IB111 Základy programování (referenční příručka)

P. Ročkai

Část 1: Jazyk první kapitoly

Jak jsme již v předchozí kapitole zmínili, v tomto kurzu budeme programovat v omezené podmnožině jazyka Python. Každá kapitola v úvodní části představí všechny jazykové prostředky, které dosud neznáte.

1 Výrazy Výrazy v Pythonu intuitivně odpovídají výrazům, které znáte z matematiky: skládají se z **konstant**, **proměnných**, **operátorů**, závorek a **volání funkcí** (o funkcích detailněji níže). Každý výraz má **hodnotu**, a smyslem výrazů je kompaktně popsat výpočet této hodnoty. Příklady:

```
a + 7
4
3 + 3 * 2
(3 + 3) * 2
a + 1 > 7
count ** 2 < 100
```

K dispozici máme tyto základní **binární** operátory (mají vždy dva operandy):

- aritmetické (odpovídají obvyklým matematickým operacím):
  - `a + b`, `a - b` – sčítání a odečítání,
  - `a * b` – násobení,
  - `a // b`, `a % b` – celočíselné dělení a zbytek po dělení (připouštíme pouze pro dva celočíselné operandy),
  - `a / b` – dělení s desetinným výsledkem (naopak připouštíme pouze v případě, kdy alespoň jedno z `a`, `b` je číslo s plovoucí desetinnou čárkou – `float`),
  - `a ** b` – mocnění a^b,
- relační (význam opět známe z matematiky):
  - `a == b` – rovnost,
  - `a != b` – různost / nerovnost,
  - `a > b`, `a < b` – ostré nerovnosti,
  - `a >= b`, `a <= b` – neostré nerovnosti,
- logické (odpovídají logickým spojkám):
  - `a and b` – logická konjunkce: platí `a` a `b` zároveň (vyhodnotí-li se `a` na `False`, podvýraz `b` **nebude vůbec vyhodnocen** protože již nemůže výsledek ovlivnit),
  - `a or b` – logická disjunkce: platí alespoň jedno z `a`, `b` (podobně, vyhodnotí-li se `a` na `True`, podvýraz `b` se **nevyhodnocuje**).

Navíc jsou k dispozici dva **unární** operátory (mají pouze jeden operand):

- `-a` – opačná hodnota,
- `not a` – logická negace.

Výrazem je také tzv. **ternární operátor**, který má podobu `x if cond else y` – vyhodnotí-li se podvýraz `cond` na pravdivou hodnotu, celý výraz se vyhodnotí na výsledek podvýrazu `x`, v opačném případě na výsledek `y` (nepoužitý podvýraz se **nevyhodnocuje**).

Několik dalších operátorů (resp. nových významů stejných operátorů) ještě přibude v příštích týdnech.

2 Příkazy Dalším stavebním prvkem programu je **příkaz**, který odpovídá pokynu k provedení nějaké akce. Nejjednodušší příkaz je tvořen **libovolným výrazem** (užitečnost takových příkazů úzce souvisí s podprogramy, které nejsou čistými funkcemi, obzvláště pak s **procedurami**). Efektem takového příkazu je, že program vypočte jeho hodnotu a pak ji zapomene.

Druhým základním typem příkazu je **přiřazení**, které podobně jako v předchozím případě **vypočte hodnotu** výrazu, ale na rozdíl od předchozího si ji zároveň **zapamatuje** a **pojmenuje**. Takto pojmenovanou hodnotu – **proměnnou** – pak můžeme s výhodou použít v pozdějších výrazech.[1] V obou případech platí, že 1 řádek = 1 příkaz.

Přiřazení zapisujeme jako `jméno = výraz`, například:

```
a = 2
b = a + 1
b = -b
average = (a + b) / 2
positive = a > 0
```

Krom obyčejného přiřazení můžeme použít ještě tzv. **složené přiřazení**, které umožňuje zápis některých častých operací zkrátit. Tato složená přiřazení zapisujeme (věnujte pozornost závorkám a rozdílu mezi `/` a `//`):

složené přiřazení	ekvivalentní zápis
`a += 2`	`a = a + 2`
`x -= 2 * b`	`x = x - (2 * b)`
`a *= b + 2`	`a = a * (b + 2)`
`x /= a + b`	`x = x / (a + b)`
`x //= 3`	`x = x // 3`

Pozor! Znak `=` v přiřazení **není operátor** a přiřazení **není výraz** – např. zápis `(a = b) + 3` nepřipouštíme.

Posledním typem příkazu, který zde uvedeme, je tzv. **tvrzení**, které vyhodnotí zadaný výraz a je-li tento pravdivý, neudělá nic. V opačném případě ukončí program s chybou. Příklad:

```
assert x > 0
```

Tento příkaz budete prozatím potkávat zejména v přiložených testech.

3 Řízení toku Krom výpočtu a zapamatování si hodnot potřebujeme pro zápis algoritmů ještě **rozhodování** a **opakování**. K tomu slouží **příkazy toku řízení**, konkrétně `if`, `for` a `while`.

Příkaz `if` realizuje rozhodnutí na základě **pravdivostní hodnoty** (výrazu). Nejjednodušší forma je:

[1] Samotné přiřazení nijak s hodnotami nemanipuluje, zejména je nevytváří ani nekopíruje. Význam přiřazení je skutečně pouze **pojmenování** hodnoty, která už musí existovat (obvykle jako výsledek vyhodnocení výrazu). Prozatím tento rozdíl není příliš důležitý – na chování programů začne mít dopad až ve třetí kapitole, kdy do jazyka přidáme složené typy. **Pozor:** některé programovací jazyky dávají přiřazení úplně jiný význam!

```
if podmínka₁:
    příkaz₁
    …
    příkazₙ
```

Význam tohoto zápisu je: vypočti hodnotu **výrazu** podmínka$_1$ a je-li výsledek pravdivý, proveď **příkazy** příkaz$_1$ až příkaz$_n$, jinak nedělej nic (výpočet pak pokračuje dalším příkazem v sekvenci). Příkaz if lze rozšířit o tzv. else větev:

```
if podmínka₁:
    příkazy₁
else:
    příkazy₂
```

který se chová stejně, ale v případě, že podmínka splněna nebyla, ještě vykoná příkazy z posloupnosti příkazy$_2$. Konečně nejobecnější podoba podmíněného příkazu je (vpravo ekvivalentní zápis pomocí výše uvedené formy):

```
if podmínka₁:          ·  if podmínka₁:
    příkazy₁           ·      příkazy₁
elif podmínka₂:        ·  else:
    příkazy₂           ·      if podmínka₂:
                       ·          příkazy₂
elif podmínka₃:        ·      else:
    příkazy₃           ·          if podmínka₃:
                       ·              příkazy₃
else:                  ·          else:
    příkazy₄           ·              příkazy₄
```

přičemž větví elif může být libovolný počet.

Pro **opakování** nějaké posloupnosti příkazů slouží **cykly**, které jsou dvojího typu: for a while. Cyklus for použijeme v případě, kdy předem známe počet **iterací** (opakování), které chceme provést:

```
for jméno in rozsah:
    příkazy
```

kde rozsah může být:

- range(počet) – vypočte hodnotu **výrazu** počet a provede sekvenci příkazy právě počet-krát (jméno je v i-té iteraci vázáno na hodnotu i),
- range(od, do) – vypočte hodnoty n_1, n_2 **výrazů** od, do a provede sekvenci příkazy pro hodnoty $i \in \langle n_1, n_2)$ (jméno je přitom opět vázáno na hodnotu i),
- range(od, do, krok) – podobně jako předchozí, ale provede sekvenci pro hodnoty $i \in I \cap \{n_1 + js \mid j \in \mathbb{N}_0\}$ kde:
  - s je výsledek vyhodnocení výrazu krok,
  - I je $\langle n_1, n_2)$ pro $n_1 \leq n_2$ nebo $(n_2, n_1\rangle$ jinak

  a jméno je vázáno na hodnoty i v **pořadí stoupajícího** j.

Naopak cyklus while použijeme v situaci, kdy umíme **výrazem** popsat, chceme-li **provést další iteraci**:

```
while podmínka:
    příkazy
```

nejprve vyhodnotí **výraz** podmínka. Je-li hodnota pravdivá, provede příkazy a výraz podmínka **opět vyhodnotí**. Cyklus je ukončen v okamžiku, kdy se podmínka vyhodnotí jako nepravdivá (v takovém případě už se příkazy neprovedou, může tedy nastat situace, kdy se příkazy neprovedou **ani jednou**).

Kdekoliv v **těle cyklu** (ale nikde jinde) se mohou objevit ještě příkazy break a continue (vztahují se k rozsahem nejmenšímu cyklu, v kterého těle jsou obsaženy – tzn. k „nejvnitřnějšímu“ aktivnímu cyklu) a mají následovný význam:

- continue okamžitě ukončí probíhající iteraci: program pokračuje další iterací (není-li to možné, cyklus je na tomto místě ukončen),
- break okamžitě ukončí vykonávání cyklu.

4 Podprogramy Podprogramy jsou základním stavebním prvkem složitějších programů. Podprogram (v Pythonu také zvaný **funkce**) zastřešuje ucelený úsek kódu, který má navíc název, parametry a návratovou hodnotu. Podprogram **definujeme** následujícím zápisem:

```
def podprogram(parametr₁, parametr₂, …, parametrₙ):
    příkaz₁
    …
    příkazₙ
```

kde podprogram je **jméno**, parametr$_1$ až parametr$_n$ jsou **jména** tzv. **formálních parametrů** a příkaz$_1$ až příkaz$_n$ jsou sekvencí příkazů, které tvoří tzv. **tělo podprogramu**.

V podprogramu se krom už známých příkazů může objevit příkaz return výsledek, který jeho vykonávání **ukončí** a určí **návratovou hodnotu** (výsledek), kterou získá vyhodnocením **výrazu** výsledek.

Chceme-li již definovaný podprogram (funkci) použít, slouží k tomu tzv. **volání funkce**. Volání je **výraz**, a zapisuje se následovně:

```
podprogram(výraz₁, výraz₂, …, výrazₙ)
```

Zde podprogram je **jméno** a výraz$_1$ až výraz$_n$ jsou tzv. **skutečné parametry**. Protože se jedná o výraz, má **hodnotu**, která odpovídá návratové hodnotě podprogramu (příkazu return, kterým byl ukončen). S touto hodnotou můžeme pracovat jako s libovolným jiným výrazem:

```
výsledek₁ = funkce(3, 4)
výsledek₂ = 1 + 2 * funkce(3, 4)
výsledek₃ = funkce(funkce(1, 2), 3)
```

Význam použití podprogramu (volání funkce) je následovný:

- jménům ze seznamu formálních parametrů jsou přiřazeny **hodnoty**, které vzniknou vyhodnocením výrazů výraz$_1$ až výraz$_n$,
- provede se **tělo** podprogramu (sekvence příkazů příkaz$_1$ až příkaz$_n$),
- **návratová hodnota** se použije jako výsledek celého podvýrazu volání funkce a pokračuje se vyhodnocováním celého výrazu, ve kterém bylo volání obsaženo.

5 Zabudované podprogramy Krom podprogramů, které si sami definujete, můžete využívat několik takových, které jsou v jazyce **zabudované** (jsou součástí jazyka). Seznam těchto podprogramů budeme během semestru postupně rozšiřovat. Prozatím jsou to tyto (všechny zde uvedené podprogramy jsou zároveň **čisté funkce**):

- min(a, b) a max(a, b): vybere nejmenší, resp. největší hodnotu mezi svými parametry,
- abs(x): spočte absolutní hodnotu parametru x,
- round(x): pro desetinné číslo x se vyhodnotí na nejbližší celé číslo (hodnoty přesně mezi se zaokrouhlí na nejbližší sudé číslo),
- float(x): pro celé číslo x se vyhodnotí na odpovídající číslo s plovoucí desetinnou čárkou (v případě, že konverzi provést nelze, protože x příliš velké, je program ukončen s chybou).

Dále máte k dispozici **proceduru** print, kterou si můžete pomoct při programování, ale kterou jinak v tomto kurzu budeme potřebovat jen výjimečně.

6 Knihovny Pomocí příkazu (píšeme vždy na začátek programu)

```
from module import name₁, name₂, …
```

můžeme požádat o zpřístupnění podprogramů nebo konstant name$_1$, name$_2$ atd. z **knihovny** module. V této chvíli můžete používat pouze tyto **čisté funkce**, které realizují výpočet **funkcí** v matematickém smyslu, a konstanty z knihovny math:

- pi – číslo π (poměr obvodu a průměru kružnice),
- goniometrické a cyklometrické funkce:
  - cos(x), sin(x), tan(x) – známé goniometrické funkce (parametr x je zadán v **radiánech**),
  - acos(x), asin(x) – cyklometrické (inverzní trigonometrické) funkce, vstupem je reálné číslo intervalu $\langle -1, 1\rangle$ a výsledkem je odpovídající úhel z intervalu $\langle 0, \pi\rangle$,
  - atan(x) – inverzní funkce k funkci tan (vstupem je libovolné reálné číslo, výsledkem úhel z intervalu $(-\pi/2, \pi/2)$,
  - atan2(y, x) – úhel svíraný x-ovou osou a polopřímkou z počátku, která prochází bodem (x, y), v rozsahu $(-\pi, \pi\rangle$,
- funkce pro převod úhlů:
  - radians(x) – stupně na radiány a
  - degrees(x) – radiány na stupně,
- funkce pro výpočet kořenů:
  - sqrt(x) – druhá odmocnina reálného čísla x a

  - isqrt(x) – největší celé číslo menší rovno odmocnině x,
- funkce pro převod reálných čísel na celá (viz též zabudovanou funkci round uvedenou výše):
  - trunc(x) – ořezání desetinné části,
  - floor(x) – největší celé číslo ≤ x,
  - ceil(x) – nejmenší celé číslo ≥ x,
- funkce isclose(x, y) která realizuje „přibližnou rovnost“ čísel s plovoucí desetinnou čárkou.

7 Shrnutí

K dispozici tedy máme:

- výrazy:
  - konstanty a proměnné,
  - operátory pro aritmetiku, srovnání, logické spojky,
  - použití podprogramů (volání funkcí),
- příkazy:
  - přiřazení,
  - podmínku if, (elif, else),
  - cykly for a while,
  - tvrzení assert,
- definice vlastních podprogramů def,
- zabudované čisté funkce min, max, abs, round
- zabudovanou proceduru print,
- knihovnu math s konstantou pi a čistými funkcemi:
  - cos, sin, tan, acos, asin, atan, atan2,
  - radians, degrees,
  - sqrt, isqrt,
  - trunc, floor, ceil,
  - isclose.

Část 2: Jazyk druhé kapitoly

Tato kapitola používá stejné jazykové prostředky a zabudované podprogramy jako kapitola první. Přibyla pouze jediná knihovní (čistá) funkce, a to factorial(n) z knihovny math, pro přímý výpočet faktoriálu přirozeného čísla n.

Část 3: Jazyk třetí kapitoly

Tato kapitola přidává do našeho jazyka důležité prostředky pro popis a práci se složenými datovými typy (doteď jsme pracovali pouze s čísly a logickými hodnotami). Protože složená data jsou **hodnoty**, podobně jako čísla, většina změn se bude týkat **výrazů**. Mezi příkazy se objeví nová varianta cyklu `for` (pro procházení seznamu) a nové varianty **přiřazení**.

1 Literály Literály jsou typem **výrazů**. V této kapitole se objeví dva typy literálů: **seznamový literál** a **literál n-tice**.

Seznamový literál má tvar $[\text{výraz}_1, \text{výraz}_2, \ldots, \text{výraz}_n]$ (výrazy oddělené čárkami, uzavřené do hranatých závorek) a jeho významem je seznam, který má na indexu i hodnotu, která vznikla vyhodnocením výrazu výraz_i. Výrazů může být libovolný počet, včetně nuly (v takovém případě má výraz podobu `[]` a jeho hodnotou je prázdný seznam). Příklady:

```
[1]
[]
[1, 3, 2]
[[1, 2], [2, 3]]
[4, a + 1, f(3)]
[1, numbers[3]]
[(0, 1), (1, 1), (2, 1)]
```

Podobně, ale s kulatými závorkami, zapisujeme **literál n-tice**; ten má 3 možné podoby:

- `()` označuje prázdnou n-tici,
- $(\text{výraz},)$ označuje 1-tici (všimněte si koncové čárky),
- $(\text{výraz}_1, \text{výraz}_2, \ldots, \text{výraz}_n)$ pro $n \geq 2$.

Význam je analogický jako v případě seznamu. V některých případech lze kulaté závorky v zápisu n-tice vynechat, je-li takový zápis jednoznačný (podobně jako lze vynechat některé závorky v aritmetických výrazech). Můžeme tedy psát např. (vpravo ekvivalentní zápis s vypsanými závorkami):

```
return 1, 2             ·  return (1, 2)
x = 7, a + 1            ·  x = (7, a + 1)
a = x + 1, f(3), 7      ·  a = (x + 1, f(3), 7)
```

Ve všech uvedených případech jsou čárkami oddělené hodnoty interpretovány jako n-tice. Tuto zkratku ale nelze použít např. v parametru podprogramu nebo v seznamovém literálu.

2 Rozbalení Pro práci s n-ticemi budeme často používat tzv. **rozbalení**. Nejedná se ani o výraz ani o příkaz: je to speciální zápis, který se může objevit na **levé straně přiřazení**, v cyklu `for` a v intenzionálních seznamech. Zápisem se podobá na **literál n-tice**, ale místo výrazů obsahuje **jména**: $(\text{jméno}_1, \text{jméno}_2, \ldots, \text{jméno}_n)$. Podobně jako v literálu lze kulaté závorky vynechat. Můžeme tedy psát např.:

```
(x, y) = (1, 2)
x, y = (1, 2)
x, y = 1, 2
x, y = point_2d
x, y, z = point_3d
x, y = y, x
```

3 Příkazy Pro práci se seznamy se nám budou hodit dvě nové varianty cyklu `for`; první z nich (základní) zapisujeme:

```
for vazby in seznam:
    příkazy
```

kde se **výraz** `seznam` vyhodnotí na **seznam** a `vazby` je buď **jméno** nebo **rozbalení**. Tělo cyklu (`příkazy`) se pak provede jednou pro každý prvek seznamu `seznam`. V i-té iteraci odpovídají `vazby` i-tému prvku seznamu `seznam`. Je-li `seznam` prázdný, tělo se neprovede ani jednou.

Rozšířená verze

```
for index, vazby in enumerate(seznam):
    příkazy
```

má stejný význam jako v předchozím případě, s těmito změnami:

- `index` je **jméno**, které váže **index** právě iterovaného prvku v seznamu `seznam` (nebo ekvivalentně váže pořadové číslo právě prováděné iterace, počítáno od 0),
- v případě, kdy jsou `vazby` **rozbalení**, musí být uzavřeny v kulatých závorkách (jinými slovy, na tomto místě nelze závorky vynechat).

Dále přidáme dvě nové varianty příkazu **přiřazení**:

- na levé straně se může krom **jména** objevit také výše popsané **rozbalení**: $\text{jméno}_1, \ldots, \text{jméno}_n$ `= výraz` s významem analogickým běžnému přiřazení (pouze je dotčeno několik proměnných najednou),
- o něco komplikovanější je **přiřazení do prvku seznamu**, které zapisujeme jako `seznam[index] = výraz` kde `seznam` je **jméno** a `index` je **výraz** s celočíselnou hodnotou.

Přiřazení do prvku seznamu (nazýváme ho též **vnitřním** přiřazením) se ale svým významem od běžného přiřazení podstatě odlišuje: tento příkaz **upraví** stávající objekt, který je přiřazen jménu `seznam`.

4 Výrazy Krom literálů přibývá se složenými datovými typy ještě několik nových výrazů. Prvním z nich je **indexace**, která má tvar `seznam[index]`, kde:

- `seznam` je **jméno** proměnné (typu seznam),
- `index` je aritmetický **výraz** (jeho hodnotou je celé číslo),
- výsledkem je hodnota, která je v seznamu `jméno` uložena na indexu i, kde i je hodnota, na kterou se vyhodnotil výraz `index`.

Například:

```
a[0]
numbers[i + 1]
names[compute_index(m, n)]
```

Dalším novým typem výrazu je **použití (volání) metody**, které má tvar $\texttt{objekt.metoda}(\text{výraz}_1, \ldots, \text{výraz}_n)$ a je obdobou **použití podprogramu** (volání funkce), který je ve speciálním vztahu s objektem vázaným ke **jménu** `objekt`:

- nejprve se vyhodnotí parametry $\text{výraz}_1, \ldots, \text{výraz}_n$,
- provede se volání samotné metody s názvem `metoda`,
- hodnotou výrazu je **návratová hodnota** volané metody.

Příklady:

```
numbers.append(a + 3)
4 + names.pop()
left.append(right.pop())
numbers.append(min(a, b))
```

Další dva nové typy výrazů nám umožní zapisovat hodnoty typu seznam:

- **seznamový literál**, který jsme již zavedli výše, nám umožňuje zapsat seznam o pevném počtu prvků, a
- **intenzionální seznam**, kterého délka může být proměnlivá, ale uložené hodnoty se řídí nějakým předpisem.

Intenzionální seznam má tyto tvary:

- $[\text{prvek for jméno in range}(\text{počet})]$, kde
  - `počet` je **výraz** s celočíselnou hodnotou,
  - výsledkem je seznam, který má `počet` prvků,
  - prvek i vznikne vyhodnocením výrazu `prvek`, přičemž `jméno` má pro dané vyhodnocení hodnotu i (počínaje nulou),
- $[\text{prvek for jméno in range}(n_1, n_2)]$, je analogický, ale hodnoty vázané na `jméno` jsou z intervalu $\langle n_1, n_2)$,
- $[\text{prvek for jméno in rozsah if podmínka}]$, kde `rozsah` je $\text{range}(\text{počet})$ nebo $\text{range}(\text{od}, \text{do})$ a který má stejný význam jako předchozí, ale obsahuje pouze ty prvky, pro které se `podmínka` vyhodnotí jako pravdivá,
- $[\text{prvek for vazby in seznam}]$, kde
  - `seznam` je **výraz** typu seznam,
  - `vazby` jsou **rozbalení** nebo **jméno**,

- hodnotou je seznam, který má stejný počet prvků jako seznam a
- na i-té pozici je hodnota, která vznikne vyhodnocením **výrazu** prvek, přičemž vazby v každém vyhodnocení odpovídají i-tému prvku seznamu seznam,

- [prvek for vazby in seznam if podmínka], který je opět ekvivalentní předchozímu, ale opět obsahuje pouze ty prvky, pro které se podmínka vyhodnotí jako pravdivá.

Výrazy podmínka se v obou případech vyhodnocují se stejnými vazbami, jako výraz prvek. Příklady:

```
[1 for i in range(5)]
[i + 1 for i in range(2 * count)]
[2 * i for i in range(7) if i != 3]
[2 * i for i in numbers]
[i ** 2 for i in numbers if i > 0]
```

Poslední nový typ výrazu je obměnou již známých relačních operátorů: výrazy `x == y`, `x != y`, `x < y`, `x > y`, `x >= y`, `x <= y` připouštíme i v případech, kdy se **oba** podvýrazy `x`, `y` vyhodnotí na seznamy, nebo se oba vyhodnotí na n-tice. Operátor $\leq$ je v tomto případě dán **lexikografickým uspořádáním**:

- je-li `x` prefixem `y` nebo naopak, jako menší se vyhodnotí hodnota s menším počtem prvků,
- jinak nechť je i nejmenší index, na kterém se `x` a `y` liší a x_i a y_i jsou prvky na této pozici; výraz `x < y` se vyhodnotí na výsledek srovnání $x_i < y_i$.

Chování ostatních operátorů je již jednoznačně určeno rovností a operátorem $\leq$.

5 Zabudované podprogramy Pro práci se složenými datovými typy také přibudou tyto zabudované **čisté funkce**:

- `len(x)` – výsledkem je délka (počet prvků) seznamu `x` (nezáporné celé číslo),
- `sum(x)` – výsledkem je suma (součet) všech prvků seznamu `x`,
- `min(x)`, `max(x)` – výsledkem je nejmenší (největší) ze všech prvků seznamu `x` (je-li seznam prázdný, program je ukončen s chybou).

Pro jednodušší práci s celými čísly přidáváme navíc čistou funkci

- `divmod(x, y)`, které výsledkem je **dvojice** `(x // y, x % y)`.

Nakonec máme nově k dispozici tyto zabudované **metody** pro hodnoty typu **seznam**:

- `l.append(x)` – přidá hodnotu `x` na konec seznamu `l`,
- `l.pop()` – odstraní ze seznamu `l` poslední prvek,
- `l.copy()` – vytvoří a vrátí kopii seznamu `l`.

Pozor, metody append a pop **nejsou čisté**: modifikují vstupní seznam `l`.

Část 4: Jazyk čtvrté kapitoly

Hlavní novinkou této kapitoly jsou **typové anotace**. Ty se dotknou zejména definice funkce a příkazu přiřazení. Rozšířený zápis definice funkce má následovný tvar:

def podprogram(p_1: typ_1, p_2: typ_2, …, p_n: typ_n) -> typ_r:

příkazy

Příkaz přiřazení dostane nový tvar, konkrétně:

jméno: typ = výraz

Význam všech anotací tvaru jméno: typ (tzn. jak v parametrech funkcí, tak v přiřazení) je „jméno **vždy** váže hodnotu typu typ“. Význam anotace -> typ v definici funkce má pak význam „návratová hodnota funkce je **vždy** typu typ“. Pravdivost těchto tvrzení pak (staticky) ověří program mypy, jak již bylo naznačeno v úvodě.

1 Typy Na místě typ se ve výše uvedených formách může objevit:

- jednoduchý typ:
  - bool – hodnota je True nebo False,
  - int – hodnota je celé číslo,
  - float – hodnota je číslo s plovoucí desetinnou čárkou,
  - str – hodnota je řetězec,
  - None – hodnota je None,
- složený typ, který vznikne použitím **typového konstruktoru** (tuple, list, atp.) a **typových parametrů** (píšeme v hranatých závorkách za konstruktor; v těchto závorkách typ představuje opět cokoliv z tohoto seznamu):
  - tuple[typ_1, typ_2, …, typ_n] – hodnota je n-tice a její i-tá složka je typu typ_i,
  - list[typ] – hodnota je seznam, kterého **každý** prvek je typu typ,
- tzv. volitelný typ, který vznikne zápisem typ | None, popisuje hodnotu, která může být typu typ, nebo může být None (ale nic jiného)[2],
- nebo tzv. **typový alias**, tedy jméno, které je přiřazením svázáno s konkrétním typem (jména typových aliasů začínají velkým písmenem):

  TypovýAlias = typ

[2] Zápis pomocí „svislítka“ | umožňuje i obecnější typy, v tuto chvíli se ale omezíme na tvar typ | None. Komplikovanější typy tohoto tvaru zavedeme v sedmé kapitole.

Část 5: Jazyk páté kapitoly

Tato kapitola přidává dva nové typy složených hodnot:[3]

- **množina** – set – podobně jako seznam obsahuje vnitřní hodnoty, s tím rozdílem, že v množině nemají hodnoty pevně určené pořadí, a každá se v dané množině může objevit nejvýše jednou,
- **slovník** – dict – obsahuje **klíče** (podobně jako v množině se daný klíč může objevit nejvýše jednou) a ke každému klíči právě jednu **přidruženou hodnotu** (obvykle nazýváme prostě **hodnota**, a mluvíme o dvojicích klíč – hodnota).

Pro hodnoty, které vkládáme do množin, nebo je používáme jako klíče ve slovníku, platí důležité omezení: taková hodnota **nesmí** mít vnitřní přiřazení, ani jiné operace, které mohou vnitřně danou hodnotu změnit. Zejména tedy nelze takto používat **seznamy**, ale ani slovníky nebo množiny. Přípustná jsou naopak zejména celá čísla, řetězce a n-tice z nich složené.

S novými typy hodnot přidáváme i nové tvary výrazů (literály, přístup k přidruženým hodnotám, množinové operace) a příkazů (přiřazení, for cyklus) a nové zabudované podprogramy.

1 Literály Jak jsme již zvyklí, hodnoty typu **množina** a **slovník** můžeme do programu zapsat pomocí speciálních výrazů – literálů (podobně jako tomu bylo u seznamů, n-tic a řetězců). Tyto literály mají tvar:

- {} je **prázdný slovník** (pozor, nikoliv množina!),
- {$klíč_1$: $hodnota_1$, $klíč_2$: $hodnota_2$, …} je **slovník**, kde $klíč_i$ jsou **výrazy**, kterých vyhodnocením vzniknou **klíče**, přičemž vyhodnocením výrazu $hodnota_i$ vznikne vždy hodnota přidružená odpovídajícímu klíči (vyhodnotí-li se dva různé výrazy $klíč_i$ na stejný výsledek, použije se dvojice více vpravo),
- {$hodnota_1$, $hodnota_2$, …} reprezentuje **množinu** s prvky, které vzniknou vyhodnocením **výrazů** $hodnota_i$.

Prázdná množina literál nemá. Chceme-li vytvořit prázdnou množinu, použijeme k tomu zabudovanou funkci set() bez parametrů.

2 Výrazy Přístup k přidružené hodnotě uložené ve slovníku[4] zapisujeme výrazem tvaru slovník[klíč], kde:

- slovník je **výraz** který se vyhodnotí na hodnotu typu slovník a
- klíč je **výraz**, který je nejprve vyhodnocen, poté je výsledná hodnota ve slovníku vyhledána,
- výraz slovník[klíč] jako celek se pak vyhodnotí na odpovídající přidruženou hodnotu byl-li klíč ve slovníku nalezen, v opačném případě je program ukončen s chybou.

Oproti seznamům jsou jak množiny tak slovníky vybaveny **efektivním** dotazem na přítomnost prvku (u slovníku klíče), a to výrazy tvaru:

```
hodnota in množina
klíč in slovník
```

kde hodnota, množina, klíč a slovník jsou **podvýrazy** a výsledkem je **pravdivostní hodnota**.

3 Zabudované podprogramy Objekty typu **slovník** mají tyto zabudované metody:

- d.keys() – výsledkem je speciální hodnota, kterou lze pouze iterovat nebo převést na seznam (viz níže), a která obsahuje pouze klíče ve slovníku přítomné (bez přidružených hodnot),
- d.values() – analogicky, ale pro přidružené hodnoty,
- d.items() – taktéž, ale obsahuje dvojice (klíč, hodnota),
- d.get(k) nebo d.get(k, fallback) – vyhledá klíč k v slovníku, a vyhodnotí se na odpovídající hodnotu, je-li tato přítomna, jinak na None (první tvar) nebo na fallback (druhý tvar),
- d.pop(k) – odstraní ze slovníku klíč k (včetně přidružené hodnoty),
- d.copy() – vytvoří kopii slovníku.

Objekty typu **množina** pak mají tyto zabudované metody:

- s.add(v) – vloží do množiny hodnotu v (byla-li již přítomna, nestane se nic),
- s.remove(v) – odstraní hodnotu v (není-li hodnota přítomna, program je ukončen s chybou),

Pro vytváření hodnot přidáváme několik zabudovaných **čistých funkcí**:

- list(x) – převede hodnotu x na seznam, kde x může být:
  - množina,
  - výsledek volání d.keys(), d.values() nebo d.items() na slovníku d,
- set() – vytvoří prázdnou množinu,
- set(l) – převede seznam l na množinu,
- dict(l) – převede seznam dvojic l na slovník.

4 Příkazy Pro práci s prvky množin a s klíči, hodnotami a dvojicemi (klíč, hodnota) ve slovníku lze použít for cykly těchto tvarů:

```
for vazby in množina:
    příkazy

for vazby in slovník.keys():
    příkazy

for vazby in slovník.items():
    příkazy

for vazby₁, vazby₂ in slovník.items():
    příkazy
```

Kde vazby je vždy buď **jméno** nebo **rozbalení** a množina a slovník jsou **výrazy**. V posledním uvedeném případě je nutné případné rozbalení uzávorkovat, například:

```
for shape, (x, y) in centers.items():
    pass
```

Posledním novým prvkem je vnitřní přiřazení do slovníku:

```
slovník[klíč] = hodnota
```

kde slovník, klíč i hodnota jsou **výrazy**. Byl-li klíč již ve slovníku přítomen, jeho přidružená hodnota se změní na výsledek vyhodnocení výrazu hodnota. V opačném případě je klíč do slovníku přidán (pozor, v tomto se slovníky liší od seznamů).

[3] Výše zmíněný **zásobník** nemá samostatný datový typ: lze jej přímo reprezentovat pomocí seznamu.

[4] Zápis je analogický k indexaci seznamů a řetězců. Oproti těmto již známým typům ale slovníky „indexujeme“ **klíčem**, který **nemusí** být celé číslo, a i v případě, kdy jím celé číslo je, **nemusí** klíče tvořit spojitou řadu začínající nulou. Množinu indexovat nelze.

Část 6: Jazyk šesté kapitoly

Tato kapitola přidává několik odvozených operací na seznamech a množinách. Pozor, tyto operace mají **lineární** složitost.

1 Výrazy Z minulé kapitoly známe operace:

`hodnota in množina`
`klíč in slovník`
Nyní přidáme analogické dotazy tohoto tvaru na přítomnost hodnoty v seznamu: `hodnota in seznam` (zde `seznam` je opět podvýraz), ale musíme si pamatovat, že pro **seznam** tento dotaz **není efektivní**: obsahuje skrytou iteraci potenciálně všemi prvky seznamu.

Pro **množiny** připouštíme nově tyto tvary výrazů (kde $\texttt{množina}_1$ a $\texttt{množina}_2$ jsou vždy **podvýrazy**, které se musí vyhodnotit na hodnoty typu množina):

- $\texttt{množina}_1$ `|` $\texttt{množina}_2$ se vyhodnotí na **sjednocení**,
- $\texttt{množina}_1$ `&` $\texttt{množina}_2$ se vyhodnotí na **průnik** a
- $\texttt{množina}_1$ `-` $\texttt{množina}_2$ se vyhodnotí na **rozdíl** příslušných množin.

Konečně pro **seznamy** přidáváme výraz tvaru $\texttt{seznam}_1$ `+` $\texttt{seznam}_2$ (kde $\texttt{seznam}_1$ a $\texttt{seznam}_2$ jsou opět podvýrazy), který se vyhodnotí na **nový seznam** s prvky z prvního i druhého seznamu (nejprve všechny prvky levého operandu, pak všechny prvky pravého, vždy v původním pořadí).

2 Zabudované podprogramy Objekty typu **množina** získají tyto nové zabudované metody:

- s_1`.update(`s_2`)` – **přidá** do množiny s_1 všechny prvky, které se **nachází** v s_2 (v s_1 tak bude po provedení operace sjednocení obou množin),[5]
- s_1`.intersection_update(`s_2`)` – **odebere** z množiny s_1 všechny prvky, které se **nenachází** v s_2 (v s_1 tedy bude po provedení průnik),
- s_1`.difference_update(`s_2`)` – **odebere** z množiny s_1 všechny prvky, které se **nachází** v s_2 (v s_1 tedy bude po provedení rozdíl).

Přidáme také několik zabudovaných metod pro práci se seznamy. **Pozor** všechny tyto metody jsou **ekvivalentní iteraci** – nelze tedy jejich použitím ušetřit výpočetní čas, jsou jen syntaktickou zkratkou pro obšírnější `for` cyklus:

- l`.reverse()` – otočí pořadí prvků v seznamu,
- l`.index(`v`)` – vyhodnotí se na index, na kterém se nachází hodnota v (je-li takových více, výsledkem je ten nejmenší; není-li takový žádný, program je ukončen s chybou),
- l_1`.extend(`l_2`)` – přidá na konec seznamu l_1 všechny prvky ze seznamu l_2 (ve stejném pořadí),
- l`.insert(`i`, `v`)` – vloží **před** index i hodnotu v (tedy hodnoty na indexech $j \geq i$ přesune o jednu pozici doprava a na index i uloží hodnotu v),
- l`.pop(`i`)` – odstraní hodnotu z indexu i (a tedy všechny hodnoty na vyšších indexech přesune o jednu pozici doleva).

[5] Pozor, s_1`.update(`s_2`)` **není** totéž, jako s_1 `=` s_1 `|` s_2 – první operace vnitřně změní existující hodnotu s_1, ta druhá vytvoří **novou množinu** a výsledek sváže se jménem s_1.

Část 7: Jazyk sedmé kapitoly

Tato kapitola přináší možnost definovat vlastní (uživatelské) datové typy. K tomuto účelu zavedeme nový typ **definice**. Definice datového typu musí stát vně jakékoliv jiné definice (tedy na stejné úrovni jako definice funkcí, které jsme doteď znali).

Definice typu má následovný tvar:

```
class Třída:
    def __init__(self, param₁: typ₁, …, paramₙ: typₙ) -> None:
        tělo
    def metoda₁(self, param₁: typ₁, …, paramₙ: typₙ) -> typ:
        tělo
    …
```

Uvnitř **definice typu** se tedy může objevit definice **inicializační funkce** a **definice metod** (a nic jiného). Tyto definice se v obou případech velmi podobají na definice funkcí – základním rozdílem (krom toho, kde stojí) je povinný první parametr s názvem self.

1 Vytváření hodnot V případě inicializační funkce (povinně nazvané __init__) reprezentuje parametr self nový objekt, který je potřeba inicializovat (zejména nastavit počáteční hodnoty atributů).

Nové **hodnoty** uživatelského typu Třída se vytvoří následovným **výrazem**:

Třída(výraz₁, …, výrazₙ)

Protože se jedná o výraz, lze jej použít jako podvýraz v jiných výrazech, nebo třeba v přiřazovacím příkazu na pravé straně takto:

objekt = Třída(výraz₁, …, výrazₙ)

Tento výraz krom samotného vytvoření objektu zavolá inicializační funkce __init__, s následovnými vazbami formálních parametrů:

- self se váže na **nově vznikající objekt**,
- param_1 se váže na hodnotu výrazu výraz_1, atd.,
- param_n se váže na hodnotu výrazu výraz_n.

2 Atributy Hlavním úkolem inicializační funkce je nastavit počáteční hodnoty **atributů** nového objektu. Atributy se velmi podobají proměnným, nejsou ale svázané s aktuálně vykonávanou funkcí, ale s objektem. Přístup k atributům objektu je **výraz**, který se podobá na použití metody. Např.:

```
person.weight
bmi = person.weight / person.height ** 2
d = sqrt(point.x ** 2 + point.y ** 2)
```

Objekty mají určitou podobnost s n-ticemi, které již dobře známe: sdružují několik hodnot (potenciálně různých typů) do jedné. Mají ale i dvě zásadní odlišnosti:

- atributy objektů jsou **pojmenované** (jsou určeny jmény, nikoliv pořadím),
- objekty mají **vnitřní přiřazení** – vazbu atributu na hodnotu lze měnit (použitím přiřazovacího příkazu).

Přiřazení do atributu je příkaz, který se podobá na ostatní druhy přiřazení, které známe (zejména na vnitřní přiřazení do seznamu nebo slovníku):

objekt.atribut = výraz

kde objekt a atribut jsou **jména**. Významem je změna vazby atributu (na hodnotu, která vznikne vyhodnocením výrazu výraz).

3 Metody V metodách parametr self reprezentuje objekt, na kterém byla metoda použita. Tedy při použití metody (druh **výrazu**, který již známe u zabudovaných typů):

objekt.metoda₁(výraz₁, …, výrazₙ)
se vážou formální parametry na skutečné parametry takto:

- self se váže na hodnotu objekt,
- param_1 se váže na hodnotu výrazu výraz_1, atd.,
- param_n se váže na hodnotu výrazu výraz_n.

Jinak jsou metody stejné jako obyčejné funkce.

Část 8: Jazyk osmé kapitoly

Tato kapitola přináší pouze dva nové prvky (oba souvisí s řazením).

1. Zabudovanou čistou funkci `sorted(x)`, které výsledkem je nový seznam, který je vzestupně uspořádaný (pro `l = sorted(x)` a `i <= j` platí `l[i] <= l[j]`), a zároveň obsahuje stejné prvky jako `x`. Parametr `x` může být:
   - seznam (`list`),
   - množina (`set`),
   - `d.items()`, `d.keys()` nebo `d.values()` je-li `d` hodnota typu slovník (`dict`).
2. Zabudovanou metodu-proceduru `l.sort()`, která přeuspořádá seznam `l` tak, aby byl vzestupně seřazený (samotné prvky se při tom opět nijak nemění).

Část 9: Jazyk deváté kapitoly

Tato kapitola přináší do jazyka dva nové prvky, které oba souvisí s typy:

1. Typovou anotaci typ_1 | typ_2 | … | typ_n, která realizuje tzv. **součtové typy**, kdy o nějaké hodnotě umíme říct, že je určitě některého z vyjmenovaných typů, ale který konkrétně to bude se rozhodne až za běhu programu.
2. Zabudovaný predikát `isinstance(value, type)`, který rozhodne, je-li hodnota `value` typu `type`. Tento predikát lze s výhodou použít v kombinaci se součtovými typy, kdy se v programu potřebujeme rozhodnout podle skutečného typu hodnoty `value`.

   V těle podmíněného příkazu `if isinstance(value, type)` pak platí, že hodnota `value` má i staticky (tzn. pro účely typové kontroly programem `mypy`) přiřazen typ `type`.

Část 10: Jazyk desáté kapitoly

V této kapitole se jazyk nemění.

Část 11: Jazyk jedenácté kapitoly

Tato kapitola přidává operace práci s **řetězci**. Krom nových **výrazů** se drobná rozšíření dotknou i příkazu for (který můžeme použít k procházení řetězce po znacích). Na rozdíl od seznamů ale pro řetězce neexistuje vnitřní přiřazení.

Tato kapitola přináší také prostředky pro jednoduchou práci se soubory a další interakci s prostředím (zejména operačním systémem).

1 Literály Podobně jako tomu bylo v případě seznamů a n-tic, řetězce můžeme do programu zapsat pomocí **řetězcových literálů**. Ty mají jeden z těchto tvarů: 'znaky', "znaky", """znaky""", '''znaky'''. Významově jsou všechny tyto tvary ekvivalentní: vytvoří hodnotu typu řetězec, která obsahuje znaky.

Pro většinu znaků je obsah vzniklého řetězce totožný se zápisem literálu, až na dva druhy výjimek:

- některé znaky nebo sekvence znaků se v literálech nesmí mimo speciální sekvence objevit:
  - znak konce řádku v literálech s jednoduchým oddělovačem ('znaky' a "znaky"),
  - samotný oddělovač (', ", ''', """) použitý pro zápis daného literálu – nebylo by zřejmé, zda se jedná o konec literálu nebo nikoliv,
- některé sekvence znaků, které začínají znakem \ (zpětné lomítko) se **přeloží** na jeden znak:
  - \', \" se přeloží na samotné znaky ' a ",
  - \\ se přeloží na znak \,
  - \n se přeloží na znak konce řádku,
  - \a, \b, \f, \r, \t, \v se přeloží na různé speciální znaky, které v tomto kurzu nebudou důležité,
  - \NNN a \xNN, \uNNNN, \UNNNNNNNN, kde N… je tříciferný osmičkový nebo dvou-, čtyř- nebo osmiciferný šestnáctkový zápis nějakého čísla n, se přeloží na znak x který má v tabulce znaků Unicode pozici n.

Snadno se přesvědčíte, že „zakázané“ znaky resp. sekvence znaků lze vždy zapsat nějakým alternativním způsobem pomocí \-sekvencí.

2 Výrazy Podobně jako seznamy, řetězce lze **indexovat**: zápis je stejný jako u seznamů: řetězec[index], kde řetězec je **jméno** a index je celočíselný výraz. Na rozdíl od seznamů, výsledkem indexace je v případě řetězce **opět řetězec**, který ale obsahuje pouze jediný znak.

Dále nově připouštíme relační operátory x == y, x != y, x < y, x > y, x <= y, x >= y i v případě, kdy se podvýrazy x a y oba vyhodnotí na řetězce. Uspořádání je dáno **lexikograficky**.

3 Příkazy Jediný nový příkaz, který souvisí s řetězci, je

```
for ch in řetězec:
    příkazy
```

kde ch je **jméno** a řetězec je **výraz**, který se vyhodnotí na hodnotu typu řetězec. Podobně jako ostatní varianty příkazu for, tento provede sekvenci příkazy jednou pro každý znak uložený v řetězci řetězec. Jméno ch je přitom v i-té iteraci vázáno na jednopísmenný řetězec odpovídající znaku na i-té pozici hodnoty řetězec.

Pro práci se soubory (a dalšími zdroji, o kterých ale v tomto předmětu nebude řeč) budeme krom zabudovaného podprogramu open (vysvětleno níže) slouží také příkaz with – je obvyklé je používat vždy společně, a to ve tvaru:

```
with open(cesta, režim) as název:
    příkazy
```

Tato konstrukce nám umožní se souborem pracovat v těle příkazu with pomocí jména název (stejně, jako kdybychom přiřadili výsledek volání open do proměnné), ale navíc máme zaručeno, že po opuštění tohoto bloku je práce se souborem korektně ukončena.

Takto otevřený a pojmenovaný soubor můžeme **iterovat** již dobře známým příkazem for:

```
for řádek in soubor:
    příkazy
```

kde řádek je jméno a soubor je výsledek volání open (obvykle vázaný příkazem with). Ke jménu řádek budou postupně vázány hodnoty typu str, které obsahují vždy jeden řádek souboru (včetně ukončovacího znaku '\n'). Cyklus je ukončen po přečtení posledního řádku.

4 Zabudované podprogramy Objekty typu řetězec navíc poskytují tyto **zabudované metody** (ve všech případech jsou zároveň **čistými funkcemi** – vstupní řetězec nikdy nemodifikují):

- s.isupper(), s.islower() – predikáty, vyhodnotí se na True v případě, že všechny abecední znaky v řetězci s jsou velká (resp. malá) písmena,
- s.isalpha(), s.isdecimal() – predikáty, které se vyhodnotí na True sestává-li s pouze z abecedních znaků (isalpha) resp. desítkových číslic (isdecimal),
- s.upper(), s.lower() – vyhodnotí se na řetězec, který vznikne ze s nahrazením všech abecedních znaků na odpovídající velká (upper) resp. malá (lower) písmena,
- s.split(delim) – vyhodnotí se na **seznam**, který vznikne rozdělením s na podřetězce oddělovačem delim (oddělovače nejsou součástí výsledných řetězců),
- s.join(parts) – vyhodnotí se na řetězec, který vznikne vložením řetězce s mezi každé dva řetězce uložené v **seznamu** parts,
- s.replace(from, to) – vyhodnotí se na řetězec, který vznikne ze s substitucí všech výskytů podřetězce from za podřetězec to,
- s.rstrip() – vyhodnotí se na řetězec, který vznikne odstraněním všech pravostranných bílých znaků (zejména mezer a znaků konce řádku).

Jak bylo naznačeno výše, práci se soubory nám umožňuje zabudovaný podprogram open(cesta, režim)[6]. Parametr cesta (typu řetězec) určuje kde v souborovém systému se má hledat soubor, se kterým chceme pracovat, řetězec režim pak určuje jakým způsobem hodláme soubor používat. Základní možnosti jsou tyto:

- 'r' – režim pouze pro čtení nám umožní ze souboru číst textová data, ale nic dalšího,
- 'w' – režim pro zápis textu, kdy je soubor při otevření zkrácen na nulovou délku (z takto otevřeného souboru nelze číst),
- 'x' – jako 'w', ale soubor je prvně vytvořen (v případě, že již existuje, je program ukončen s chybou),
- 'a' – jako 'w', ale soubor není zkrácen, nová data jsou zapisována na konec souboru.

Tyto základní možnosti lze kombinovat se specifikátorem 't' nebo 'b', který určí, chceme-li se souborem pracovat v **textovém** nebo **binárním** režimu. Neuvedeme-li ani jedno z nich, implicitní je textový režim. V tomto předmětu se omezíme na textový režim.

S hodnotou f, které vznikne voláním podprogramu open v textovém režimu, můžeme použít také několik zabudovaných metod:

- f.close() – ukončí práci se souborem (obvykle **nepoužíváme**, ukončení provedeme místo toho správným použitím příkazu with),
- f.read(n) – přečte nejvýše n znaků a vrátí je jako hodnotu typu str,
- f.readline() – přečte znaky od aktuální pozice až do konce řádku a vrátí je jako hodnotu typu str,
- f.readlines() – přečte celý zbytek souboru po řádcích, výsledkem je hodnota typu list, která obsahuje pro každý přečtený řádek jednu položku typu str,
- f.write(s) – zapíše řetězec s (t.j. hodnotu typu str) do souboru.

5 Knihovny Většina funkcionality pro interakci s vnějším světem je k dispozici formou knihoven (obdoba knihovny math, kterou známe z první kapitoly). Zde uvádíme pouze stručný přehled, bližší informace k použití jednotlivých knihoven získáte v 11. přednášce. Použití knihovny je potřeba

[6] Nejedná se v tomto případě ani o čistou funkci, ale ani o klasickou proceduru.

vždy na začátku souboru deklarovat řádkem

`from knihovna import jméno₁, jméno₂, …`
K dispozici máme tyto knihovny:

- `gzip` – práce s komprimovanými soubory `*.gz`,
  - `open` – otevře komprimovaný soubor (dále s ním lze pracovat jako s obyčejným souborem, liší se ale implicitním použitím binárního režimu) – voláme pomocí příkazu `with`,
- `csv` – práce s textovými soubory, které obsahují tabulky hodnot oddělené čárkou (nebo jiným oddělovačem),
- `sys` – obecná interakce se systémem:
  - `argv` – seznam hodnot typu `str`, které byly programu předány při spuštění na příkazové řádce,
- `os` – další podprogramy (zejména procedury) pro práci se systémem (`cesta` je hodnota typu `str`):
  - `remove(cesta)` – odstraní (smaže) soubor,