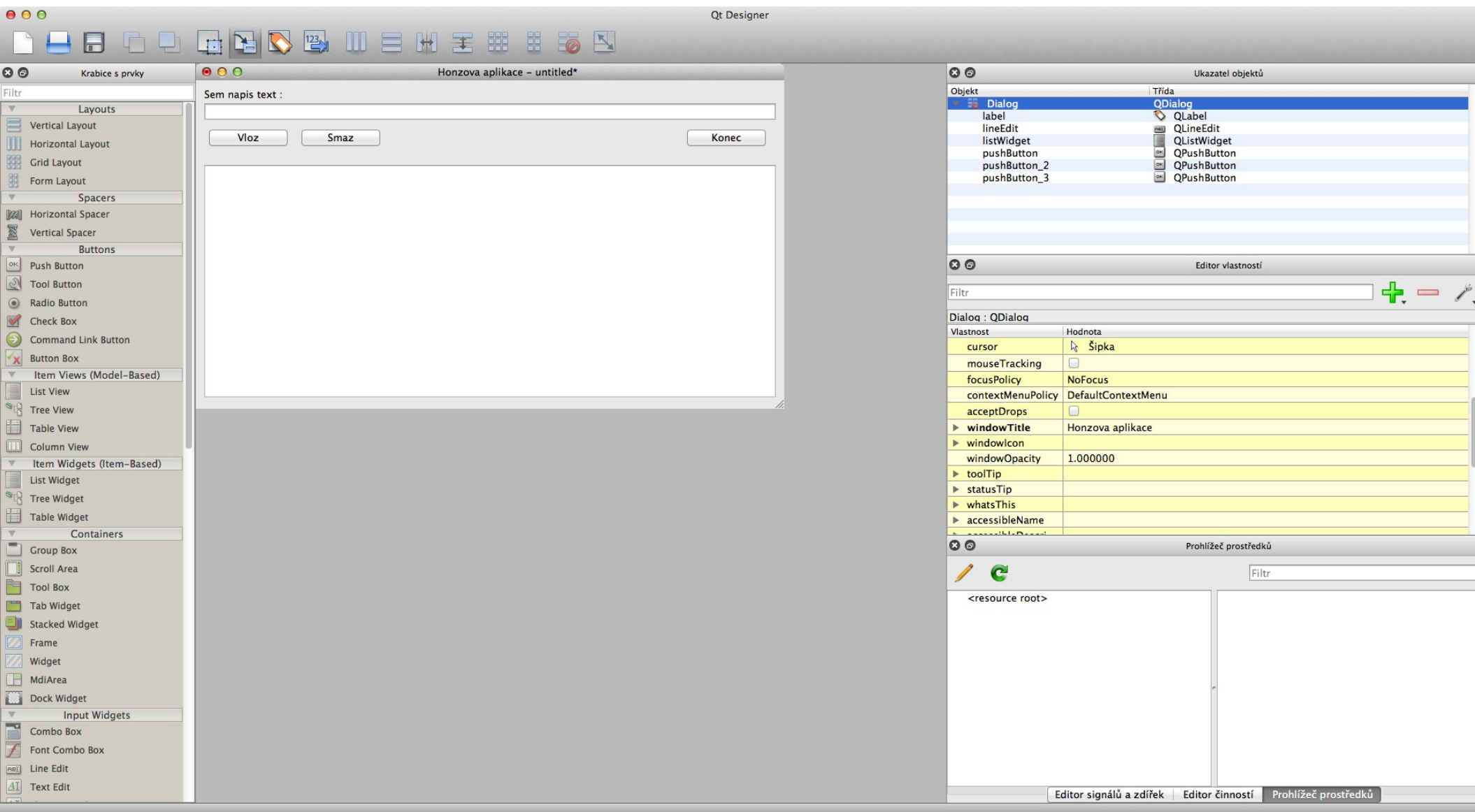
4) PyGTK

5) PyQT !!!!!

6) PyGame

Grafické uživatelské rozhraní GUI

Qt a PyQt – Qt Designer



Grafické uživatelské rozhraní GUI

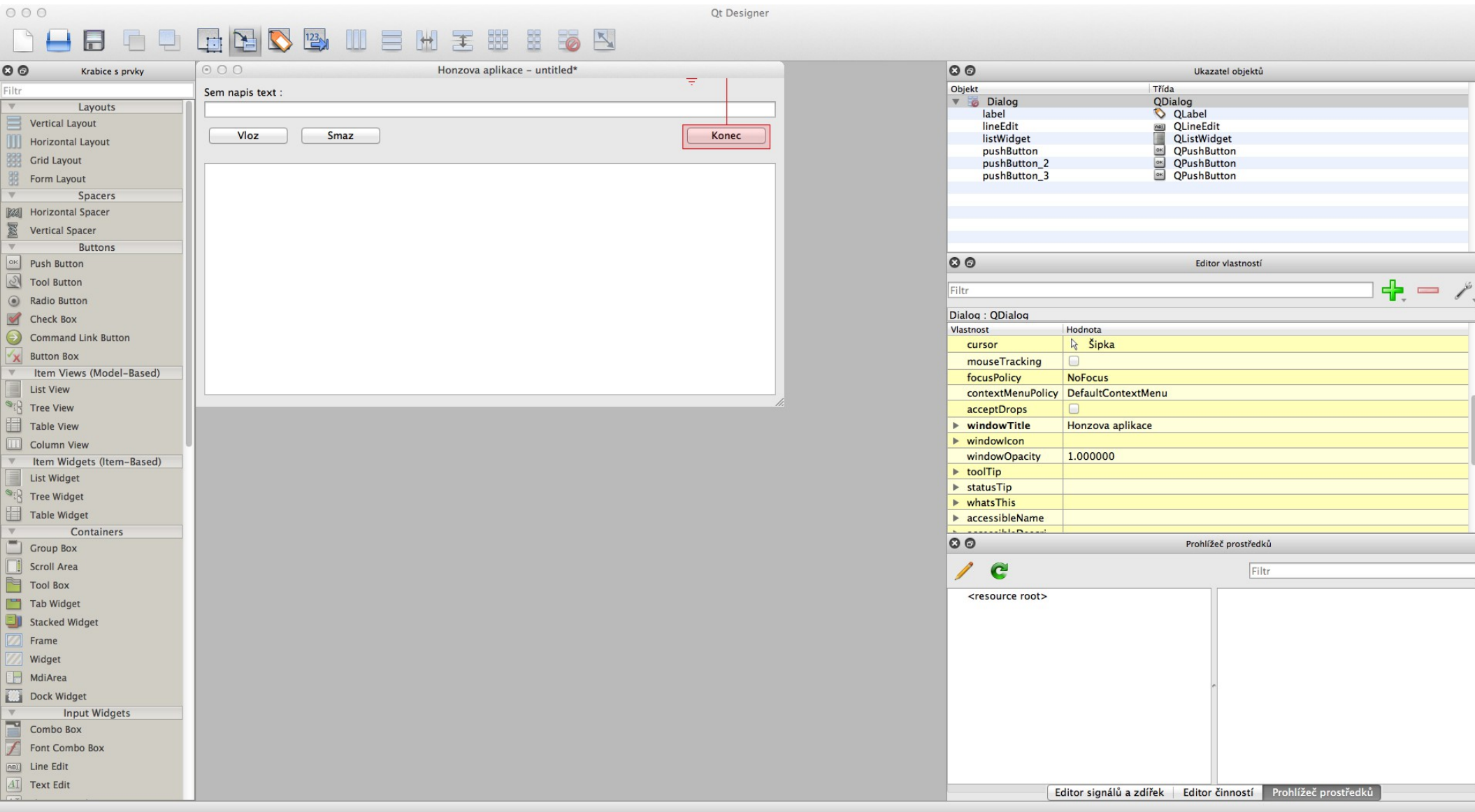
Qt a PyQt – Qt Designer

Princip fungování Qt grafické aplikace :

- každý widget (obrazový prvek) generuje při událostech různé **signály** a poskytuje určitou množinu **slotů**, které mohou jiné signály volat.
- je možné vytvářet zcela nové uživatelské sloty.
- **propojením signálů se sloty** a naprogramováním chování se vytváří grafická aplikace.

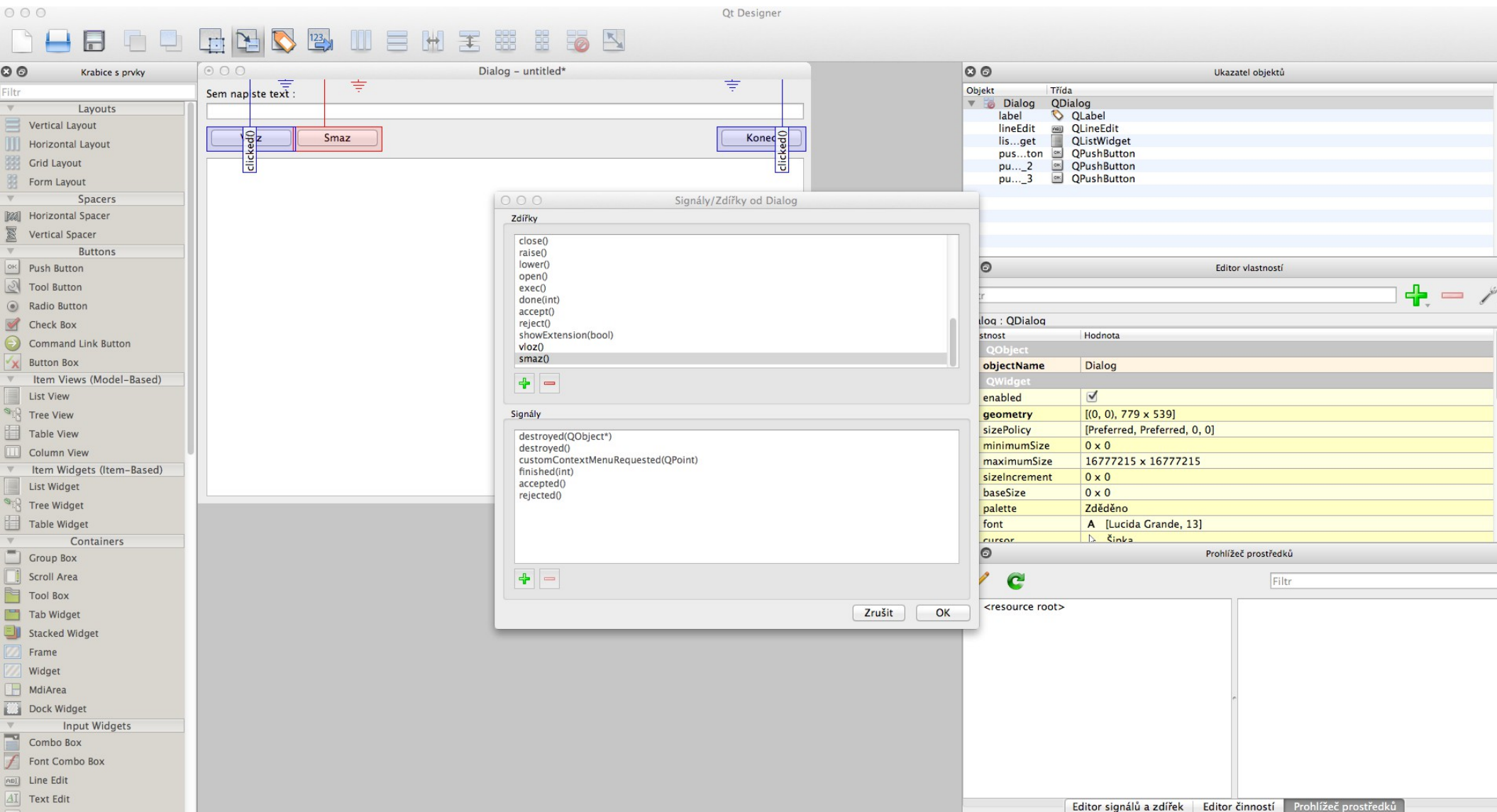
Grafické uživatelské rozhraní GUI

Qt a PyQt – Qt Designer



Grafické uživatelské rozhraní GUI

Qt a PyQt – Qt Designer



Grafické uživatelské rozhraní GUI

Qt a PyQt

- Výstupem z QtDesigneru je XML (.ui) soubor s popisem grafického rozvržení aplikace, interakcí mezi **signály** a **sloty**, atd.
- S tímto souborem je možné v zásadě nakládat dvěma způsoby :
 - **Zkompilovat** nástrojem pyuic XML UI soubor do zdrojového kódu Pythonu a ten dále používat.
pyuic form1.ui > form1.py
 - **UI soubor bude součástí aplikace** a bude zpracován až při jejím spuštění.
- Dále je třeba doprogramovat obsluhu uživatelských slotů a vytvořit kostru Qt aplikace.

Grafické uživatelské rozhraní GUI

Qt a PyQt

```
import sys
from PyQt4 import QtGui, uic

class MyDialog(QtGui.QDialog):
    def __init__(self):
        QtGui.QDialog.__init__(self)
        uic.loadUi("aplikace.ui", self)

    def vloz(self):
        text = self.lineEdit.text()
        self.listWidget.addItem(text)
        self.lineEdit.clear()

    def smaz(self):
        radek = self.listWidget.currentRow()
        self.listWidget.takeItem(radek)

app = QtGui.QApplication(sys.argv)
dialog = MyDialog()
dialog.show()
app.exec_()
```

Přístup k databázím

- Python obsahuje jednotné rozhraní pro přístup k databázím **DB API 2**.
- Díky tomuto je možné jednotně přistupovat k různým typům databází.
- Jako příklad vytvoříme SQLite databázi s jednou tabulkou, seznamem telefonních čísel.

Přístup k databázím

```
import sqlite3
```

```
conn=sqlite3.connect("seznam.sqlite")
```

```
cursor=conn.cursor()
```

```
cursor.execute("select * from seznam")
```

```
for zaznam in cursor.fetchall():
```

```
    print("Jmeno : %s, cislo : %s" %(zaznam[0],zaznam[1]))
```

```
conn.close()
```


Přístup k databázím

```
import sqlite3
```

```
conn=sqlite3.connect("seznam.sqlite")
```

```
cursor1=conn.cursor()
```

```
cursor2=conn.cursor()
```

```
cursor1.execute("insert into seznam values ('Hasici','155')")
```

```
conn.commit()
```

```
cursor2.execute("select * from seznam")
```

```
for zaznam in cursor2.fetchall():
```

```
    print("Jmeno : %s, cislo : %s" %(zaznam[0],zaznam[1]))
```

```
conn.close()
```

PyPI

- **Python Package Index** je repozitář doplňkového software pro Python.
- V současné době obsahuje více jak 70000 balíčků.
- Management nástroj *pip*.

Virtualenv

- Pomocí nástroje *virtualenv* je možné vytvářet izolované instalace Pythonu.

virtualenv adresar

- Zde je možné instalovat moduly a balíčky, mimo hlavní systémovou instalaci Pythonu.

Distribuce programu

- Jelikož je Python **interpretovaný jazyk**, je nutné aby měl uživatel našeho programu instalován interpret Pythonu v příslušné verzi.
- Pokud náš program využívá různé doplňky třetích stran, instalované z repozitáře **PyPi** nebo odjinud, je nutné instalovat i tyto.
- To mohou být poměrně rozsáhlé požadavky pro spuštění našeho programu.
- Praktickou cestou je tzv. "**zmrazení**", tedy **zabalení** interpretu Pythonu, všech potřebných balíčků, modulů a našeho programu do jednoho samospustitelného celku.

Distribuce programu - freezery

- K vlastnímu freezingu je možné použít nástroje pomocí kterých lze tuto operaci provádět víceméně automaticky.
- Oblíbené freezery : **cx_Freeze**, **PyInstaller**, **py2exe**, ...