Martin Trnečka

Základy programování v Pythonu

první část

Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci

Tento text slouží jako doplňkový učební materiál pro první 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 osvojení základů programování a algoritmizace, které jsou klíčové pro pozdější studium. Jednotlivé koncepty 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 bez předchozích znalostí programování. Text je průběžně rozšiřován a doplňován. Poslední změna proběhla 24. února 2023. Pokud v textu naleznete chyby, prosím, kontaktujte mne prostřednictvím e-mailové adresy martin.trnecka@upol.cz.

Děkuji Janu Laštovičkovi, Petru Osičkovi, Karlu Panchártkovi, Markétě Trnečkové a Jirku Zacpalovi za jejich připomínky a cenné podněty ke kvalitě textu.

Martin Trnečka

Obsah

Úvod do programování

„Everybody in this country should learn to program a computer, because it teaches you how to think."

Steve Jobs

V této kapitole vysvětlíme pojmy programování, program a programovací jazyk. Stručně zmíníme některé vlastnosti programovacích jazyků a vytvoříme první jednoduchý program v jazyce Python.

Co je to programování?

Aby počítač mohl vykonat specifickou úlohu, potřebuje, aby mu někdo, obvykle programátor, zadal instrukce řešící danou úlohu. Sekvence takovýchto instrukcí se nazývá *program*. Ve své podstatě programování není nic jiného než: (i) nalezení postupu řešení dané úlohy a (ii) zadání instrukcí (programu) počítači, které mu umožní nalezený postup vykonat. Příkladem úlohy je třeba nalezení největšího společného dělitele dvou celých čísel.[1] Pokud bychom chtěli naprogramovat počítač, aby řešil tuto úlohu, musíme nejprve vymyslet postup (algoritmus), jak největšího společného dělitele dvou čísel vypočítat, a následně tento postup zapsat pomocí instrukcí, kterým počítač rozumí.

Zde je dobré upozornit, že (i) je tou náročnější částí programování, jež vyžaduje teoretické znalosti, algoritmické myšlení a také kreativitu.[2] Tyto znalosti jsou ale univerzální a lze na nich, podobně jako v matematice, postupně stavět. Oproti tomu (ii), tedy zapsání programu, vyžaduje znalosti konkrétních technologií. Tyto znalosti nejsou vždy přenositelné a časem zastarávají. Jejich osvojení je ale jednodušší.

[1] Klasická „učebnicová" úloha. Stejně tak bychom mohli napsat zobrazení webové stránky, hraní počítačové hry nebo udržování informací o skladových zásobách.

[2] Osvojit si tyto dovednosti vyžaduje čas. Je naprosto běžné, že začínajícím programátorům zabere nalezení řešení více času. Stejně tak jejich řešení nemusí být tak dobré jako řešení zkušených programátorů. Tímto se nesmí nikdo nechat odradit.

Program

Pokud známe postup řešení problému, můžeme začít psát program. Přestože jsou počítače velice složité stroje, „rozumí" jen několika primitivním instrukcím.[3] Programování pomocí těchto primitivních instrukcí je poměrně náročné a zdlouhavé. Místo toho se

[3] Tyto instrukce se označují jako instrukční sada či strojový kód. Jedná se o velice jednoduché instrukce, například: „přičti k číslu jedna".

pro zápis programů používají *programovací jazyky*. Zápis programu v konkrétním jazyce se označuje *zdrojový kód* programu.

Průvodce studiem
Je důležité si uvědomit, že naučit se programovat, neznamená naučit se konkrétní programovací jazyk, ale naučit se obecně řešit problémy, bez ohledu na programovací jazyk.

Programovací jazyky

Existují stovky programovacích jazyků. Ty se liší především možnostmi, které programátorům nabízí. V následující části stručně zmíníme některé vlastnosti programovacích jazyků.

Programovací jazyky se dělí podle úrovně poskytované *abstrakce* na *nízkoúrovňové* a *vysokoúrovňové*.[4] Nízkoúrovňové programovací jazyky, například různé assemblery nebo jazyk C, neposkytují žádnou nebo malou abstrakci nad instrukcemi, kterým počítač rozumí. Typické pro tyto jazyky je, že programátor má, respektive musí mít, vše pod kontrolou.[5] Vysokoúrovňové programovací jazyky, mezi které patří například jazyky Python nebo Java, poskytují výrazně větší abstrakci. Při zápisu programů v těchto jazycích je možné využívat složitějších instrukcí, které jsou blíže přirozenému jazyku. Navíc tyto jazyky odstiňují[6] programátora od nízkoúrovňových záležitostí, jako je například správa paměti.

[4] Dodejme, že striktní vymezení hranice mezi těmito dvěma skupinami neexistuje.

[5] Například se musí starat o práci s pamětí, způsob uložení dat a další.

[6] Programátor se o ně nemusí starat, ale ani je nemůže ovlivnit.

Programovací jazyky mohou být *kompilované* nebo *interpretované*. Rozdíl spočívá ve způsobu zpracování programu. Program zapsaný v kompilovaném programovacím jazyce, například v jazyce C, je nejprve pomocí *kompilátoru* (jiného programu) převeden do spustitelné podoby. Tomuto převodu se říká *kompilace*. Jejím výsledkem je spustitelný program obsahující instrukce, kterým počítač rozumí.[7] Při kompilaci je možné provést kontrolu správnosti *syntaxe* (zápisu) a částečně i *sémantiky* (chování) programu.

[7] Počítače se mohou lišit hardwarem, ale i operačním systémem. Při kompilaci je vytvářen spustitelný program pro konkrétní hardware a operační systém (pro konkrétní platformu).

Průvodce studiem
Syntaxe programu popisuje pravidla pro zápis programu v konkrétním programovacím jazyce. Sémantika popisuje chování programu, tedy to co program dělá.

Program zapsaný v interpretovaném programovacím jazyce, například v jazyce Scheme, je postupně, po instrukcích, vykonáván (*interpretován*) *interpretem* (jiným programem). Při interpretaci dochází k postupnému překladu programu do instrukcí, kterým počítač rozumí. Pro spuštění programu v interpretovaném jazyce musí být na počítači interpret jazyka, ve kterém je program zapsán. Na rozdíl od kompilace dochází při interpretaci ke kontrole syntaxe a částečně sémantiky až při běhu programu.[8] Interpretace programu navíc (obvykle) umožňuje interaktivní práci, při které je spuštěn interpret, kterému jsou přímo zadávány instrukce.[9]

[8] Kompilovaný program se v době svého běhu již nemusí kontrolovat, což šetří čas.

[9] Tento proces bývá označován jako cyklus REPL (z anglického Read Eval Print Loop). V tomto cyklu se čeká na načtení instrukce (read), která je následně vykonána (eval) a její výsledek je zobrazen (print). Následně se cyklus opakuje (loop).

V dnešní době řada programovacích jazyků využívá kombinaci kompilace a interpretace. Příkladem jsou jazyky Java, Python nebo C#. U těchto jazyků není program překládán přímo do instrukcí, kterým počítač rozumí, ale do instrukcí pro interpret, který je následně překládá do instrukcí, kterým počítač rozumí. Kompilace v tomto případě představuje pouze mezikrok a interpret pomyslný virtuální počítač.[10] Tyto jazyky, přestože používají kompilaci, jsou běžně označovány za interpretované.

[10] Běžně označovaný jako virtuální stroj.

Výběr programovacího jazyka

Nabízí se přirozená otázka: „Který programovací jazyk je ten nejlepší?“ Správná odpověď neexistuje. Každý programovací jazyk má své pro i proti. Programovací jazyky se liší především v tom, jaký poskytují programátorovi komfort a možnosti při řešení konkrétního problému.
Například jazyk C se hodí (má prostředky) pro systémové programování, které je blíže hardware počítače. Na druhou stranu, vytvářet pomocí tohoto jazyka webovou aplikaci, přestože je to možné, je nesmysl. V tomto případě je výhodnější zvolit jazyk, který je určený pro tvorbu webových aplikací. Správná volba programovacího jazyka je klíčová k úspěšnému vyřešení problému.
Přestože ideální volba neexistuje, pro účely výuky programovaní, si alespoň jeden programovací jazyk vybrat musíme. Tímto jazykem bude programovací jazyk Python.

Průvodce studiem
Všechny běžně používané programovací jazyky, například jazyky z rodiny C, Java, Python nebo JavaScript, jsou tzv. *Turingovsky úplné* a lze pomocí nich popsat každý algoritmus. Zjednodušeně řečeno, je-li problém řešitelný v jednom z těchto jazyků, je řešitelný v libovolném z nich.

Jazyk Python

Programovací jazyk Python vytvořil v roce 1989 Guido van Rossum, který se na jeho vývoji podílí dodnes.[11] Jazyk je pojmenován po britské televizní show Monty Pythonův létající cirkus.[12] V průběhu let se stal jazyk Python velice populární[13] a to zejména pro svoji jednoduchost a nesčetné množství dostupných knihoven. Běžně je používán pro programování, vytváření prototypů, data science, webové aplikace a řadu dalších. Jazyk samotný ovlivnil vývoj dalších programovacích jazyků, které z něj převzali některé koncepty. Mezi tyto jazyky patří například: Go, Swift, Ruby, Julia nebo JavaScript.
Jazyk Python dlouhá léta existoval ve dvou separátních řadách, v řadě 2.X a řadě 3.X, které nejsou navzájem kompatibilní. Řada 2.X již není od roku 2020 podporována,[14] byť je právě v této řadě, především ve verzi 2.7, napsáno a provozováno značné množství programů. V rámci tohoto kurzu budeme využívat aktuální verzi řady 3.X. Můžeme si ale dovolit použít i starší varianty z řady 3.X. Jednotlivé verze jazyka se liší především v pokročilejší funkcionalitě, které se věnovat nebudeme.

[11] Každý rok vycházejí průběžné aktualizace jazyka.

[12] V anglickém originále Monty Python's Flying Circus. Hodnocení 8,8 dle IMDb.

[13] To dokazuje každoroční obsazování prvních příček v různých žebříčcích nejpopulárnějších programovacích jazyků.

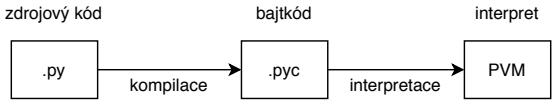
[14] Není doporučeno ji používat a to zejména z důvodu bezpečnosti.

První program v jazyce Python

Již víme, že zdrojový kód programu je sekvence instrukcí zapsaných v konkrétním jazyce. Zdrojový kód v jazyce Python je uložen v textovém souboru s příponou `.py`. Při jeho prvním spuštění je program kompilátorem přeložen do tzv. *bajtkódu*,[15] který je uložen

[15] Bajtkód jsou instrukce pro interpret jazyka Python.

(obvykle) v souboru s příponou `.pyc`. Bajtkód je následně vykonán interpretem jazyka.[16] Postup je ilustrován na obrázku 1.



Obrázek 1: Zdrojový kód v jazyce Python je nejprve kompilován do bajtkódu, který je následně interpretován interpretem jazyka Python (PVM).

Pokud nedojde ke změně programu (jeho zdrojového kódu), nedochází ke kompilaci do bajtkódu a bajtkód je rovnou interpretován.
Nyní si vyzkoušíme vytvořit první program v jazyce Python. Budeme k tomu potřebovat vývojové prostředí[17] a interpret jazyka Python.[18] Jako vývojové prostředí je možné použít prakticky jakýkoliv textový editor, který umožňuje uložení souboru v čistě textovém formátu. Při programování je ale výhodnější zvolit vývojové prostředí podporující programování v daném jazyce.[19] Vývojových prostředí podporující jazyk Python je celá řada. Mezi ty nejpopulárnější patří Visual Studio Code,[20] které budeme používat.
Postup vytvoření programu v jazyce Python, za předpokladu, že máme nainstalován interpret jazyka Python a vývojové prostředí, je následující:

1. Spustíme vývojové prostředí.
2. Vytvoříme nový soubor do kterého zapíšeme následující zdrojový kód jednoduchého programu.[21]

```
print("Ahoj světe!")
```

3. Soubor uložíme jako `muj_prvni_program.py`.
4. Program spustíme. To můžeme udělat přímo ve vývojovém prostředí v menu „Run“, položka „Run without debugging“,[22] Program je také možné spustit přímo z terminálu (a to z terminálu ve vývojovém prostředí nebo terminálu operačního systému) příkazem `python muj_prvni_program.py`.

Úkol 1
Nainstalujte interpret jazyka Python, vývojové prostředí Visual Studio Code a vytvořte program v jazyce Python dle uvedeného postupu.

Postupně si budeme osvojovat správné programátorské návyky. Tím nejzákladnějším vůbec je ukládání provedené práce. Dobrým zvykem je, že pokud programátor zvedne ruce z klávesnice, svou práci si uloží.[23]

[16] Iterpret jazyka Python se nazývá Python virtual machine (PVM).

[17] Anglicky Integrated Development Environment běžně označované zkratkou IDE.

[18] Interpretů existuje několik. My budeme používat ten nejrozšířenější, CPython, volně dostupný na URL `https://www.python.org/`.

[19] Takové vývojové prostředí výrazně usnadňuje programování. Například zvýrazňuje syntaxi jazyka, napovídá programátorovi, vypisuje chyby a další.

[20] Volně dostupné na URL: `https://code.visualstudio.com/`.

[21] Jedná se o klasický první program, kterým začíná téměř každá učebnice věnující se programování. Program vypíše `Ahoj světe!`.

[22] Lze využít i klávesovou zkratku. `CTRL+F5` (Windows, Linux), `control + F5` (MacOS). Spuštění programu je jeden z nejčastějších úkonů. Je tedy dobré si klávesové zkratky osvojit.

[23] Stačí `ctrl + S`, či `command + S`. Byť toto může znít úsměvně, každý programátor přišel o kus své práce právě tím, že si ji takto neuložil.

Shrnutí

Stručně jsme vysvětlili základní pojmy z oblasti programování a programovacích jazyků. V rámci prvního úkolu nainstalovali překladač jazyka Python, vývojové prostředí a vytvořili jednoduchý program v jazyce Python.

Kontrolní otázky
Odpovězte na následující otázky:

1. Co je to program?
2. Co je syntaxe a sémantika programu?
3. Jaký je rozdíl mezi kompilovaným a interpretovaným programovacím jazykem?
4. Jakým způsobem je zdrojový kód jazyce Python přeložen na instrukce, kterým počítač rozumí?

Základy programování

„The only way to learn a new programming language is by writing programs in it.“

Dennis Ritchie

V následující kapitole se zaměříme na základní prvky, ze kterých je složen program. Vysvětlíme pojmy hodnota, výraz a příkaz. Dále se budeme věnovat proměnným a operátorům a to zejména operátoru přiřazení.

Základní pojmy

Základními stavebními kameny každého programovacího jazyka jsou: *hodnota*, *výraz* a *příkaz*. Hodnoty reprezentují informace (data), se kterými programy pracují.[24] Výrazy slouží pro vytváření hodnot. Pokud například v programu napíšeme výraz

```
42
```

jeho vyhodnocením vytvoříme hodnotu. Hodnoty jsou v jazyce Python reprezentovány jako *objekty*.[25] Ve výše uvedeném příkladu, výraz 42 vytváří objekt „číslo“, který reprezentuje konkrétní číselnou hodnotu 42. Příkazy dávají počítači instrukce, co má dělat. Následující příkaz zobrazí hodnotu 42.[26]

```
print(42)
```

Počítačový program není nic jiného než sekvence příkazů.

Proměnné

Proměnné chápeme jako entity, které mohou obsahovat hodnoty. Každá proměnná má své jedinečné jméno, označované jako *identifikátor proměnné*,[27] které ji identifikuje a skrze něž můžeme s proměnnou dále pracovat.

[24] Hodnotou je například číslo 42, text „spam“, množina čísel.

[25] Pro jednoduchost si vystačíme s intuitivním chápáním tohoto pojmu: entita reprezentující data.

[26] Přesněji řečeno zobrazí objekt číslo, reprezentující hodnotu 42.

Průvodce studiem
Objekty, kromě hodnoty samotné, nesou i další informace, například typ hodnoty. Programátor pracuje s těmito objekty a jejich vnitřní struktura jej nemusí zajímat. Tímto nás Python odstiňuje od nízkoúrovňových záležitostí, jako je správa paměti.

[27] Pojem identifikátor je běžný termín pro jména používaná v programovacích jazycích.

Identifikátor v jazyce Python je kombinace malých písmen (`a-z`), velkých písmen (`A-Z`), číslic (`0-9`) a znaku `_` (podtržítko). Identifikátor nesmí začínat číslicí a jako identifikátor nelze použít *klíčová slova* jazyka (tabulka 1), která mají speciální význam.

`and`	`del`	`from`	`not`	`while`
`as`	`elif`	`global`	`or`	`with`
`assert`	`else`	`if`	`pass`	`yield`
`break`	`except`	`import`	`print`	
`class`	`exec`	`in`	`raise`	
`continue`	`finally`	`is`	`return`	
`def`	`for`	`lambda`	`try`	

Tabulka 1: Klíčová slova jazyka Python.

Jazyk Python rozlišuje velká a malá písmena.[28] Například `velikost` a `Velikost` jsou v jazyce Python dva různé identifikátory.
S každou proměnnou je kromě hodnoty, kterou obsahuje, spojen i její typ. Existují dva druhy typování: *statické* a *dynamické*. Rozdíl spočívá ve vazbě mezi proměnnou a typem, která je buď pevně daná nebo ji lze měnit. Podle použitého typování se programovací jazyky dělí na: *staticky typované* a *dynamicky typované*.
Staticky typované programovací jazyky spojují typ se jménem proměnné. V těchto programovacích jazycích, je nutné při vytváření proměnné určit i její typ, který je již neměnný. Proměnná pak může obsahovat pouze data tohoto typu.
Jazyk Python je dynamicky typovaný jazyk. To znamená, že typ proměnné je určen typem objektu, který reprezentuje hodnotu, jež proměnná obsahuje a tento typ lze měnit. Například `42` je objekt typu „celé číslo“, zatímco `42.0` je objekt typu „desetinné číslo“.[29]

[28] Programovací jazyky, které mají tuto vlastnost se označují jako *case-sensitive*.

[29] Běžněji se používá anglické označení *integer* pro celá čísla a *float* pro desetinná čísla.

Průvodce studiem
V programovacích jazycích existuje pro jednotlivé entity doporučená podoba identifikátorů, také známá jako *pojmenovávací konvence*. V jazyce Python se pro identifikátory proměnných (a funkcí) používá tzv. *snake case* konvence. V této konvenci se identifikátory zapisují pomocí malých písmen a znak `_` (podtržítko) je použit pro oddělení slov (pokud jich je více). Například `odpoved_na_otazku` nebo `promena_42`. Mezi další běžně používané konvence patří: *camel case* konvence (první písmeno malé, každé další slovo má první písmeno velké, slova se nijak neoddělují), *pascal case* konvence (stejné jako camel, ale první písmeno je velké) a *kebab case* konvence (jako snake, ale používá se `-` (pomlčka) na místo podtržítka.

Komentáře

Do zdrojového kódu programu je možné zapisovat komentáře. Ty slouží pro zápis poznámek a dokumentování zdrojového kódu. Komentáře chování programu nijak neovlivňují, ale jejich význam je zcela zásadní. Zdrojové kódy jsou častěji čteny než psány. Je běžné, že se programátor ke zdrojovým kódům vrací a to mnohdy i po velice dlouhé době. Navíc se zdrojovými kódy nemusí pracovat pouze jejich autor, ale i další programátoři. Výstižný komentář výrazně usnadní pochopení zdrojového kódu. Komentáře se v jazyce Python zapisují pomocí symbolu `#` (čte se heš).

```
# tento text je komentář v programu
```

Operátor	Příklad použití	Popis operátoru
`-`	`-42`	mění znaménko operandu (unární operátor)
`+`	`40 + 2`	součet dvou operandů
`-`	`40 - 2`	rozdíl dvou operandů
`*`	`40 * 2`	násobení dvou operandů
`/`	`40 / 2`	dělení dvou operandů
`%`	`40 % 2`	operace modulo (zbytek po celočíselném dělení prvního operandu druhým)
`**`	`40 ** 2`	mocnina (40^2)
`//`	`40 // 2`	výsledek dělení prvního operandu druhým zaokrouhlený směrem dolů (floor division)

Tabulka 2: Aritmetické operátory a jejich popis.

Operátory

Operátory umožňují kombinovat jednodušší výrazy a tím vytvářet výrazy složitější. Operátor obvykle chápeme jako operaci, kterou lze provést s nějakými *operandy* (argumenty).[30]

[30] Například `40 + 2`. `+` představuje operaci sčítání dvou čísel, `40` a `2` operandy této operace.

Aritmetické operátory

Pro základní operace s číselnými hodnotami slouží *aritmetické operátory*, které jsou shrnuty v tabulce 2. Ke každému operátoru je třeba vědět, na kolik operandů jej lze aplikovat. Tomuto číslu se říká *arita* operátoru.[31]

[31] Operátory s aritou jedna označujeme jako *unární*, operátory s aritou dva označujeme jako *binární* a operátory s aritou tři označujeme jako *ternární* operátory.

Pokud výraz obsahuje více různých operátorů, je důležité vědět, v jakém pořadí se budou operátory aplikovat. Pořadí, ve kterém se operátory vyhodnocují, se nazývá *priorita operátorů*. Operátory s větší prioritou mají přednost před operátory s menší prioritou. Pro aritmetické operátory platí stejná pravidla, jako pro jejich matematické protějšky.[32] Například výraz

[32] Včetně toho, že nelze dělit nulou.

```
1 + 2 * 3
```

se vyhodnotí na číslo `7`. Nejprve se vynásobí `2` a `3` (násobení má přednost před sčítáním), následně se k výsledku přičte `1`.

Pořadí, ve kterém se operátory vyhodnocují, je možné ovlivnit pomocí kulatých závorek.[33] Například výraz

[33] Opět platí stejná pravidla jako v matematice.

```
(1 + 2) * 3
```

se vyhodnotí na hodnotu `9`. Nejprve se vyhodnotí obsah závorky (na hodnotu `3`) a následně je tato hodnota vynásobena `3`.

V případě, že operátory mají stejnou prioritu, rozhoduje o pořadí jejich vyhodnocení *asociativita* operátoru.

Průvodce studiem
Kromě ovlivnění pořadí vyhodnocování operátorů, se závorky používají také pro zvýšení čitelnosti zdrojového kódu. Například následující dva výrazy:

```
a + b * c / 2
a + (b * c) / 2
```

se vyhodnotí stejně. Druhý uvedený je ale přehlednější.

Téměř všechny operátory v jazyce Python jsou *asociativní zleva*, a proto se vyhodnocují zleva doprava.[34] Například výraz

```
1 * 2 * 3 * 4 * 5
```

se vyhodnocuje jako by byl zapsán následovně.

```
(((1 * 2) * 3) * 4) * 5
```

Argumenty operátoru mohou být i proměnné.

```
a = 1
b = 2
c = 3
-a + b * c / 2
```

[34] Výjimku tvoří operátor `**`, který je asociativní zprava.

Úkol 2
Napište kód, který ukáže, že operátor `**` je asociativní zprava.

Příkaz přiřazení

Ve výše uvedeném příkladu jsme použily symbol `=`, který představuje jeden z nejdůležitějších příkazů vůbec, *příkaz přiřazení*. Ten slouží pro přiřazení hodnoty k proměnné. Obecný zápis příkazu přiřazení následuje.

```
l-value = r-value
```

Příkaz přiřazení funguje tak, že na místo v paměti[35] uloží výsledek vyhodnocení výrazu `r-value` (objekt reprezentující vzniklou hodnotu) a následně do proměnné určené `l-value` výrazem uloží *referenci*[36] na toto místo. Například příkaz

```
a = 42
```

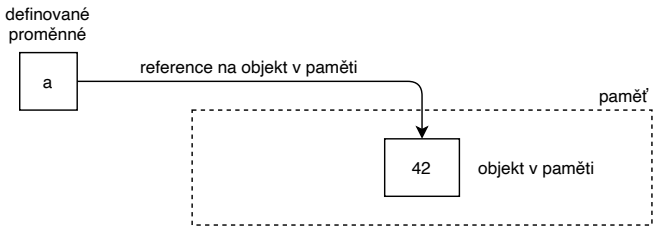
přiřadí proměnné `a` hodnotu 42.[37] V příkazu přiřazení se výraz vlevo od symbolu `=` označuje jako *l-value výraz*, a výraz vpravo od symbolu `=` se označuje jako *r-value výraz*.
Obrázek 2 ilustruje vytvoření reference na objekt v paměti po vyhodnocení výše uvedeného příkazu přiřazení.
Proměnná, která doposud nebyla použita, se označuje jako *nedefinovaná proměnná*. Pokud je proměnná použita jako l-value výraz v příkazu přiřazení poprvé, dochází k její *definici*.[38] V jazyce Python dochází vždy při definici proměnné i k její *deklaraci*. Deklarace označuje přiřazení reference k proměnné, tedy přiřazení hodnoty. V jazyce Python nelze vytvořit proměnou bez hodnoty. První nastavení hodnoty proměnné se také označuje jako *inicializace proměnné*.

[35] O konkrétním místě v paměti rozhoduje operační systém a programátor tuto volbu nemůže nijak ovlivnit.

[36] Nebo také odkaz, či ukazatel.

[37] Zjednodušeně říkáme, že proměnná `a` má hodnotu 42.

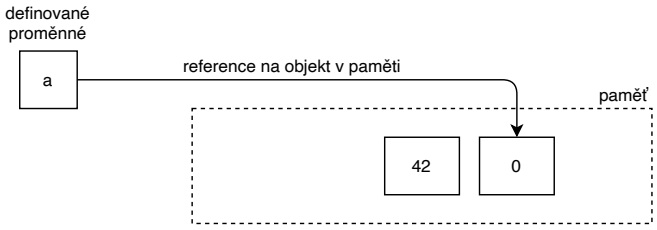
[38] To si lze jednoduše představit tak, že jméno proměnné je přidáno do seznamu použitelných (definovaných) proměnných.



Obrázek 2: Přiřazení proměnné `a = 42`. Nejprve se do paměti uloží objekt reprezentující výsledek vyhodnocení r-value výrazu (v našem případě hodnota 42). Následně se do l-value výrazu (v našem případě je jím proměnná `a`) uloží reference na vytvořený objekt v paměti.

Pokud je jako l-value výraz použita již deklarována proměnná, dochází ke změně její reference (obrázek 3).

```
a = 42  # proměnná a obsahuje hodnotu 42
a = 0
```



Obrázek 3: Změna reference proměnné `a`. Původní objekt v paměti (42) zaniká. O toto se stará automatická správa paměti tzv. *garbage colector*, která z paměti odstraňuje objekty, na které nejsou žádné reference.

Nelze pracovat s proměnnou, která není definována. Nedefinovanou proměnnou nelze použít v žádném výrazu ani příkazu. Jedinou výjimkou je příkaz přiřazení, kde může být nedefinovaná proměnná použita jako l-value výraz.

```
# b není definována
a = b # způsobí chybu: NameError: name 'b' is not defined
42 + b # způsobí chybu: NameError: name 'b' is not defined
```

Průvodce studiem
O běžných programátorských chybách budeme mluvit později. Interpret či překladač programovacího jazyka dokáže některé chyby (zejména syntaktické) odhalit a upozornit na ně pomocí chybového hlášení. Tyto hlášení obvykle velice přesně určují místo, kde k chybě došlo a s jejich pomocí je možné chyby rychle opravit. Je tedy důležité těmto chybovým hlášením dobře porozumět.

V případě, že l-value výraz není proměnná, dojde k chybě.[39]

```
42 = 42 # způsobí chybu: SyntaxError: can't assign to
    literal
```

[39] Později ukážeme další entity, které lze použít jako l-value výraz.

Výsledkem příkazu přiřazení není žádná hodnota.[40] V důsledku toho není možné s výsledkem přiražení pracovat jako s operandem. Například následujících několik výrazů způsobí chyby.

```
a = 40 + b = 2  # způsobí chybu: SyntaxError: can't assign
    to operator
(a = 40) + (b = 2) # způsobí chybu: SyntaxError: can't
    assign to operator
b = (a = 42)  # způsobí chybu: SyntaxError: invalid syntax
```

[40] Toto není úplně přesné. Výsledkem aplikace operátoru přiřazení je ve skutečnosti *vedlejší efekt*, což je důležitý pojem zejména ve funkcionálních programovacích jazycích. Pro úplnost dodejme, že v některých jazycích, například v jazyce C, je přiřazení výrazem a jeho vyhodnocením se získá hodnota.

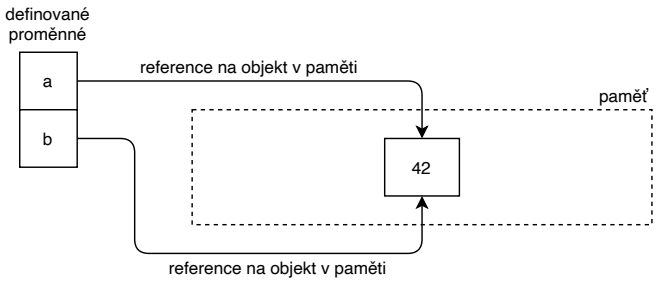
Na druhou stranu výraz

```
b = a = 42
```

chybu nezpůsobí a proměnná `a` i `b` budou mít hodnotu 42. Důvodem je, že tento typ zápisu[41] představuje *syntaktický cukr*, tedy alternativní, pohodlnější zápis jiného výrazu. V tomto případě je výše uvedený kód vyhodnocován tak, jako by byl zapsán následovně.[42]

```
a = 42
b = a
```

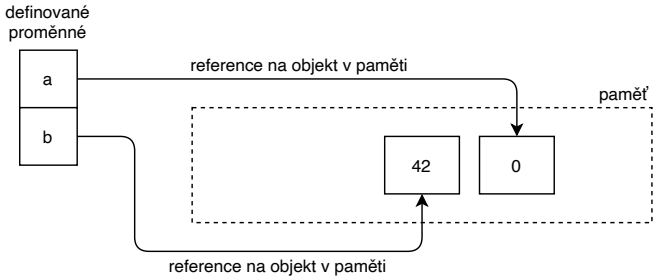
Obrázek 4 ukazuje jak vypadají reference po vykonání výše uvedeného kódu.



Pokud nyní dojde ke změně `a`, například

```
a = 0
```

nedojde k ovlivnění `b`, jelikož změnou `a` se mění pouze reference, tak jak je to ukázáno na obrázku 5.



Příkaz = lze kombinovat s aritmetickými (a dalšími) operátory. Například `a += b` odpovídá zápisu `a = a + b`. Ukázky těchto kombinovaných příkazu jsou uvedeny v tabulce 3.

[41] Označovaný jako zřetězení přiřazení.

[42] Zde je nutné upozornit na častou chybu, a sice `b = a` nikoliv `b = 42`.

Průvodce studiem
Syntaktický cukr je terminus technicus. Téměř každý programovací jazyk nějaký syntaktický cukr obsahuje.

Obrázek 4: Po vykonání `a = 42` obsahuje `a` referenci na 42, která je následně příkazem `b = a` uložena do `b`. `b` tedy obsahuje referenci na stejný objekt.

Průvodce studiem
V jazyce Python příkaz přiřazení vždy mění pouze referenci na objekt nikoliv objekt samotný.

Obrázek 5: Po vykonání `a = 0` je změněna reference uložená v `a`. Reference v `b` je nezměněna.

Kombinovaný operátor	Příklad použití	Klasický zápis
`+=`	`x += 42`	`x = x + 42`
`-=`	`x -= 42`	`x = x - 42`
`*=`	`x *= 42`	`x = x * 42`
`/=`	`x /= 42`	`x = x / 42`
`%=`	`x %= 42`	`x = x % 42`
`//=`	`x //= 42`	`x = x // 42`
`**=`	`x **= 42`	`x = x ** 42`

Tabulka 3: Příkaz přiřazení kombinovaný s aritmetickými operátory a jejich alternativní zápis.

Standardní výstup

Abychom mohli začít vytvářet jednoduché programy, potřebujeme, aby program akceptoval vstup a vrátil výstup. My si situaci zjednodušíme tak, že vstup do programu budeme chápat jako námi zadanou hodnotu proměnné[43] a výstup z programu budeme vypisovat do tzv. *standardního výstupu*, tedy do terminálu či konzole.
K tomuto účelu slouží funkce `print`.[44] O funkcích budeme mluvit později. Nyní si vystačíme s jejich intuitivním chápáním. Funkce `print` akceptuje jako vstup[45] entity, které mají být zobrazeny. Entitou se rozumí například textový řetězec, číslo nebo proměnná. Například

```
print(42) # zobrazí: 42
print("Ahoj světe") # zobrazí: Ahoj světe

a = 42
print(a) # zobrazí 42
```

Funkci `print` je možné předat i více argumentů, které se oddělují čárkou. Funkce tyto argumenty zobrazí v pořadí v jakém jsou uvedeny.

```
a = 42
print("Ahoj světe", a) # zobrazí: Ahoj světe 42
```

V základním nastavení přidává funkce `print` na konec výstupu znak konce řádku[46] a mezeru mezi každý argument. Byť je toto chování možné ovlivnit, je mnohem výhodnější použít *formátovací řetězec*.[47] Ten obsahuje text, jenž má být zobrazen, a ve složených závorkách (znaky `{` a `}`) místa, která jsou nahrazena výrazy uvedenými mezi složenými závorkami. Následující příklad ukazuje použití formátovacího řetězce.

```
a = 42
print(f"Ahoj světe {a}") # zobrazí: Ahoj světe 42
```

[43] Uvedenou ve zdrojovém kódu programu. Je ale možné tuto hodnotu nechat zadat uživatele v době běhu programu. K tomuto účelu se používá funkce `input()`, která vrací uživatelem zadaný textový řetězec.

[44] Funkce `print` má rozsáhlé možnosti. Pro naši aktuální potřebu si uvedeme pouze to nejzákladnější.

[45] argumenty funkce

[46] Konec řádku je ve většině programovacích jazyků reprezentován znaky zpětné lomítko a písmeno n, tedy `\n`.

[47] V jazyce Python existuje několik způsobů jak používat formátovací řetězce. My si ukážeme pro jazyk Python ten nejpřirozenější.

Funkci `print` je nejprve předán formátovací řetězec (uvozený písmenem `f`),[48]. Ten obsahuje kromě textu i ve složených závorkách uvedený výraz `a`, tedy název proměnné. Funkce postupuje tak, že výraz ve složených závorkách nahradí jeho hodnotou na kterou se vyhodnotí. V našem případě je touto hodnotou 42. Stejně tak je možné napsat například

[48] Od tohoto zápisu je odvozený název *f-string*.

```
print(f"40 + 2 = {40 + 2}") # zobrazí: 40 + 2 = 42
```

Pokud je ve složených závorkách uveden název proměnné, musí tato proměnná existovat, jinak dojde k chybě.

Úkol 3
Napište program, který pro zadanou stranu čtverce a vypíše jeho obsah.

```
print(f"{neexistujici_promenna}") # způsobí chybu: NameError
    : name 'neexistujici_promenna' is not defined
```

Pokud chceme, aby se ve formátovacím řetězci vyskytoval znak `{` nebo `}`, je třeba jej psát zdvojeně, tedy `{{` nebo `}}`.
Ve složených závorkách mohou být kromě jména proměnné či výrazu uvedeny další informace řídící tvar výstupu. Například počet zobrazených desetinných míst, zarovnání řetězce a další. Tyto údaje se zapisují za symbol `:` (dvojtečka) za výraz či název proměnné. Následující příklad ukazuje jak lze ovlivnit počet zobrazených desetinných míst.

```
pi = 3.14159265359

print(f"{pi}")
print(f"{pi:.2f}") # vypíše 3.14
print(f"{pi:.0f}") # vypíše 3
```

Shrnutí

V této kapitole jsme se seznámili se základními pojmy z programovacích jazyků a jejich zápisem v jazyce Python. Představili jsme proměnné a operátory, zejména příkaz přiřazení, a ukázali funkci `print()`, která umožňuje zobrazit výstup z programu do standardního výstupu. Nyní již známe vše potřebné pro vytvoření primitivních programů.

Kontrolní otázky
Odpovězte na následující otázky:

1. Co je to hodnota a jak je reprezentována v jazyce Python?
2. Co je to výraz a k čemu slouží?
3. Co je to příkaz?
4. Co je to operátor?
5. Jak funguje příkaz přiřazení v jazyce Python?
6. Co je to definice a deklarace proměnné?

Úkoly

Úkol 4
Napište program, který vymění hodnoty dvou proměnných.

Úkol 5
Napište program, který převede teplotu ve stupních Fahrenheita na teplotu ve stupních Celsia[49] a následně tuto hodnotu vypíše.

[49] Pro převod slouží vzorec: $c = \frac{5 \cdot (f-32)}{9}$, kde c jsou stupně Celsia a f jsou stupně Fahrenheita.

Úkol 6
Napište program, který převede částku v CZK na částku USD.

Úkol 7
Napište program, který převede úhel ve stupních na úhel v radiánech a ten vypíše.[50]

[50] $1\,^\circ = 1{,}745 \cdot 10^{-2}$ rad

Úkol 8
Napište program, který pro zadané tříciferné číslo vypíše všechny jeho číslice.

Základní datové typy

„Data! Data! Data! I can't make bricks without clay."

Sherlock Holmes

V následující kapitole si představíme základní datové typy: čísla, logické hodnoty a textové řetězce. Ukážeme, jak je s nimi možné pracovat v jazyce Python.

S každým datovým typem je spojena i jeho velikost,[51] která je určena reprezentací v paměti počítače. Jazyk Python nás od této reprezentace odstiňuje a samotná velikost datového typu není klíčová.[52]

[51] Používá se pojem rozsah datového typu nebo přesnost datového typu.

[52] To je typický rys vysokoúrovňových programovacích jazyků. V nízkoúrovňových jazycích, například v jazyce C, je naopak velikost datového typu zcela zásadní.

Čísla

Čísla jsou nejzákladnějším datovým typem vůbec.

Celá čísla

V předchozí kapitole jsme již celá čísla (například 42) používali. V jazyce Python není velikost celočíselného datového typu nijak omezena.[53] Lze tedy psát libovolně velká čísla.

[53] Respektive je omezena celkovou dostupnou pamětí počítače.

```
123456789123456789123456789123456789123456789
```

Kromě čísel v desítkové soustavě, lze používat i celá čísla ve dvojkové, osmičkové a šestnáctkové soustavě. Čísla ve dvojkové (binární) soustavě začínají prefixem `0b`, čísla v osmičkové soustavě začínají prefixem `0o`, čísla v šestnáctkové soustavě začínají prefixem `0x`. Po prefixu následuje zápis čísla v dané soustavě. Následující příklad ukazuje zápisy čísla 42 v různých soustavách.

Průvodce studiem
Na místo pojmu celé číslo se v programovacích jazycích běžněji používá anglicky pojem *integer*.

```
# zápis čísla 42
0b101010 # dvojková soustava
0o52 # osmičková soustava
0x2A # šestnáctková soustava
```

Pří zápisu čísel je možné libovolně používat velká i malá písmena.

Desetinná čísla

Desetinná čísla, například $42,0$, jsou v počítačích reprezentována jako čísla s plovoucí řádovou čárkou. V jazyce Python je lze zapsat řadou způsobů tak, jak je ukázáno na následujícím příkladu. Všechny způsoby jsou ekvivalentní.

Průvodce studiem
Na místo zdlouhavého pojmu číslo s plovoucí řádovou čárkou se v programovacích jazycích běžněji používá anglicky pojem *float*.

```
4.2
4. # 4.0
.2 # 0.2
4e2 # 400.0
4e-2 # 0.04
4.2e-2 # 0.042
```

Při zápisu desetinných čísel se používá desetinná tečka. Hodnota za symbolem `e` říká o kolik míst má být posunuta desetinná čárka. Pokud je číslo kladné jedná se o posun doprava. Pokud je číslo záporné, jedná se o posun doleva.[54] Je možné použít `E` na místo `e`. Největší desetinné číslo, které lze v jazyce Python reprezentovat je `1.79e308`. Cokoliv většího je označeno jako nekonečno.[55] Nejmenší kladné nenulové desetinné číslo je `5e-324`. Jakékoliv menší je již bráno jako nula.

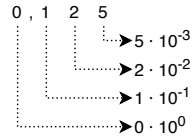
[54] Tento tvar čísla se označuje jako *vědecká notace*.

[55] Nekonečno je v jazyce Python reprezentováno jako číslo, které je větší než všechna ostatní.

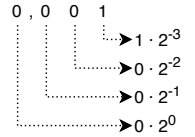
Problém s přesností

Čísla s plovoucí řádovou čárkou v desítkové soustavě lze reprezentovat jako součet zlomků, kde jmenovatel obsahuje mocniny 10. V počítači jsou tato čísla reprezentována jako součet zlomků, kde jmenovatel obsahuje mocniny 2 (binární zlomky).[56] Problém této reprezentace je nepřesnost. Například číslo $\frac{1}{3}$ v desítkové soustavě nelze pomocí zlomků vyjádřit.[57] Analogicky číslo $\frac{1}{10}$ nelze vyjádřit pomocí binárních zlomků. Problém nastává při zobrazování těchto čísel, kdy je obvykle zobrazena zaokrouhlená hodnota, samotné číslo je ale uchováváno nezaokrouhlené.

[56] Například číslo $0,125$ lze pomocí zlomků reprezentovat následovně



V případě binárních zlomků jsou mocniny čísla 10 nahrazeny mocninami čísla 2. Například číslo $0,001$ zapsané pomocí binárních zlomků má stejnou hodnotu jako číslo $0,125$ zapsané pomocí zlomků o základu 10.




```
print(0.1) # zobrazí 0.1
print(1/10) # zobrazí 0.1
```

Toto často působí problémy například při porovnání hodnot.

```
print(0.1 + 0.1 + 0.1) # zobrazí 0.30000000000000004
```

Důsledkem toho je, že v počítači $0,1 + 0,1 + 0,1 \neq 0,3$.[58]

[57] Nezáleží zda použijeme $0,3$, $0,33$, $0,333$, ..., vždy získáme pouze přibližnou reprezentaci čísla $\frac{1}{3}$.

[58] Tento problém je snadno řešitelný tak, že neporovnáváme čísla samotná, ale jejich absolutní rozdíl porovnáme s požadovanou přesností.

Logické hodnoty

Součástí jazyka Python je logický datový typ,[59] který reprezentuje logickou pravdu (`True`) a logickou nepravdu (`False`). Jak uvidíme později, celá řada výrazů se vyhodnocuje právě na tento datový typ.

[59] Někdy označovaný jako booleovský.

```
a = True
b = False
```

Ve své podstatě se jedná o celočíselný datový typ. Hodnota `True` je v jazyce Python chápána jako hodnota 1 a hodnota `False` jako hodnota 0.

Řetězce

Řetězce jsou tvořeny sekvencí *znaků*, které jsou v počítači reprezentovány pomocí čísel. Čísla základních znaků udává ASCII tabulka, která je zobrazena v tabulce 4.

Průvodce studiem
Na místo českého pojmu řetězec se v programovacích jazycích běžněji používá anglicky pojem *string*.

číslo	znak	číslo	znak	číslo	znak	číslo	znak	číslo	znak	číslo	znak	číslo	znak	číslo	znak
0	(nul)	16	(dle)	32	␣	48	0	64	@	80	P	96	‘	112	p
1	(soh)	17	(dc1)	33	!	49	1	65	A	81	Q	97	a	113	q
2	(stx)	18	(dc2)	34	"	50	2	66	B	82	R	98	b	114	r
3	(etx)	19	(dc3)	35	#	51	3	67	C	83	S	99	c	115	s
4	(eot)	20	(dc4)	36	$	52	4	68	D	84	T	100	d	116	t
5	(enq)	21	(nak)	37	%	53	5	69	E	85	U	101	e	117	u
6	(ack)	22	(syn)	38	&	54	6	70	F	86	V	102	f	118	v
7	(bel)	23	(etb)	39	'	55	7	71	G	87	W	103	g	119	w
8	(bs)	24	(can)	40	(	56	8	72	H	88	X	104	h	120	x
9	(tab)	25	(em)	41	)	57	9	73	I	89	Y	105	i	121	y
10	(lf)	26	(eof)	42	*	58	:	74	J	90	Z	106	j	122	z
11	(vt)	27	(esc)	43	+	59	;	75	K	91	[	107	k	123	{
12	(np)	28	(fs)	44	’	60	<	76	L	92	\	108	l	124	|
13	(cr)	29	(gs)	45	-	61	=	77	M	93	]	109	m	125	}
14	(so)	30	(rs)	46	.	62	>	78	N	94	^	110	n	126	~
15	(si)	31	(us)	47	/	63	?	79	O	95	_	111	o	127	(del)

Tabulka 4: Základní ASCII tabulka. Jednotlivé znaky jsou reprezentovány pomocí 7-bitových čísel. Znaky v závorkách jsou speciální řídící znaky.

Některé jazyky, včetně jazyka Python, kromě základních znaků, z ASCII tabulky, podporují i další znaky.[60]
Jazyce Python lze řetězec vytvořit pomocí znaku ' (apostrof) nebo " (uvozovka).

[60] Například Unicode znaky.

```
s = 'spam'
s = "spam"
```

Oba způsoby jsou ekvivalentní. Obvykle, pokud textový řetězec obsahuje uvozovky, volí se zápis s apostrofem a naopak. Znak uvozovky a apostrofu je také možné vložit pomocí *escape sekvence*.[61]

[61] S escape sekvencí jsme se již setkali. `\n` vkládající znak nového řádku (hodnota 10 v ASCII tabulce) je escape sekvence.

```
# escape sekvence \' a \"
s = 'spam který obsahuje \' (apastrof)'
s = "spam který obsahuje \" (uvozovku)"
```

Délka řetězce není nijak omezena.[62] Jazyk Python umožňuje vytvořit i řetězec, který zabírá více řádků. Příklad následuje.

[62] Je limitována pouze dostupnou pamětí počítače.

```
# víceřádkový řetězec pomocí """
"""spam,
který zabírá
několik řádků"""

# víceřádkový řetězec pomocí '''
'''spam,
který zabírá
několik řádků'''
```

Číslo odpovídající konkrétnímu znaku v ASCII tabulce je možné zjistit pomocí funkce `ord()`. A znak odpovídající konkrétnímu číslu v ASCII tabulce lze zjistit pomocí funkce `chr()`.

```
print(ord("A")) # zobrazí 65
print(chr(65)) # zobrazí A
```

Úkol 9
Napište program, který zadané velké písmeno (znaky `A-Z`) převede na odpovídající malé písmeno (znaky `a-z`). Nápověda: použijte ASCII tabulku.

Převody mezi datovými typy

Mezi datovými typy je možné provádět převody. Například je možné textový řetězec převést na číslo. V jazyce Python se pro převody používají funkce. Pro převod na celé číslo se používá funkce `int()`, pro převod na desetinné číslo se používá funkce `float()` a pro převod na řetězec se používá funkce `str()`. Ukázka jejich použití následuje.

```
int(4.2) # převod desetinného čísla na celé číslo 4
int("42") # převod řetězce čísla na celé číslo 42
float(42) # převod celého čísla na desetinné číslo 42.0
float("42") # převod řetězce na desetinné číslo  42.0
```

```
str(4.2) # převod desetinného čísla na řetězec "4.2"
str(42)  # převod celého čísla na řetězec "42"
```

Převod z textového řetězce na celá či desetinná čísla je možný pouze v případě, že tento řetězec obsahuje syntakticky správně zapsané hodnoty.

```
int("4.2") # způsobí chybu
int("text") # způsobí chybu
```

Převod jednoho typu na druhý se označuje jako *přetypování*. Výše uvedené příklady jsou příklady *explicitního přetypování*.[63] K přetypování může dojít i automaticky (*implicitní přetypování*). Například při operaci sčítání celého a desetinného čísla dojde k implicitnímu přetypování výsledku na desetinné číslo.

[63] Přetypování vynucené programátorem.

```
print(40 + 2.0) # vypíše 42.0
```

V jazyce Python dochází vždy k implicitnímu přetypování výsledku operace dělení.

```
print(42 / 2) # zobrazí: 21.0
print(42 / 2.0) # zobrazí: 21.0
print(42.0 / 2) # zobrazí: 21.0
```

Implicitní přetypování je možné pouze v případě, že datové typy jsou kompatibilní.[64] Pokud je prováděná operace s nekompatibilními typy, dojde k chybě.[65]

[64] Například čísla.

[65] Programovací jazyky je možné rozdělit na *silně typované* a *slabě typované*. Silně typované programovací jazyky předcházejí provádění operací s nekompatibilními datovými typy. Slabě typované provádí implicitní přetypování.

```
4 + "2" # způsobí chybu: TypeError: unsupported operand type
    (s) for +: 'int' and 'str'
```

Python je silně typovaný programovací jazyk. Například není možné implicitně přetypovat textový řetězec na číslo.[66]

[66] Samozřejmě je toto možné vyřešit explicitním přetypováním. Například `int("2")`.

Operátory porovnání

V předchozí kapitole jsme si ukázali aritmetické operátory. Dalším typem operátorů jsou operátory porovnání, které jsou shrnuty v tabulce 5.
Vyhodnocením výrazu obsahující operátory porovnání je logická hodnota `True` v případě, že je výraz pravdivý, `False` pokud je výraz nepravdivý. Tyto výrazy se běžně označují jako *logické výrazy*. Několik příkladů logických výrazů následuje.

Operátor	Popis operátoru	Příklad použití
==	rovnost	x == y
!=	nerovnost	x != y
>	větší než	x > y
<	menší než	x < y
>=	větší než nebo rovnost	x >= y
<=	menší než nebo rovnost	x <= y

Tabulka 5: Operátory porovnání.

```
10 < 100 # True
10 < 0 # False
1 == 1 # True
```

Operátory porovnání mají menší prioritu než aritmetické operátory. Například výraz

```
40 + 2 >= 40 # True
```

se vyhodnotí tak, že je nejprve vypočítána hodnota `40 + 2` a následně se vyhodnotí logický výraz, který je pravdivý.

Úkol 10
Jaká bude výsledná hodnota výrazu `40 + (2 >= 40)`.

Logické operátory

Logické výrazy je možné spojovat pomocí logických operátorů `and`, `or` a `not`, které mají význam logické konjunkce, logické disjunkce a negace.[67]
Logické operátory mají menší prioritu než operátory porovnání. Navíc jsou jednotlivé operandy logických operátorů vyhodnocovány postupně zleva doprava. Uvažme následující kód

```
x = 4
y = 2

x < 10 and y < 10 # True
```

Nejprve se vyhodnotí první porovnání `x < 10` na hodnotu `True` následně se vyhodnotí druhé porovnání `y < 10` opět na hodnotu `True`. Celý výraz se tedy vyhodnotí na `True`. Pokud kód upravíme následovně

```
x = 4
y = 2

x > 10 and y < 10 # True
```

[67] Výsledná hodnota logické konjunkce (logický součin) dvou proměnných `A` a `B`

A	B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

Výsledná hodnota logické disjunkce (logický součet) dvou proměnných `A` a `B`

A	B	A or B
False	False	False
False	True	True
True	False	True
True	True	True

Výsledná hodnota negace proměnné `A`

A	not A
False	True
True	False

dojde i ke změně vyhodnocování. Opět je nejprve vyhodnoceno první porovnání tentokráte na hodnotu `False`. Tím celé vyhodnocování končí a výsledek je vyhodnocen na `False`. Důvodem je, že výsledek vyhodnocení operátoru `and` je pravdivý pouze tehdy, pokud jsou pravdivé oba jeho operandy. Druhý operand tedy není třeba vyhodnocovat. Operátor `or` se chová analogicky.
V případě, že je výraz tvořen kombinací logických operátorů je proces vyhodnocení totožný. Například

Průvodce studiem
Postupné vyhodnocování logických operátorů zleva doprava je známo jako *líné vyhodnocování*. Jedná se o klíčový koncept, který se používá v mnoha programovacích jazycích.

```
x = 4
y = 2

x < 10 and (y < 10 or y > 0)  # True
```

Vyhodnotí se `x < 10` a `y < 10`. `y > 0` se nevyhodnocuje. Výsledek vyhodnocení operátoru `or` je pravdivý pokud je pravdivý alespoň jeden jeho operand.
Při programování často potřebujeme ověřit, zda je proměnná v daném rozsahu. Například

```
x = 42

print(x > 0 and x < 100)
```

V jazyce Python lze použít zkrácený zápis výše uvedeného výrazu.

```
x = 42

print(0 < x < 100)
```

Trocha logiky

Nemálo lidí má nemalé problémy s nepochopením nekrátkých nezáporných vět. Jinak řečeno, mnoho lidí má problémy s porozuměním větám obsahujících mnoho záporů. Při zápisu logických výrazů je lepší se vyhnout složitým konstrukcím a tím zvýšit čitelnost kódu. Logické výrazy je možné zjednodušit pomocí zákonu asociativity, distributivity a DeMorganových zákonů.
Asociativní zákon pro logické výrazy `A` a `B`:

```
A and (B and C) = (A and B) and C,
  A or (B or C) = (A or B) or C.
```

Zákon distributivity pro logické výrazy `A` a `B`:

```
A and (B or C) = (A and B) or (A and C),
 A or (B and C) = (A or B) and (A or C).
```

DeMorganovy zákony pro logické výrazy `A` a `B`:

```
not (A and B) = not A or not B,
not A and not B = not (A or B).
```

Úkol 11
Napište program, který ověří správnost DeMorganových zákonů pro zadané hodnoty `A` a `B`.

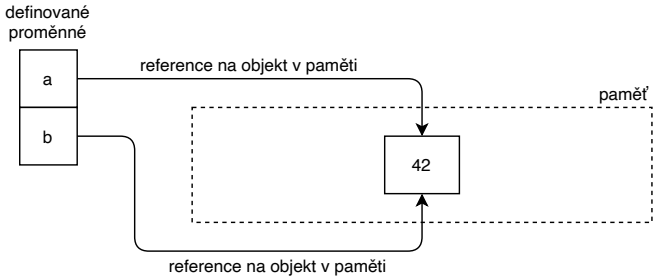
Operátor identity

Jak již víme, hodnoty jsou v jazyce Python reprezentované jako objekty. Operátory porovnání (tabulka 5) neporovnávají přímo tyto objekty, ale hodnoty, které reprezentují. Jazyk Python umožňuje ověřit, zda se jedná o stejný objekt. Pro tento účel slouží *operátor identity* `is`.[68] Uveďme si příklad.

[68] Případně jeho negovaná verze `is not`.

```
a = 42
b = 42

print(a == b) # True
print(a is b) # True
```

Tento příklad je velice zajímavý. Výraz `a == b` je samozřejmě pravdivý, ale to, že `a is b` je pravdivý znamená, že obě proměnné obsahují referenci na stejný objekt (obrázek 6).

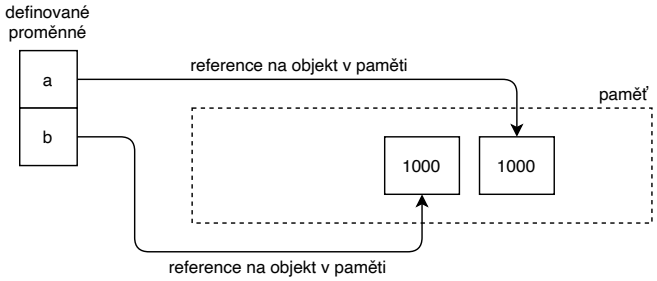


Obrázek 6: Jazyk Python optimalizuje kód tak, že objekty reprezentující menší číselné hodnoty a krátké texty vytváří pouze jednou.

Důvodem je optimalizace, kterou za nás provádí jazyk Python. Malé číselné hodnoty a krátké řetězce jsou vytvářeny pouze jednou. Pokud bychom použili větší hodnotu, například 1000, bude výsledek takový, jaký bychom očekávali (obrázek 7).

```
a = 1000
b = 1000

print(a == b) # True
print(a is b) # False
```

Úkol 12
Jakou hodnotu bude mít výraz `a is b` v případě, že `a = b = 1000`?



Obrázek 7: V případě větších hodnot již k optimalizaci nedochází.

Shrnutí

V této kapitole jsme se věnovali základním datovým typům, jejich reprezentaci a použití v jazyce Python. Navíc jsme rozšířili naše znalosti operátorů a přiblížili pojem a fungování logického výrazu, jehož hlavní využití uvidíme v nadcházejících kapitolách.

Kontrolní otázky
Odpovězte na následující otázky:

1. Jaké jsou základní datové typy jazyka Python?
2. Co je to přetypování?
3. Co je to logický výraz?
4. Jak jsou vyhodnocovány logické operátory?
5. Jak funguje operátor identity?

Úkoly

Úkol 13
Upravte logický výraz `not (not a and not b)` tak, aby se vyhodnocoval na stejné hodnoty jako původní výraz a neobsahoval negaci (`not`).

Úkol 14
Napište program, který pro dvě zadaná čísla vypočítá jejich součet, rozdíl a součin.

Větvení programu

„Simplicity is prerequisite for reliability.“

Edsger W. Dijkstra

V následující kapitole vysvětlíme pojem větvení programu, jeho zápis a fungování v jazyce Python.

Všechny programy, které jsme doposud uvažovali, byly vykonávány tak, že se postupně (sekvenčně) prováděly příkazy v nich uvedené. Pořadí, ve kterém se jednotlivé příkazy provádějí, se nazývá *tok programu*. Ten je možné ovlivnit pomocí příkazů pro řízení toku programu.[69] Mezi tyto příkazy patří *větvení programu* a *cykly*. Větvení programu umožňuje rozhodnout, na základě *podmínky*, jaká část programu se bude vykonávat. Pro zápis větvení programu se v jazyce Python používají příkazy `if`, `elif` a `else`. Jedná se o *složené příkazy*, tedy příkazy, které mohou obsahovat další příkazy.
Každé větvení programu se skládá z logického výrazu reprezentujícího podmínku a *bloků* (příkazů), které se vykonávají v závislosti na splnění či nesplnění podmínky.
Blok kódu představuje ohraničenou skupinu příkazů.[70] V jazyce Python se pro zápis bloků používá, oproti jiným jazykům poněkud netradičně, odsazení pomocí tabulátoru. V následující části si ukážeme různé způsoby zápisu větvení programu.

[69] Pro řízení programu se používá anglický termín *control flow*.

[70] Skupina příkazů, která má začátek a konec.

Příkaz `if`

Jedná se o základní příkaz pro větvení programu. Obecný zápis vypadá následovně.

```
if podmínka:
  příkazy
```

Ve výše uvedeném je `podmínka` logický výraz. Pokud se tento výraz vyhodnotí na hodnotu `True`, říkáme, že *podmínka je splněna*, dojde k vykonání bloku kódu označeného `příkazy`. Tomuto bloku se také říká *tělo podmínky*. V případě, že `podmínka` je vyhodnocena

na `False`, říkáme, že *podmínka není splněna*, příkazy v těle podmínky (blok `příkazy`) se nevykonávají a program pokračuje dalšími příkazy, které jsou uvedeny až za tělem podmínky. Průběh větvení programu při vykonávání příkazu `if` je ukázán na obrázku 8. Uveďme si jednoduchý příklad.

```
1  # výpočet absolutní hodnoty rozdílu kladných čísel x a y
2  x = 40
3  y = 2
4  absolutni_hodnota_rozdilu_x_y = 0
5
6  if x == y:
7    print("x je rovno y")
8
9  if x > y:
10   print("x je větší než y")
11   absolutni_hodnota_rozdilu_x_y = x - y
12
13 if x < y:
14   print("x je menší než y")
15   absolutni_hodnota_rozdilu_x_y = y - x
16
17 print(f"|x - y| = {absolutni_hodnota_rozdilu_x_y}")
```

Postupně jsou vykonány příkazy na řádcích 2–4. Výraz `x == y` na řádku 6 je nepravdivý (vyhodnotí se na `False`). Podmínka není splněna a její tělo (řádek 7) se nevykoná. Pokračuje se příkazem na řádku 9. Tato podmínka je pravdivá (vyhodnotí se na `True`) a dojde k vykonání jejího těla (řádky 10 a 11). Podmínka na řádku 13 je nepravdivá, nedojde k vykonání jejího těla. Jako poslední se vykoná příkaz na řádku 17.[71]
Za dvojtečkou v příkazu `if` se očekává blok. Ten musí být odsazen, jinak dojde k chybě.

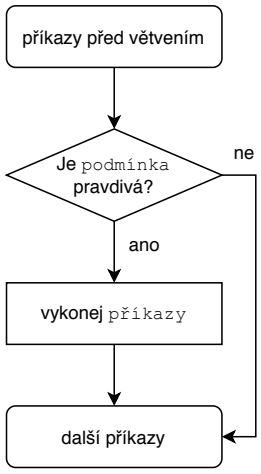
```
if x > y:
print("x je větší než y") # způsobí chybu: IndentationError:
      expected an indented block
```

Výjimkou je případ, kdy blok kódu obsahuje pouze jeden příkaz (řádek). V tomto případě lze větvení programu `if` zapsat následovně.[72]

```
if x > y: print("x je větší než y")
```

V podmínkách lze efektivně využít líného vyhodnocování logických operátorů. Například chceme ověřovat hodnotu výrazu `a / b`. Následující kód je nedostatečný, jelikož nepočítá s případem (skončí

Obrázek 8: Grafické znázornění větvení programu při vykonávání příkazu `if`.



Průvodce studiem
Blok uvedený za příkazem `if` a za příkazy `elif` a `else`, které uvedeme později) se běžně označuje jako *větev* programu.

[71] **Výstup programu:**

```
x je větší než y
|x - y| = 38
```

[72] Tento způsob zápisu ale není moc přehledný.

chybou), kdy `b = 0`.

```
if (a / b) > 1:
  print("a je měnší než b")
```

Uvedený kód můžeme snadno upravit tak, aby k dělení nulou nedocházelo, pomocí logického operátoru `and`.

```
if b != 0 and (a / b) > 1:
  print("a je měnší než b")
```

Jelikož je první operand operátoru `and` vyhodnocen na `False`, další operand se již nevyhodnocuje.

Příkaz `else`

Za tělo podmínky je možné uvést nepovinný příkaz `else`, který určuje větev programu, jež se vykoná v případě, že podmínka před ním uvedená není splněna. Tento příkaz může být pouze jeden. Zápis větvení programu s příkazem `else` vypadá následovně.

```
if podmínka:
  příkazy_1
else:
  příkazy_2
```

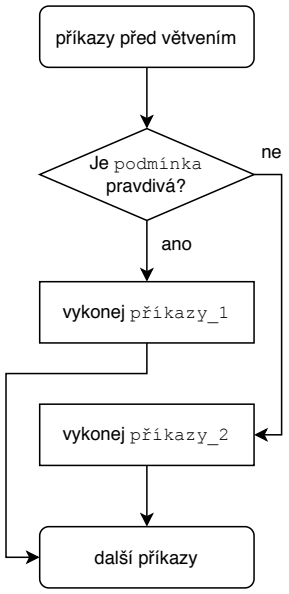
`podmínka` opět představuje logický výraz. Pokud se tento výraz vyhodnotí na hodnotu `True`, dojde k vykonání bloku `příkazy_1`. V případě, že je `podmínka` vyhodnocena na `False`, vykoná se blok kódu označený `příkazy_2`.[73] Větvení programu, používající příkazy `if` a `else`, je ukázáno na obrázku 9. Uveďme si příklad.

```
1  # absolutní hodnota čísla x
2  x = -42
3  absolutni_hodnota_x = x
4
5  if x >= 0:
6    print("x je kladné")
7  else:
8    print("x je záporné")
9    absolutni_hodnota_x *= -1
10
11 print(f"|x| = {absolutni_hodnota_x}")
```

Postupně se vykonají příkazy na řádcích 2–3. Podmínka na řádku 5 není splněna, řádek 6 je přeskočen a vykoná se else-větev programu, tedy blok na řádcích 8 a 9. Následně je vykonán příkaz

[73] Běžně označovaný jako *else-blok* případně *else-větev*.

Obrázek 9: Grafické znázornění větvení programu při vykonávání příkazů `if...else`.



na řádku 11. Pokud by `x = 42`, vykonal by se příkaz na řádku 6, else-větev by byla přeskočena a program by pokračoval příkazem na řádku 11.
Blok za příkazem `else` musí být opět správně odsazen, jinak dojde k chybě. Stejně jako v případě `if`, pokud blok za `else` obsahuje pouze jeden příkaz (řádek), je možné jej zapsat přímo na řádek, kde se příkaz `else` nachází.

Příkaz `elif`

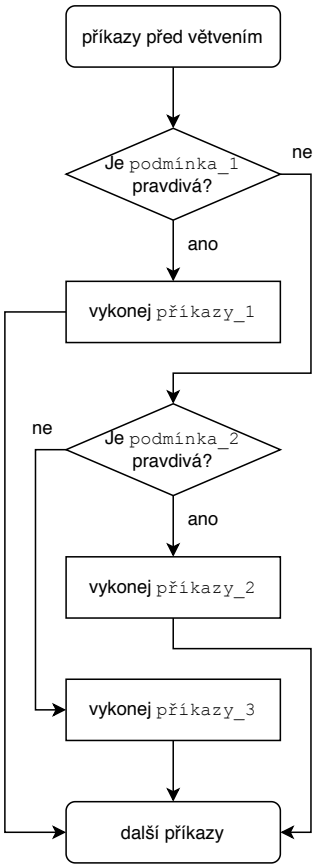
Další nepovinný příkaz kterým je možné rozšířit zápis větvení programu je příkaz `elif`. Ten určuje alternativní větve programu, které se vykonají za předpokladu, že všechny výše uvedené podmínky[74] jsou nepravdivé a podmínka s tímto příkazem spojená je pravdivá. Obecný zápis vypadá následovně.

[74] Logické výrazy za příkazy `if` a `elif`

```
if podmínka_1:
  příkazy_1
elif podmínka_2:
  příkazy_2
else:
  příkazy_3
```

`podmínka_1` představuje logický výraz. Pokud se tento výraz vyhodnotí na hodnotu `True`, dojde k vykonání bloku `příkazy_1` a pokračuje se vykonáváním kódu za posledním blokem (tedy za `příkazy_3`). Pokud se `podmínka_1` vyhodnotí na `False`, pokračuje se příkazem `elif`. Ten se chová analogicky jako příkaz `if`, tedy vyhodnocuje se `podmínka_2` a v závislosti na jejím splnění se vykoná nebo přeskočí blok následující za tímto příkazem. Dodejme, že příkazů `elif` může být v podmínce více. Blok `příkazy_3` se vykoná za předpokladu, že žádná z výše uvedených podmínek nebude pravdivá. Větvení programu používajícího příkazy `if`, `elif` a `else` je ukázáno na obrázku 10.
Příkaz `elif` nám například umožní lépe vyřešit výpočet absolutní hodnoty rozdílu dvou kladných čísel, které jsme si uvedli dříve.

Obrázek 10: Grafické znázornění větvení programu při vykonávání příkazů `if...elif...else`.




```
1  # výpočet absolutní hodnoty rozdílu kladných čísel x a y
2  x = 40
3  y = 2
4  absolutni_hodnota_rozdilu_x_y = 0
5
6  if x > y:
7    print("x je větší než y")
8    absolutni_hodnota_rozdilu_x_y = x - y
9
10 elif x < y:
```

```
11    print("x je menší než y")
12    absolutni_hodnota_rozdilu_x_y = y - x
13
14  else:
15    print("x je rovno y")
16
17  print(f"|x - y| = {absolutni_hodnota_rozdilu_x_y}")
```

Pro úplnost dodejme, že příkaz `else` na řádku 14, je možné nahradit `elif x == y` aniž by došlo ke změně funkce programu.

Zanořené větvení programu

Příkazy pro větvení programu je možné do sebe libovolně *zanořovat*.[75] Při zanoření větvení programu je nutné dodržet správné odsazení jednotlivých bloků, jelikož odsazení bloků řídí sémantiku programu.

[75] Bloky s podmínkami spojené, mohou obsahovat další příkazy pro větvení programu. Toto se běžně označuje jako zanořené větvení programu.

```
1  # výpočet absolutní hodnoty rozdílu kladných čísel x a y
2  x = 40
3  y = 2
4  absolutni_hodnota_rozdilu_x_y = 0
5
6  if x != y:
7    if x > y:
8      print("x je větší než y")
9      absolutni_hodnota_rozdilu_x_y = x - y
10   else:
11     print("x je menší než y")
12     absolutni_hodnota_rozdilu_x_y = y - x
13
14 else:
15   print("x je rovno y")
16
17 print(f"|x - y| = {absolutni_hodnota_rozdilu_x_y}")
```

Větvení programu na řádcích 7–12 je zanořeno ve větvení programu na řádcích 6–15.

Úkol 15
Upravte uvedený kód, aniž by došlo ke změně fungování programu tak, aby bylo tělo podmínky na řádku 6 (tedy řádky 7–12) zanořeno v bloku else na řádku 15.

Ternární operátor

Ternární operátor[76] je speciálním případem podmínky, který se používá zejména ve spojení s přiřazením. Zápis vypadá následovně.

[76] Tento název, byť běžně používaný, je poněkud matoucí. Každý operátor s aritou tři je ternární. V jazyce Python, stejně jako v řadě dalších programovacích jazyků, existuje pouze jeden operátor s aritou tři. Z tohoto důvodu jej lze takto pojmenovat.

```
proměnná = výraz_1 if podmínka else výraz_2
```

`proměnná` bude obsahovat hodnotu vzniklou vyhodnocením výrazu `výraz_1` pokud je logický výraz `podmínka` pravdivá, jinak bude obsahovat hodnotu vzniklou vyhodnocením výrazu `výraz_2`. Například výpočet absolutní hodnoty dvou kladných čísel by se pomocí ternárního operátoru dal zjednodušit následovně.

```
# výpočet absolutní hodnoty rozdílu kladných čísel x a y
absolutni_hodnota_rozdilu_x_y = x - y if x >= y else y - x
```

Ternární operátor je syntaktický cukr.[77] Výše uvedený kód je možné přepsat následovně.

[77] Hodně používaný.

```
# výpočet absolutní hodnoty rozdílu kladných čísel x a y
if x >= y:
  absolutni_hodnota_rozdilu_x_y = x - y
else:
  absolutni_hodnota_rozdilu_x_y = y - x
```

Ternární operátor je možné použít i jako běžnou podmínku.

```
příkaz_1 if podmínka else příkaz_2
```

Ternární operátory je také možné zanořovat.

Přetypování v podmínkách

Doposud byly vždy podmínky určeny logickými výrazy, tedy výrazy, které se vyhodnocují na hodnoty `True` nebo `False`. Ve skutečnosti lze v podmínce použít libovolný výraz. Jeho výsledná hodnota je implicitně přetypována na logickou hodnotu. Toto přetypování funguje tak, že vše je přetypováno na hodnotu `True` s výjimkou hodnoty `0` a prázdných objektů.[78]

[78] Objekty, které mají jako hodnotu prázdnou entitu. Například prázdný řetězec (řetězec, který neobsahuje žádné znaky). Další příklady uvedeme později.

```
if 42: # True, vypíše se
  print("Podmínka je splněna")

if -1: # True, vypíše se
  print("Podmínka je splněna")

if 0: # False, nevypíše se
  print("Podmínka je splněna")

if "spam": # True, vypíše se
  print("Podmínka je splněna")

if "": # False, nevypíše se
  print("Podmínka je splněna")
```

K výše uvedenému přetypování dochází i v případě vyhodnocování logických výrazů. Například.

```
print(0 and 42) # vypíše: 0
print(1 and 42) # vypíše: 42
print(1 or 42) # vypíše: 1
print(0 or 42) # vypíše: 42
```

Úkol 16
Proč příkaz `print(1 and 42)` vypíše 42 na místo hodnoty `True`?

Pro úplnost dodejme, že jazyk Python od verze 3.10 umožňuje používat příkazy `match` a `case`. Jejich triviálním použitím lze simulovat příkaz `switch`,[79] který je běžně používaný v řadě jiných programovacích jazyků pro větvení programu. Toto použití je ale nevhodné, proto jej neuvádíme.

[79] Jazyk Python tímto příkazem nedisponuje.

Shrnutí

V této kapitole jsme představili příkazy umožňující řízení toku programu. Ty nám umožňují vytvářet mnohem komplexnější programy, jejichž chování je ovlivněno aktuálním stavem proměnných.

Kontrolní otázky
Odpovězte na následující otázky:

1. K čemu slouží podmínky?
2. Co je to ternární operátor?

Úkoly

Úkol 17
Napište program, který vypíše známku na základě počtu bodů získaných v testu. Známka je dána následující tabulkou.

Body	Známka
90–100	A
80–89	B
76–79	C
71–75	D
60–70	E
0–59	F

Úkol 18
Napište program, který zjistí zda je zadané číslo liché nebo sudé.

Úkol 19
Napište program bez použití příkazu `if`, který vytiskne zadanou

číselnou hodnotu, pokud je menší než 100. Nápověda: použijte logický operátor.

Úkol 20
Napište program, který rozhodne, zda je zadaný rok přestupný nebo ne.[80]

[80] Pokud číslo roku není dělitelné 4, rok není přestupný. Pokud je číslo roku dělitelné 4 a není dělitelné 100, rok je přestupný. Pokud je číslo roku dělitelné 100 a není dělitelné 400, rok není přestupný. Pokud je číslo roku dělitelné 400, rok je přestupný.

Cykly

„First, solve the problem. Then, write the code.“

John Johnson

V následující kapitole si vysvětlíme pojem cyklus v programu, jeho zápis a použití v jazyce Python.

Cykly umožňují opakovaně provádět určitý blok kódu.[81] V jazyce Python existují dva typy cyklů, cyklus `while` a cyklus `for`.

[81] Stejně jako podmínky, tak i cykly patří do skupiny příkazů, které umožňují řízení toku programu.

Cyklus `while`

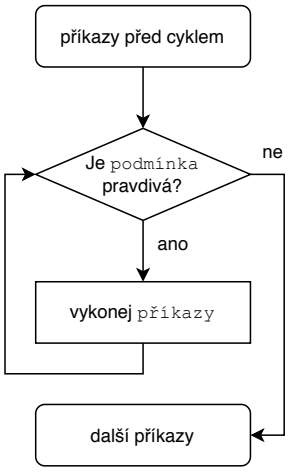
Obecný zápis cyklu `while` vypadá následovně.

```
while podmínka:
  příkazy
```

Ve výše uvedeném je `podmínka` logický výraz. Pokud se tento výraz vyhodnotí na hodnotu `True`, dojde k vykonání bloku kódu označeného `příkazy`. Tomuto bloku se říká *tělo cyklu*. Po vykonání těla cyklu je opět vyhodnocen logický výraz `podmínka` a situace se opakuje. V případě, že `podmínka` je vyhodnocena na `False`, příkazy v těle cyklu (blok `příkazy`) se nevykonávají a program pokračuje dalšími příkazy, které jsou uvedeny až za cyklem (za tělem cyklu), běžně říkáme, že cyklus končí. Průběh programu při vykonávání příkazu `while` je ukázán na obrázku 11.
Každé vykonání těla cyklu se označuje jako *iterace*. Ty jsou obvykle číslovány. Běžně tedy hovoříme o první iteraci, druhé iteraci a tak dále. Uveďme si jednoduchý příklad, který vypíše čísla 1–9.

```
1 # vypíše čísla 1-9
2 i = 1
3
4 while i < 10:
5   print(i)
6   i += 1
```

Obrázek 11: Grafické znázornění průběhu programu při vykonávání příkazu `while`.



Výše uvedený kód se vykonává následovně. `i` je inicializováno na hodnotu 1. Podmínka `i < 10` v cyklu `while` je pravdivá a dojde k první iteraci cyklu. Provede se příkaz na řádku 5, který vytiskne hodnotu `i` (hodnotu 1) a následně dochází k *inkrementaci* proměnné `i` (k obsahu proměnné `i` je přičtena jednička). Následně se opět ověří platnost podmínky, vykoná se tělo cyklu a tak dále dokud hodnota `i` je menší než 10. V okamžiku kdy `i` obsahuje hodnotu 10 (konec 9. iterace cyklu), podmínka `i < 10` není splněna a cyklus končí.

Průvodce studiem
Pro přičtení a odečtení jedničky se v programování běžně používá termín *inkrementace* a *dekrementace*.

Jako další si uvedeme příklad výpočtu největšího společného dělitele dvou nenulových kladných čísel (odečítací verze Eukleidova algoritmu pro nalezení největšího společného dělitele).

Průvodce studiem
Proměnné, které se používají pro uložení iterace cyklu se běžně pojmenovávají `i`, `j`, `k`, ...

```
# největší společný dělitel nenulových kladných čísel x a y
x = 12
y = 8

while x != y:
  if x > y:
    x -= y
  else:
    y -= x

print(f"Největší společný dělitel je {x}")
```

Úkol 21
Upravte program počítající největšího společného dělitele dvou kladných nenulových čísel `x` a `y` tak, aby v něm nebyl použit příkaz `if`.

Při psaní cyklu musíme dávat pozor, aby nevznikl *nekonečný cyklus*.[82] Následující příklad ukazuje nekonečný cyklus.

[82] Nekonečné vykonávání těla cyklu. Pokud k tomuto dojde, program tzv. *cyklí* a nikdy neskončí, respektive skončí až jej uživatel explicitně ukončí.

```
while i < 10:
  print(i)
```

Cyklus `for`

Dalším typem cyklu je cyklus `for`.[83] Jeho obecný zápis vypadá následovně.

[83] Ve většině programovacích jazyků existuje cyklus `for`, ten má ale obvykle jinou syntaxi než v jazyce Python. Cyklus `for` v jazyce Python odpovídá spíše jinému typu cyklu, cyklu `foreach`.

```
for proměnná in sekvence:
  příkazy
```

`sekvence` představuje objekt, který obsahuje uspořádanou posloupnost jiných objektů. Takový objekt je v terminologii jazyka Python označován jako *sekvence*.[84] `proměnná` označuje proměnnou, která nabývá v každé iteraci cyklu právě jedné hodnoty ze `sekvence`. V první iteraci cyklu `for` obsahuje `proměnná` první hodnotu v sekvenci, v druhé iteraci druhou hodnotu a tak dále. Průběh programu při vykonávání příkazu `for` je ukázán na obrázku 12.

[84] V jazyce Python je možné v cyklu `for` použít jakýkoliv objekt, který je *iterovatelný* (anglicky iterable).

Než si ukážeme příklad cyklu `for`, ukážeme funkci `range()`, která umožňuje vytvořit číselnou sekvenci.

Funkce `range()`

Funkce `range()` vrací sekvenci celých čísel $c_0, c_1, c_2, \ldots, c_k$. Syntaxe je následující.

```
range(start, stop, krok)
```

Nepovinný argument `start` určuje první číslo sekvence c_0. Pokud není uveden, je nastaven na hodnotu `0`. Argument `stop` je povinný a udává poslední číslo sekvence, které ale již v sekvenci není ($c_k <$ `stop`). Nepovinný argument `krok` ovlivňuje výpočet následujícího čísla v sekvenci c_{i+1}, které je počítáno jako součet předchozího čísla c_i a hodnoty `krok` ($c_{i+1} = c_i +$ `krok`). Ve výchozím nastavení je `krok = 1`. Poslední číslo sekvence je největší číslo c_k pro které platí $c_k = c_{k-1} +$ `krok` $<$ `stop`. Uveďme si několik příkladů.

```
range(10) # sekvence: 0, 1, 2, ..., 9
range(1, 11) # sekvence: 1, 2, 3, ..., 10
range(0, 10, 2) # sekvence: 0, 2, 4, 6, 8
range(0, 5, 10) # sekvence: 0
range(0, 0) # prázdná sekvence
range(0) # prázdná sekvence
```

Argumenty funkce `range()` mohou být i záporné.

```
range(10, 0, -2) # 10, 8, 6, 4, 2
range(-10, 0) # -10, -9, -8, ..., -1
```

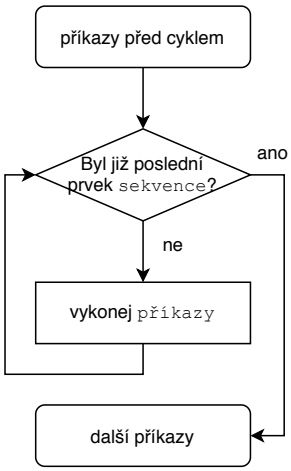
Nyní se vrátíme k cyklu `for`. Následující příklad, využívající funkci `range()`, vypíše čísla 1–9.

```
# vypíše čísla 1-9
for i in range(10):
  print(i)
```

Cyklus `for` nedělá nic jiného, než že postupně prochází jednotlivé položky sekvence `range(10)`.[85] V každé iteraci je proměnné `i` přiřazena hodnota ze sekvence. V našem příkladu v první iteraci proměnná `i` obsahuje hodnotu 1, v druhé iteraci hodnotu 2 a tak dále. Pokud není v cyklu `for` použita sekvence, dojde k chybě.[86]

```
for i in 1:
  print(i) # způsobí chybu: TypeError: 'int' object is not
      iterable
```

Obrázek 12: Grafické znázornění průběhu programu při vykonávání příkazu `for`.



[85] Běžně se používá termín *iteruje*. Můžeme tedy říct, že cyklus `for` iteruje přes sekvenci.

[86] Pojem sekvence významně rozšíříme později. Nyní si vystačíme pouze s celočíselnými sekvencemi vytvořenými pomocí funkce `range()`.

Výběr cyklu

Přirozenou otázkou je, kdy vybrat cyklus `while` a kdy vybrat cyklus `for`. Cyklus `while` je možné převést na cyklus `for` a naopak. Ukázka převodu mezi cykly následuje.

```
# cyklus while
i = start

while i < konec:
  print(i)
  i += krok

# je možné přepsat na cyklus for
for i in range(start, konec, krok):
  print(i)
```

Přestože jsou tyto cykly navzájem zaměnitelné,[87] cyklus `while` je výhodnější použít v případech, kdy v každé iteraci chceme testovat nějakou podmínku. Obvykle se jedná o situace, ve kterých předem nevíme počet iterací cyklu. Cyklus `for` je výhodnější použít v případech, kdy iterujeme přes nějakou sekvenci, tedy víme, kolik bude iterací.

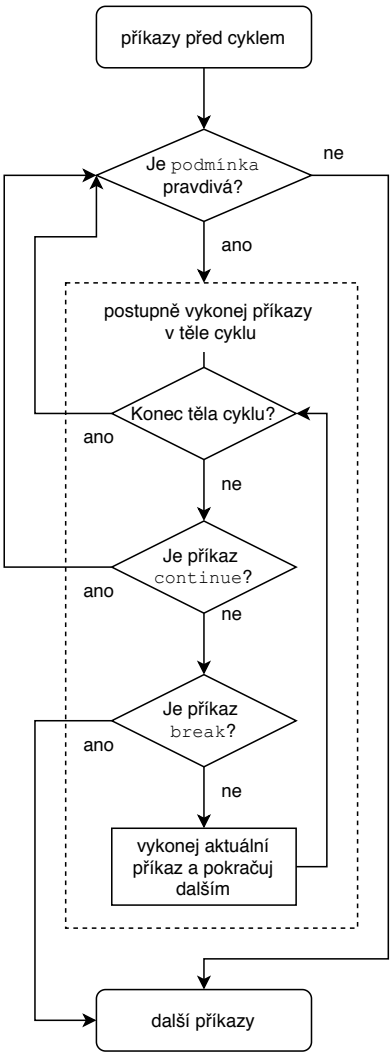
Příkazy `break` a `continue`

V těle cyklu (`while` i `for`) je možné použít dva speciální příkazy `break` a `continue`. Příkaz `break` způsobí okamžité opuštění těla cyklu v místě jeho použití. Kód v těle cyklu za tímto příkazem se již nevykonává. Zároveň dojde k ukončení celého cyklu. Nedojde tedy k vykonání další iterace cyklu. Příkaz `continue` způsobí, stejně jako `break`, okamžité opuštění těla cyklu v místě jeho použití, ale nedojde k ukončení celého cyklu. Cyklus normálně pokračuje. Grafické znázornění průběhu programu při vykonávání cyklu `while` s příkazy `break` a `continue` je znázorněno na obrázku 13. V případě cyklu `for` je chování analogické. Uveďme si několik příkladů.

```
# vypíše čísla 1-5
for i in range(10):
  if i > 5:
    break
  print(i)

# vypíše čísla 1-4, přeskočí 5 a vypíše čísla 6-9
for i in range(10):
  if i == 5:
    continue
```

[87] Dokonce je možné zaměnit nekonečný `while` cyklus za nekonečný `for` cyklus. Tato záměna ale vyžaduje znalosti nad rámec tohoto kurzu.

Obrázek 13: Grafické znázornění průběhu programu při vykonávání cyklu `while` s příkazy `break` a `continue`.



```
  print(i)
```

Dodejme, že příkazy: `break` a `continue` nejsou v mnoha případech vůbec nutné. Výše uvedené je možné zapsat i bez jejich použití. Například.

Průvodce studiem
Příkazy `break` a `continue` by měly být používány obezřetně. Jejich použití, zejména v případě programu s mnoha řádky, zhoršuje čitelnost zdrojového kódu.

```
# vypíše čísla 1-5
for i in range(5):
  print(i)

# vypíše čísla 1-4, přeskočí 5 a vypíše čísla 6-9
for i in range(10):
  if i != 5:
    print(i)
```

Zanoření cyklů

Podobně jako podmínky, lze i cykly do sebe zanořovat.[88] Při zanoření je třeba dodržovat správné odsazení jednotlivých bloků.
Uvažme následující program, který obsahuje dva cykly, jeden zanořený do druhého.[89]

[88] Stejně tak je možné zanořovat cykly do podmínek a naopak.

```
n = 10
range_i = range(0, n)
range_j = range(0, n)

for i in range_i:
  for j in range_j:
    print("*", end="") # end="" zabrání vložení znaku \n

  print() # vloží pouze znak \n
```

[89] **Výstup programu:**

```
**********
**********
**********
**********
**********
**********
**********
**********
**********
**********
```

Shrnutí

V této kapitole jsme představili další příkazy pro řízení toku programu, které umožňují opakované vykonávání bloku kódu v závislosti na aktuálním stavu proměnných. Podmínky a cykly se vyskytují prakticky ve všech komplexnějších programech.

Kontrolní otázky
Odpovězte na následující otázky:

1. Co je to tělo cyklu?
2. Co je to iterace?
3. Co je index?
4. K čemu slouží příkazy `break` a `continue`?

Úkoly

Úkol 22
Napište program, který vypočítá součet a průměr čísel $1, \ldots, n$. Program napište tak, aby fungoval pro libovolné (i záporné)[90] celé číslo `n`.

[90] V případě záporného čísla bude sekvence ve tvaru $-n, \ldots, -1$

Úkol 23
Napište program, který pro zadané `n` vypíše součet řady $1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$. Program napište tak, aby fungoval pro libovolné přirozené číslo a nulu. Pro $n = 0$ je součet řady roven 0.

Úkol 24
Napište program, který vypíše všechna prvočísla menší rovno n. Program napište tak, aby fungoval pro libovolné kladné číslo `n`.

Úkol 25
Napište program, který pro zadanou hodnotu `n` vypíše následující trojúhelník (v ukázce výstup pro hodnotu `n = 10`).

```
*
**
***
****
*****
******
*******
********
*********
**********
```

Program napište tak, aby fungoval pro libovolné celé kladné `n`.

Úkol 26
Napište program, který určí počet číslic zadaného čísla `n`.

Úkol 27
Napište program, který pro zadanou hodnotu `n` vypíše následující tvar (v ukázce výstup pro hodnotu `n = 9`).

```
*.......*
.*.....*.
..*...*..
...*.*...
....*....
...*.*...
..*...*..
.*.....*.
*.......*
```

Program napište tak, aby fungoval pro libovolné celé kladné `n`.

Úkol 28
Napište program, který vypíše všechna n-ciferná čísla. Program navrhněte tak, aby fungoval pro libovolné celočíselné $n > 0$.

Úkol 29
Napište program, který se bude chovat stejně jako program:

```
for i in range(start, konec, krok):
    print(i)
```

Při řešení použijte pouze funkci `range(n)`, kde `n` je vhodně vypočítané celé číslo.[91]

[91] Pro řešení tohoto úkolu lze využít funkci `round(n)`, která vrací zaokrouhlené číslo `n`. Zaokrouhlení je směrem k nejbližšímu celému číslu.

Hledání chyb v programu

„Before software can be reusable, it first has to be usable."

Ralph Johnson

V následující krátké kapitole si představíme základní postup hledání chyb v programu.

Chyby v programu

Doposud jsme se setkávali převážně se *syntaktickými chybami*, tedy chybami, které byly způsobeny nesprávným zápisem programu.[92] Příkladem syntaktické chyby v jazyce Python je třeba opomenutí dvojtečky v zápisu podmínky.

[92] Syntaktické chyby jsou způsobeny porušením syntaxe daného jazyka.

```
# způsobí chybu: SyntaxError: invalid syntax
if 42
  print("42")
```

Pokud se ve zdrojovém kódu nachází syntaktická chyba nedojde ke spuštění programu.[93] Odhalit syntaktickou chybu je velice jednoduché. Chybové hlášení, které syntaktické chyby způsobují, velmi přesně určují místo a důvod chyby. Například pro výše uvedený kód vypadá chybové hlášení následovně (část chybového výpisu není zobrazena).

[93] Chyba je odhalena a ohlášena při generování bajtkódu.

```
File "./chyba.py", line 2
  if 42
       ^
SyntaxError: invalid syntax
```

Vidíme, že k chybě došlo v souboru `chyba.py` na druhém řádku, místo chyby na tomto řádku a informaci o typu chyby.
Oproti tomu *sémantické chyby*, je možné odhalit až za běhu programu. Tyto chyby se označují jako *run-time chyby*.[94] Sémantické chyby je možné rozdělit do dvou kategorií na chyby způsobené nekorektním použitím jazyka. Například práce s nekompatibilními datovými typy[95].

[94] Pojem run-time odkazuje na to, že k chybě došlo až při běhu programu.

[95] U staticky typovaných jazyků je možné tyto chyby odhalit i bez spuštění programu.

Druhou kategorií jsou chyby způsobené špatně napsaným zdrojovým kódem. Například chybou implementace správného postupu nebo chybným postupem samotným (nalezené řešení nefunguje správně pro všechny přípustné hodnoty). Při hledání chyb je klíčové identifikovat, proč k chybě dochází. To lze efektivně provést právě pomocí ladění programu.

Ladění programu

Hledání chyb v programu se označuje jako *ladění*.[96] Nejjednodušším, a mnohdy nejlepším, způsobem ladění je postupné vypisování proměnných programu, například funkcí `print()`.
Alternativou k tomto postupu je *krokování programu*. Při krokování program neběží od začátku do konce tak, jak jsme zvyklí, ale je vykonáván postupně (krokován), po jednotlivých řádcích. Při každém kroku programu je možné sledovat hodnoty jednotlivých proměnných.[97] Dodejme, že krokování programu neslouží pouze pro hledání chyb. Mnohdy je obtížné prostým čtením zdrojového kódu programu pochopit chování programu. Postupné krokování můžete usnadnit pochopení programu.
Další možností je využít IDE a v něm zakomponovaný *debugger*, tedy nástroj, který ladění výrazně usnadňuje. Ve Visual Studio Code se debugger spouští zkratkou `F5`.[98]. Po spuštění ladění program běží dokud nenarazí na *breakpoint* (místo zastavení programu), na kterém se zastaví a zobrazí aktuální stav všech proměnných.[99] Od tohoto okamžiku můžeme program krokovat po jednotlivých řádcích pomocí ovládacích prvků debuggeru, které se automaticky zobrazí v okamžiku, kdy program stojí na nějakém breakpointu.[100]
Debugger je možné ovládat i z kontextového menu, kde jsou navíc dostupné další možnosti. Například skok na označený řádek. Breakpoint je možné umístit po kliknutí na mezeru před požadovaným číslem řádku. Řádky, na kterých je umístěn breakpoint jsou označeny červeným kolečkem. Kliknutím na toto červené kolečko je možné breakpoint odebrat.
Kromě proměnných je možné sledovat i části zdrojového kódu, například celé výrazy či jejich části. K tomuto účelu slouží sekce „Watch“. Požadovanou část kódu do ní musíme nejprve přesunout. Stačí vybrat požadovanou část a skrze kontextové menu (položka „Add to Watch“) tuto část do sekce „Watch“ přidáme. Při krokování programu vidíme aktuální stav všech přidaných částí.

[96] Anglicky debugging. V češtině se často používá česko-anglická zkomolenina „debugování“ (čte se „debagování“).

[97] Tím získáváme úplný přehled o chování programu.

[98] Případně odpovídajícím tlačítkem nebo z menu.

[99] Ve Visual Studio Code je stav všech proměnných (jejich hodnoty) zobrazeny v sekci „Variables“.

[100] Ovládání debuggeru je poměrně intuitivní. „Step over“ provede jeden krok programu (posun na další řádek), „Step into“ pokud je na aktuálním řádku příkaz volání funkce, provede posun na první řádek uvnitř funkce, „Step out“ provede posun na řádek za voláním funkce v níž se aktuálně nacházíme, „Continue“ provede posun na další breakpoint.

Shrnutí

Hledání chyb v programu je běžnou součástí programování. Čím jsou programy složitější, tím je pravděpodobnější, že se při jejich zápisu dopustíme chyby. Zatímco syntaktické chyby je velmi snadné odstranit, nalezení chyb sémantických může stát značné úsilí. Ladění programu pomocí debuggeru hledání chyb značně ulehčuje.

Kontrolní otázky
Odpovězte na následující otázky:

1. Co je to syntaktická chyba?
2. Co je to sémantická chyba?
3. Co je debugger?
4. Co je to breakpoint?

Úkoly

Úkol 30
Odkrokujte programy uvedené v předchozích dvou kapitolách.

Sekvence

„Any fool can write code that a computer can understand. Good programmers write code that humans can understand.“

Martin Fowler

V nadcházející kapitole rozšíříme naše znalosti sekvencí, přestavíme nové typy sekvencí a ukážeme, jak je s nimi možné v jazyce Python pracovat.

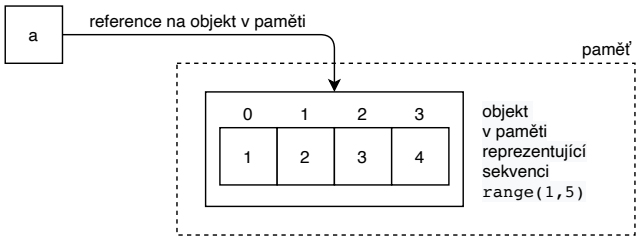
Číselné sekvence

V předchozích kapitolách jsme ukázali, že pro vytvoření číselné sekvence slouží funkce `range()`. Objekt, který reprezentuje sekvenci, obsahuje uspořádané jednotlivé položky dané sekvence. Ty je možné očíslovat, čísluje se od nuly.[101] Tomuto číslování se říká *indexace* a jednotlivým číslům *indexy*. Můžeme tedy říct, že položky sekvence jsou indexovány od nuly. Například

[101] Číslování od nuly je v programování běžné.

```
a = range(1, 5)
```

vytvoří referenci na objekt sekvence (obrázek 14). Ten obsahuje čísla $1, 2, 3, 4$, přičemž číslo 1 se nachází na indexu nula, číslo 2 na indexu jedna a tak dále.



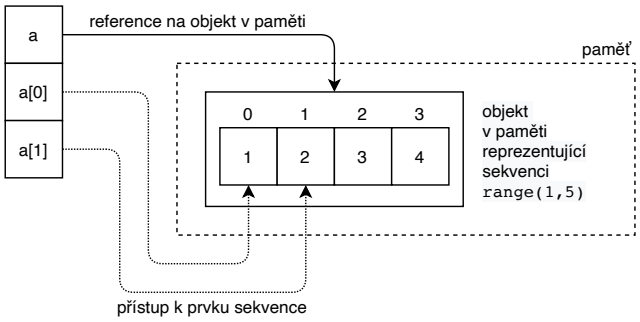
Obrázek 14: Sekvence jsou v paměti reprezentovány jako objekty, které obsahují další objekty. Proměnná `a` obsahuje objekt sekvence vytvořený funkcí `range(1,5)`. Ten obsahuje další objekty reprezentující hodnoty $1, 2, 3, 4$.

K jednotlivým položkám sekvence lze přistupovat pomocí *indexačního operátoru* `[]` a indexu. Například.

```
a = range(1, 5)
```

```
print(a[0]) # vypíše první prvek v a (hodnotu 1)
print(a[1]) # vypíše druhý prvek v a (hodnotu 2)
```

Obrázek 15 ilustruje přístup pomocí indexačního operátoru k jednotlivým položkám sekvence.



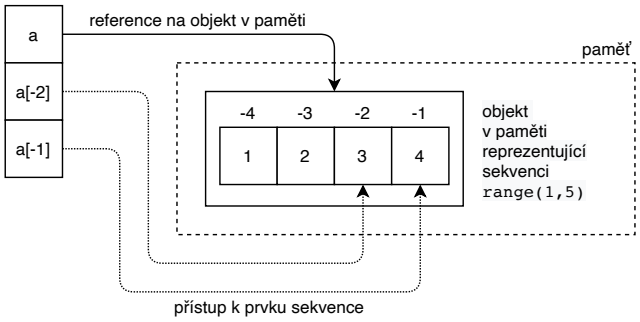
Obrázek 15: Přístup pomocí indexačního operátoru k jednotlivým položkám sekvence.

Kromě indexace od nuly, kterou jsme doposud uvažovali, je v Pythonu možné použít i *zpětnou indexaci*.[102] Ta čísluje jednotlivé položky sekvence od jejího konce. Jednotlivé indexy jsou značeny zápornými čísly (začíná se od -1). Například.

[102] Označovanou také jako reverzní.

```
a = range(1, 5)

print(a[-1]) # vypíše poslední prvek v a (hodnotu 4)
print(a[-2]) # vypíše předposlední prvek v a (hodnotu 3)
```

Obrázek 16 ilustruje přístup pomocí indexačního operátoru a zpětného indexování k jednotlivým položkám sekvence.



Obrázek 16: Přístup pomocí indexačního operátoru a zpětného indexování k jednotlivým položkám sekvence.

Indexační operátor je možné použít nejen pro přístup k jednomu konkrétnímu prvku sekvence, ale i pro výběr několika prvků současně. Tomuto výběru se říká *řez* (anglicky slice). Výsledek řezu je podsekvence, která je tvořena výběrem některých prvků sekvence. Zápis řezu, vybírající podsekvenci ze sekvence `sekvence`, je následující.[103]

Úkol 31
Napište program, který vypíše prvky v sekvenci `range(10)`. Při řešení použijte pouze cyklus `while`.

[103] Syntaxe řezu je podobná syntaxi funkce `range()`.

```
sekvence[index_start:index_stop:krok]
```

Podsekvence je vybírána tak, že jsou vybrány prvky `sekvence` na indexech $i_0, i_1, \ldots, i_k$ určených v `[]`. `index_start`, `index_stop` a `krok` jsou nepovinné. `index_start` určuje index i_0. Ve výchozím nastavení je `index_start = 0`. `index_stop` určuje poslední index v `sekvence`, který ale již v podsekvenci není ($i_k <$ `index_stop`). Ve výchozím nastavení je `index_stop` nastaven na poslední index v `sekvence` zvětšený o jedna ($i_k + 1$). Hodnota `krok` ovlivňuje výpočet následujícího indexu. Ten je počítán jako součet předchozího indexu a `krok`. Ve výchozím nastavení je `krok = 1`. Poslední index i_k je největší index pro který platí $i_k = i_{k-1} +$ `krok` $<$ `index_stop`. Pokud není uveden `krok`, je možné v zápise vynechat poslední : (dvojtečku). Uveďme si několik příkladů.

```
a = range(0, 10)

a[2:8] # vrací range(2, 8)
a[2:] # vrací prvky od 2. indexu do konce, range(2, 10)
a[:2] # vrací prvky od začátku do 2. indexu, range(0, 2)
a[2::4] #range(2, 10, 4)
a[::2] # vrací range(0, 10, 2)
a[::] # vrací range(0, 10)
a[:] # stejné jako a[::]
a[::-1] # vrací range(9, -1, -1)
a[-1:-4:-2] # vrací range(9, 6, -2)
```

Délku sekvence[104] je možné zjistit pomocí funkce `len()` tak, jak je to ukázáno na následujícím příkladu.

[104] Počet položek dané sekvence.

```
a = range(1, 5)

print(len(a)) # vypíše délku sekvence (hodnotu 4)
```

Typickou úlohou je ověření, zda určitý prvek je či není v sekvenci. Například chceme ověřit, zda sekvence obsahuje číslo 42.

```
1  # ověření zda prvek je či není v sekvenci
2  sekvence = range(0, 100, 7)
3  hledany_prvek = 42
4  hledany_prvek_nalezen = False
5
6  for prvek in sekvence:
7    if prvek == hledany_prvek:
8    hledany_prvek_nalezen = True
9
10 if hledany_prvek_nalezen:
```

```
11    print(f"{sekvence} obsahuje {hledany_prvek}.")
12  else:
13    print(f"{sekvence} neobsahuje {hledany_prvek}.")
```

Program pracuje tak, že nastaví proměnnou indikující nalezení hledaného prvku na `False` (řádek 3). Následně je na řádcích 5–6 iterováno přes prvky sekvence. Pokud prvek odpovídá hledanému prvku, je `hledany_prvek_nalezen` nastavena na `True`. Jazyk Python umožňuje program zjednodušit pomocí logického operátoru `in` (případně `not in`) testujícího zda je (či není) hledaný prvek v zadané sekvenci. Zjednodušení je ukázáno v následujícím příkladu.

Úkol 32
Program ověřující zda prvek je či není v sekvenci vždy projde celou sekvenci. Upravte program tak, aby v případě, že je prvek v sekvenci nalezen, již neprocházel zbytek sekvence.

```
1  # ověření zda prvek je či není v sekvenci
2  sekvence = range(0, 100, 7)
3  hledany_prvek = 42
4
5  if hledany_prvek in sekvence:
6    print(f"{sekvence} obsahuje {hledany_prvek}.")
7  else:
8    print(f"{sekvence} neobsahuje {hledany_prvek}.")
```

Na místo cyklu `for` je použit operátor `in` (řádek 5).
Pro úplnost dodejme, že indexační operátor, řez, funkce `len()` a logický operátor `in` souvisejí se sekvencemi a to nejen těmi číselnými. Později ukážeme, že v jazyce Python existuje několik typů sekvencí a výše zmíněné je možné aplikovat na libovolnou z nich.
Číselné sekvence vytvořené pomocí funkce `range()` nelze měnit. Jednotlivé položky je tedy možné číst pomocí indexačního operátoru, ale nelze jim přiřadit hodnotu.

Průvodce studiem
Indexační operátor, řez, funkci `len()` a logický operátor `in` je možné aplikovat na libovolné sekvence.

```
a = range(1, 5)

a[0] = 0 # způsobí chybu: TypeError: 'range' object does not
     support item assignment
```

Průvodce studiem
Pojmy mutabilita a imutabilita jsou v jazyce Python klíčové.

To je důsledkem toho, že jednotlivé položky číselné sekvence jsou čísla, která jsou v jazyce Python *imutabilní* (neměnná). Není tedy možné změnit obsah objektu reprezentující číselnou hodnotu.[105] Uvažujme následující kód.

[105] Tuto skutečnost jsme pro jednoduchost v úvodních kapitolách zamlčeli.

```
a = 42
a = 0
```

Příkaz na druhém řádku nezmění obsah objektu reprezentující číslo 42, ale vytvoří objekt nový a změní původní referenci na referenci na nový objekt (obrázek 3).

Řetězce

Řetězce jsou dalším příkladem sekvencí.[106] Stejně jako číselné sekvence je možné řetězce používat v cyklu `for`, pro přístup k prvkům indexační operátor, vybírat podřetězec pomocí řezu, pro zjištění délky řetězce použít funkci `len()`. Například

[106] V předchozích kapitolách jsme uvedli, že řetězce jsou tvořeny sekvencí znaků. To, že jsou řetězce sekvence, tedy není moc překvapivé.

```
retezec = "spam"
print(len(retezec)) # vypíše délku řetězce (4)

print(retezec[0]) # vypíše první prvek v řetězci (znak s)
print(retezec[1]) # vypíše druhý prvek v řetězci (znak p)
print(retezec[1:3]) # vypíše pa
print(retezec[1:]) # vypíše pam

# vypíše jednotlivé znaky řetězce
for c in retezec:
  print(c)
```

Dále, stejně jak u číselných sekvencí, je možné použít logický operátor `in`. Ten v případě řetězců funguje nejen pro jednotlivé prvky (znaky) sekvence (řetězce), ale i pro podsekvence. Například.

```
retezec = "spam"

"m" in retezec # vrací True
"am" in retezec # vrací True
"pam" in retezec # vrací True
"spam" in retezec # vrací True
"x" in retezec # vrací False
```

Řetězce jsou imutabilní sekvence. Nelze je tedy měnit.[107]

[107] Nepříjemným důsledkem je, že pokud chceme v jazyce Python změnit řetězec, například změnit první písmeno na velké, musíme vytvořit nový objekt.

```
retezec[0] = "S" # způsobí chybu: TypeError: 'str' object
    does not support item assignment
```

Řetězce lze spojovat pomocí operátoru spojení + a opakovat pomocí operátoru *. Například

```
"spam" + "ham" # vrací řetězec "spamham"
"spam" * 4 # vrací řetězec "spamspamspamspam"
```

S řetězci je spojeno několik specifických operací, které je možné provádět vždy nad konkrétním řetězcem (objektem jenž jej reprezentuje). Formálně se nejedná o operace, ale o tzv. *metody objektu*.[108] Některé z těchto metod jsou shrnuty v tabulce 6. Příklad použití následuje.

[108] Pro naše účely vystačíme s jejich intuitivním chápáním.

```
"spam".upper() # vrací "SPAM"
"Spam".lower() # vrací "spam"
```

Metody	Význam
`.upper()`	převede všechna písmena v řetězci na velká
`.lower()`	převede všechna písmena v řetězci na malá
`.isalpha()`	vrací `True`, pokud řetězec obsahuje pouze písmena, jinak `False`
`.isdigit()`	vrací `True`, pokud řetězec obsahuje pouze číslice, jinak `False`
`.replace(co, čím)`	v řetězci nahradí všechny výskyty řetězce `co` řetězcem `čím`

Tabulka 6: Vybrané metody nad řetězci.

Pokud bychom chtěli napsat program, který vytiskne zadaný řetězec, kde je první písmeno převedeno na velké, mohli bychom to provést například následovně.

```
retezec = "spam"
novy_retezec = ""
i = 0

# iterujeme přes řetězec
for c in retezec:
  # všechny znaky kromě prvního překopírujeme
  # z původního řetězce do nového řetězce
  if i > 0:
    novy_retezec += c

  # první písmeno převedeme na velké písmeno
  else:
    novy_retezec += c.upper()
  i += 1

# vytiskneme nový řetězec
print(novy_retezec)
```

Případně je možné využít řez.

```
retezec = "spam"
novy_retezec = retezec[0].upper() + retezec[1:]

# vytiskneme nový řetězec
print(novy_retezec)
```

Seznamy

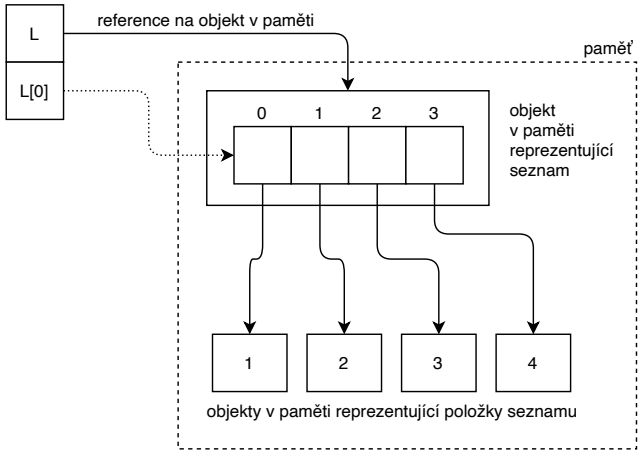
Poslední typ sekvence, který si představíme, je seznam (anglicky list).[109] Obecný zápis seznamu je následující.

```
[první_položka_seznamu, druhá_položka_seznamu, ..., poslední
    _položka_seznamu]
```

Na rozdíl od číselných sekvencí a řetězců nejsou jednotlivé položky seznamu omezeny na předem určený typ.[110] Položky seznamu jsou reference na (libovolné) objekty, které reprezentují tyto položky. Můžeme například vytvořit seznam obsahující čísla, znaky, řetězce nebo objekty různých typů, tak, jak je to ukázáno v následujícím příkladu.

```
L = [1, 2, 3, 4]
L = ["s", "p", "a", "m"]
L = ["spam", "ham"]
L = ["spam", 2, 3, 4]
```

Indexační operátor automaticky následuje referenci na daný objekt. Reprezentace seznamu `L = [1, 2, 3, 4]` v paměti počítače a použití indexačního operátoru je znázorněno na obrázku 17.



S jednotlivými položkami seznamu se pracuje jako by to byly přímo konkrétní objekty. Například.

```
L = [1, 2, 3, 4]
print(L[1]**L[2]) # vypíše: 8
```

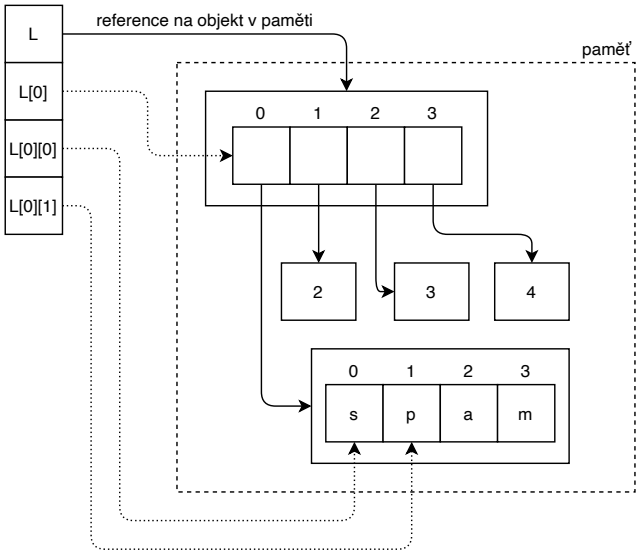
V případě, že je položka seznamu sekvence, je možné přistupovat k položkám této sekvence stejně jako jsme to dělali doposud. Například `L = ["spam", 2, 3, 4]`. První položka seznamu je řetězec,

[109] Je tedy jasné, že seznamy, stejně jako číselné sekvence a řetězce, lze použít v cyklu `for`, pro přístup k prvkům seznamu indexační operátor, vybírat podseznam pomocí řezu, ověřit zda je prvek v sekvenci pomocí operátoru `in` a pro zjištění délky seznamu použít funkci `len()`.

[110] Položky číselné sekvence jsou pouze čísla, položky řetězce jsou pouze znaky.

Obrázek 17: Přístup pomocí indexačního operátoru k jednotlivým položkám seznamu. `L[0]` obsahuje referenci na objekt reprezentující hodnotu 1, která je automaticky následována. `L[0]` tedy obsahuje hodnotu 1.

který je přístupný pomocí `L[0]`. K jednotlivým položkám tohoto řetězce můžeme přistupovat opět pomocí indexačního operátoru, tedy `L[0][0]` je první položka tohoto řetězce (znak `"s"`). Použití indexačního operátoru je znázorněno na obrázku 18.



Obrázek 18: Přístup pomocí indexačního operátoru k jednotlivým položkám sekvence. První položka seznamu, řetězec `spam`, je přístupná pomocí `L[0]`. Jednotlivé položky tohoto řetězce jsou přístupné opět pomocí indexačního operátoru. Například znak `"s"` je přístupný pomocí `L[0][0]`.

Jednotlivé položky seznamu mohou být i další seznamy.

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

Průvodce studiem
Jelikož jsou jednotlivé položky seznamu pouze reference na objekty, které je reprezentují, je validní, aby položka seznamu `L`, byla opět seznam `L`. Tomuto se budeme věnovat až v následujícím semestru.

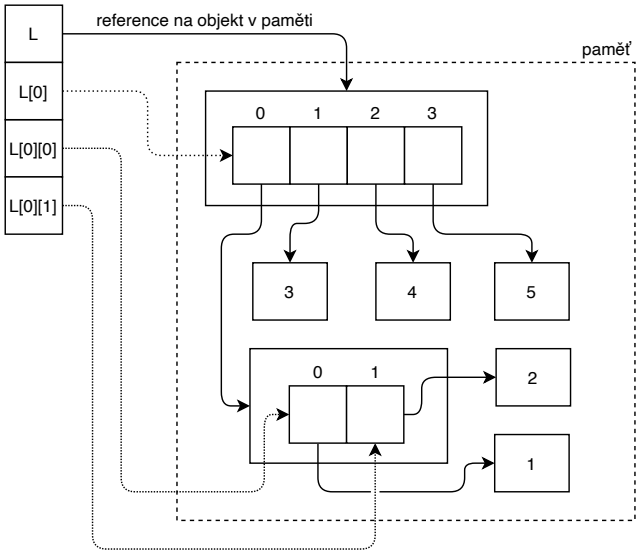
Jelikož je seznam sekvence, je použití indexačního operátoru v takovémto případě stejné, jako v předchozím příkladu s řetězcem (obrázek 19).
Seznamy jsou na rozdíl od číselných sekvencí a řetězců *mutabilní*. Jejich jednotlivé položky lze měnit. Například

```
L = [1, 2, 3, 4, 5]
L[0] = 42
print(L) # vypíše: [42, 2, 3, 4, 5]
```

Mutabilita přináší drobnou komplikaci. Ke změně objektu může dojít skrze různé reference. Například

```
L = [1, 2, 3, 4, 5]
M = L # M nyní obsahuje referenci na L
M[0] = 42 # změní referenci v L
print(L) # vypíše: [42, 2, 3, 4, 5]
```

V následujícím kódu výše uvedené nenastává.



Obrázek 19: Přístup pomocí indexačního operátoru k jednotlivým položkám sekvence.

```
L = [1, 2, 3]
M = [4, 5]
N = [L, M]
M = 42
print(N) # vypíše: [[1, 2, 3], [4, 5]]
```

Pokud vykonáme příkaz `M = 42` dojde ke změně reference proměnné `M`, seznam `N` to neovlivní, jelikož ten stále obsahuje referenci na seznam `[4, 5]`.[111]

Za pomocí řezu je možné nejen vytvářet podseznamy, ale také měnit jejich položky. Například

[111] Připomeňme, že příkaz přiřazení vždy mění referenci, nikoliv objekt samotný.

```
L = [1, 2, 3, 4, 5]
L[2:4] = [41, 42, 43]
print(L) # vypíše: [1, 2, 41, 42, 43, 5]

L = [1, 2, 3, 4, 5]
L[2:4] = [41]
print(L) # vypíše:[ 1, 2, 41, 5]

L = [1, 2, 3, 4, 5]
L[2:4] = 41 # zpsůobí chybu: TypeError: can only assign an
    iterable
```

Při zápisu seznamu jsou jednotlivé položky seznamu nepovinné. Je tedy možné vytvořit i *prázdný seznam*,[112] který nemá žádné položky.

[112] Délka, vrácená funkcí `len()`, prázdného seznamu je rovna nule.

```
L = [] # prázný seznam
```

Metody	Význam
`.append(objekt)`	přidá `objekt` na konec seznamu
`.extend(sekvence)`	přidá všechny položky `sekvence` na konec seznamu
`.remove(hodnota)`	odstraní ze seznamu všechny objekty s hodnotou `hodnota`
`.pop()`	odstraní poslední prvek seznamu
`.pop(index)`	odstraní prvek seznamu na daném indexu
`.clear()`	odstraní všechny položky seznamu
`.copy()`	vrací kopii seznamu (nový objekt)
`.reverse()`	vrací seznam s prvky v obráceném pořadí

Tabulka 7: Vybrané metody nad seznamy.

Při vyhodnocování logických výrazů je prázdný seznam přetypován na hodnotu `False`.
Seznamy lze spojovat pomocí operátoru spojení + a opakovat pomocí operátoru *. Například.

```
[1, 2, 3] + [4, 5]  # vrací seznam [1, 2, 3, 4, 5]
[1, 2, 3] * 3 # vrací seznam [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

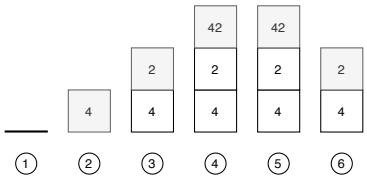
Stejně jako řetězce i seznamy disponují řadou speciálních metod. Vybrané z nich jsou shrnuty v tabulce 7.

Základní datové struktury

Pomocí metod uvedených v tabulce 7, lze v jazyce Python snadno vytvořit datovou strukturu zásobník.[113] Ukázka následuje.

```
# implementace zásobníku
zasobnik = []

# operace is_empty (ověření prázdnosti zásobníku)
if not zasobnik:
  print("Zásobník je prázdný.")

# operace push (přidání na zásobník)
zasobnik.append(4)
zasobnik.append(2)
zasobnik.append(42)

# operace top (objekt na vrcholu zásobníku)
print(zasobnik[-1]) # vypíše: 42

# operace pop (odebrání objektu z vrcholu zásobníku)
a = zasobnik.pop() # a = 42
```

[113] Zásobník je abstraktní datová struktura typu LIFO (poslední dovnitř první ven) s metodou `push(hodnota)` umožňující uložení hodnoty `hodnota` na zásobník, metodou `top()` umožňující vrácení hodnoty z vrcholu zásobníku, metodou `pop()` odstraňující hodnotu z vrcholu zásobníku a metodou is_empty() vracející `True` pokud je zásobník prázdný, `False` jinak. Příklad následuje..



① Prázdný zásobník `Z`, `Z.is_empty()` vrací `True`.
② `Z.push(4)`, uloží hodnotu 4 na vrchol zásobníku (znázorněn šedou barvou).
③ `Z.push(2)`, uloží hodnotu 2 na vrchol zásobníku.
④ `Z.push(42)`, uloží hodnotu 42 na vrchol zásobníku.
⑤ `Z.top()` vrací hodnotu 42 z vrcholu zásobníku. Zásobník se nemění.
⑥ `Z.pop()` odebere hodnotu (na vrcholu) ze zásobníku a vrátí ji. Na vrcholu zásobníku je nyní 2.

```
print(zasobnik) # vypíše: [4, 2]
```

Dodejme, že uvedeném zdrojovém kódu nijak nekontrolujeme, zda je možné operaci provést. Například operace `pop()` nad prázdným zásobníkem způsobí chybu.
Analogicky je možné vytvořit datovou strukturu fronta.[114]

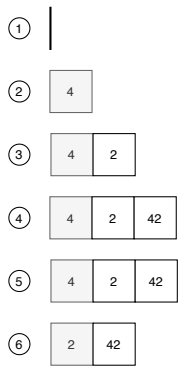
```
# implementace fronty
fronta = []

# operace is_empty (ověření prázdnosti fronty)
if not fronta:
  print("Fronta je prázdná.")

# přidání na konec fronty
fronta.append(4)
fronta.append(2)
fronta.append(42)

# prvek na začátku fronty
print(fronta[0]) # vypíše: 4

# odebrání prvku ze začátku fronty
a = fronta.pop(0) # a = 4
print(fronta) # vypíše: [2, 42]
```

Ani v tomto případě nijak nekontrolujeme, zda je možné operaci provést. Pomocí seznamu je v jazyce Python možné provést také implementaci pole, což je přímočaré, jelikož seznam můžeme chápat jako pole. Mnohem zajímavější je implementace dvojrozměrného pole, které lze implementovat například pomocí seznamu seznamů tak, jak je ukázáno níže.[115]

```
# dvojrozměrné pole m x n
m = 10
n = 10

L = [] # dvojrozměrné pole
for i in range(m):
  L.append([0] * n) # inicializace pole všechny prvky rovny
      0

L[1][1] = 1 # zápis prvku do pole

print(L)
```

Případně lze dvojrozměrné pole reprezentovat jako pole jednorozměrné (jako seznam).

[114] Fronta je abstraktní datová struktura typu FIFO (první dovnitř první ven) s metodou `push(hodnota)` umožňující uložení hodnoty `hodnota` na konec fronty, metodou `top()` umožňující vrácení hodnoty ze začátku fronty, metodou `pop()` odstraňující hodnotu ze začátku fronty a metodou is_empty() vracející `True` pokud je fronta prázdná, `False` jinak. Příklad následuje.



① Prázdná fronta `F`, `F.is_empty()` vrací `True`.
② `F.push(4)`, uloží hodnotu 4 na konec fronty, který je nyní zároveň začátkem (znázorněn šedou barvou).
③ `F.push(2)`, uloží hodnotu 2 na konec fronty.
④ `F.push(42)`, uloží hodnotu 42 na konec fronty.
⑤ `F.top()` vrací hodnotu 42 ze začátku fronty. Fronta se nemění.
⑥ `F.pop()` odebere hodnotu ze začátku fronty a vrátí ji. Na začátku fronty je nyní 2.

[115] **Výstup programu:**

```
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

```
# reprezentace pole m x n seznamem s m * n prvky
m = 10
n = 10

L = [] # seznam uchovávající pole
for i in range(m * n):
  L.append([0]) # inicializace pole všechny prvky rovny 0

# přístup k L[i][j] je ekvivalentní L[i * n + j]
```

Na závěr dodejme, že seznamy je možné vytvářet pomocí funkce `list()`. Ta akceptuje jako argument objekt ze kterého lze vytvořit seznam. Několik příkladů následuje.

```
list() # vrací prázdný seznam
list("spam") # vrací ['s', 'p', 'a', 'm']
list(range(10)) # vrací [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
list([1, 2, 3]) # vrací kopii seznamu [1, 2, 3]
```

Shrnutí

Sekvence v jazyce Python, zejména pak řetězce a seznamy, kterým jsme se věnovali v této kapitole, jsou užitečnou datovou reprezentací, jež je často využívána. Většina programů pracuje právě s daty v podobě sekvence prvků, byť mohou být v různých programovacích jazycích nazývány jinak. Zejména pochopení seznamu a práce se seznamem je klíčové pro další studium programování.

Kontrolní otázky
Odpovězte na následující otázky:

1. Co je to sekvence?
2. Co je to indexační operátor?
3. Jakým způsobem se v jazyce Python indexují prvky sekvence?
4. Co je to řez sekvence?
5. Jak lze v jazyce Python ověřit, zda se prvek nachází v sekvenci?
6. K čemu slouží funkce `len()`?
7. Co znamená mutabilita a imutabilita datového typu?

Úkoly

Úkol 33
Napište program, který vymaže z řetězce všechny výskyty zadaného podřetězce. Například pro podřetězec `"spam"` a řetězec `"Dnes mi přišel superspam schospamvaný spamemail."` program vypíše řetězec `"Dnes mi přišel super schovaný email."`. Program napište (a) bez použití metody `.replace()` a (b) s metodou `.replace()`.

Úkol 34
Napište program, který ověří, zda je zadaný řetězec palindromem (čte se stejně od konce i od začátku).

Úkol 35
Napište program, který pro zadaný seznam čísel vypočítá jejich průměr.

Úkol 36
Napište program, který otočí pořadí prvků v seznamu. Například pro seznam `[1, 2, 3, 4, 5, 6]` program vrací seznam `[6, 5, 4, 3, 2, 1]`, pro seznam `[1, 2, 3, 4, 5]` program vrací seznam `[5, 4, 3, 2, 1]`. Program napište tak, aby nevytvářel nový seznam, ale pouze měnil zadaný seznam. Při řešení nepoužívejte řez ani metodu `.reverse()`.

Úkol 37
Napište program, který spojí dva zadané seznamy. Stejné hodnoty se neopakují. Například pro `L = [1, 2, 3, 4]` a `M = [3, 4, 5]` bude výsledný seznam `[1, 2, 3, 4, 5]`.

Úkol 38
Napište program, který pro zadaný seznam číselných seznamů vypočítá průnik těchto seznamů. Například pro seznam `[[3, 4, 5], [1, 2, 3, 4, 5], [1, 4, 5]]` program vypíše seznam `[4, 5]`.

Úkol 39
Napište program, který ověří, zda jsou čísla v seznamu uspořádány vzestupně, sestupně nebo vůbec.

Úkol 40
Napište program, který vrátí seznam všech prvočísel menších nebo rovno zadané hodnotě `n`.

Úkol 41
Napište program, který zašifruje textový řetězec pomocí Caesarovy šifry.[116] Pro jednoduchost uvažujte pouze malá písmena `a-z`.

[116] Každé písmeno v šifrovaném řetězci je posunuto o k pozic v abecedě. Například slovo „python“ je pro $k = 2$ (posun o 2 písmena) zašifrováno do podoby „ravjqp“. Písmeno „z“ je pro $k = 2$ zašifrováno do „b“

Úkol 42
Napište program,[117] který dešifruje řetězec `"bdasdmyahmzu h bkftazg vq lmnmhm"` šifrovaný pomocí Caesarovy šifry. Mezery odpovídají mezerám v původním řetězci.

[117] Pro řešení tohoto úkolu je třeba využít hrubou sílu a vypočítat všechny možné posuny.

Úkol 43
Napište program, který pro zadanou hodnotu `n` vytvoří matici obsahující převrácenou diagonálu. (v ukázce výstupu pro hodnotu `n = 8`).

```
[[0, 0, 0, 0, 0, 0, 0, 1],
 [0, 0, 0, 0, 0, 0, 1, 0],
 [0, 0, 0, 0, 0, 1, 0, 0],
 [0, 0, 0, 0, 1, 0, 0, 0],
 [0, 0, 0, 1, 0, 0, 0, 0],
 [0, 0, 1, 0, 0, 0, 0, 0],
 [0, 1, 0, 0, 0, 0, 0, 0],
 [1, 0, 0, 0, 0, 0, 0, 0]]
```

Úkol 44
Napište program, který vypíše číslo zadané v desítkové soustavě jako binární číslo.

Úkol 45
Napište program, který ověří, zda jsou dva zadané seznamy stejné.[118]

[118] Obsahují stejné hodnoty na stejných indexech

Úkol 46
Napište program, který ověří zda dva zadané seznamy obsahují stejné hodnoty. Jednotlivé prvky mohou být v jiném pořadí.[119]

[119] Pozor na případy, kdy se hodnoty opakují.

Funkce

„So much complexity in software comes from trying to make one thing do two things.“

Ryan Singer

V předchozích kapitolách jsme mnohokrát využívali předem definované funkce. V této kapitole se budeme věnovat vytváření vlastních funkcí. Navíc osvětlíme pojmy rozsah proměnné a rekurze.

Funkci lze chápat jako pojmenovanou část kódu, které předáme vstupní data, s nimiž funkce pracuje.[120] Funkce je vykonána při jejím *zavolání*. Například `print("Ahoj světe!")` není nic jiného než volání funkce, které předáme textový řetězec. Kromě používání již existujících funkcí může programátor vytvářet funkce vlastní. Funkce umožňují lépe strukturovat program,[121] vytvářet znovupoužitelný neredundantní kód a především poskytují základní nástroj pro *programátorskou abstrakci*.[122]
Pro zápis funkcí v jazyce Python se používá klíčové slovo `def`. Základní zápis (deklarace a definice funkce) je následující.

```
def jméno_funkce(parametr_1, parametr_2, ..., parametr_n):
  příkazy
```

`jméno_funkce` představuje identifikátor funkce.[123] `parametr_1, ..., parametr_n` jsou *parametry funkce*. Parametry funkce jsou proměnné, jež jsou přístupné v *těle funkce*. Tělo funkce, tvořené `příkazy`, je blokem kódu a musí být korektně odsazeno. Například funkce vypisující druhou mocninu čísla `x`.

```
def na_druhou(x):
  print(x**2)
```

`na_druhou` je identifikátorem funkce. `x` je parametrem této funkce. Tělo funkce je tvořeno příkazem `print(x**2)`. Volání funkce se provádí pomocí identifikátoru funkce, přičemž v kulatých závorkách jsou uvedeny jednotlivé *argumenty* funkce.

[120] Funkce si lze představit jako podprogramy v našem programu.

[121] Program rozdělit na podprogramy.

[122] O funkci víme co dělá, ale již nás nemusí zajímat jakým způsobem to dělá.

Průvodce studiem
Programátorská abstrakce umožňuje lépe se soustředit na řešení daného problému (řešíme méně problémů), přináší větší srozumitelnost, menší redundaci a tím i chybovost.

[123] Formálně se jedná o proměnnou, která obsahuje referenci na objekt reprezentující danou funkci.

Průvodce studiem
Jaký je rozdíl mezi parametrem a argumentem? Argument je hodnota předaná do funkce, například číslo 10. Parametr je proměnná, například `x`, skrze kterou jsou přístupné předané argumenty. Parametry tedy obsahují argumenty.

```
na_druhou(42)
```

Při volání funkce se argumenty předané do funkce naváží na odpovídající parametry.[124] V předchozím případě je argumentem funkce hodnota 42. Ta je při volání navázána na parametr x. Jakmile funkce skončí, program pokračuje za místem volání funkce. Například.

[124] Přesný mechanismus předávání hodnot do funkce popíšeme později.

```
1 def na_druhou(x):
2   print(x**2, end=" ")
3
4 n = 10
5 na_druhou(n)
6 print(f"je druhá mocnina čísla {n}")
```

Příkazy jsou postupně vykonávány. Nejprve je definována a deklarována funkce (řádky 1–2). Řádek 5 obsahuje volání funkce, které je jako *argument* předána hodnota 10. Následně je vykonáno tělo funkce (řádek 2). Program pokračuje na řádku 6.
V jazyce Python je možné volat pouze definované funkce. Analogicky jako v případě proměnných, jazyk Python neumožňuje provést deklaraci funkce bez její definice.
Tělo funkce je možné opustit předčasně[125] pomocí příkazu `return`. Příkaz `return` způsobí okamžité ukončení funkce. Kromě toho umožňuje příkaz `return` *vrátit* hodnotu z funkce.[126] Návratová hodnota se zapisuje přímo za příkaz `return`.

[125] Dříve než je vykonán celý blok, který tvoří tělo funkce.

[126] Ta bývá označována jako *návratová hodnota*.

```
return hodnota
```

Tuto hodnotu je možné dále použít. Například.

```
1 def na_druhou(x):
2   return x**2
3
4 n = 10
5 n_na_druhou = na_druhou(n)
6 print(f"{n_na_druhou} je druhá mocnina čísla {n}")
```

Při vykonání příkazu na řádku 5, je nejprve zavolána funkce `na_druhou`. Ta vypočítá druhou mocninu předaného argumentu a tuto hodnotu vrátí. Výsledná hodnota je uložena do proměnné a následně použita na řádku 6.
Funkce, která ve svém těle neobsahuje příkaz `return`, případně obsahuje příkaz `return` bez uvedené návratové hodnoty, vrací speciální datový typ `None` (žádný).[127]

[127] Funkce, které nevrací žádnou hodnotu, se někdy označují jako *procedury*.

```
def moje_funkce():
  n = 42

print(moje_funkce()) # zobrazí: None
```

Rozsah platnosti proměnných

Každá proměnná má svůj *rozsah platnosti*. Ten určuje část zdrojového kódu, ve které proměnná existuje.[128] Proměnné, které jsou definovány v těle funkce se označují jako *lokální* a existují pouze lokálně v těle funkce. Například.

[128] Je definována (deklarována).

```
def moje_funkce(a):
  b = 42 # b je lokální proměnná
  print(a)
  print(b)

moje_funkce(7)
print(b) # způsobí chybu: NameError: name 'b' is not defined
```

Důsledkem toho se mohou lokální proměnné v těle funkce jmenovat stejně jako proměnné v jiné části programu.

```
a = 42
def moje_funkce(a):
  a = 7  # a je lokální proměnná existující v těle funkce
  print(a)

moje_funkce(a)  # vypíše: 7
print(a) # vypíše: 42
```

Pokud je v těle funkce přistupováno k proměnné, která není definována, je tato proměnná hledána v nadřazeném bloku kódu.[129]

[129] Pokud v nadřazeném bloku neexistuje, hledá se v dalších nadřazených blocích (pokud existují). Když proměnná není nalezena program skončí chybou.

```
b = 42

def moje_funkce(a):
  print(b) # proměnná b není definována v těle funkce

moje_funkce(7) # vypíše: 42
```

Proměnné, které existují v celém programu se označují jako *globální*. V předchozím příkladu byla proměnná `b` globální. Globální proměnné je třeba používat velice obezřetně. Zatímco čtení globálních proměnných nepůsobí žádné problémy, jejich změna v těle funkce

Průvodce studiem
Funkce by měli být co nejvíce nezávislé na ostatních částech programu.

problémy přináší. Neuvážené používání globálních proměnných vede k *špagety kódu*.[130] Způsob, jak měnit globální proměnné v těle funkce, záměrně zamlčíme.
Globální proměnné jsou udržovány v paměti, která se označuje jako *halda*.[131] Lokální proměnné jsou udržovány v části paměti, která se označuje jako *systémový zásobník*.[132] V tomto zásobníku jsou udržovány všechny lokální proměnné používané uvnitř funkce. Pokud v těle nějaké funkce dojde k volání funkce, je na systémovém zásobníku vytvořen prostor, do kterého jsou uloženy všechny lokální proměnné. V zásobníku volání jsou tedy udržovány lokální proměnné všech neukončených funkcí. Jakmile je funkce opuštěna (ukončena), dojde k smazání prostoru vymezeného pro lokální proměnné dané funkce.[133].
Aktuální stav všech lokálních a globálních proměnných je možné v IDE sledovat při ladění v sekci „Variables“. Stejně tak je možné sledovat systémový zásobník (proměnné v něm uložené) v sekci „Call stack“.

[130] Špagety kód je označení pro složitě strukturovaný a obtížně srozumitelný zdrojový kód.

[131] Anglicky heap. V jazyce Python se o správu této paměti stará garbage collector.

[132] Případně *zásobník volání* (call stack).

[133] Lokální proměnné tak nevratně zanikají.

Předávání hodnot

V jazyce Python se hodnoty do funkcí předávají vždy pomocí reference.[134] Do funkce není předáván objekt samotný, ale vždy reference na tento objekt. Například.

[134] Tento způsob se běžně označuje jaké *předání hodnoty odkazem*.

```
1 def moje_funkce(L):
2   L[0] = 42
3
4 L = [1, 2, 3, 4, 5]
5 print(L) # vypíše: [1, 2, 3, 4, 5]
6 moje_funkce(L)
7 print (L) # vypíše: [42, 2, 3, 4, 5]
```

Druhým způsobem předání proměnných je *předávání hodnotou*. Do funkce není předána reference, ale kopie objektu.[135] Při tomto způsobu nedojde ke změně původního objektu.[136] Například, ve výše uvedeném příkladu, stačí řádek 6 nahradit tímto řádkem

```
moje_funkce(L.copy())
```

Metoda `.copy()` vytvoří kopii seznamu `L`. V těle funkce dojde ke změně této kopie a původní seznam zůstane nezměněn.[137] Dodejme, že analogicky je možné použít funkci `list()`, případně řez, které rovněž vytváří kopii objektu.

[135] Jazyk Python tento způsob předání proměnných neumožňuje. Programátor musí ručně vytvořit kopii objektu a tu předat.

[136] Připomeňme, že měnit lze v jazyce Python pouze mutabilní objekty.

[137] Metoda `.copy()` vytváří tzv. *mělkou kopii* (shallow copy). Při vytváření mělké kopie je vytvořen nový seznam a do jednotlivých prvků seznamu jsou zkopírovány reference z položek původního seznamu. V důsledku tohoto, vnořené seznamy nejsou skutečně kopírovány. Této problematice se budeme více věnovat v dalším semestru.

Rekurze

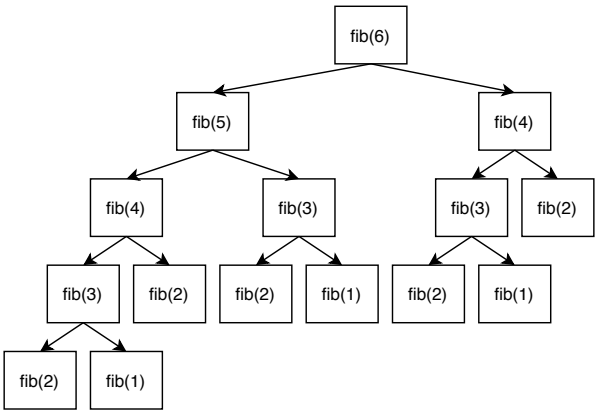
Průvodce studiem
Při rekurzi je řešení problému závislé na řešení menší instance (menšího případu) stejného problému. V informatice má rekurze zcele zásadní význam.

Pojem rekurze, v programování, označuje situaci, kdy funkce volá ve svém těle sebe sama. Rekurze obvykle umožňuje jednoduché řešení problémů, které je možné rozložit na menší podproblémy. Klasickým příkladem používající rekurzi je výpočet *n*-tého členu Fibonacciho posloupnosti.[138]

[138] Fibonacciho posloupnost je nekonečná posloupnost přirozených čísel, začínajících hodnotami 0 a 1, přičemž každý další člen této posloupnosti je součet předchozích dvou členů. Začátek Fibonacciho posloupnosti je následující: $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$

```
# výpočet n-tého členu Fibonacciho posloupnosti
# rekurzivní výpočet
def fib(n):
  if n <= 1: # limitní podmínka
    return n
  return fib(n - 1) + fib(n - 2) # rekurzivní volání
```

Pokud funkce obsahující rekurzi, jsou označovány jako rekurzivní funkce. Každá rekurzivní funkce obsahuje *limitní podmínku*, která určuje zda má dojít k opětovnému (rekurzivnímu) volání či nikoliv. Obrázek 20 zachycuje průběh rekurzivního volání pro `fib(6)`. Tento průběh se také označuje jako *strom rekurzivního volání*.



Obrázek 20: Průběh rekurzivního volání pro `fib(6)`.

Rekurzivní funkce jsou obvykle jednodušší a kratší než jejich iterativní protějšky.[139] Následující zdrojový kód zachycuje iterativní verzi funkce `fib`.

[139] Iterativní funkce jsou funkce bez rekurze. Každou rekurzivní funkci je možné přepsat na iterativní.

```
# výpočet n-tého členu Fibonacciho posloupnosti
# iterativní verze
def fib(n):
  prvni_clen = 0
  druhy_clen = 1
  for i in range(0, n):
    pomocna = prvni_clen
    prvni_clen = druhy_clen
    druhy_clen = pomocna + druhy_clen
  return prvni_clen
```

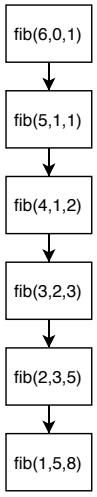
Rekurzivní varianta je zjevně přímočařejší. Nevýhodou rekurzivního řešení je právě samotná podstata rekurze a sice opakované volání. To je prováděno vždy před dokončením dané funkce, přičemž funkce s výsledkem rekurzivního volání dále pracuje. Při výpočtu je třeba neustále udržovat v zásobníku volání všechna nedokončená rekurzivní volání, což může vyžadovat nemalé systémové prostředky. Například zavoláním funkce `fib` s parametrem `n` nastaveným na větší hodnotu, například 40, se odezva programu značně prodlouží. Důvodem je, že v případě funkce `fib` narůstá počet rekurzivních volání exponenciálně v závislosti na parametru `n`.[140] Speciálním případem rekurze je *koncová rekurze*.[141] Při koncové rekurzi je rekurzivní volání funkce posledním příkazem v těle funkce, je pouze jedno a výsledek rekurzivního volání je použit jako návratová hodnota funkce. V případě koncové rekurze obvykle není nutné udržovat všechna nedokončená rekurzivní volání.[142] Následuje příklad funkce `fib` využívající koncovou rekurzi.[143]

[140] Strom rekurze je exponenciálně velký.

[141] Anglicky tail recursion.

[142] V době volání koncové rekurze je již tělo funkce téměř celé vykonané. Jediné, co zbývá, je právě rekurzivní volání.

[143] Průběh (koncově) rekurzivního volání pro `fib(6, 0, 1)` vypadá následovně.




```
# výpočet n-tého členu Fibonacciho posloupnosti
# koncová rekurze
def fib(n, prvni_clen, druhy_clen):
  if n < 1:
    return prvni_clen
  return fib(n - 1, druhy_clen, prvni_clen + druhy_clen)

print(fib(6, 0, 1))
```

Při výpočtu n-tého členu Fibonacciho posloupnosti vyžaduje funkce používající koncovou rekurzi mnohem méně rekurzivních volání a vrací výsledek téměř okamžitě i pro mnohem větší hodnoty `n`.

Shrnutí

Ukázali jsme, jakým způsobem je možné v jazyce Python vytvářet funkce. Funkce v programovacích jazycích jsou důležitým nástrojem, který umožňuje modularizaci a znovupoužitelnost zdrojového kódu.

Úkoly

Úkol 47
Napište rekurzivní funkci pro výpočet faktoriálu zadaného čísla n.

Kontrolní otázky
Odpovězte na následující otázky:

1. K čemu slouží funkce?
2. Co jsou lokální a globální proměnné?
3. Co je to rekurze?
4. Jaký je rozdíl mezi funkcí a procedurou?

Úkol 48
Naprogramujte funkci `mocnina(zaklad, exponent)` vracející hodnotu $zaklad^{exponent}$. Při implementaci funkce nepoužívejte operátor `**`.

Úkol 49
Napište rekurzivní funkci pro výpočet délky seznamu.

Úkol 50
Napište funkci `rez(seznam, start, stop, krok)`, která se chová stejně jako řez seznamu.

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,[144] až je úloha vyřešena vlastními silami.
Dodejme, že mohou existovat i jiná řešení, ne nutně horší, než ta, která jsou zde uvedena.

[144] Rozhodně jej ale není dobré vynechat.

Řešení úkolu 3:

```
a = 10
print(f"Obsah čtvrece o straně a = {a} je {a*a}")
```

Řešení úkolu 5:

```
fahrenheit = 42
celsius = (5 * (fahrenheit - 32)) / 9
print(f"Teplota {farnhait} (F) je {celsius} (C)")
```

Řešení úkolu 8:[145]

```
cislo = 123

# nepěkné, přímočaré řešení
print((cislo - (cislo % 100)) / 100)
print(((cislo % 100) - ((cislo % 100) % 10)) / 10)
print(((cislo % 100) % 10) / 1)
```

[145] Uvedené řešení vypisuje výsledek jako desetinné číslo. To lze odstranit pomocí formátovacího řetězce. Nebo je možné operátor / nahradit // jeho výsledkem je celé číslo. Obě možnosti vyzkoušejte.

```
# elegantnější řešení pomocí proměnných
desitky = cislo % 100
jednotky = desitky % 10

print((cislo - desitky) / 100)
print((desitky - jednotky) / 10)
print(jednotky)
```

Řešení úkolu 9:[146]

```
znak = "A"
posun = ord("A") - ord("a") # 32
print(chr(ord(znak) + posun))
```

[146] Jedná se o klasickou programovací úlohu. Řešení spočívá v tom, že rozdíl mezi čísly reprezentující velká písmena a čísly reprezentující odpovídající malá písmena je v ASCII tabulce je 32.

Řešení úkolu 11:[147]

```
# na samotných hodnotách nezáleží
A = True
B = False

# True pro libovolnou kombinaci A a B
print(not (A and B) == (not A or not B))

# True pro libovolnou kombinaci A a B
print((not A and not B) == (not (A or B)))
```

[147] Výrazy je třeba správně uzávorkovat, jelikož == má vetší prioritu než `and` a `or`.

Řešení úkolu 21:[148]

```
# nejvetší společný dělitel nenulových kladných čísel x a y
x = 12
y = 8

while y != 0:
  t = y
  y = x % y
  x = t

print(f"Největší společný dělitel je {x}")
```

[148] Jedná se o Eukleidův algoritmus pro nalezení největšího společného dělitele dvou čísel. Pro lepší představu, jak program funguje, je v tabulce níže zachycen stav proměnných před začátkem každé iterace (poslední sloupec tabulky).

x	y	y != 0	t	iterace
12	8	True	-	0.
8	4	True	8	1.
4	0	False	4	2.

Řešení úkolu 27:

```
n = 9
range_i = range(0,n)
range_j = range(0,n)
```

```
for i in range_i:
  for j in range_j:
    if i == j or (n-i-1) == j:
      print("*", end="")
    else:
      print(".", end="")

  print()
```

Řešení úkolu 29:

```
n = round((konec - start) / krok)

for i in range(n):
    print(i * krok + start)
```

Řešení úkolu 33(a):

```
retezec = "Dnes mi přišel superspam schospamvaný spamemail."
hledany_retezec = "spam"
novy_retezec = ""
i = 0

for znak in retezec:
  if znak != hledany_retezec[i]:
    novy_retezec += hledany_retezec[:i] + znak
    i = 0
  else:
    i = (i+1) % len(hledany_retezec)

# vytiskneme řetězec bez spamu
print(novy_retezec)
```

Řešení úkolu 33(b):

```
retezec = "Dnes mi přišel superspam schospamvaný spamemail."
novy_retezec = retezec.replace("spam", "")

print(novy_retezec)
```

Řešení úkolu 36

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

for i in range(len(L)//2):
```

```
  temp = L[i] # dočasné uložení
  L[i] = L[-(i+1)]
  L[-(i+1)] = temp

print(L)
```

Řešení úkolu 40[149]

```
# Eratosthenovo síto
n = 101
prvocisla = []
neni_prvocislo = [] # seznam čísel, které nejsou prvočísla

for i in range(2, n+1):
  if i not in neni_prvocislo:
    prvocisla.append(i) # i je prvočíslo
    for j in range(i*i, n+1, i):
      neni_prvocislo.append(j) # všechny násobky i nejsou
          prvočísla

print(prvocisla)
```

[149] Klasická úloha na algoritmizaci. Pokud najdeme prvočíslo i, můžeme vyloučit všechny jeho násobky. Tento algoritmus je známý jako Eratosthenovo síto.

Rejstřík