5. Praktické aspekty programování. Objekty.

BAB37ZPR – Základy programování

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Přehled témat

- Část 1 – Praktické aspekty programování
  - Číselné typy
  - Globální a lokální proměnné
  - Funkce
  - Ošetření chyb
  - Soubory
- Část 2 – Objekty
- Část 3 – Abstraktní datové typy
  - Datový typ seznam
  - Zásobník

Část I

Praktické aspekty programování

I. Praktické aspekty programování

Číselné typy

- `int` – celá čísla
- `float`
  - floating-point number
  - čísla s plovoucí desetinnou čárkou
  - reprezentace: mantisa $\times$ báze ^{exponent}
  - nepřesnosti, zaokrouhlování
- `complex` – komplexní čísla

Nepřesnosti

$$((1 + \frac{1}{x}) - 1) * x = 1$$

```
>>> x = 2**50
>>> ((1 + 1 / x) - 1) * x
1.0
>>> x = 2**100
>>> ((1 + 1 / x) - 1) * x
0.0
```

Číselné typy – poznámky

- Explicitní přetypování: `int(x)`, `float(x)`
- Automatické nafukování typu `int`
  - viz např. `2**100`
  - pomalejší, ale korektní
  - v ostatních programovacích jazycích zpravidla dochází k přetečení

Kvíz

```
1 n = 1
2 while n > 0:
3   print(n)
4   n = n / 10
5 print("done")
```

- Co udělá program?
  - Co když změníme výraz na `n=n*10`
  - Co když změníme výraz na `n=n*10.0`
- Jaký bude výsledek programu v jiných jazycích?

Příklad – ciferný součet

- vstup: číslo `x`
- výstup: ciferný součet čísla `x`
- příklady: $8 \to 8$, $15 \to 6$, $297 \to 18$

Jak na to?

- opakovaně provádíme:
  - dělení 10 se zbytkem – hodnota poslední cifry
  - celočíselné dělení – okrajování čísla

Naivní řešení

```
1    if n % 10 == 0:
2      f = 0 + f
3    elif n % 10 == 1:
4      f = 1 + f
5    elif n % 10 == 2:
6      f = 2 + f
```

Příklad – ciferný součet – řešení

```
1  def digit_sum(n):
2    result = 0
3    while n > 0:
4      result += n % 10
5      n = n // 10
6    return result
```

Pro zajímavost

```
1  def digit_sum(n):
2    return sum(map(int, str(n)))
```

Připomenutí

- `print` – výpis hodnoty
- `return` – návratová hodnota funkce, se kterou lze dále pracovat

Příklad – Collatzova posloupnost

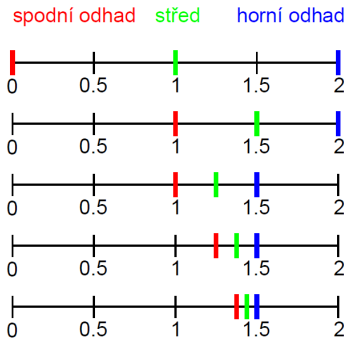
- vezmi přirozené číslo:
  - pokud je sudé, vyděl jej dvěma
  - pokud je liché, vynásob jej třemi a přičti jedničku
- tento postup opakuj, dokud nedostaneš číslo jedna

Řešení

```
1  def collatz_sequence(n):
2    while n != 1:
3      print(n, end=", ")
4      if n % 2 == 0:
5        n = n // 2
6      else:
7        n = 3*n + 1
8    print(1)
```

Výpočet odmocniny

- Vstup: číslo `x`
- Výstup: odhad $\sqrt{(x)}$
- Existuje mnoho metod, ukázka metody binárního půlení



Výpočet odmocniny – binární půlení

```
1  def square_root(x, precision=0.01):
2    upper = x
3    lower = 0
4    middle = (upper + lower) / 2
5    while abs(middle**2 - x) > precision:
6      print(lower, upper, sep="\t")
7      if middle**2 > x:
8        upper = middle
9      if middle**2 < x:
10       lower = middle
11     middle = (upper + lower) / 2
12   return middle
```

Výpočet odmocniny – binární půlení

Drobný problém: program není korektní
Kde je chyba?

- Funguje korektně jen pro čísla $x \geq 1$
- Co program udělá pro čísla $x < 1$
- Proč?
- Jak to opravit?

I. Praktické aspekty programování

Globální a lokální proměnné

Globální proměnné

- definovány globálně (tj. ne uvnitř funkce)
- jsou viditelné kdekoli v programu

Lokální proměnné

- definovány uvnitř funkce
- jsou viditelné jen ve své funkci

Rozsah proměnných obecněji

- proměnné jsou viditelné v rámci svého **rozsahu**
- rozsahem může být
  - funkce
  - moduly (soubory se zdrojovým kódem)
  - třídy
  - a jiné (záleží na konkrétním jazyce)
- terminologie: `namespace`, `scope`, ...

Příklad – globální a lokální proměnná

```
1  a = "This is global."
3  def example1():
4    b = "This is local."
5    print(a)
6    print(b)
8  example1() # This is global.
9             # This is local.
11 print(a) # This is global.
12 print(b) # ERROR!
14 # NameError:  name 'b' is not defined
```

Příklad – zastínění globální proměnné

```
1  a = "Think global."
3  def example2():
4    a = "Act local."
5    print(a)
7  print(a) # Think global.
8  example2() # Act local.
9  print(a) # Think global.
```

Příklad – změna globální proměnné

```
1  a = "Think global."
3  def example3():
4    global a
5    a = "Act local."
6    print(a)
8  print(a) # Think global.
9  example3() # Act local.
10 print(a) # Act local.
```

Lokální proměnné – deklarace

- lokální proměnná vzniká, pokud je přiřazení kdekoliv uvnitř těla funkce

```
1  a = "Think global."
3  def example4(change_opinion=False):
4    print(a)
5    if change_opinion:
6      a = "Act local."
7      print("Changed opinion:", a)
9  print(a) # Think global.
10 example4() # ERROR!
```

Rozsah proměnných – cyklus `for`

- Rozsah proměnné v Pythonu není pro dílčí blok kódu, ale pro celou funkci (resp. globální kód)
- Častá chyba: řídicí proměnná `for` cyklu použita po ukončení cyklu

```
1  n = 9
2
3  for i in range(n):
4    print(i)
6  if i % 2 == 0:
7    print("I like even length lists")
```

Slovník proměnných

- proměnné jsou uloženy ve slovniku
- výpis: `globals()`, `locals()`

```
1  def function():
2    x = 100
3    s = "dog"
5  print(locals())
7  a = [1, 2, 3]
8  x = 200
9    function()
10
11 print(globals())
```

Závěry

Doporučení

- spíše se vyhýbat globálním proměnným
- omezit na specifické případy, např. globální konstanty

Proč?

- horší čitelnost kódu
- náročnější testování, možný zdroj chyb

Obecně: lokalita kódu je užitečná

Alternativy

- předávání parametrů funkcím a využití návratových hodnot
- objekty

I. Praktické aspekty programování

Funkce a vedlejší efekty

Čistá funkce

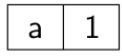
- funkce bez vedlejších efektů
- výstup závisí jen na vstupních parametrech

Vedlejší efekty

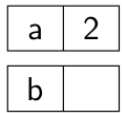
- změna měnitelných parametrů
  - OK, ale nemíchat s návratovou hodnotou, vhodně pojmenovat, dokumentovat
- změna globálních proměnných (které nejsou parametry)
  - většinou cesta do pekla
- změna stavu systému (libovolné výpisy, zápis do souboru, databáze, ...)
  - nutnost, ale nemíchat chaoticky s výpočty

Proměnné a pamět

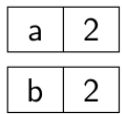
```
int a, b;
a = 1;
```




```
a = 2;
```




```
b = a;
```

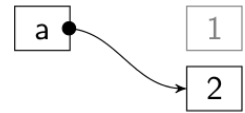


Jazyk C

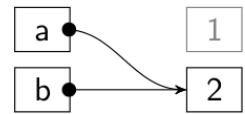
Proměnné jako hodnoty

```
a = 1;
```


```
a = 2;
```




```
b = a;
```



Jazyk Python

Proměnné jako odkazy

Identita a rovnost

- Funkce `id()` – vrací identitu objektu (adresa v paměti)

```
a = 1000
b = a
print(a, b)
print(id(a), id(b))
b += 1
print(a, b)
print(id(a), id(b))
```

```
a = [1]
b = a
print(a, b)
print(id(a), id(b))
b.append(2)
print(a, b)
print(id(a), id(b))
```

- Operátor `is` – stejná identita

```
a = [1, 2, 3]
b = [1, 2, 3]
print(a == b) # True
print(id(a) == id(b)) # False

```

Předávání parametrů funkcím 1/3

- hodnotou (call by value)
  - předá se hodnota proměnné (kopie)
  - standardní v C, C++, apod.
- odkazem (call by reference)
  - předá se odkaz na proměnnou
  - lze použít v C++
- jiné možnosti (jménem, hodnotou-výsledkem, ...)
- jazyk Python: něco mezi voláním hodnotou a odkazem
  - podobně funguje např. Java
  - někdy nazýváno `call by object sharing`

Předávání parametrů funkcím 2/3

Předávání parametrů hodnotou

- parametr je vlastně lokální proměnná
- funkce má svou vlastní lokální kopii předané hodnoty
- funkce nemůže měnit hodnotu předané proměnné

Předávání parametrů odkazem

- nepředává se hodnota, ale odkaz na proměnnou
- změny parametru jsou ve skutečnosti změny předané proměnné

Předávání parametrů funkcím 3/3

Předávání parametrů v Pythonu

- parametr drží odkaz na předanou proměnnou
- změna parametru změní i předanou proměnnou
- pro **neměnitelné typy** tedy v podstatě funguje jako předávání hodnotou
  - čísla, řetězce, n-tice (tuples)
- pro **měnitelné typy** jako předávání odkazem
  - pozor: přiřazení znamená změnu odkazu

Připomenutí

- neměnitelné typy: `int`, `str`, `tuple`, ...
- měnitelné typy: `list`, `dict`, ...

Předávání parametrů funkcím – příklad 1/4

- číselný parametr je neměnitelný, nestane se nic

```
1  def update_param_int(x):
2    x = x + 1
4  a = 1
5  print(a)
6  update_param_int(a)
7  print(a)
```

```
$ python update_param_int
1
1
```

Předávání parametrů funkcím – příklad 2/4

- seznam je měnitelný, změna se projeví i mimo funkci

```
1  def update_param_list(x):
2    x.append(3)
3
4  a = [1, 2]
5  print(a)
6  update_param_list(a)
7  print(a)
```

```
$ python update_param_list
[1, 2]
[1, 2, 3]
```

Předávání parametrů funkcím – příklad 3/4

- odkaz se změní na nový seznam, původní je nezměněn

```
1  def change_param_list(x):
2    x = [1, 2, 3]
3
4  a = [1, 2]
5  print(a)
6  change_param_list(a)
7  print(a)
```

```
$ python change_param_list
[1, 2]
[1, 2]
```

Předávání parametrů funkcím – příklad 4/4

- kvíz

```
1  def test(s):
2    s.append(3)
3    s = [42, 17]
4    s.append(9)
5    print(s)
6
7  t = [1, 2]
8  test(t)
9  print(t)
```

Práce s parametry

```
1  def change_list(alist, value):
2    alist.append(value)
4  def return_new_list(alist, value):
5    newlist = alist[:]
6    newlist.append(value)
7    return newlist
```

Práce s parametry

- operátor `+=` – různé chování pro neměnné typy a seznamy

```
1  def increment(x):
2    print(x, id(x))
3    x += 1
4    print(x, id(x))
6  p = 42
7  increment(p)
8  print(p, id(p))
```

lec05/increment.py

```
42 1906340608
43 1906340640
42 1906340608
```

```
1  def add_to_list(s):
2    print(s, id(s))
3    s += [1]
4    print(s, id(s))
6  t = [1, 2]
7  add_to_list(t)
8  print(t, id(t))
```

lec05/add_to_list.py

```
[1, 2] 2657138746184
[1, 2, 1] 2657138746184
[1, 2, 1] 2657138746184
```

Práce s parametry

- pozor na rozdíl mezi = a += u seznamů

```python
def add_to_list1(s):
  print(s, id(s))
  s += [1]
  print(s, id(s))

t = [1, 2]
add_to_list1(t)
print(t)
```

lec05/add_to_list1.py

```
[1, 2] 2319764595464
[1, 2, 1] 2319764595464
[1, 2, 1]
```

```python
def add_to_list2(s):
  print(s, id(s))
  s = s + [1]
  print(s, id(s))

t = [1, 2]
add_to_list2(t)
print(t)
```

lec05/add_to_list2.py

```
[1, 2] 1528528741192
[1, 2, 1] 1528528823752
[1, 2]
```

I. Praktické aspekty programování

Blok `try` – `except`

- Ošetření se v Pythonu provádí pomocí bloku `try` – `except`

```
try:
  # blok prikazu
except jmeno_prvni_vyjimky:
  # blok prikazu
except jmeno_dalsi_vyjimky:
  # blok příkazů
# zde je bud konec, nebo zachyceni dalsich vyjimek
```

- Pokud chceme zachytit i chybovou zprávu, musíme napsat:

```
except jmeno_vyjimky as chyba:
  # text vyjimky se ulozi do promenne chyba
```

Výjimky

Základní výjimky v Pythonu

- **SyntaxError** – chyba ve zdrojovém kódu
- **ZeroDivisionError** – dělení nulou
- **TypeError** – nesprávné použití datových typů, např. sčítání řetězce a čísla apod.
- **ValueError** – nesprávná hodnota

Více o výjimkách: `https://docs.python.org/3/library/exceptions.html`

```
(x,y) = (5,0)
try:
  z = x/y
except ZeroDivisionError:
  print("divide by zero")
```

```
import sys
try:
  z = 5/0
except:
  print(sys.exc_info()[0])
```

Příklad

```
1  def nacti_cislo ():
2    spatne = True
3    while spatne:
4      try:
5        cislo = float(input("float: "))
6        spatne = False
7      except ValueError as err:
8        print(err)
9      else:
10       return cislo
12 print(nacti_cislo())
```

```
float: ahoj
could not convert string to float: 'ahoj'
float: 3.14
3.14
```

I. Praktické aspekty programování

Práce se soubory

Proč?

- Vstupní data
- Uložení výstupu programu
- Udržovaní stavu programu mezi jednotlivými běhy
- Větší projekty: databáze

Základní operace

- Otevření souboru
- Práce se souborem (čtení, zápis)
- Zavření souboru

Práce se soubory – otevření a uzavření souboru

- `f = open(filename, mode)`
- jméno souboru: řetězec
- způsob otevření:
  - čtení – `"r"`
  - zápis – `"w"` – přepíše soubor, pokud ještě neexistuje, vytvoří jej
  - přidání na konec – `"a"`
  - další možnosti: čtení i zápis, binární režim
- uzavření: `f.close()`

Práce se soubory – čtení a zápis do souboru

- `f.write(text)` – zapíše řetězec do souboru
  - raw výstup – neukončuje řádky, je třeba explicitně použít `'\n'`
- `f.readline()` – přečte jeden řádek
- `f.readlines()` – přečte všechny řádky, vrací seznam řádků
- `f.read(count)` – přečte daný počet znaků
- `f.read()` – přečte celý soubor, vrací řetězec
- `f.tell()` – aktuální pozice v souboru
- `f.seek(position)` – přesun pozice v souboru

Práce se soubory – iterování po řádcích

- Intuitivní způsob

```
for line in f.readlines():
  print(line)

```

- Alternativní způsob

```
for line in my_file:
  print(line)

```

- Načtení pole ze souboru

```
f=open('line.txt','r')
line = f.readline()
pole = list(map(int, line.split()))

```

Práce se soubory – `with`

Speciální blok `with`

- není třeba soubor zavírat – uzavře se automaticky po ukončení bloku
- souvislost s výjimkami

```
with open("/tmp/my_file", "r") as my_file:
  lines = my_file.readlines()
print(lines)
```

Část II

Objekty

Záznam (Record)

Záznam (obecně)

- strukturovaný/složený datový typ
- obsahuje položky *(fields)* / prvky *(elements)*/ členy
- položek je (obvykle) pevný počet, mají každá svůj typ
- položky jsou identifikované jménem
- typ záznamu má své jméno

Příklady

- Datum = rok, měsíc, den
- Osoba = jméno, příjmení, datum narození
- Adresa = jméno ulice, číslo popisné, město, PSČ
- Bod = x, y
- Kruh = střed, poloměr

Abstrakce

- **funkční abstrakce** *(function abstraction)*
  - Klient ví, co funkce dělá. *(rozhraní, interface)*
  - Klient nemusí vědět, jak je funkce implementována.
- **datová abstrakce** *(data abstraction)*
  - Klient ví, co datový typ představuje, jak ho vytvořit a jaké operace podporuje.
  - Klient nemusí vědět, jaká je vnitřní struktura.
- Klient musí znát rozhraní *(interface)*.
- Implementace je skrytá, lze ji změnit.
- Implementace je rozdělena mezi klientský kód a knihovny (např. ve formě modulů).
- Znovupoužitelnost kódu, možnost nezávislého vývoje.
- Zjednodušení psaní klientského kódu.

Záznam v Pythonu

V Pythonu záznamy nejsou..., ale jsou (dynamické) **objekty**.

objekt = *záznam* + *metody*

Objekt

- Obsahuje **datové položky** / **atributy** / **proměnné**
- Může obsahovat **metody** – funkce pracující s datovými položkami
- Strukturu definuje **třída**, **objekty** jsou instance třídy.
- Základ **objektově orientovaného programování – OOP)**.

Varování

Toto není objektově orientované programování.

OOP je významně komplexnější, než je prezentováno v tomto předmětu.

Slovníček pojmů

třída	obecný uživatelsky definový typ, charakterizován atributy
objekt	konktétní instance třídy
atribut	vlastnost konkrétního objektu
metoda	funkce, která je vázaná na danou třídu
konstruktor	inicializační metoda, vytváří objekt

Intuitivní ilustrace

- třída: Pes
- instance: Alík
- datové atributy: rasa, jméno, věk, poloha
- metody: štěkej, sedni, lehni, popoběhni

Objekty v Pythonu

- Definice třídy:

```
1  class Osoba:
2    pass # prázdná třída
```

- Vytvoření objektu:

```
o=Osoba()
```

- Přidání a použití datových položek (atributů):

```
1  o.jmeno="Karel"
2  o.prijmeni="Novak"
4  print(o.jmeno+" "+o.prijmeni)
```

```
Karel Novák
```

- V Pythonu jsou datové položky přidávané *dynamicky.*
- K datovým položkám přistupujeme pomocí *tečkové notace.*

Metody

- Metody jsou funkce definované uvnitř třídy.

```
class Osoba:
  def print(self):
    print(self.jmeno+" "+self.prijmeni)
o=Osoba()
o.jmeno="Karel"
o.prijmeni="Novak"
```

- Pracují s konkrétním objektem.
- Mají přístup k proměnným objektu pomocí prvního parametru.

self není klíčovým slovem, jen silnou konvencí.

- Voláme je na objekt tečkovou notací.

```
o.print()
```

```
Karel Novak
```

Vytvoření objektu

- Objekt s konstruktorem:

```
class Osoba:
  def __init__(self,jmeno,prijmeni):
  self.jmeno=jmeno
  self.prijmeni=prijmeni

  def print(self):
    print(self.jmeno+" "+self.prijmeni)
```

- Konstruktor má speciální jméno `__init__`.
- Je volán při vytvoření objektu.

```
o=Osoba("Karel","Novak")
o.print()
```

```
Karel Novak
```

Přístup k atributům

Přímý

- přistupujeme přímo k atributu, můžeme ho přímo měnit
- `person.name`

Nepřímý

- pomocí metod (getters and setters)
- `person.get_name()`, `person.set_name()`

Který zvolit?

- závisí na použití
- objekt jen pro udržení dat – přímý přístup OK
- skrytí implementace (zapouzdření) – metody

Přístup k atributům

Přímý

- přistupujeme přímo k atributu, můžeme ho přímo měnit
- `person.name`

Nepřímý

- pomocí metod (getters and setters)
- `person.get_name()`, `person.set_name()`

Který zvolit?

- závisí na použití
- objekt jen pro udržení dat – přímý přístup OK
- skrytí implementace (zapouzdření) – metody

Příklad – body ve 2D prostoru 1/2

- definice třídy

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

- inicializace proměnných

```
r = Point(10,20)
s = Point(13,24)
print("r.x=",r.x)
```

```
r.x = 10
```

- změna atributu

```
s.y =20
print("s.y =",s.y)
```

```
s.y = 20
```

Příklad – body ve 2D prostoru 2/2

- Funkce s argumenty datového typu `Point`:

```
1  import math
3  def distance(r,s):
4    """ Vypocita vzdalenost dvou bodu 'r','s' typu Point """
5    return math.sqrt((r.x-s.x)**2+(r.y-s.y)**2)
7  r=Point(10,20)
8  s=Point(13,24)
9  print(distance(r,s))
```

```
5.0
```

Příklad – knihy 1/3

- práce se seznamem knih
- atributy knihy: název, autor, rok
- chceme seznam načítat/ukládat do souboru
- implementace třídy:

```
class Book:
  def __init__(self, title, author, year):
    self.title = title
    self.author = author
    self.year = year
```

- vytvoření záznamů

```
a = Book ("Dva roky prazdnin", "Jules Verne", 1888)
b = Book ("Honzikova cesta", "Bohumil Riha", 1954)
```

Příklad – knihy

- načítání ze souboru

```
def load_library(filename):
  book_list = []
  with open(filename, "r") as f:
    for line in f:
      a, t, i = line.split(";")
      book_list.append(Book(t, a, i))
  return book_list
```

- ukládání do souboru

```
def save_library(filename, book_list):
  with open(filename, "w") as f:
    for b in book_list:
      f.write(b.title + ";" + b.author + ";"
      + b.year + "\n")
```

Příklad – knihy 3/3

- použití

```
1  save_library("library.csv", [a, b])
2  books = load_library("library.csv")
3  for b in books: print(b.title)
```

Reklamní vstup

CSV jsou super, ty chcete!

- CSV = Comma-separated values
- formát pro reprezentaci tabulkových dat
- jednoduchý textový formát, položky odděleny čárkami (nebo podobnými znaky)

Příklad – studenti a kurzy 1/3

- reprezentace studentů a kurzů
- kurz je seznamem zapsaných studentů
- implementace třídy `Student`

```
1  class Student:
2    def __init__(self, id, name):
3      self.id = id
4      self.name = name
```

Příklad – studenti a kurzy 2/3

```
1  class Course:
2    """ list of the students """
3    def __init__(self, code):
4      """ constructor """
5      self.code = code
6      self.students = []
8    def add_student(self, student):
9      """ add student to the list """
10     self.students.append(student)
11
12   def print_students(self):
13     """ print student enrolled to the course """
14     i = 1
15     for s in self.students:
16       print(str(i) + ".", str(s.id), s.name, sep="\t")
17       i += 1
```

Příklad – studenti a kurzy 3/3

- použití

```
jimmy = Student(555007, "James Bond")
c = Course("ZPR")
c.add_student(Student(555000, "Luke Skywalker"))
c.add_student(jimmy)
c.add_student(Student(555555, "Bart Simpson"))
c.print_students()
```

```
# 1. 555000 Luke Skywalker
# 2. 555007 James Bond
# 3. 555555 Bart Simpson
```

- co kdybychom chtěli řadit?
  - `sorted(self.students)` nefunguje – není definováno
  - definovat `__lt__(self, other)`
  - `sorted(self.students, key=lambda s: s.name)`

Část III

Abstraktní datové typy

Abstraktní datové typy

Datový typ

- rozsah hodnot, které patří do daného typu
- operace, které je možno s těmito hodnotami provádět

Abstraktní datový typ (ADT)

- rozhraní
- popis operací, které chceme provádět

Konkrétní datová struktura

- implementace
- přesný popis uložení dat v paměti
- definice funkcí pro práci s těmito daty

Dva pohledy na data

Abstraktní

- operace, které budu s daty provádět
- co musí operace splňovat
- například množina: ulož, najdi, vymaž
- tento předmět

Výhody: snadný vývoj, jednodušší přemýšlení o problémech

Riziko: svádí k ignorování efektivity

Implementační

- jak jsou data uložena v paměti
- jak jsou operace implementovány
- například binární vyhledávací strom
- navazující předmět :-)

Nejpoužívanější ADT

- seznam
- zásobník
- fronta
- množina
- slovník (asociativní pole)

III. Abstraktní datové typy

Datový typ seznam

Zásobník

Různé varianty seznamu

- obsahuje posloupnost prvků
  - stejného typu
  - různého typu
- přidání prvku
  - na začátek
  - na konec
  - na určené místo
- odebrání prvku
  - ze začátku
  - z konce
  - konkrétní prvek
- test prázdnosti, délky
- případně další operace, např. přístup pomocí indexu

Seznamy v Pythonu – operace

```
s = [4, 1, 3]  # seznam
s.append(x)    # prida prvek x na konec
s.extend(t)    # prida vsechny prvky t na konec
s.insert(i, x) # prida prvek x pred prvek na pozici i
s.remove(x)    # odstrani prvni prvek rovny x
s.pop(i)       # odstrani (a vrati) prvek na pozici i
s.pop()        # odstrani (a vrati) posledni prvek
s.index(x)     # vrati index prvniho prvku rovneho x
s.count(x)     # vrati pocet vyskytu prvku rovnych x
s.sort()       # seradi seznam
s.reverse()    # obrat seznam
x in s         # test, zda seznam obsahuje x
# (linearni pruchod seznamem!)
```

III. Abstraktní datové typy

Datový typ seznam

Zásobník

Zásobník

- strukturovaný/složený datový typ
- obsahuje předem neznámé množství položek, typicky stejného typu *(jako seznam/pole)*
- podporuje následující operace (se složitostí $O(1)$)
  - přidání položky na konec *(push)*
  - odebrání položky z konce *(pop)*
  - test, jestli je zásobník prázdný *(is_empty)*
- položky jsou odebírány v opačném pořadí, než byly přidány. *(LIFO — last in first out)*
- zásobník může podporovat i další operace
  - nedestruktivní čtení z konce *(peek)*
  - zjištění počtu položek na zásobníku *(size)*

Zásobník – příklad použití

```
1  from stack import Stack
3  s=Stack()
5  s.push(1)
6  s.push(2)
7  s.push(3)
9  print(s.pop())
10 print(s.pop())
12 s.push(10)
14 print(s.pop())
15 print(s.is_empty())
16 print(s.pop())
17 print(s.is_empty())
```

Zásobník – implementace pomocí pole

```
1  class Stack:
2    def __init__(self):
3      self.items = []
5    def size(self):
6      return len(self.items)
8    def is_empty(self):
9      return self.size()==0
11   def push(self, item):
12     self.items+=[item]
14   def pop(self):
15     return self.items.pop()
17   def peek(self):
18     return self.items[-1]
```

Příklad - převod do jiné číselné soustavy

```
1  from stack import Stack
3  def to_str(n,base):
4    cislice = "0123456789ABCDEF"
5    assert(n>=0)
6    stack=Stack()
7    while True:
8      stack.push(n % base)
9      n //= base
10     if n==0:
11       break
12   result=""
13   while not stack.is_empty():
14     result+=cislice[stack.pop()]
15   return result
17 print(to_str(67,2))
```