Martin Trnečka

Základy programování v Pythonu

druhá část

Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci

Tento text slouží jako doplňkový učební materiál pro druhou část dvousemestrálního kurzu *Základy programování pro IT*, jenž je vyučován v rámci bakalářského studia na Katedře informatiky Přírodovědecké fakulty Univerzity Palackého v Olomouci. Kurz je zaměřen na mírně pokročilé programátorské koncepty, které jsou demonstrovány prostřednictvím jazyka Python. Kurz není primárně zaměřen na jazyk samotný. Některé pokročilé aspekty jazyka Python, které si studenti osvojí v pozdější části studia, jsou záměrně zamlčeny. Text je určen především studentům se znalostí programování na úrovni kurzu *Základy programování pro IT 1*, na který tento kurz navazuje. Text je průběžně rozšiřován a doplňován. Poslední změna proběhla 13. září 2022. Pokud v textu naleznete chyby, prosím, kontaktujte mne prostřednictvím e-mailové adresy martin.trnecka@upol.cz.

Děkuji Janovi Laštovičkovi, Petru Osičkovi, Markétě Trnečkové a Jirkovi Zacpalovi za jejich připomínky a cenné podněty ke kvalitě textu.

Martin Trnečka

Obsah

Funkce

V nadcházející kapitole rozšíříme naše znalosti funkcí z předchozího semestru. Představíme důležité programátorské koncepty: anonymní funkce, vnořené funkce a funkce vyšších řádů.

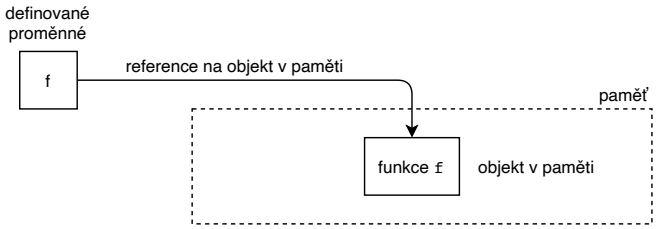
Funkce jsme doposud používali tak, že jsme pomocí identifikátoru funkce a kulatých závorek, v nichž jsou uvedeny jednotlivé argumenty funkce, provedli volání dané funkce. Zamlčeli jsme ale, že v jazyce Python[1] je identifikátor funkce (bez kulatých závorek) proměnná, která obsahuje referenci na objekt,[2] jenž danou funkci reprezentuje. Pokud například vytvoříme funkci `f()`.

[1] Stejně tak je tomu i v jiných programovacích jazycích.

[2] Pro jednoduchost si opět vystačíme s intuitivním chápáním pojmu objekt.

```
# funkce
def f(x):
  return x
```

Identifikátor funkce se stává proměnnou tak, jak je to ukázáno na obrázku 1.



Obrázek 1: Identifikátor funkce je proměnná, která obsahuje referenci na objekt reprezentující danou funkci.

S touto proměnnou můžeme pracovat tak, jak jsme to dělali doposud. Například.

```
# funkce
def na_druhou(x):
  return x**2

# volání funkce
print(na_druhou(4)) # vypíše: 16

# vytvoření nové reference na funkci na_druhou
f = na_druhou
```

```
# zavolání funkce pomocí nové reference
print(f(4)) # vypíše: 16
```

V programovacích jazycích se používá pojem *first-class citizen*[3] pro označení entit, které mohou být: (i) uloženy v proměnných; (ii) použity jako argument funkce; (iii) použity jako návratová hodnota funkce.[4] V jazyce Python jsou základní datové typy, například, čísla, řetězce nebo seznamy, first-class citizens. Navíc v jazyce Python, stejně jako v řadě jiných programovacích jazyků, patří mezi first-class citizens také funkce.[5]

Průvodce studiem
To, že je možné pracovat s funkcemi jako s proměnnými, značně rozšiřuje funkcemi poskytovanou programátorskou abstrakci.

[3] Případně také first-class object. Tento pojem zavedl v 60. letech minulého století informatik Christopher Strachey, který, mimo jiné, formalizoval také pojmy l-value a r-value.

[4] Pro naše účely jsme definici first-class citizen zjednodušili.

[5] Běžně se používá termín *first-class function*.

Anonymní funkce

Pokud vytváříme funkci pomocí klíčového slova `def`, vždy dojde k uložení reference na objekt reprezentující tuto funkci do proměnné s názvem odpovídajícím identifikátoru funkce. V případě, že chceme pracovat s funkcí jako s objektem, je obvykle toto automatické uložení do proměnné nežádoucí.
Některé programovací jazyky umožňují vytvářet funkce beze jména (bez identifikátoru). Takovéto funkce se nazývají *anonymní funkce*. Pro zápis anonymní funkce, jenž je uvedený níže, se používá klíčové slovo `lambda`.[6]

```
lambda parametr_1, parametr_2, ..., parametr_n: výraz
```

Ve výše uvedeném zápisu, také označovaném jako *lambda výraz*, `parametr_1, ..., parametr_n` jsou parametry anonymní funkce, jež jsou přístupné v těle anonymní funkce tvořeném jediným výrazem `výraz`. Výsledná hodnota, na kterou se `výraz` vyhodnotí, je vrácena jako návratová hodnota anonymní funkce. Vyhodnocením lambda výrazu vznikne v paměti objekt reprezentující funkci, na který nevede žádná reference.[7] Pro anonymní funkce platí stejná pravidla pro rozsah platnosti lokálních proměnných a předávání hodnot do funkce jako u běžných funkcí. Následující příklad ukazuje dvě ekvivalentní funkce.

[6] Ve většině programovacích jazyků se anonymní funkce vytvářejí právě pomocí klíčového slova `lambda` a často se nazývají *lambda funkce*. Toto názvosloví zavedl Alonzo Church ve svých pracích o lambda kalkulu (teoretickém základu funkcionálních programovacích jazyků), ve kterých mají anonymní funkce klíčovou roli.

[7] V minulém semestru jsme uvedli, že výrazy vytvářejí hodnoty. Lambda výrazy toto nijak nenarušují. Z pohledu jazyka Python lze na objekty reprezentující funkce nahlížet jako na objekty reprezentující (speciální) hodnoty.

```
# zápis funkce pomocí def
def f1(x, y):
  return x + y

# zápis funkce pomocí lambda výrazu
f2 = lambda x, y: x + y

# obě funkce se volají stejně
f1(4,2)
```

Průvodce studiem
Na rozdíl od příkazu `def` je `lambda` skutečně výrazem. V důsledku toho lze lambda výraz použít i v jiných výrazech.

```
f2(4,2)
```

Anonymní funkce je možné zavolat i přímo bez přiřazení do proměnné. Například.

```
# zápis funkce pomocí lambda výrazu a volání této funkce
(lambda x, y: x + y)(4,2)
```

Funkci `na_druhou()`, kterou jsme použili na začátku této kapitoly, můžeme zapsat pomocí lambda výrazu takto.

```
na_druhou = lambda x: x**2

# zavolání funkce
na_druhou(4)
```

Lambda výrazy se obvykle používají pro zápis jednoduchých funkcí, které nepotřebují identifikátor a ve funkcích vyšších řádů, kterým se budeme věnovat později. V těchto případech značně zjednodušují syntaxi.

Průvodce studiem
Anonymní funkce se používají především pro zápis krátkých funkcí, které nepotřebují jméno.

Výchozí parametry funkce

V některých situacích je žádoucí předem určit výchozí hodnotu parametru funkce. Ta je použita v případě, kdy do funkce není pro daný parametr předán žádný argument. Výchozí hodnoty se zapisují za parametry funkce pomocí symbolu = tak, jak je to ukázáno na následujícím příkladu.

```
def f(x=1): # výchozí hodnota parametru x je 1
  return x

print(f(42)) # vypíše: 42
print(f()) # vypíše: 1 (výchozí hodnota parametru x)
```

Analogicky lze zapsat výchozí hodnoty parametrů i v lambda výrazu.

```
f = lambda x=1: x # výchozí hodnota parametru x je 1

print(f(42)) # vypíše: 42
print(f()) # vypíše: 1 (výchozí hodnota parametru x)
```

Pokud funkce obsahuje parametry s výchozí hodnotou i bez výchozí hodnoty, musí být parametry s výchozí hodnotou uvedeny až za všemi parametry bez výchozí hodnoty.

```
# parametr x nemá výchozí hodnotu, y má výchozí hodnotu 1
def f(x, y=1):
  return x*y

print(f(42, 10))
print(f(42))
```

Pokud zaměníme pořadí parametrů,[8] dojde k syntaktické chybě.

[8] Parametr s výchozí hodnotou bude před parametrem bez výchozí hodnoty.

```
def f(y=1, x):
  return x*y
# způsobí chybu: SyntaxError: non-default argument follows
    default argument
```

Dodejme, že nastavení výchozí hodnoty parametru na mutovatelný datový typ (seznam) může způsobit nechtěné problémy. Například.

```
1  def pridej_a_secti(x, seznam=[]):
2    seznam.append(x)
3    soucet = 0
4    for i in seznam:
5      soucet += i
6
7    return soucet
8
9  print(pridej_a_secti(1, [42, 43])) # vypíše: 86
10 print(pridej_a_secti(42)) # vypíše: 42
11 print(pridej_a_secti(43)) # vypíše: 85!
```

Jazyk Python vyhodnocuje výchozí hodnotu pouze jednou a to při prvním volání funkce, kde není předán odpovídající argument. Ve všech nadcházejících volání k tomuto již nedochází. Při volání na řádku 10 je vytvořen nový prázdný seznam, který je použit jako výchozí hodnota. Ten je v těle funkce `pridej_a_secti()` změněn. Při dalším volání je tento pozměněný seznam použit jako výchozí hodnota.

Vnořené funkce

Tělo funkce může obsahovat další funkce, které se běžně označují jako *vnořené funkce* (nested functions). Příklad následuje.

```
1  # funkce
2  def f1():
3    # tělo funkce
4    # vnořená funkce
```

```
5     def f2():
6       # tělo vnořené funkce
7       print("Zanořená funkce")
8
9     # volání vnořené funkce f2()
10    f2()
11
12  # volání funkce f1()
13  f1()
```

Funkce `f1()` obsahuje vnořenou funkci `f2()` a také její volání (řádek 10). Funkce `f2()` je definována lokálně,[9] tedy existuje pouze v těle funkce `f1()` a není možné ji volat mimo tuto funkci. Pokud řádek 13 nahradíme následujícím kódem, dojde k chybě, jelikož f2() není definována.

[9] Identifikátor `f2` je lokální proměnná.

```
# volání funkce f2()
f2() # způsobí chybu: NameError: name 'f2' is not defined
```

Zanořovat lze i anonymní funkce.[10]

[10] Rovněž je možné v lambda výrazu použít další lambda výraz.

```
def f1():
  f2 = lambda zprava: print(zprava)
  f2("Zanořená funkce")
```

Rozsah platnosti

Doposud jsme uvažovali pouze globální proměnné (definované vně funkce) a lokální proměnné (definované v těle funkce). Možnost vnořovat funkce toto rozdělení mírně komplikuje. Každá proměnná má svůj rozsah platnosti,[11] který vymezuje místo, kde je daná proměnná definována. Rozsahy platností jsou v programovacích jazycích reprezentovány pomocí *prostředí*.[12] Globální proměnné jsou definovány v tzv. *globálním prostředí*, zatímco lokální proměnné v tzv. *lokálním prostředí*. Lokální prostředí funkce je vytvořeno při jejím zavolání a je vymezeno tělem funkce.[13] Každé volání funkce má své vlastní lokální prostředí. Uvažme následující příklad.

[11] Viz sekce Rozsah platnosti proměnných v textu *Úvod do programování v jazyce Python, první část*.

[12] Rozsah platnosti je syntaktický pojem. Prostředí, které uchovávají platnosti jednotlivých proměnných, vznikají až za běhu programu.

[13] Lokální prostředí funkcí jsou uchovávána v paměti počítače.

```
1   x = 42
2   y = 1
3
4   # funkce
5   def f1():
6     x = 4
7
8     # vnořená funkce
```

```
9     def f2():
10      x = 2
11      y = 3
12
13      # vnořená funkce ve vnořené funkci
14      def f3():
15        x = 0
16
17      f3()
18    f2()
19  f1()
```

Funkce `f3()` má ve svém lokálním prostředí definovánu (řádek 15) lokální proměnnou `x`. Prostředí funkce, obsahující vnořenou funkci, se stává *nadřazeným prostředím* pro tuto funkci. Přesněji řečeno nadřazené prostředí je prostředí, ve kterém byla funkce definována (ve kterém funkce vznikla). Například lokální prostředí funkce `f1()` je nadřazené prostředí pro lokální prostředí funkce `f2()`.[14] Nadřazené prostředí pro lokální prostředí funkce `f1()` je globální prostředí. To jako jediné nemá žádné nadřazené prostředí.[15] Obrázek 2 schématicky zachycuje výše uvedený příklad.

Pokud je v těle funkce přistupováno k proměnné, která není v jejím lokálním prostředí definována, je tato proměnná postupně hledána v nadřazených prostředích.[16] Jakmile je proměnná nalezena, vyhledávání končí. Pokud nalezena není, dojde k chybě programu. Například v těle funkce `f3()` není definována proměnná `y`. Pokud bychom přidali do `f3()` na řádek 16 výpis `print(y)` byla by vypsána hodnota 3, jelikož proměnná `y` existuje v nadřazeném lokálním prostředí funkce `f3()`.[17] Pokud bychom na stejné místo umístili výpis `print(z)`, budou při pokusu vypsat tuto proměnou, postupně prohledána všechna nadřazená lokální prostředí. Jako poslední je prohledáno globální prostředí, ve kterém rovněž není proměnná `z` definována a program skončí chybou.

Ve výše uvedeném příkladu jsme záměrně použili stejné názvy proměnných, abychom ukázali, že jednotlivá prostředí jsou nezávislá a že, pokud je v prostředí definována daná proměnná, již se tato proměnná nehledá v prostředí nadřazeném.[18] Pro úplnost je ale třeba dodat, že v některých situacích mohou stejné názvy způsobit chybu programu. Uvažme následující kód.

```
1  a = 42 # globální proměnná
2
3  def f1():
4    a = 0 # nová lokální proměnná
5    print(a) # vytiskneme lokální proměnnou
6
7  f1() # vypíše: 0
```

Průvodce studiem
Pokud je jako nadřazené prostředí použito prostředí definice funkce, mluvíme o tzv. *lexikálním rozsahu platnosti*.

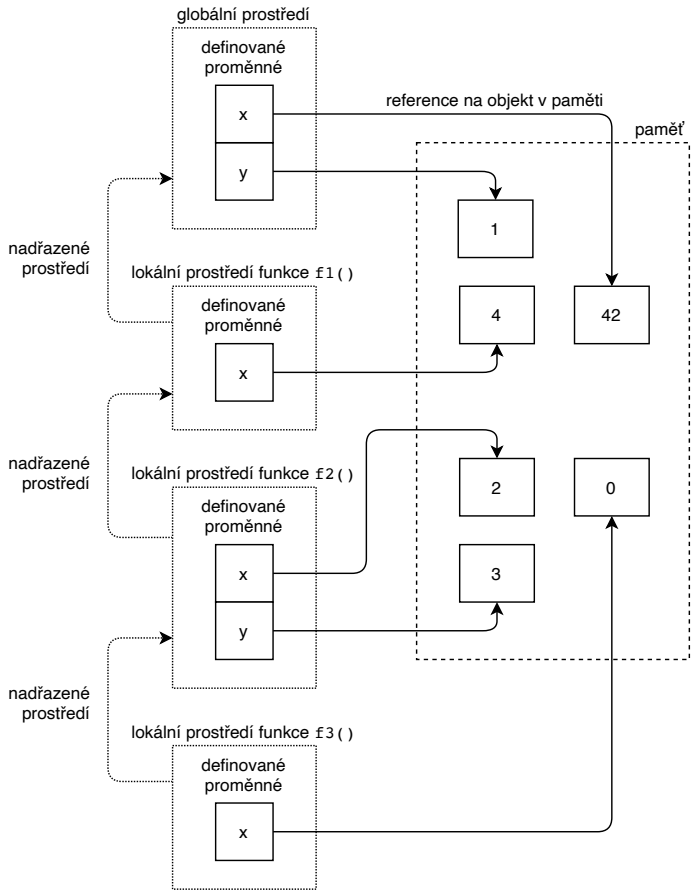
[14] Funkce `f2()` je vnořena ve funkci `f1()`.

[15] V jazyce Python existuje pro globální prostředí nadřazené prostředí, které se nazývá built-in (vestavěné) prostředí. V tomto prostředí jsou definovány vestavěné funkce jazyka Python, například funkce `print()`. Programátor do tohoto prostředí nemůže přistoupit.

[16] Všechny proměnné, které jsou definovány v nadřazených lokálních prostředích se označují jako *enclosed* (přiložené) proměnné.

[17] Nadřazené lokálním prostředí funkce `f3()` je lokální prostředí funkce `f2()`

[18] Důsledkem toho je možné dočasně „překrýt“ hodnotu dané proměnné. To může být v některých případech žádoucí, ale stejně tak to může vést na nechtěné sémantické chyby.



Obrázek 2: Jednotlivá prostředí a v nich definované proměnné.

```
8
9  def f2():
10   print(a) # chceme vytisknout globální proměnnou
11   a = 0 # nová lokální proměnná
12
13 f2() # způsobí chybu: UnboundLocalError: local variable 'a'
       referenced before assignment
```

Zatímco volání funkce `f1()` proběhne normálně, při volání funkce `f2()` dojde k chybě. Interpret jazyka Python se bude snažit při vykonávání `print(a)` na řádku 10 vytisknout lokální proměnnou `a` na místo té globální.[19]

Ke stejnému problému dochází v případě překrytí jmen lokálních proměnných a proměnných v nadřazeném lokálním prostředí. Například.

```
1  def f1():
2    a = 42
3    def f2():
```

[19] Při volání funkce je v systémovém zásobníku vytvořen prostor pro všechny lokální proměnné. Funkce tedy „ví“, že obsahuje lokální proměnnou `a` a tu se pokusí vypsat (napřed je prohledáváno lokální prostředí). Jelikož proměnná `a` ještě nebyla v lokálním prostředí inicializována (inicializace proběhne až na řádku 11), proměnná `a` nemá přiřazenou hodnotu a dojde k chybě. Dodejme, že existence globální proměnné v tomto případě nehraje žádnou roli. K chybě by došlo i kdybychom tuto proměnou neuvedli.

```
4       print(a)
5       a = 0
6     f2()
7
8   f1() # způsobí chybu: UnboundLocalError: local variable 'a'
        referenced before assignment
```

V obou výše uvedených případech jazyk Python umožňuje explicitně určit s jakou proměnnou se má pracovat.[20]

[20] Tento způsob opět záměrně zamlčíme, jelikož změna proměnných v lokálně nadřazeném prostředí obvykle přináší problémy, viz sekce Rozsah platnosti proměnných v textu *Úvod do programování v jazyce Python první část*.

Funkce vyšších řádů

Jako *funkce vyššího řádu* se označuje funkce, která akceptuje jako parametry jiné funkce nebo vrací funkci jako návratovou hodnotu. Následující kód ukazuje funkci vyššího řádu `filtrovani()`, která akceptuje jako vstup funkci (parametr `f`) vracející `True` nebo `False`, na základě které jsou vyfiltrovány položky seznamu (parametr `sekvence`).

Průvodce studiem
Funkce vyšších řádů poskytují výrazně větší programátorskou abstrakci než běžné funkce.

```
def filtrovani(f, sekvence):
  sekvence_f = []
  for i in sekvence:
    if f(i):
      sekvence_f.append(i)

  return sekvence_f
```

Příklad použití funkce `filtrovani()` následuje.

```
sekvence = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

licha_cisla = lambda x: x % 2 != 0
suda_cisla = lambda x: x % 2 == 0

sekvence_licha_cisla = filtrovani(licha_cisla, sekvence)
print(sekvence_licha_cisla) # vypíše: [1, 3, 5, 7, 9]

sekvence_suda_cisla = filtrovani(suda_cisla, sekvence)
print(sekvence_suda_cisla) # vypíše: [2, 4, 6, 8, 10]
```

Dodejme, že jazyk Python disponuje funkcí `filter()`, která funguje stejně jako výše uvedená funkce.

```
sekvence = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sekvence_licha_cisla = filter(lambda x : x % 2 != 0,
    sekvence)
print(sekvence_licha_cisla) # vypíše: [1, 3, 5, 7, 9]
```

Uvažme následující jednoduchý, ale velmi zajímavý příklad.

```
1  def vytvor_funkci_na_n(n):
2    # parametr n je lokální proměnná
3    def na_n(x):
4      print(x**n)
5
6    # lokální funkce je vrácena jako návratová hodnota
7    return na_n
8
9  na_druhou = vytvor_funkci_na_n(2)
10 na_druhou(4) # vypíše: 16
11
12 na_treti = vytvor_funkci_na_n(3)
13 na_treti(4) # vypíše: 64
```

Funkce `vytvor_funkci_na_n()` je funkce vyššího řádu, která vrací jako návratovou hodnotu lokálně definovanou funkci `na_n()`. Na první pohled by se mohlo zdát, že uvedený kód nemůže fungovat. Při volání funkce `vytvor_funkci_na_n(2)` (řádek 9) má proměnná `n` v lokálním prostředí této funkce hodnotu 2. V prostředí této funkce je definována funkce `na_n()`, která je pouze navrácena (řádek 7) a není zavolána.[21] Při volání funkce `na_druhou(4)` (řádek 10), je volána funkce `na_n(4)`, jež byla vrácená při volání (řádek 9) funkce `vytvor_funkci_na_n(2)`. V jejím lokálním prostředí má proměnná `x` hodnotu 4. Proměnná `n`, která je použita v těle funkce (řádek 4), ale v tomto prostředí není definována. Je tedy otázka, jak funkce `na_n(4)` získá hodnotu proměnné `n`.

V jazyce Python si každá vnořená funkce udržuje informaci o všech nadřazených lokálních prostředích,[22] které jsou k dispozici v době její definice (vzniku). Analogickým způsobem se chovají i jiné programovací jazyky.[23] Funkce `na_n()` má v době svého běhu přístup k proměnným v nadřazeném lokálním prostředí (lokální prostředí funkce `vytvor_funkci_na_n(2)`) a to i v době, kdy již byla tato funkce ukončena. Lokální prostředí a proměnné v nich definované pro výše uvedený příklad jsou ilustrovány na obrázku 3.

Dodejme, že pro vnořené anonymní funkce platí stejná pravidla, jež byla popsána výše. Například pomocí lambda výrazu můžeme zápis funkce `vytvor_funkci_na_n()` zjednodušit následovně.

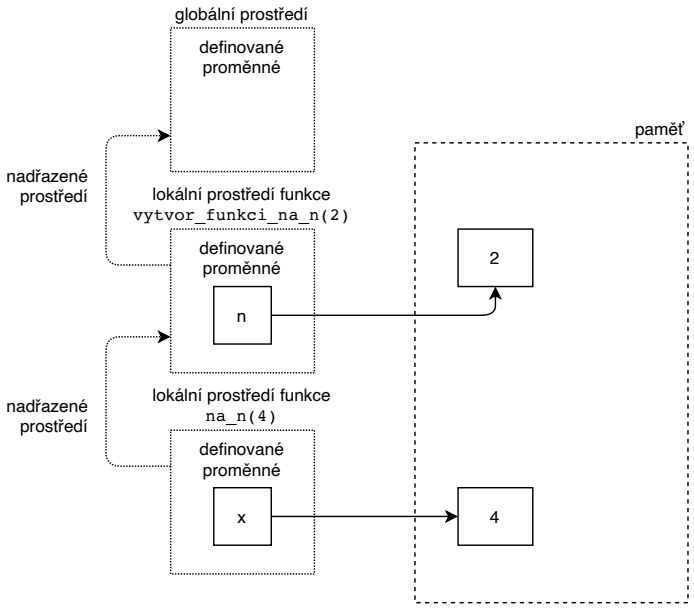
```
def vytvor_funkci_na_n(n):
  return lambda x : print(x**n)

na_druhou = vytvor_funkci_na_n(2)
na_druhou(4) # vypíše: 16
```

Případně můžeme celou funkci `vytvor_funkci_na_n()` nahradit

[21] Zde je důležité si uvědomit, že lokální prostředí funkce `na_n()` v době běhu funkce `vytvor_funkci_na_n(2)` neexistuje, jelikož funkce `na_n()` není volána.

[22] Nadřazená lokální prostředí, která existují v době vzniku funkce, se označují *closure* (uzávěr).

[23] Například Java nebo C#.



Obrázek 3: Při volání funkce `na_n(4)` je vytvořeno lokální prostředí, jehož nadřazené lokální prostředí je prostředí vytvořené při volání funkce `vytvor_funkci_na_n(2)`, ve kterém má proměnná `n` hodnotu 2.

dvěma vnořenými lambda výrazy.

```
vytvor_funkci_na_n = lambda n : lambda x : print(x**n)
na_druhou = vytvor_funkci_na_n(2)
na_druhou(4) # vypíše: 16
```

Přístup k nadřazeným lokálním prostředím funkcí, které již byly ukončeny má jedno úskalí. Předpokládejme, že chceme naprogramovat funkci, která pro zadané číslo `n` vrátí `n`-prvkový seznam, jehož *i*-tá položka bude funkce akceptující parametr `x`, počítající *i*-tou mocninu čísla `x`. Tato funkce by mohla vypadat následovně.

```
1 def vytvor_funkce_na_n(n):
2   na_n = []
3   for i in range(n):
4     na_n.append(lambda x: x**i)
5
6   return na_n
```

Na první pohled se zdá být řešení funkční, jenže není.

```
# test funkce vytvor_funkce_na_n()
na_n = vytvor_funkce_na_n(5)

print(na_n[0](4)) # chceme aby vypsalo: 1, vypíše 256
print(na_n[1](4)) # chceme aby vypsalo: 4, vypíše 256
print(na_n[2](4)) # chceme aby vypsalo: 16, vypíše 256
print(na_n[3](4)) # chceme aby vypsalo: 64, vypíše 256
```

```
print(na_n[4](4)) # chceme aby vypsalo: 256, vypíše 256
```

Při běhu funkce `vytvor_funkce_na_n()` má proměnná `i` v každé iteraci cyklu (řádky 3–4) jinou hodnotu. Po ukončení cyklu má proměnná `i` hodnotu rovnu `n-1`. Vnořená funkce (lambda výraz na řádku 4), která ve svém těle používá proměnnou `i`, nalezne definici této proměnné v nadřazeném lokálním prostředí tak, jak bylo popsáno dříve. V tomto prostředí má, v době běhu této anonymní funkce, proměnná `i` vždy hodnotu `n-1`.[24]
Tento problém je možné obejít tak, že z proměnné `i` uděláme lokální proměnnou v těle anonymní funkce. Tím zabráníme hledání definice proměnné v nadřazeném lokálním prostředí, jelikož definice bude nalezena již v prostředí lokálním. Lokální proměnnou `i` v těle funkce na řádku 4 vytvoříme například takto.[25]

[24] Do nadřazeného lokálního prostředí se přistupuje až v době, kdy cyklus (řádky 3–4) skončil. V našem případě má proměnná `i` hodnotu 4, proto obdržíme jako výsledek číslo $256 = 4^4$.

[25] Parametr funkce `i` je lokální proměnou a bude mít hodnotu argumentu předanému funkci, tedy `i`.

```
na_n.append(lambda x, i=i: print(x**i))
```

Shrnutí

V této kapitole jsme se seznámili s pokročilejšími, běžně používanými možnostmi funkcí a to zejména s anonymními funkcemi, vnořenými funkcemi a funkcemi vyšších řádů. Díky uvedeným konceptům je možné dosáhnou výrazně vyšší programátorské abstrakce.

Kontrolní otázky
Odpovězte na následující otázky:

1. Co je to anonymní funkce?
2. Jak lze nastavit výchozí hodnotu parametru funkce?
3. Co je to vnořená funkce?
4. Co je to prostředí?
5. Co je funkce vyššího řádu?

Úkoly

Úkol 1
Napište funkci `test_all(f, sekvence)`, která ověří, zda je předaná funkce `f()` pravdivá pro všechny prvky sekvence `sekvence`. Pokud ano, funkce `test_all()` vrací `True`, jinak `False`.

Úkol 2
Napište funkci `test_any(f, sekvence)`, která ověří, zda je předaná funkce `f()` pravdivá alespoň pro jeden prvek sekvence `sekvence`. Pokud ano, funkce `test_any()` vrací `True`, jinak `False`.

Úkol 3
Napište funkci `mapovani(f, sekvence)`, která na každý prvek sekvence `sekvence` použije funkci `f()` a výslednou sekvenci vrátí. Například `mapovani(lambda x: x + x, [1, 2, 3, 4, 5])` vrátí seznam `[2, 4, 6, 8, 10]`.

Úkol 4
Napište funkci `aplikuj(f, sekvence)`, která akceptuje operaci se dvěma operandy, jež je zadána funkcí `f()` a vrací výsledek postupné aplikace této operace na všechny prvky sekvence `sekvence`. Například `aplikuj(lambda x, y: x + y, [1, 2, 3, 4, 5])` sečte všechny hodnoty v `[1, 2, 3, 4, 5]` a vrátí výslednou hodnotu `15`.

Úkol 5
Napište funkci `pricti_konstantu(k)`, která vrátí funkci `f(cislo)`, jež přičítá zadanou konstantu `k` k hodnotě `cislo`.

Úkol 6
Napište funkci `linearni_funkce(a, b)`, která vrátí funkci počítající hodnotu lineární funkce[26] v bode `x` zadanou parametry `a` a `b`.

[26] $f(x) = a \cdot x + b$

Úkol 7
Napište funkci `vytvor_prumer()`, která vytvoří funkci `prumer(x)` vracející průměrnou hodnotu ze všech jí předaných hodnot. Příklad volání funkce následuje.

```
p = vytvor_prumer()
p(10) # vrátí 10.0
p(11) # vrátí 10.5
p(12) # vrátí 11
```

Úkol 8
Napište funkci `spocitej(vyraz)`, kde je `vyraz` textový řetězec obsahující zápis matematického výrazu, například `"1+1"`, která vrátí výsledek výrazu v zadaném řetězci. Řetězec může obsahovat pouze celá čísla a binární operace `+` a `-`. Pro jednoduchost uvažujte pouze správně zadané řetězce.

Úkol 9
Rozšiřte předchozí úkol tak, aby funkce `spocitej(vyraz)` akceptoval operace `+`, `-`, `*` a `/` a symboly závorek `(` a `)`.[27] Pro jednoduchost uvažujte pouze správně zadané řetězce.

[27] Při implementaci je třeba dát pozor na správnou prioritu operací.

Propojené datové struktury

V minulém semestru jsme ukázali, že seznamy mohou ve svých položkách obsahovat (vnořené) seznamy. V následující kapitole se na tuto problematiku podíváme podrobněji. Vysvětlíme si pojmy mělká a hluboká kopie seznamu a ukážeme, jak lze seznamy využít při implementaci propojených datových struktur: spojový seznam, strom a graf.

Kopie seznamu

Připomeňme, že jednotlivé položky seznamu neobsahují přímo hodnoty, ale reference na tyto hodnoty. Pokud je hodnotou seznam, může jeho mutabilita způsobovat problémy. Uvažme následující příklad.[28]

[28] Dodejme, že kopii seznamu (řádek 5) je v jazyce Python rovněž možné vytvořit pomocí řezu: `L2 = L1[:]`.

```
1 # seznam obsahující vnořený seznam
2 L1 = [1, [2, 3, 4], 5]
3
4 # vytvoření kopie seznamu
5 L2 = L1.copy()
```

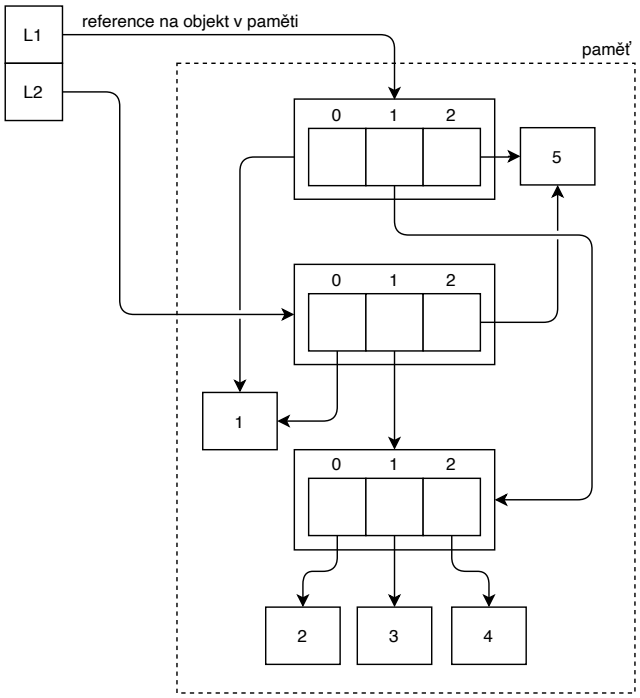
Proměnná `L2` obsahuje kopii seznamu `L1`.

```
# vypíše: True (senamy obsahují stejné hodnoty)
print(L2 == L1)

# vypíše: False (seznamy nejsou stejné objekty)
print(L2 is L1)
```

Metoda `.copy()` vytváří tzv. *mělkou kopii* (shallow copy). Při vytváření mělké kopie je vytvořen nový objekt[29] (seznam) a do jednotlivých prvků seznamu jsou zkopírovány reference z položek původního seznamu. Nejedná se tedy o plnohodnotnou kopii seznamu, jelikož prvky kopie a původního seznamu mohou být mezi sebou stále provázané skrze referenci. Tak je tomu i v našem případě (ob-

[29] Pro ověření, zda jsou dva objekty stejné jsme doposud používali operátor `is`. Podobně je možné použít funkci `id()`, která vrací jedinečný (číselný) identifikátor objektu, jenž je uložen v proměnné. Například `id(L1)` vrací identifikátor seznamu, který je v této proměnné uložený. Pokud mají hodnoty stejný identifikátor, jedná se o stejné objekty.

rázek 4). Pokud například změníme kopii `L2` seznamu `L1`, může dojít k ovlivnění původního seznamu `L1`.



Obrázek 4: Stav objektů v paměti po vykonání `L2 = L1.copy()`. Je vytvořen nový objekt (seznam `L2`) do jehož položek jsou zkopírovány reference uložené v položkách kopírovaného seznamu `L1`.

```
# změna L2
L2[1][1] = 0

# ovlivní L1
print(L1) # vypíše: [1, [2, 0, 4], 5]
```

Výše popsaný problém je důsledkem toho, že seznamy jsou mutabilní. Pokud seznam obsahuje nemutabilní prvky, například čísla, tak k tomuto nedochází.

```
L1 = [1, 2, 3, 4, 5]
L2 = L1.copy()
L2[1]= 0
print(L1) # vypíše: [1, 2, 3, 4, 5]
```

Kopie, ve které je kopírován na místo reference na objekt samotný, se nazývá *hluboká kopie* (deep copy). Jednoduchá implementace hluboké kopie seznamu obsahujícího pouze číselné seznamy a samotná čísla[30] může vypadat například následovně.

[30] Příklad se omezuje pouze na celá čísla.

```
1 # hluboká kopie pro seznamy
2 # obsahující pouze číselné seznamy a samotná čísla
```

```
3  def hluboka_kopie(data):
4    kopie = []
5
6    # ověříme zda jsou data číslo (int)
7    if isinstance(data, int):
8      kopie = data
9
10   # data jsou seznam
11   else:
12     for polozka in data:
13       kopie.append(hluboka_kopie(polozka))
14
15   return kopie
```

Ve funkci `hluboka_kopie()` jsme použili funkci `isinstance()` (řádek 7), která vrací `True` v případě, že je (datový) typ objektu uložený v proměnné, jež je předána jako první argument, shodný s typem, jenž je předán jako druhý argument.[31] Podmínka na řádku 7 je pravdivá, pokud proměnná data obsahuje celé číslo.

```
# použití funkce hluboka_kopie()
L1 = [1, [2, 3, 4], 5]
L2 = hluboka_kopie(L1)
```

Reprezentace seznamů po použití funkce `hluboka_kopie(L1)` je ukázána na obrázku 5.
V jazyce Python je přípustné, aby položkou seznamu byla reference na seznam samotný. Například.

```
L = [1, 2, 3, 4, 5]

L[0] = L
print(L) # vypíše [[...], 2, 3, 4, 5]
```

Ve výpisu pomocí funkce `print()` je skutečnost, že seznam obsahuje referenci na sebe sama, znázorněna pomocí znaků `[...]`. Na takto vnořený seznam je možné běžně použít indexační operátor.[32]

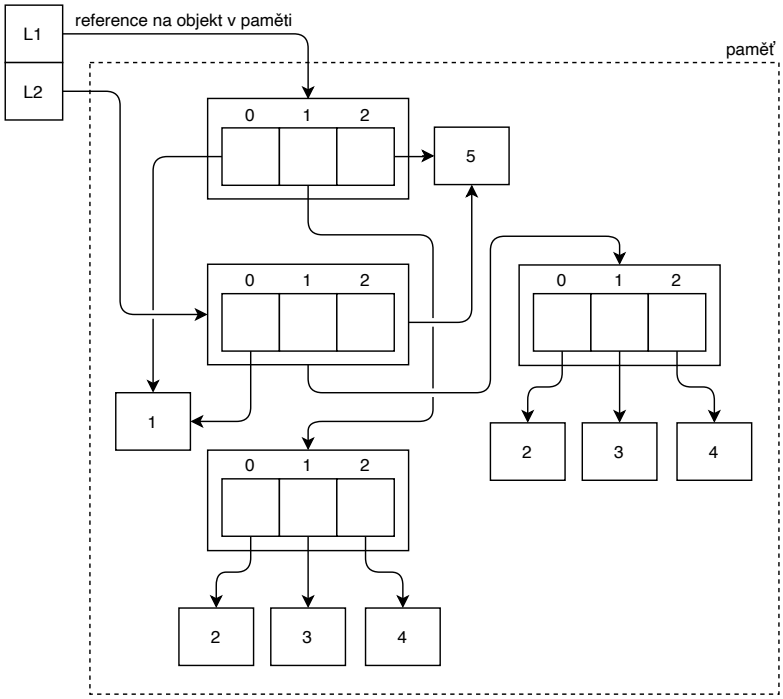
```
L[0][0][1] = 42 # změní seznam L
print(L) # vypíše [[...], 42, 3, 4, 5]
```

V nadcházející části si ukážeme, jak je možné pomocí referencí implementovat (základní) propojené datové struktury.

[31] Další typy jsou například: `float` (desetinná čísla), `str` (textové řetězce), `bool` (logické hodnoty), `list` (seznamy). Datový typ konkrétní proměnné je možné zjistit pomocí funkce `type()`.

Průvodce studiem
Vytváření hluboké kopie velkých dat (například data obsahující mnohonásobně zanořené seznamy) je výpočetně velmi náročné. Pokud je to možné, je lepší hluboké kopie nepoužívat.

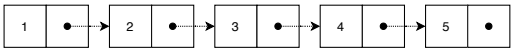
[32] Dodejme, že stejného výsledku bychom dosáhli i `L[1] = 42`, `L[0][1] = 42`, `L[0][0][1] = 42`, ...



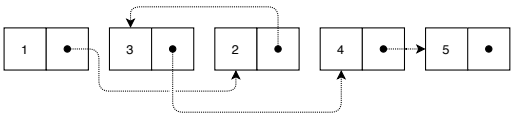
Obrázek 5: Stav objektů v paměti po vytvoření hluboké kopie seznamu `L1`.

Spojový seznam

Spojový seznam[33] (linked list) je abstraktní datová struktura uchovávající posloupnost hodnot. Každý prvek spojového seznamu uchovává hodnotu a odkaz na následující prvek seznamu. Graficky si spojový seznam obsahující hodnoty `1`, `2`, `3`, `4` a `5` můžeme představit následovně.



Jelikož je propojení mezi prvky realizováno formou odkazu, nemusí být tyto prvky uloženy v paměti počítače lineárně za sebou.

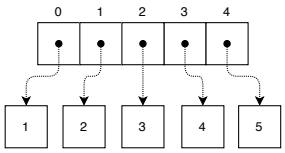


Spojové seznamy poskytují celou řadu metod. Například přístup k *i*-tému prvku, přidání prvku (na *i*-tou pozici) do seznamu, odebrání prvku (na *i*-té pozici) ze seznamu a mnoho dalších.
Nabízí se otázka, jaký je rozdíl mezi seznamem v jazyce Python a datovou strukturou (spojový) seznam. Seznamy[34] v jazyce Python jsou implementovány pomocí dynamicky alokovaného pole.[35] Schématicky tedy seznam v jazyce Python vypadá takto.

[33] Někdy též lineární spojový seznam.

[34] Stejným způsobem jsou v jazyce Python implementovány i řetězce.

[35] O alokaci se stará interpret jazyka Python.



Jednotlivé prvky (reference na položky) seznamu jsou uloženy v paměti lineárně za sebou.[36] Příjemným důsledkem je, že časová složitost přístupu k prvkům seznamu (v jazyce Python) pomocí indexačního operátoru je $\mathcal{O}(1)$. Tato implementace je ale nevýhodná v případech, kdy je seznam často modifikován (přidávání a odebírání prvků seznamu).
V následující části si ukážeme, jak lze v jazyce Python implementovat spojový seznam. Prvek spojového seznamu je možné reprezentovat jako seznam dvou prvků, kde první prvek uchovává hodnotu a druhý referenci na další prvek seznamu. Pokud další prvek neexistuje, použijeme prázdný seznam. Například seznam

[36] Samotné hodnoty již takto nemusí být uloženy.

```
[42, []]
```

představuje jeden prvek spojového seznamu.

Na prvek spojového seznamu lze nahlížet jako na jednoduchou datovou strukturu reprezentující uspořádanou dvojici.[37] Nás bude zajímat především abstrakce nad touto datovou strukturou.[38]
Metody pro přístup k prvkům uspořádané dvojice lze implementovat například pomocí funkcí.

[37] Analogická datová struktura nazývaná tečkový pár (cons) má velký význam v programovacím jazyce Lisp.

[38] Uspořádaná dvojice je abstraktní datová struktura s metodou `car()` sloužící pro přístup k prvnímu prvku uspořádané dvojice a metodou `cdr()` pro přístup k druhému prvku uspořádané dvojice. Názvy metod mají historický význam, ale s ohledem na lepší čitelnost kódu je používat nebudeme.

```
# přístup k prvnímu prvku uspořádaná dvojice
def prvni_prvek(data):
  return data[0]

# přístup k druhému prvku uspořádaná dvojice
def druhy_prvek(data):
  return data[1]
```

Na místo výše uvedeného řešení využijeme toho, že uspořádaná dvojice je reprezentována pomocí seznamu a pro přístup k jednotlivým prvkům použijeme indexační operátor. Zavedeme si dvě konstanty.[39]

[39] Konstanta je proměnná, která po celou dobu běhu programu nemění svoji hodnotu. V jazyce Python se konstanty pro lepší pochopení zdrojového kódu zapisují velkými písmeny.

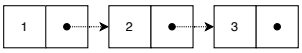
```
PRVNI_PRVEK = 0
DRUHY_PRVEK = 1

data = [1, []]
```

```
data[PRVNI_PRVEK] # přístup k prvnímu prvku uspořádaná
    dvojice
data[DRUHY_PRVEK] # přístup k druhému prvku uspořádaná
    dvojice
```

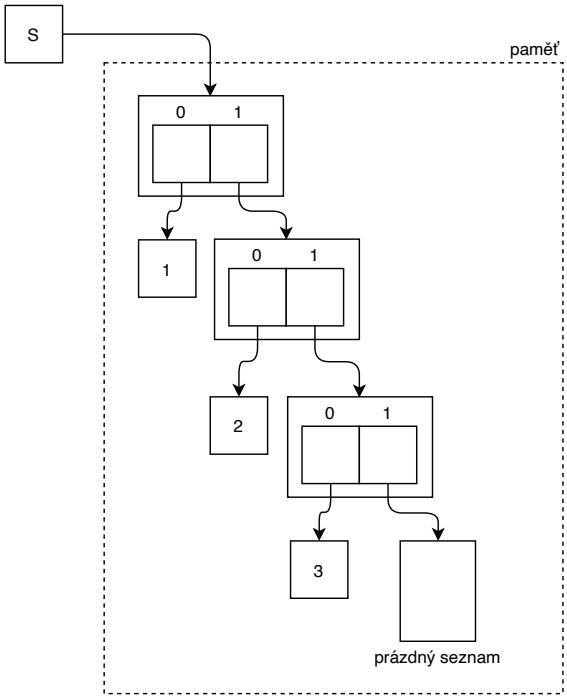
Propojení uspořádaných dvojic (prvků seznamu) lze snadno provést zanořením seznamu. Například

```
S = [1, [2, [3, []]]]
```

odpovídá spojovému seznamu.



Pro úplnost je reprezentace seznamu `[1, [2, [3, []]]]` v paměti počítače zobrazena na obrázku 6.



Obrázek 6: Reprezentace seznamu `S = [1, [2, [3, []]]]` v paměti.

S tímto seznamem můžeme pracovat pomocí metod popsaných výše. Například.

```
HODNOTA = 0
DALSI_PRVEK = 1

print(S[HODNOTA]) # vypíše: 1
print(S[DALSI_PRVEK]) # vypíše: [2, [3, []]]
```

Nyní máme vše potřebné pro naprogramování funkce vytvářející spojový seznam. Následující kód implementuje jednoduchou funkci `pridej_do_seznamu()`, která na konec seznamu (parametr `seznam`) přidá hodnotu (parametr `x`).

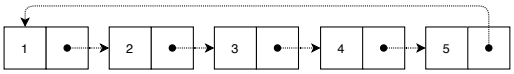
```
HODNOTA = 0
DALSI_PRVEK = 1

# funkce pro přidání prvku do seznamu
def pridej_do_seznamu(seznam, x):
  while seznam:
    seznam = seznam[DALSI_PRVEK]

  seznam.extend([x, []])

# použití funkce pridej_do_seznamu()
seznam = []
pridej_do_seznamu(seznam, 1)
pridej_do_seznamu(seznam, 2)
pridej_do_seznamu(seznam, 3)
```

Varianta funkce `pridej_do_seznamu()`, která bude udržovat položky seznamu seřazené dle velikosti uchovávané hodnoty od nejmenší po největší může vypadat například takto.[40]

[40] Za předpokladu, že seznam obsahuje pouze porovnatelné hodnoty.

```
HODNOTA = 0
DALSI_PRVEK = 1

# funkce pro zatřízení prvku do seznamu
def zatrid_do_seznamu(seznam, x):
  while seznam and seznam[HODNOTA] < x:
    seznam = seznam[DALSI_PRVEK]

  if not seznam:
    seznam.extend([x, []])
  else:
    seznam[DALSI_PRVEK] = [seznam[HODNOTA], seznam[
        DALSI_PRVEK]]
    seznam[HODNOTA] = x

# použití funkce zatrid_do_seznamu()
seznam = []
zatrid_do_seznamu(seznam, 2)
zatrid_do_seznamu(seznam, 1)
zatrid_do_seznamu(seznam, 3)
```

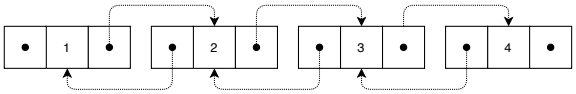
Často používanou variantou spojového seznamu je *cyklický spojový seznam*, kde poslední prvek cyklického spojového seznamu ukazuje na první prvek. Cyklický seznam si můžeme představit následovně.



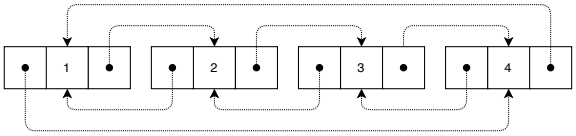
Implementace takovéto varianty je jednoduchá.

```
1  HODNOTA = 0
2  DALSI_PRVEK = 1
3
4  # funkce pro přidání do cyklického spojového seznamu
5  def pridej_do_cyklickeho_seznamu(seznam, x):
6
7    if not seznam:
8      seznam.extend([x, seznam])
9    else:
10     prvni_prvek_seznamu = seznam
11     seznam[DALSI_PRVEK] = [seznam[HODNOTA], seznam[
           DALSI_PRVEK]]
12     seznam[HODNOTA] = x
```

Uvedený kód je modifikací funkce `zatrid_do_seznamu()`. Poslední prvek vždy ukazuje na první (řádek 8) a hodnoty jsou přidávány na začátek seznamu.
Další často používanou variantou spojového seznamu je *obousměrný spojový seznam*



a *cyklický obousměrný spojový seznam*.



Pro jejich implementaci je třeba namísto uspořádané dvojice použít uspořádanou trojici. Například.

```
PREDCHOZI_PRVEK = 0
HODNOTA = 1
DALSI_PRVEK = 2

# funkce pro zatřízení prvku do seznamu
def pridej_do_obousmerneho_seznamu(seznam, x):
  predchozi_prvek = []
  while seznam:
    predchozi_prvek = seznam
```

```
    seznam = seznam[DALSI_PRVEK]

  seznam.extend([predchozi_prvek, x, []])
```

Stromy

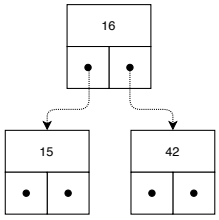
Analogickým způsobem, jako jsme vytvořili obousměrný spojový seznam, je možné vytvořit i komplexnější datové struktury. V následující části si stručně představíme, jak je možné naprogramovat pomocí uspořádané trojice *binární strom*.[41] Binární strom je abstraktní datová struktura uchovávající data v uzlech, které obsahují samotná data a odkazy na další prvky binárního stromu: *levý potomek* a *pravý potomek*.

[41] Připomeňme pojem binárního stromu z kurzu Diskrétní struktury 1. Binární strom je kořenový strom (neorientovaný souvislý graf bez kružnic), přičemž každý vrchol stromu (prvek) má nejvýše dva potomky.

Uzel stromu budeme opět reprezentovat seznamem (uspořádanou trojicí), ve které první prvek uchovává hodnotu uzlu, druhý a třetí prvek obsahují reference na potomky.

```
uzel = [42, [], []]
```

Speciálním případem je *uspořádaný binární strom*, ve kterém jsou prvky stromu lineárně uspořádány dle hodnoty uzlu. Levý potomek obsahuje hodnotu menší než hodnota uložená v daném uzlu a pravý potomek obsahuje hodnotu větší než je hodnota uložená v daném uzlu. Například.



Tento strom odpovídá následujícímu seznamu.

```
S = [16, [15, [], []], [42, [], []]]
```

Funkce vytvářející uspořádaný binární strom by mohla vypadat takto.

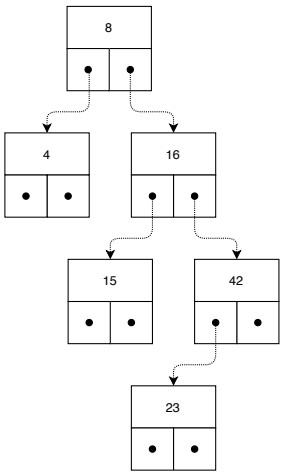
```
HODNOTA = 0
LEVY_POTOMEK = 1
```

```
PRAVY_POTOMEK = 2

def pridej_do_stromu(uzel, x):
  if not uzel:
    uzel.extend([x, [], []])
    return

  while uzel[HODNOTA] != x:
    if x < uzel[HODNOTA]:
      if uzel[LEVY_POTOMEK]:
        uzel = uzel[LEVY_POTOMEK]
      else:
        uzel[LEVY_POTOMEK] = [x, [], []]
        return

    elif x > uzel[HODNOTA]:
      if uzel[PRAVY_POTOMEK]:
        uzel = uzel[PRAVY_POTOMEK]
      else:
        uzel[PRAVY_POTOMEK] = [x, [], []]
        return
```

Příklad použití funkce `pridej_do_stromu()` následuje.[42]

```
koren_stromu = []

# přidání prvků do stromu
pridej_do_stromu(koren_stromu, 8)
pridej_do_stromu(koren_stromu, 4)
pridej_do_stromu(koren_stromu, 16)
pridej_do_stromu(koren_stromu, 15)
pridej_do_stromu(koren_stromu, 42)
pridej_do_stromu(koren_stromu, 23)
```

[42] V příkladu je postupně vytvořen binární strom.



Grafy

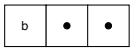
Pomocí provázaných datových struktur je rovněž možné v jazyce Python snadno implementovat *grafy*,[43] byť tento způsob reprezentace nemusí být vždy ten nejvhodnější. Následující příklad ukazuje reprezentaci neorientovaného grafu pomocí propojené datové struktury, přičemž každý uzel grafu je reprezentován jménem a seznamem sousedů.[44]

```
JMENO = 0
HRANY = 1
```

[43] Viz kurz Diskrétní struktury 1.

[44] Uzel se jménem `a`, který nemá žádné sousedy je reprezentován takto.

Uzel s jedním sousedem a jménem `b` je reprezentován takto.

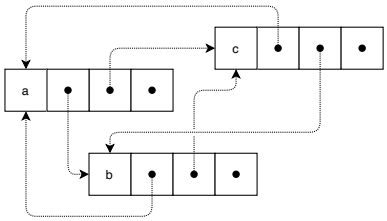


```
# přidání uzlu
def add_node(graf, jmeno):
  graf.append([jmeno, []])

# přidání neorientované hrany
def add_edge(graf, z, do):
  for uzel in graf:
    if uzel[JMENO] == z:
      v = uzel

    elif uzel[JMENO] == do:
      w = uzel

  v[HRANY].append(w)
  w[HRANY].append(v)
```

Použití funkcí výše může vypadat například takto.[45]

[45] V příkladu je postupně vytvořen graf.

```
graf = []

# přidání uzlů do grafu
add_node(graf, "a")
add_node(graf, "b")
add_node(graf, "c")

# přidání hran do grafu
add_edge(graf, "a", "b")
add_edge(graf, "a", "c")
add_edge(graf, "c", "b")
```



Shrnutí

V této kapitole jsme se věnovali problematice vnořených seznamů a vytváření jejich kopií. Ukázali jsme, jakým způsobem je možné využít reference uložené v položkách seznamu pro implementaci propojených datových struktur a to konkrétně spojového seznamu, binárního stromu a grafu.

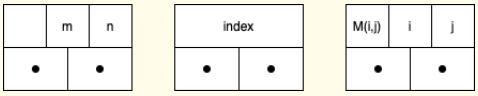
Kontrolní otázky
Odpovězte na následující otázky:

1. Co je mělká kopie seznamu?
2. Co je to hluboká kopie seznamu?

Úkoly

Úkol 10
Napište funkci `odeber_ze_seznamu(seznam, x)`, která odstraní prvek `x` ze spojového seznamu.

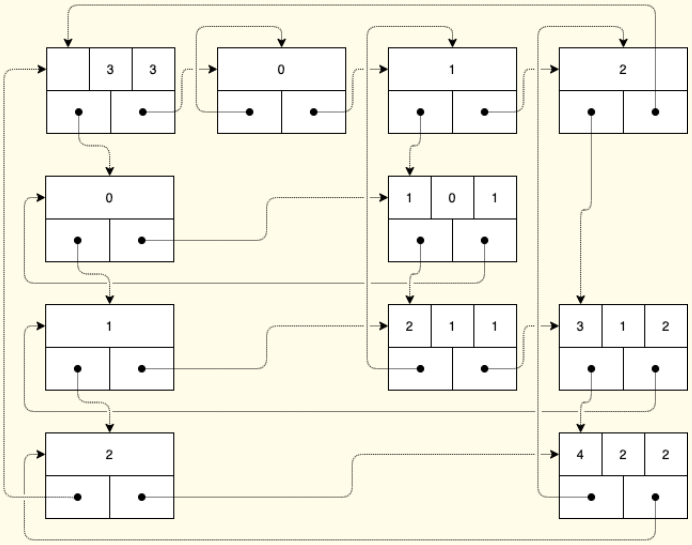
Úkol 11
Pomocí následujících struktur implementujte reprezentaci matice s `m` řádky a `n` sloupci.[46]



První struktura uchovává počet řádků matice, počet sloupců matice, odkaz na první řádek a první sloupec v matici. Druhá struktura uchovává index řádku nebo sloupce a odkaz na další řádek (sloupec), případně první hodnotu na řádku (sloupci). Poslední struktura uchovává nenulovou hodnotu v matici, číslo sloupce a řádku a odkaz na další hodnotu v řádku a sloupci.
Propojení datových struktur pro matici

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 2 & 3 \\ 0 & 0 & 4 \end{pmatrix}$$

je ukázáno na následujícím obrázku.



[46] Tento způsob reprezentace je variantou reprezentace popsané v prvním díle knihy Umění programování (The Art of Computer Programming), kapitola Informační struktury, část 2.2.6, ukládá pouze nenulové hodnoty v matici. Zejména pro velké matice, kde je jen několik nenulových hodnot je tento formát velmi úsporný.

Úkol 12
Napište funkci `odeber_z_obousmerneho_seznamu(seznam, x)`, která odstraní prvek `x` z obousměrného spojového seznamu.

Úkol 13
Pomocí spojového seznamu implementujte zásobník a frontu.

Úkol 14
Napište funkci `najdi_ve_stromu(strom, x)`, která projde uspořádaný binární strom `strom` a vrací `True`, pokud je v něm hodnota `x` nalezena, jinak `False`.

Úkol 15
Implementujte uspořádaný binární strom, ve kterém každý uzel obsahuje odkaz na svého rodiče.

Úkol 16
Rozšiřte funkci `add_edge(graf, od, do)` tak, aby akceptovala parametr `hodnota` reprezentující ohodnocení (celé číslo) hrany.

Ošetřování chyb při běhu programu

Následující kapitola popisuje standardní mechanizmy umožňující ošetření chyb, které vzniknou za běhu programu.[47]

Pokud při běhu programu nastane chyba (například při dělení nulou), je vykonávání programu okamžitě zastaveno a následně dojde k vyvolání tzv. *výjimky*. Tu si lze pro jednoduchost představit jako informační zprávu o vzniklé chybě. Pokud tato zpráva není programem *zachycena* (zpracována) program skončí chybou.[48] Například.

```
def deleni(a, b):
  return a/b

print(deleni(42, 0)) # způsobí chybu: ZeroDivisionError:
    division by zero
```

Pokud chceme takovouto chybu ošetřit a zabránit ukončení programu, můžeme: (i) vzniku chyby předcházet, například pomocí příkazu `if`, nebo (ii) zachytit výjimku vyvolanou při této chybě. Je třeba si uvědomit, že mezi (i) a (ii) je značný rozdíl. Zatímco (i) řeší chyby proaktivně, (ii) přistupuje k řešení chyb reaktivně.
Pokud je výjimka programem zachycena[49] program nekončí způsobenou chybou, ale je mu umožněno pokračovat od místa, kde byla výjimka zpracována. Pro práci s výjimkami se v jazyce Python používají příkazy `try`, `except` a `finally`.

Příkazy `try` a `except`

Příkazy `try` a `except`[50] jsou základní příkazy pro ošetření výjimek. Obecný zápis vypadá následovně.

```
try:
  příkazy_1
except:
```

[47] Budeme tedy mluvit o ošetření runtime chyb, viz sekce Hledání chyb v programu v textu *Úvod do programování v jazyce Python první část*.

[48] V tomto případě mluvíme o *neošetřené výjimce* případně *neošetřené chybě*.

[49] Běžně se používá termín *ošetřena*.

Průvodce studiem
Práce s výjimkami se v různých programovacích jazycích může lišit. Některé jazyky dokonce výjimky nepodporují (například jazyk C).

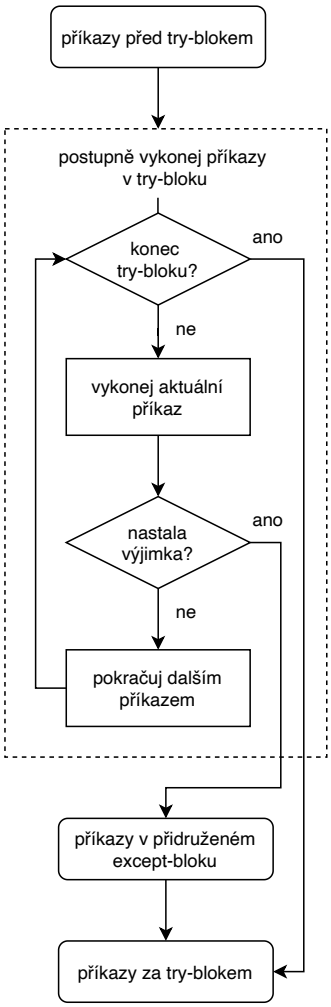
[50] Jazyk Python používá poněkud netradičně příkaz `except`, jehož analogie je ve většině programovacích jazyků označena jako příkaz `catch`.

```
  příkazy_2
```

Příkaz `try` vymezuje blok kódu `příkazy_1`, běžně označovaný jako *try-blok*, při jehož vykonávání jsou zachytávány výjimky. Příkaz `except` *přidružený*[51] k try-bloku specifikuje blok kódu, označíme jej *except-blok*, který je vykonán v případě, že v bloku `příkazy_1` došlo k vyvolání výjimky. Větvení programu při zachycení výjimky je znázorněno na obrázku 7.
Pomocí `try` a `except` můžeme funkci `deleni()` upravit následovně.

[51] Uvedený bezprostředně za try-blokem.

Obrázek 7: Grafické znázornění větvení programu při vykonávání příkazů v try-bloku.




```
def deleni(a, b):
  try:
    return a/b
  except:
    print("nelze dělit nulou")

deleni(42, 0) # vypíše: nelze dělit nulou a vrátí None
```

Upřesnění typu výjimky

K try-bloku přidružený příkaz `except` ošetřuje (stejně) všechny výjimky. V některých případech ale chceme, aby byly různé výjimky ošetřeny různým způsobem. Bezprostředně za příkaz `except` je možné uvést jméno výjimky, čímž určujeme, že příslušný blok kódu ošetřuje výjimku daného jména (typu). K jednomu try-bloku může být přidruženo více příkazů `except`. Rozšířený zápis vypadá následovně.

```
try:
  příkazy
except jméno_1:
  příkazy_1
except jméno_2:
  příkazy_2
except:
  příkazy_3
```

V případě, že dojde v try-bloku `příkazy` k vyvolání výjimky, jsou postupně procházeny přidružené příkazy `except`. Pokud název vyvolané výjimky odpovídá `jméno_1`, je vykonán blok `příkazy_1`. V opačném případě se analogicky pokračuje s dalším příkazem `except`. Dodejme, že ve výše uvedeném je poslední příkaz `except` nepovinný.[52] Názvy nejběžnějších výjimek v jazyce Python jsou shrnuty v tabulce 1.
Následující kód ukazuje ošetření několika různých výjimek při provádění funkce `deleni()`.

[52] Pokud by nebyl uveden a žádné jméno uvedené za přidruženými příkazy `except` by neodpovídalo názvů vyvolané výjimky, byla by tato výjimka neošetřena.

Výjimka	Příčina vzniku
`AssertionError`	podmínka v příkazu aserce (vysvětlíme později) není splněna
`IndexError`	přístup na neexistující index v kolekci
`NameError`	přístup k nedefinované proměnné
`TypeError`	operace s nekompatibilními datovými typy, například: `"4"+ 2`
`ValueError`	operace s kompatibilními datovými typy ale s chybnou hodnotou, například: `int("dva")`
`OverflowError`	výsledek operace je příliš velký a nelze jej reprezentovat v paměti
`ZeroDivisionError`	druhý operand operace dělení nebo modulo je roven nule
`RuntimeError`	blíže nespecifikovaná chyba
`Exception`	obecná výjimka (zahrnuje všechny výjimky)

Tabulka 1: Vybrané výjimky v jazyce Python a jejich příčiny vzniku.

```
1  def deleni(a, b):
2    try:
3      return a/b
4    except ZeroDivisionError:
5      print("nelze dělit nulou")
6    except TypeError:
7      print("nekompatibilní datové typy")
8    except:
9      print("nespecifikovaná výjimka:")
10
11 deleni(42, 0) # vypíše: nelze dělit nulou
12 deleni(42, "dva") # vypíše: nekompatibilní datové typy
13 deleni(10**1000,2) # vypíše: nespecifikovaná výjimka
```

Dodejme, že při volání `deleni(10**1000,2)` na řádku 13 je vyvolána `OverflowError` výjimka. V našem případě je tato výjimka ošetřena poslední příkazem `except` na řádku 8.
Kromě názvu výjimky je k výjimce přidružen krátký komentář,[53] upřesňující chybu při která byla výjimka vyvolána. Například výše uvedené by bez zachycení výjimek způsobily následující chyby.

[53] Interpret jazyka Python jej zobrazí ve výpisu chyby za symbolem : (dvojtečka).

```
deleni(42, 0)
# způsobí chybu: ZeroDivisionError: division by zero

deleni(42, "dva")
# způsobí chybu: TypeError: unsupported operand type(s) for
    /: 'int' and 'str'

deleni(10**1000,2)
# způsobí chybu: OverflowError: integer division result too
    large for a float
```

Při zachycení výjimky můžeme chtít pracovat (například uložit do

logu) i s tímto dodatečným komentářem, případně získat celý detailní výpis chyby. To lze provést následovně.

```
except jméno_výjimky_1 as proměnná:
  příkazy_1
```

Ve výše uvedeném bude `proměnná` lokálně definovaná proměnná v bloku `příkazy_1` a bude obsahovat komentář k vzniklé chybě. Například.

```
1  def deleni(a, b):
2    try:
3      return a/b
4    except ZeroDivisionError:
5      print("nelze dělit nulou")
6    except TypeError:
7      print("nekompatibilní datové typy")
8    except Exception as e:
9      print("nespecifikovaná výjimka:", e)
10
11 deleni(10**1000,2) # vypíše: nespecifikovaná výjimka:
       integer division result too large for a float
```

Příkaz na řádku 8 ošetřuje obecnou výjimku `Exception`.[54] V našem případě bude proměnná `e` obsahovat komentář k chybě, jež je v této části kódu ošetřena.[55]

[54] Příkaz `except` bez specifikované výjimky je pouze syntaktický cukr pro příkaz `except Exception`.

[55] Detailní vysvětlení a pochopení příkazu na řádku 8 vyžaduje znalosti objektově orientovaného programování, které bude představeno v kurzu *Základní programovací paradigmata*. Nyní se spokojíme s tím, že to takto funguje.

Manuální vyvolávání výjimky

Doposud jsme uvažovali pouze výjimky, které vznikly při chybě. Výjimky je možné manuálně vyvolat[56] pomocí příkazu `raise`, jehož použití je ukázáno níže.

[56] A to kdykoliv nejen při výskytu chyby.

```
raise jmeno_vyjimky
```

Případně je možné doplnit k výjimce komentář.

```
raise jmeno_vyjimky("komentář")
```

Například.

```
def deleni(a, b):
  try:
    if b == 0:
      raise ZeroDivisionError("operand b ve funkci deleni()
          je roven 0")

    return a/b
```

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

deleni(42, 0) # vypíše: operand b ve funkci deleni() je
    roven 0
```

Pro manuálně vyvolané blíže nespecifikované výjimky, které souvisí s chybou, se běžně používá `RuntimeError` případně `Exception`. Druhý zmíněný je možné použít i pro případy, které nesouvisí s žádnou chybou.

Vnořené try-bloky

Try-blok může obsahovat další (vnořené) try-bloky. Pokud není daná výjimka ošetřena v try-bloku je předána do nadřazeného try-bloku.[57] Pokud žádný takový není, program končí neošetřenou chybou. Předávání výjimek mezi vnořenými try-bloky se označuje jako *propagace* výjimek.

[57] Do try-bloku v němž je vnořena.

```
def deleni(a, b):
  try:
    return a/b
  except RuntimeError: # tato výjimka při dělení nenastane
    print("RuntimeError")

try:
  deleni(42, 0)
except Exception as e:
  print("chyba:", e) # vypíše: chyba: division by zero
```

Při ošetření výjimky je možné vyvolat další výjimku. Ta je zachycena nadřazeným try-blokem.

```
def deleni(a, b):
  try:
    return a/b
  except:
    raise RuntimeError("chyba ve funkci deleni()")

# dojde k vypsání: chyba ve funkci deleni()
try:
  deleni(42, 0)
except Exception as e:
  print(e)
```

Dodejme, že druhá ukázka je evidentně výhodnější, jelikož zůstává zachována informace o tom, kde k chybě došlo.

Průvodce studiem
Pokud je výjimka spojena s chybou, je vždy výhodné zachovat informaci o místě vzniku chyby, aby jí bylo možné co nejsnadněji dohledat.

Příkaz `finally`

Jazyk Python, stejně jako většina jiných programovacích jazyků, které podporují výjimky, disponuje příkazem `finally`, jenž je stejně jako příkazy `exception` přidružen k nějakému try-bloku. Příkaz `finally` určuje blok kódu, označíme jej *finally-blok*, který je vykonán vždy a to bez ohledu na to, zda byl try-blok opuštěn vyvolanou výjimkou či nikoliv. Příkaz `finally` se zapisuje za poslední příkaz `except`.

```
try:
  příkazy_1
except:
  příkazy_2
finally:
  příkazy_3
```

Příkaz `finally` je možné použít i bez příkazu `except`.[58]

[58] Ke každému bloku musí být přidružený alespoň jeden z příkazů `except` nebo `finally`.

```
try:
  příkazy_1
finally:
  příkazy_2
```

Kód ve finally-bloku se běžně používá pro „úklid“ po kódu v try-bloku k němuž je příkaz `finally` přidružen.[59]

[59] Například uzavření otevřených souborů. Této problematice se budeme věnovat později.

Příkaz `assert`

V programování se podmínky, o kterých předpokládáme, že jsou za všech okolností pravdivé, označují jako *aserce*.[60] Aserce se používají pro kontrolu, zda jsou vždy splněny dané předpoklady. Pro zápis asercí se v jazyce Python používá příkaz `assert`.

[60] Někdy také *aserční podmínky* či *invariant* programu.

```
assert výraz, "komentář"
```

Pokud je výraz `výraz` pravdivý program pokračuje dále. Pokud je nepravdivý dojde k vyvolání výjimky `AssertationError` s komentářem: `komentář`. Například.

```
x = 46
assert x==42, "x != 42" # způsobí chybu: AssertionError: x
    != 42
```

Průvodce studiem
Aserce se běžně používají při vývoji a testování. V produkčním kódu (hotový program) se ale obvykle nepoužívají.

Výjimku `AssertationError` je možné odchytit pomocí příkazu `try` stejně jako jiné dříve uvedené výjimky.[61]

[61] Stejně tak je možné ji vyvolat příkazem `raise`.

Aserce se běžně používají pro jednoduché testování funkčnosti části programu. Například lze s nimi ověřovat, zda změna jedné části programu neovlivnila funkčnost jiných částí nebo celku.[62]

[62] Toto testování, běžně označované jako *unit testování*, je velmi důležité a nemělo by být opomíjeno.

Proaktivní vs. reaktivní ošetření chyb

Přirozenou otázkou je, zda je lepší k ošetření chyb přistupovat proaktivně či reaktivně. Univerzální odpověď neexistuje. Vždy je lepší chybě předcházet, než na ni reagovat. Na druhou stranu předcházení vzniku chyby je náročnější, jelikož programátor musí předem počítat s každou možnou chybou. Běžně se používá kombinace obou přístupů. Běžné chyby, ke kterým může docházet,[63] jsou ošetřeny proaktivně a méně časté chyby, které mohou mít mnoho příčin,[64] jsou ošetřovány reaktivně.

[63] Například chybně zadaný vstup do programu.

[64] Například chyby při práci s externími soubory.

Výjimky nejsou jen chyby

Mnoho programátorů nesprávně spojuje výjimky pouze s chybami. Výjimka je zpráva informující o tom, že nastala nějaká situace. Výjimky tedy nemusí být nutně vyvolány pouze při chybě.[65] Následující příklad ukazuje, jak je možné využít výjimku pro okamžité opuštění několika vnořených cyklů.

[65] Byť se k tomuto účelu používají nejčastěji.

```
L = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
najdi = 2
nalezeno = False

try:
  for polozka_v_L in L:
    for polozka in polozka_v_L:
      if polozka==najdi:
        nalezeno = True
        raise Exception("položka nalezena")

except:
  print(nalezeno) # vypíše True
```

Výše uvedené by mělo být použito pouze v opodstatněných případech.

Shrnutí

V této kapitole jsme se seznámili s výjimkami a možnostmi jejich zpracování pomocí příkazů `try`, `except` a `finally`. Ukázali jsme, jak lze vyvolat výjimku manuálně a představili aserce, které umožňují základní testování programu.

Kontrolní otázky
Odpovězte na následující otázky:

1. Co je to výjimka?
2. Kdy dojde k vyvolání výjimky?
3. Jak lze výjimku odchytit.
4. K čemu slouží příkaz `finally`?
5. Co je to aserce?

Úkoly

Úkol 17
Dřívější úkoly doplňte o výjimky.

Bitové operace

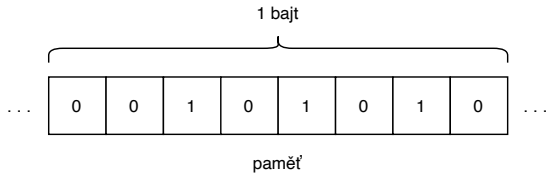
Jak již víme, jazyk Python, stejně jako většina jiných vysokoúrovňových programovacích jazyků, programátory odstiňuje od řady nízkoúrovňových záležitostí. Jednou z výjimek jsou bitové operace, jež umožňují manipulaci s daty na úrovni jednotlivých bitů, kterým se budeme věnovat v této kapitole.

Veškerá data v počítači jsou uložena ve formě jedniček a nul. Pokud například vytvoříme proměnnou obsahující celé číslo, je toto číslo uloženo v paměti v binární podobě. Nejmenší adresovatelnou jednotkou paměti je jeden bajt (8 bitů). Uvažme následující příklad.

```
a = 42
```

Samotná hodnota[66] 42 je fyzicky uložena v jednom bajtu paměti v binární podobě tak, jak je to ukázáno na obrázku 8.

[66] Nyní mluvíme skutečně o čísle 42 nikoliv o objektu, který dané číslo reprezentuje.



Obrázek 8: Reprezentace čísla 42 v paměti počítače.

Bitové operátory

V jazyce Python je pro manipulaci s jednotlivými bity k dispozici šest *bitových operátorů*, které jsou shrnuty v tabulce 2.
Tyto operátory pracují s binární reprezentací celých čísel a znaků.[67] Bitové operace jsou obvykle podporovány přímo procesorem počítače.[68] V důsledku toho je výpočet výsledků těchto operací velice rychlý. Postupně si jednotlivé operace vysvětlíme.

Průvodce studiem
Bitové operátory lze aplikovat pouze na celočíselné hodnoty a znaky (ty jsou reprezentovány pomocí celočíselných hodnot).

[67] Tedy s jejich skutečnou reprezentací v paměti počítače.

[68] Procesor má pro tyto operace speciální instrukce.

Operátor	Popis
`&`	bitový součin (AND)
`|`	bitový součet (OR)
`^`	bitový exklusivní součet (XOR, znak stříška)
`~`	bitová negace (NOT, znak tilda)
`<<`	bitový posun vlevo
`>>`	bitový posun vpravo

Tabulka 2: Bitové operátory jazyka Python a jejich popis.

Bitový součin

Při operaci bitového součinu se provede logický součin (AND) mezi korespondujícími bity. Například.[69]

```
# bitový součin
print(10 & 7) # vypíše: 2
```

[69] Postup výpočtu výrazu `10 & 7`:

$$\begin{aligned} \texttt{10} &= \texttt{00001010}_2 \\ &\quad\ \ \& \\ \texttt{7} &= \texttt{00000111}_2 \\ \hline \texttt{2} &= \texttt{00000010}_2 \end{aligned}$$

Bitový součet

Při operaci bitového součinu se provede logický součet (OR) mezi korespondujícími bity. Například.[70]

```
# bitový součet
print(10 | 7) # vypíše: 15
```

[70] Postup výpočtu výrazu `10 | 7`:

$$\begin{aligned} \texttt{10} &= \texttt{00001010}_2 \\ &\quad\ \ | \\ \texttt{7} &= \texttt{00000111}_2 \\ \hline \texttt{15} &= \texttt{00001111}_2 \end{aligned}$$

Exklusivní bitový součet

Při operaci bitového součinu se provede logický součet (XOR) mezi korespondujícími bity. Například.[71]

```
# exklusivní bitový součet
print(10 ^ 7) # vypíše: 13
```

[71] Postup výpočtu výrazu `10 ^ 7`:

$$\begin{aligned} \texttt{10} &= \texttt{00001010}_2 \\ &\quad\ \ \hat{} \\ \texttt{7} &= \texttt{00000111}_2 \\ \hline \texttt{13} &= \texttt{00001101}_2 \end{aligned}$$

Bitová negace

Bitová negace je unární operátor. Při operaci bitové negace je každý bit invertován (bit s hodnotou jedna je přepsán na hodnotu nula a naopak). Formálně je tato operace pro číslo a definována následovně: `~a = -(a + 1)`. Například.[72]

```
# bitová negace
print(~10) # vypíše: -11
```

[72] Přestože je bitová negace jednoduchou operací, výsledek vyhodnocení výrazu `~10` nemusí být na první pohled zřejmý. Důvodem je dvojkový doplňkový kód, který je použit pro reprezentaci záporných čísel v paměti počítače. Výsledná hodnota `-11` = $\texttt{11110101}_2$ je výsledkem bitové negace čísla `10` = $\texttt{00001010}_2$.

Bitový posun vlevo

Při operaci bitového posunu vlevo jsou bity v paměti počítače posunuty o zadaný počet bitů směrem doleva a zprava doplněny nulami. Například.

```
# bitový posun vlevo
print(10 << 2) # vypíše: 40
```

Ve výše uvedeném je číslo `10 = 1010`$_2$ zprava doplněno dvěma nulami, čímž dojde k danému posunu. Výsledkem je `101000`$_2$ `= 40`.
Bitový posun vlevo je možné použít k velmi rychlému násobení mocninou čísla dva.

```
42 << 1 # 84 = 42 * (2**1)
42 << 2 # 168 = 42 * (2**2)
42 << 3 # 336 = 42 * (2**3)
```

Bitový posun vpravo

Při operaci bitového posunu vpravo jsou bity v paměti počítače posunuty o zadaný počet bitů směrem doprava (bity se ztrácí) a zleva doplněny nulami. Například.

```
# bitový posun vpravo
print(10 >> 2) # vypíše: 2
```

Ve výše uvedeném jsou z čísla `10 = 1010`$_2$ odstraněny poslední dva bity (vpravo) a číslo je zleva doplněno dvěma nulami, čímž dojde k danému posunu. Výsledkem je `0010`$_2$ `= 2`.
Bitový posun vpravo je možné použít k velmi rychlému celočíselnému dělení mocninou čísla dva.[73]

[73] Za předpokladu, že je dané číslo dělitelné mocninou čísla dva beze zbytku.

```
48 >> 1 # 24 = 48 / (2**1)
48 >> 2 # 12 = 48 / (2**2)
48 >> 3 #  6 = 48 * (2**3)
```

Priorita operátorů

Bitové operátory, stejně jako ty aritmetické, mají svoji prioritu, která určuje pořadí jejich vyhodnocování ve výrazu. Priority všech operátorů jsou shrnuty v tabulce 3.
Připomeňme, že ve složitějších výrazech je lepší explicitně určit pořadí vyhodnocování jednotlivých operátorů pomocí závorek.

Priorita	Operátor	Popis
největší	`()`	závorky
	`[]`	indexace
	`**`	mocnina
	`~`	bitová negace
	`-`	unární mínus
	`*, /, %`	násobení, dělení, modulo
	`+, -`	operace plus, operace mínus
	`<<, >>`	bitový posun
	`&`	bitový součin
	`^`	exklusivní bitový součet
	`|`	bitový součet
	`in, not in, is, is not, <, <=, >, >=, <>, !=, ==`	operátory porovnání a identity
	`not`	logická negace
	`and`	logická konjunkce
	`or`	logická negace
nejmenší	`lambda`	lambda výraz

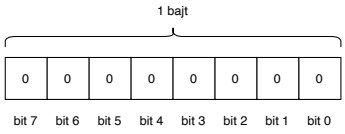
Tabulka 3: Priority operátorů jazyka Python. Operátory na stejném řádku mají stejnou prioritu.

Praktické použití

Pomocí bitových operátorů můžeme adresovat jednotlivé bity a dále s nimi pracovat. V následující ukázce jsou implementovány funkce `nastav_bit()` a `smaz_bit()`.[74]

```
def nastav_bit(cislo, bit):
  maska = 1 << bit
  return cislo | maska

def smaz_bit(cislo, bit):
  maska = ~(1 << bit)
  return cislo & maska

# použít funkce nastav_bit()
nastav_bit(0, 0) # vrací: 1 (0b1)
nastav_bit(0, 7) # vrací: 128 (0b10000000)

# použít funkce smaz_bit()
smaz_bit(255, 0) # vrací: 254 (0b11111110)
smaz_bit(255, 2) # vrací: 251 (0b11111011)
```

[74] Jednotlivé bity jež jsou použity pro reprezentaci daného čísla jsou číslovány od nuly zprava doleva. Například.



Bitové operátory nám umožňují efektivnější uložení informací. Následující příklad implementuje klasický učebnicový příklad repre-

zentace data (den, měsíc, rok) pomocí čísel. Při vhodném nastavení je možné datum reprezentovat jako dvoubajtové číslo.[75]

[75] Při běžné reprezentaci jsou potřeba 4 bajty (jeden pro měsíc, jeden pro den a dva pro rok).

```
POCATEK = 2020
POCET_BITU_DEN = 5
POCET_BITU_MESIC = 4
POCET_BITU_ROK = 7

def preved_datum_na_cislo(den, mesic, rok):
  datum = den
  datum <<= POCET_BITU_MESIC
  datum |= mesic
  datum <<= POCET_BITU_ROK
  datum |= (rok - POCATEK)
  return datum

def preved_cislo_na_datum(datum):
  rok = POCATEK + (datum & 2**POCET_BITU_ROK-1) # rok
  mesic = (datum >> POCET_BITU_ROK) & 2**POCET_BITU_MESIC-1 # měsíc
  den = (datum >> (POCET_BITU_MESIC + POCET_BITU_ROK)) & 2**POCET_BITU_DEN-1 # den
  print(f"{den}. {mesic}. {rok}")

datum = preved_datum_na_cislo(4, 8, 2112)
preved_cislo_na_datum(datum)

print(datum) # vypíše: 9308
print(bin(datum)) # vypíše: 0b10010001011100
```

Trik, který umožňuje datum reprezentovat jako dvoubajtové číslo, spočívá v tom, že neukládáme číslo roku, ale pouze rozdíl roku a pevně zvoleného počátku (konstanta `POCATEK`).

Shrnutí

V této kapitole jsme se seznámili s bitovými operátory, které pracují s binární reprezentací celých čísel a znaků v paměti počítače.

Kontrolní otázky
Odpovězte na následující otázky:

1. K čemu slouží bitové operátory?
2. Jak lze přistoupit k jednotlivým bitům?

Úkoly

Úkol 18
Napište funkci `pocet_bitu(x)`, která vrátí počet bitů potřebných pro reprezentaci celého čísla `x`.

Úkol 19
Napište funkci `rotuj_doprava(x, n)`, která posune číslo `x` o `n` bitů doprava a tyto posunuté bity jsou přidány na začátek v pořadí, ve kterém byly odsunuty. Například `rotuj_doprava(0b10010011, 2)` vrátí `0b11100100`.

Úkol 20
Napište funkci `invertuj_doprava(x, p)`, která v čísle `x` invertuje všechny bity od pozice `p`. Ostatní bity zůstanou nezměněny.

Úkol 21
Napište funkci `preved(znak)`, která pomocí bitových operací převede `znak` na velké písmeno `A-Z`, pokud je `znak` malé písmeno `a-z`, nebo `znak` převede na malé písmeno `a-z`, pokud je `znak` velké písmeno `A-Z`.[76]

[76] Nápověda: velká a malá písmena se liší v pátém bitu.

Práce se soubory

V této kapitole popíšeme způsob práce se soubory v počítačových programech.

Práce se soubory je ve většině programovacích jazyků velmi podobná. Každý soubor musí být nejprve otevřen. Tímto krokem získá program přístup k obsahu souboru, přičemž k souboru je možné přistoupit v režimu (i) čtení, (ii) zápisu nebo (iii) čtení a zápisu.
Následně probíhá samotná práce se souborem.[77] Ta by měla být vždy co nejkratší, jelikož od okamžiku, kdy byl soubor otevřen, může být (záleží na zvoleném režimu) blokován přístup k tomuto souboru jiným programům.[78] Soubory otevřené v režimu čtení mohou být po dobu práce se souborem čteny dalšími programy, nelze ale do nich zapisovat. Soubory otevřené v režimu umožňující zápis do souboru nejsou přístupné ostatním programům.
Po skončení práce se souborem je zapotřebí otevřený soubor uzavřít. Tím dojde k „uvolnění“ souboru a ostatní programy k němu mohou přistupovat.
Práce se soubory je obecně časově náročná.[79] Operační systémy se snaží tyto operace, zejména zápis do souboru, co nejvíce optimalizovat. Jedna z nejčastějších technik je použití tzv. *bufferu*.[80] Například při zápisu do souboru jsou změny ukládány do bufferu, který je při jeho zaplnění vyprázdněn, čímž dojde k provedení požadovaných změn. Teprve v tento okamžik je otevřený soubor změněn. Pokud by náš program skončil chybou ještě před vyprázdnění bufferu, změny by nebyly uloženy. Při uzavření souboru dojde vždy k vyprázdnění bufferu a zapsání změn do souboru.

[77] V případě, že je soubor otevřen v režimu čtení, je možné jej pouze číst a nelze do něj zapisovat. V režimu zápisu je možné do souboru zapisovat, ale nelze jej číst. Režim čtení a zápisu umožňuje obě operace.

[78] Blokování přístupu k souboru, jež je zajištěno operačním systémem, zabraňuje vzniku nekonzistencí v obsahu souboru.

[79] Přistupuje se k pevnému disku počítače, tedy k jedné z jeho nejpomalejších částí.

[80] Vyrovnávací paměť.

Otevření souboru

Pro otevření souboru slouží funkce `open()`, která akceptuje jako parametr adresu souboru, jenž má být otevřen, a režim[81] přístupu k danému souboru. Funkce `open()` vrací objekt reprezentující daný soubor, se kterým je možné dále pracovat. Pro objekt reprezentující soubor se běžně používá termín *handler*. Jednotlivé módy, které lze při práci se soubory využít, jsou shrnuty v tabulce 4.

[81] Případně *mód*.

Režim	Popis
`r`	čtení (výchozí mód), neexistující soubor je vytvořen
`a`	přidání, neexistující soubor je vytvořen
`w`	zápis, neexistující soubor je vytvořen, existující je přepsán
`x`	vytvoří soubor, pokud soubor existuje nastane chyba
`r+`	stejné jako režimy `r` a `a`
`a+`	stejné jako režimy `r` a `a`
`w+`	stejné jako režimy `r` a `w`
`t`	čtení/zápis po znacích (výchozí mód)
`b`	čtení/zápis po bajtech

Tabulka 4: Jednotlivé režimy práce se souborem a jejich popis. Písmena označující režim jsou odvozeny z anglických slov. Režimy `r+` a `a+` se liší výchozí polohou kurzoru (viz dále).

Otevření souboru v režimu čtení po znacích, což je výchozí režim, se provádí následovně.

```
# otevření souboru pro čtení po znacích (výchozí)
f = open("soubor.txt")

# stejné jako předchozí
f = open("soubor.txt", "rt")
```

Oba výše uvedené zápisy jsou ekvivalentní. Objekt reprezentující otevřený soubor poskytuje řadu metod umožňující práci s tímto souborem.
Otevřený soubor je možné uzavřít pomocí metody `.close()`.

```
# uzavření souboru
f.close()
```

Průvodce studiem
Každý programem otevřený soubor by jím měl být také uzavřen. Uzavření souboru by mělo být provedeno ideálně v okamžiku, kdy se souborem již nebude zapotřebí pracovat.

Čtení ze souboru

Pro čtení souboru (v textovém režimu) slouží metody `.read()` a `.readline()`. V následujících ukázkách budeme předpokládat, že existuje soubor s názvem `soubor.txt`, který obsahuje

```
Programování
v Pythonu je zábava.
```

Soubor je umístěn ve stejném adresáři, ve kterém se nachází náš program.[82]

[82] Pokud by byl soubor umístěn jinde, je třeba funkci `open()` předat cestu k tomuto souboru. Lze použít jak relativní tak absolutní cestu.

Metoda `.read()` přečte celý soubor (od začátku do konce) a vrátí jej v podobě textového řetězce.[83]

```
f = open("soubor.txt")
print(f.read())
f.close()
```

[83] **Výstup programu:**

```
Programování
Pythonu je zábava.
```

Program vypíše textový řetězec včetně znaku `\n`. Výpis tedy bude skutečně na dva řádky. Metoda `.read()` akceptuje jako parametr počet znaků, které mají být ze souboru přečteny.[84]

```
f = open("soubor.txt")
print(f.read(7)) # přečteme 7 znaků
f.close()
```

[84] **Výstup programu:**

```
Program
```

Při čtení souboru je uchován *kurzor* (ukazatel) na doposud nepřečtený znak. Při dalším čtení ze souboru se pokračuje od tohoto kurzoru.[85]

```
f = open("soubor.txt")
print(f.read(7)) # přečteme 7 znaků
print(f.read(7)) # přečteme dalších 7 znaků
print(f.read(7)) # přečteme dalších 7 znaků
f.close()
```

[85] **Výstup programu:**

```
Program
ování
v
 Python
```

Jakmile je celý soubor přečten, metoda `.read()` vrací prázdný řetězec. Pokud bychom chtěli nějakou část souboru číst opakovaně, je třeba přesunout kurzor na tuto část. Pro přesun kurzoru slouží metoda `.seek()` akceptující pořadové číslo znaku, na který má být kurzor nastaven. Například následující kód dvakrát přečte obsah souboru `soubor.txt`.[86]

```
f = open("soubor.txt")
print(f.read()) # přečte celý soubor
f.seek(0) # nastavíme kurzor na začátek souboru
print(f.read()) # znovu přečteme celý soubor
f.close()
```

[86] **Výstup programu:**

```
Programování
v Pythonu je zábava.
Programování
v Pythonu je zábava.
```

Analogicky jako lze číst celý soubor metodou `.read()`, je možné číst soubor po jednotlivých řádcích pomocí metody `.readline()`. Rovněž je možné metodě `.readline()` předat počet znaků, které mají být z řádku přečteny. Pokud již není žádný další nepřečtený řádek, případně znaky na řádku, vrací metoda `.readline()` prázdný řetězec.

```
f = open("soubor.txt")
print(f.readline()) # vypíše první řádek souboru
```

```
f.close()
```

Následující kód ukazuje, jak je možné přečíst celý soubor řádek po řádku.

```
f = open("soubor.txt")
radek = f.readline()
while radek:
  print(radek, end="")
  radek = f.readline()
f.close()
```

Případně je možné využít cyklus `for`. Tento zápis ale uvedeme pouze pro úplnost.

```
f = open("soubor.txt")
for radek in f:
  print(radek)
f.close()
```

Výše uvedený kód může vypadat poněkud zvláštně, jelikož neobsahuje žádnou metodu pro čtení souboru.[87]

[87] Pro jednoduchost si můžeme představit, že objekt reprezentující soubor se chová jako kolekce jednotlivých řádků v souboru. Detailní vysvětlení a pochopení tohoto kódu vyžaduje znalosti objektově orientovaného programování, které bude představeno v kurzu *Základní programovací paradigmata*. Nyní se spokojíme s tím, že to takto funguje.

Zápis do souboru

Aby bylo možné do souboru zapisovat, je zapotřebí jej otevřít v režimu umožňující zápis. Zápis probíhá vždy na aktuální pozici kurzoru. Pokud je soubor otevřen v režimu `a` (přidávání), je zachován původní obsah tohoto souboru a kurzor je nastaven na poslední znak v tomto souboru. V případě, že je soubor otevřen v režimu `w`, je původní obsah souboru smazán a kurzor je přesunut na začátek tohoto souboru.[88]

[88] V tomto režimu dojde ke smazání obsahu souboru vždy, tedy i tehdy pokud soubor pouze otevřeme a uzavřeme (aniž bychom provedli cokoliv dalšího).

V režimech `r+`, `a+` a `w+` je soubor otevřen pro čtení a zápis. V případě `r+` a `w+` je kurzor nastaven na první znak v souboru, při použití `w+` je navíc původní soubor přepsán. V režimu `a+` je kurzor nastaven na poslední znak v souboru.

Pro zápis textového řetězce do souboru slouží metoda `.write()`. Následující ukázka vytvoří soubor, se kterým jsme doposud pracovali.

```
f = open("soubor.txt", "w")
f.write("Programování\n")
f.write("v Pythonu je zábava.")
f.close()
```

Ošetření chyb při práci se soubory

Doposud jsme při práci se soubory vůbec nepočítali s možnými chybami. K těm může dojít naprosto běžně. Chyby při práci se soubory mohou být zapříčiněny celou řadou okolností. Například program nemá dostatečná oprávnění pracovat se souborem, soubor je poškozen, soubor je blokován jiným programem, chyba při přenosu dat[89] a další. Při práci se soubory je vždy nutné s těmito chybami počítat.

Průvodce studiem
Chyby při práci se soubory se běžně vyskytují. Programátor s nimi musí vždy počítat.

[89] V případě, že je soubor uložen na síťovém disku.

Ošetření chyb při práci se soubory je možné provést pomocí příkazů `try`, `except` a `finally` tak, jak jsme to již ukázali. Veškerá práce se souborem (včetně jeho otevření) by měla být umístěna v try-bloku. V případě, že dojde k chybě (ke vzniku výjimky), je třeba všechny otevřené soubory uzavřít.[90] To lze provést v except-bloku vymezeném přidruženým příkazem `except`.

[90] Bez tohoto by mohl již neběžící program blokovat přístup k souboru ostatním programům.

```
try:
  # práce se souborem
  f = open("soubor.txt", "w+")
  f.write("Programování\n")
  f.write("v Pythonu je zábava.")
  f.close() # uzavření souboru
except:
  # ošetření výjimek
  print("Chyba při práci se souborem.")
  # uzavření souboru při výjimce
  f.close()
```

Mnohem výhodnější je uzavření otevřených souborů v bloku vymezeném přidruženým příkazem `finally`, jelikož je tento blok vykonán bez ohledu na to, zda došlo či nedošlo k vyvolání výjimky. Tím se vyhneme redundantnímu volání metody `.close()`.

```
try:
  # práce se souborem
  f = open("soubor.txt", "w+")
  f.write("Programování\n")
  f.write("v Pythonu je zábava.")
except:
  # ošetření výjimek
  print("Chyba při práci se souborem.")
finally:
  # uzavření souboru
  f.close()
```

Dodejme, že jazyk Python podporuje různá kódování. Například při práci se soubory je možné používat i unicode znaky.[91]

[91] Pro zápis unicode znaků se používá escape sekvence \u pro 16-bitové znaky a \U pro 32-bitové znaky. Například `print(\U0001F600)` vytiskne (smějícího se) smajlíka.

Práce s binárními soubory

Binární soubory, na rozdíl od textových souborů, neukládají data po jednotlivých znacích ale po bajtech. Předpokládejme, že chceme uložit číslo 42. V textovém souboru zabírá toto číslo 2 bajty.[92] Číslo 42 lze uložit do jednoho bajtu.[93]
V jazyce Python je bajt imutabilní datový typ a lze jej vytvořit pomocí funkce `bytes()`, která akceptuje seznam hodnot (0–255) ze kterých je vytvořena posloupnost bitů. Například.

[92] Každý znak (tedy znak 4 a znak 2) zabírá jeden bajt.

[93] $42_{10} = 101010_2$

```
bytes([0]) # 00000000
bytes([16]) # 00010000
bytes([42]) # 00101010 (42)
bytes([255]) # 11111111
bytes([255, 255]) # 1111111111111111 (dva bajty)
```

Pro zápis a čtení binárních souborů se používají již dříve popsané metody, je ale zapotřebí je otevřít v režimu `b`.[94] Například zápis jednoho bajtu do již otevřeného souboru.

[94] V tomto režimu čtení a zápis probíhá po jednotlivých bajtech.

```
f.write(bytes([42])) # zapíšeme jeden bajt
```

Následuje příklad přečtení jednoho bajtu ze souboru.[95]

[95] Funkce `ord()` je použita pouze pro účely výpisu. Bez ní bude vypsána hodnota bajtu ve tvaru `b"*"`. Důvod pro tento, na první pohled zvláštní, výpis je ten, že interpret jazyka Python se snaží pro výpis dat používat ASCII tabulku. V té má znak `*` hodnotu 42.

```
f = open("soubor.bin", "rb")
print(ord(f.read()))
f.close()
```

Výpis můžeme vylepšit pomocí formátovacího řetězce.[96]

[96] `:b` zobrazí hodnotu v binární reprezentaci, `:x` zobrazí hodnotu v hexadecimální reprezentaci.

```
f = open("soubor.txt", "rb")
bajt = ord(f.read(1))
print(f"{bajt}") # vypíše: 42
print(f"{bajt:x}") # vypíše: 2a
print(f"{bajt:b}") # vypíše: 101010
f.close()
```

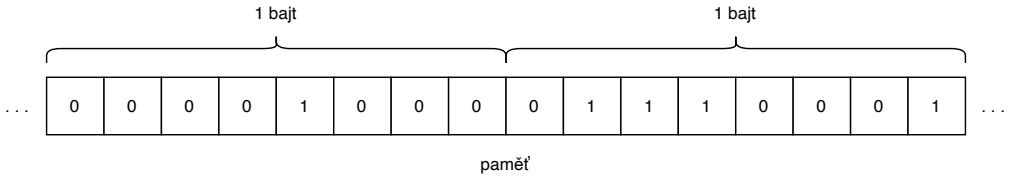
Rozdělení na více bajtů

Čísla, která zabírají více než jeden bajt, jsou v paměti uložena za sebou. Například.

```
a = 2161
```

Hodnota 2161[97] je uložena tak, jak je to ukázáno na obrázku 9.

[97] $2161_{10} = 100001110001_2$



Obrázek 9: Reprezentace čísla 2161 v paměti počítače.

Pořadí jednotlivých bajtů, tzv. *endianita*, je určené architekturou počítače. Pro lepší názornost jsme použili Big-endian,[98] který je ale na rozdíl od Little-endian,[99] méně používaný.
Pokud tedy chceme uložit číslo 2161 do souboru, musíme jej rozdělit na jednotlivé bajty a ty následně uložit do souboru. Následující funkce realizuje zmíněné rozdělení.

[98] Nejvíce významný bajt je uložen jako první (ukládá se zleva doprava).

[99] Nejméně významný bajt je uložen jako první (ukládá se zleva doprava).

Průvodce studiem
Endianita je nejčastější příčina nekompatibility binárních souborů na různých architekturách.

```
def preved_cislo_na_bajty(cislo):
  bajty = []
  if cislo:
    while cislo:
      bajt = cislo & 0xff
      bajty.append(bajt)
      cislo = cislo >> 8
  else:
    bajty.append(0)

  return bajty[::-1]
```

Funkce pomocí bitových operací rozdělí číslo po jednotlivých bajtech a vrátí seznam hodnoty (čísla 0–255) těchto bajtů. Výsledek můžeme zapsat do binárního souboru. Například.

```
cislo = preved_cislo_na_bajty(9308) # vrátí: [36, 92]

try:
  f = open("soubor.bin", "wb")
  f.write(bytes(cislo))
except:
  # ošetření výjimek
  print("Chyba při práci se souborem.")
finally:
  # uzavření souboru
  f.close()
```

Dodejme, že pokud soubor otevřeme v textovém režimu a přečteme jeho obsah, bude obsahovat dva znaky: $\ (dolar a zpětné lomítko). Binární soubory obvykle zabírají výrazně méně místa než jejich textové protějšky. Navíc je práce s nimi rychlejší. Velkou výhodou binárních souborů je možnost libovolného strukturování souboru.[100]

[100] Pěkný příklad jsme již uvedli v kapitole Bitové operace, kde jsme ukázali uložení data do dvou bajtů.

Zjevnou nevýhodou binárních souborů je absence jejich přímé čitelnosti, například pomocí textového editoru. Toto ale v některých případech, například při ukládání obrázků, není nikterak omezující.

Shrnutí

V této kapitole jsme popsali, jakým způsobem lze v jazyce Python pracovat s textovými a binárními soubory.

Kontrolní otázky
Odpovězte na následující otázky:

1. Jakým způsobem se pracuje se soubory?
2. Co je to režim přístupu k souboru?
3. Proč je třeba soubor uzavřít?
4. Jak se pracuje s binárními soubory?

Úkoly

Úkol 22
Napište funkci `uloz_matici(soubor, M)`, která uloží číselnou matici (reprezentovanou jako seznam seznamů) do souboru v CSV formátu.[101] Například matice `M = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]` bude uložena v souboru následovně.

```
1,2,3
4,5,6
7,8,9
```

[101] Každý řádek matice je uložen na samostatném řádku. Jednotlivé hodnoty matice jsou odděleny čárkou s výjimkou poslední hodnoty na řádku.

Úkol 23
Napište funkci `nacti_matici(soubor)`, která načte matici uloženou ve formátu z předchozího úkolu.

Úkol 24
Napište program, který pomocí cyklu `while` přečte celý soubor po znacích.

Úkol 25
Napište funkci `preved_cislo_na_bajty(x, pocet_bajtu)`, která zadané číslo `x` uloží do `pocet_bajtu` bajtů. Pokud se číslo do zadaného počtu nevejde, je uložena pouze část čísla, která se vejde do daného počtu bajtů a je ohlášena výjimka.[102]

[102] Dojde k přetečení čísla.

Úkol 26
Napište funkci, `zapis_cisla(soubor, pocet_bajtu, cisla)`,

která zapíše seznam čísel (`cisla`) do binárního souboru. Jako první je do souboru uložena (do jednoho bajtu) hodnota `pocet_bajtu` a následně jednotlivá čísla zabírající odpovídající počet bajtů. V těle funkce použijte funkci z předchozího příkladu.

Úkol 27
Napište funkci `preved_bajty_na_cislo(bajty)`, která zadané bajty převede na číslo.

Úkol 28
Napište funkci `precti_cisla(soubor)`, která přečte čísla ze zadaného souboru.[103] Funkce nejprve zjistí počet `pocet_bajtu` použitý v souboru a následně načte odpovídající bajty a ty pomocí funkce z předchozího úkolu převede na čísla, jež vrátí jako seznam.

[103] Soubor je vytvořen funkcí z Úkolu 26.

Organizace zdrojového kódu

V závěrečné kapitole tohoto textu popíšeme způsob, jakým lze v jazyce Python organizovat zdrojový kód.

Doposud jsme používali funkce jako základní nástroj pro programátorskou abstrakci, čímž jsme přirozeně vytvářeli opakovaně použitelné funkce v rámci jednoho programu (jednoho souboru). Pro udržení pořádku ve zdrojovém kódu je někdy žádoucí rozdělit jej do logických celků a ty uložit do různých souborů, které jsou do našeho programu vkládány. Například různé pomocné funkce můžeme vyčlenit do samostatného souboru.
Pro ucelenou sadu funkcí,[104] které slouží k řešení specifického problému, se používá termín *knihovna*. Knihovny tedy nejsou z našeho pohledu ničím novým. Přináší ale výraznou přehlednost ve zdrojovém kódu. V následující části si představíme, jakým způsobem je možné v jazyce Python knihovny vytvářet.

[104] Obecně se nemusí jednat pouze o funkce, ale i další struktury.

Průvodce studiem
Knihovny jsou dalším stupněm programátorské abstrakce.

Moduly

V jazyce Python *modul* označuje soubor s příponou `.py`, který obsahuje příkazy jazyka. Jedná se tedy o klasický zdrojový kód, jakých jsme již napsali stovky. Na rozdíl od nich se ale o modulu předpokládá, že nebude přímo spuštěn,[105] ale jeho obsah bude vložen do jiného zdrojového kódu. Například vytvoříme soubor `matematika.py` s následujícím obsahem.

[105] Byť je to samozřejmě možné.

```
PRESNOST = 0.00000001

# výpočet druhé odmocniny čísla x pomocí Newtonovy metody
# https://cs.wikipedia.org/wiki/Metoda_tečen
def odmocnina(x):
  odhad = x
  while True:
    novy_odhad = 0.5 * (odhad + x / odhad)
    if abs(novy_odhad - odhad) < PRESNOST:
      return novy_odhad
    odhad = novy_odhad
```

```
# výpočet n-té mocniny čísla x
def mocnina(x, n):
  return float(x**n)

# test funkcí
print(odmocnina(2)) # vypíše: 1.414213562373095
print(odmocnina(4)) # vypíše: 2.0
print(mocnina(2,2)) # vypíše: 4.0
```

Jedná se tedy o běžně spustitelný program, který obsahuje jednu konstantu, dvě funkce a volání těchto funkcí.

Vložení modulu

Soubor `matematika.py`, respektive jeho obsah, můžeme vložit do jiného zdrojového kódu pomocí příkazu `import`.[106]

[106] Příkaz dokáže vložit pouze soubory s příponou `.py`. Ta se v příkaze import nepíše.

```
import soubor
```

Tento příkaz způsobí importování souboru `soubor` za předpokladu, že nebyl doposud importován. Při importování dojde k volání programu zapsaného v `soubor` a vytvoření nové proměnné, se jménem odpovídajícím názvu souboru (`soubor`), skrze něhož je možné přistupovat k celému jeho obsahu. Přístup k jeho obsahu[107] se provádí podobně jako se používají metody, tedy pomocí symbolu `.` (tečka). Příklad následuje.[108]

[107] Definované funkce a proměnné.

[108] Předpokládáme, že importovaný soubor `matematika.py` je umístěn ve stejné složce, jako náš program.

```
# importování souboru matematika.py
import matematika

# zavolání funkce odmocnina
matematika.odmocnina(2)

# výpis konstanty PRESNOST
print(matematika.PRESNOST)
```

Jméno pomocí kterého přistupujeme k obsahu souboru (v našem případě `matematika`), se běžně označuje jako *namespace* (jmenný prostor).[109] Ten nám umožňuje udržovat přehled o původu obsahu a především odděluje zdrojový kód importovaného programu od zdrojového kódu našeho programu, případně jiných importovaných programů.
Jméno jmenného prostoru je možné definovat v příkazu `import`.

[109] V jazyce Python je možné, že obsah objektu (jeho vlastnosti) je možné si prohlédnou příkazem `dir(objekt)`. Například `dir(matematika)` vrátí seznam proměnných (a tedy i funkcí) definovaných v souboru `matematika.py`.

```
import soubor as jmeno
```

Například.

```
import matematika as m

m.odmocnina(2)
```

Při použití příkazu `import` dochází k volání importovaného programu. `matematika.py` obsahuje několik výpisů pomocí funkce `print()`. Ty můžeme vidět v programu, do kterého soubor `matematika.py` importujeme. V našem případě je to spíše nežádoucí,[110] ale v mnoha případech se může jednat o užitečnou vlastnost.[111]
V jazyce Python je možné importovat pouze část importovaného souboru, například pouze jednu funkci nebo proměnnou.

[110] Jednou možností je tyto výpisy v modulu neuvádět. Jazyk Python má prostředky pro potlačení volání programu v případě, že je importován. Ty ale potřebovat nebudeme.

[111] Program můžeme rozdělit na menší logické celky.

```
from soubor import část
```

Následující kód importuje ze souboru `matematika.py` pouze funkci `odmocnina()`.

```
from matematika import odmocnina
```

Během tohoto typu importu nedochází k volání programu[112] a vytvoření jmenného prostoru. Importovaná část je přímo vložena do programu.

[112] Výpisy v souboru `matematika.py` se nezobrazí.

```
from matematika import odmocnina

odmocnina(2)
```

Importovanou část programu je možné přejmenovat (pomocí `as`) a je možné importovat více částí současně. Například.

```
# import a přejmenování více částí programu
from matematika import odmocnina as f1, mocnina as f2

f1(2) # volání funkce odmocnina()
f2(2, 2) # volání funkce mocina()
```

V případě, že chceme importovat celý obsah souboru, lze použít symbol `*` (hvězdička). Například.

```
from matematika import *
```

Pro úplnost dodejme, že při použití `from` příkazu dochází k importování pouze specifikovaných částí.

```
from matematika import odmocnina

# funkce funguje správně (má přístup ke konstantě PRESNOST)
odmocnina(2) # vypíše: 1.414213562373095

# náš program nemá přístup ke konstantě PRESNOST
print(PRESNOST) # způsobí chybu: NameError: name 'PRESNOST'
    is not defined
```

Pro lepší organizaci zdrojových kódů můžeme moduly umísťovat do adresářů.[113] V příkazech `import` a `from` se cesta zapisuje pomocí symbolu `.` (tečka) a stává se součástí jmenného prostoru. Například přesuneme soubor `matematika.py` do složky `moje_knihovna` a importujeme jej do programu.

[113] V terminologii jazyka Python se moduly uložené v adresářích označují jako balíčky (packages).

```
import moje_knihovna.matematika

# použití
moje_knihovna.matematika.odmocina(2)
```

Případně.

```
import moje_knihovna.matematika as m

# použití
m.odmocina(2)
```

Existující knihovny

Součástí programovacího jazyka Python je mnoho existujících knihoven.[114] Ty je možné vkládat stejným způsobem, jaký jsme popsali výše. Jako příklad uveďme knihovnu `math` implementující základní matematické operace.

[114] Jejich úplný přehled je k dispozici na stránce `https://docs.python.org/3/library/`.

```
# import knihovny math (součást jazyka Python)
import math

# druhá odmocnina
print(math.sqrt(2))
```

Shrnutí

V závěrečné kapitole jsme ukázali jakým způsobem je možné vkládat do programu existující zdrojové kódy, či jejich části a tím organizovat kód.

Kontrolní otázky
Odpovězte na následující otázky:

1. Proč je důležité organizovat zdrojový kód do souborů?
2. Co je to modul?
3. Jak pracuje příkaz `import`?

Dodatek

Učit se programovat je s trochou nadsázky nikdy nekončící proces. Kurzy *Základy programování pro IT 1* a *Základy programování pro IT 2* jsou pouze začátkem na této dlouhé cestě. Na úplný závěr tohoto textu uveďme The Zen of Python, také známý jako PEP20, jehož autorem je Tim Peters.[115]

[115] PEP20 je sada 19 neformální doporučení, které by měl respektovat každý programátor a to ne jen při psaní programů v jazyce Python.

1. Beautiful is better than ugly.
2. Explicit is better than implicit.
3. Simple is better than complex.
4. Complex is better than complicated.
5. Flat is better than nested.
6. Sparse is better than dense.
7. Readability counts.
8. Special cases aren't special enough to break the rules.
9. Although practicality beats purity.
10. Errors should never pass silently.
11. Unless explicitly silenced.
12. In the face of ambiguity, refuse the temptation to guess.
13. There should be one—and preferably only one—obvious way to do it.
14. Although that way may not be obvious at first unless you're Dutch.
15. Now is better than never.
16. Although never is often better than right now.
17. If the implementation is hard to explain, it's a bad idea.

18. If the implementation is easy to explain, it may be a good idea.
19. Namespaces are one honking great idea—let's do more of those!

Řešení vybraných úkolů

„I hear and I forget. I see and I remember. I do and I understand.“
Confucius

Všechny úkoly uvedené v tomto textu jsou jednoduché či dokonce triviální. Jejich vyřešení ale může stát nemalé množství času a úsilí a to zejména začínající programátory. Naučit se programovat znamená toto úsilí vynaložit.
Řešení téměř každého zde uvedeného úkolu je možné nalézt na Internetu. Tato řešení ale obvykle obsahují pokročilejší konstrukce, jsou obecná či naopak příliš konkrétní a co hůř mnohdy obsahují chyby. Hledání řešení, a to ať už v této kapitole nebo na Internetu, je dobré nechat na úplný konec,[116] až je úloha vyřešena vlastními silami.
Dodejme, že mohou existovat i jiná řešení, ne nutně horší, než ta, která jsou zde uvedena.

[116] Rozhodně jej ale není dobré vynechat.

Řešení úkolu 4:

```
def aplikuj(f, sekvence):
  delka_sekvence = len(sekvence)

  if delka_sekvence == 0:
    return []
  elif delka_sekvence == 1:
    return sekvence[0]
  else:
    value = sekvence[0]
    for element in sekvence[1:]:
      value = f(value, element)
    return value

aplikuj(lambda x, y: x + y, [1, 2, 3, 4, 5])
```

Řešení úkolu 6:

```
def linearni_funkce(a, b):
```

```
  def vysledek(x):
    return a*x + b
  return vysledek

f1 = linearni_funkce(4, 2)
print(f1(2))
```

Řešení úkolu 7:[117]

```
def vytvor_prumer():
  hodnoty = [] # closure

  def prumer(x):
    hodnoty.append(x)
    soucet = sum(hodnoty)
    return soucet/len(hodnoty)

  return prumer

# ověření funkčnosti
p = vytvor_prumer()
print(p(10))
print(p(11))
print(p(12))
```

[117] Při řešení úkolu je výhodné použít closure (uzávěr).

Řešení úkolu 14:

```
def najdi_ve_stromu(uzel, x):
  while uzel:
    if x < uzel[HODNOTA]:
      uzel = uzel[LEVY_POTOMEK]
    elif x > uzel[HODNOTA]:
      uzel = uzel[PRAVY_POTOMEK]
    else:
      return True

  return False
```

Rejstřík