Python

Základy programování

Petr Gregor

ICT Pro

3.-5. prosince 2025

Petr Gregor

- Lektor
  - Python
  - Java
  - AI
  - Data Science
- Analytik na MUNI
- Programátor (Pascal, C, C++, C#, Java, JavaScript, PHP, VB, Python, Ruby,...)
- 25+ let zkušeností s výukou
- 20+ let zkušeností s programováním

ictPRO

Obsah školení: Část 1/4

1. Základy programování
   - Práce s daty a vývoj software
   - Výběr programovacího jazyka
   - Silné a slabé stránky Pythonu
2. Pracovní prostředí
   - Instalace Pythonu
   - Práce v příkazovém řádku
   - Úprava programů v textovém editoru
   - Online nástroje
3. První kroky
   - Počítání a práce s čísly
   - Komunikace s uživatelem
   - Proměnné a hodnoty
   - Práce s textem

Obsah školení: Část 2/4

4. Přiřazení a kopírování
   - Objekty a reference
   - Uložení globálních proměnných
   - Význam operátoru přiřazení
   - Kopírování objektů
   - Seznamy objektů
5. Struktura kódu
   - Příkazy a odsazení
   - Podmínky a cykly
   - Ošetření chyb
6. Organizace kódu
   - Psaní funkcí
   - Předávání parametrů
   - Lokální proměnné
   - Dokumentace

Obsah školení: Část 3/4

7. Objektově orientované programování
   - Instance a třídy
   - Datové atributy
   - Psaní metod
   - Základy polymorfismu
   - Jednoduchá dědičnost
8. Objektový model v Pythonu
   - Konstruktory
   - Speciální metody
   - Dynamické vlastnosti
   - Duck-typing
   - Volání metod předka

Obsah školení: Část 4/4

9. Standardní knihovna
   - Využívání hotových nástrojů
   - Ukládání dat
   - Práce s HTTP

10. Diskuze a další zdroje
    - Vlastní témata
    - Dokumentace
    - Česká komunita

Základy programování

Práce s daty a vývoj software (Data jako základ)

Proč se učíme programovat?

Moderní software je postaven na jedné věci: **Zpracování dat**.

- **Data:** Surové informace, které systém přijímá (např. vstupy od uživatele, hodnoty ze senzorů, soubory).
- **Program:** Sada instrukcí, která definuje, jak se tato data **načtou, zpracují, uloží a prezentují**.
- **Cíl:** Převést surová data na **užitečné informace** a řešení pro uživatele.

Příklad

Kód kalkulačky bere **data** (čísla 5, 3 a operátor +) a vrací **informaci** (výsledek 8).

Práce s daty a vývoj software (Cyklus vývoje)

Práce s daty a vývoj software (Cyklus vývoje)

Software Development Life Cycle (SDLC)

Cyklus vývoje softwaru (SDLC)

Proces vývoje nového programu je systematický a cyklický.

1. **Analýza a Plánování:** Definování cílů a požadavků. (*Co chceme?*)
2. **Návrh (Design):** Tvorba architektury, datových modelů a rozhraní. (*Jak to uděláme?*)
3. **Implementace (Kódování):** Převod návrhu do funkčního kódu (to je naše role!).
4. **Testování:** Ověření, že kód splňuje požadavky a neobsahuje chyby.
5. **Nasazení a Údržba:** Uvedení do provozu a řešení budoucích problémů.

Výběr programovacího jazyka (Principy)

Rozdíly v přístupu

Programovací jazyky lze rozdělit podle způsobu, jakým se jejich kód spouští.

- **Kompilované jazyky**
- **Interpretované jazyky**

Výběr programovacího jazyka (Principy)

- **Kompilované jazyky:**
  - Kód je před spuštěním **přeložen (kompilován)** do strojového kódu.
  - Výsledkem je samostatný spustitelný soubor (např. `.exe`).
  - **Výhody:** Rychlejší běh, lepší optimalizace, včasná detekce chyb.
  - **Nevýhody:** Delší vývojový cyklus, závislost na platformě.
  - *Příklady: C++, Java (částečně), Go.*

Výběr programovacího jazyka (Principy)

- **Interpretované jazyky:**
  - Kód je **čten a vykonáván řádek po řádku** interpreterem.
  - Není potřeba kompilace před spuštěním.
  - **Výhody:** Rychlejší vývoj, platformní nezávislost.
  - **Nevýhody:** Pomalejší běh (většinou), chyby se objeví až za běhu.
  - *Příklady: Python, JavaScript, Ruby.*

Výběr programovacího jazyka (Praktické faktory)

Kritéria pro praktické nasazení

Volba jazyka je málokdy jen o rychlosti; často rozhodují externí faktory.

- **Cíl projektu / Účel:**
  - Je projekt určen pro **web** (např. Python, JavaScript)? Pro **mobilní aplikace** (např. Kotlin, Swift)? Pro **vědecké výpočty** (např. Python, R)?
  - **Python** vyniká ve skriptování, analýze dat a backendu.
- **Ekosystém a Knihovny:**
  - Jaké **hotové nástroje** jsou k dispozici, aby se nemuselo „vynalézat kolo“?
  - Python má obrovskou standardní knihovnu a tisíce externích balíčků (`NumPy`, `Pandas`, `Django`).

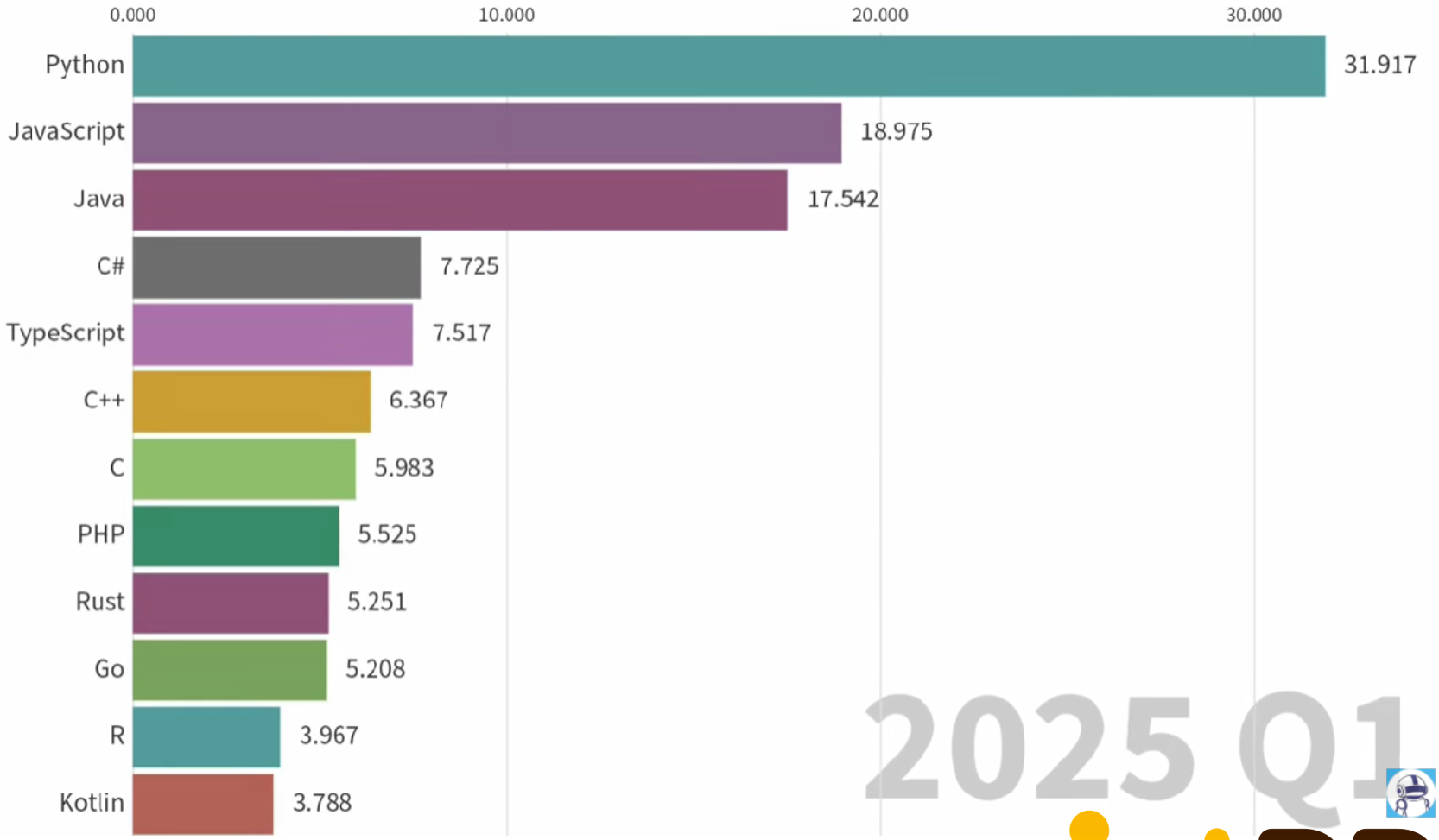
Výběr programovacího jazyka (Praktické faktory)

Kritéria pro praktické nasazení

Volba jazyka je málokdy jen o rychlosti; často rozhodují externí faktory.

- **Komunita a Dokumentace:**
  - Velká a aktivní komunita znamená **rychlejší pomoc** a více zdrojů k učení.
  - Kvalitní, aktuální dokumentace je klíčová pro řešení problémů.
- **Učení a Nábor:**
  - Jak obtížné je **naučit** se jazyk (nízká vstupní bariéra)?
  - Jak snadné je najít **zkušené vývojáře**? (Python je populární, a proto je nábor snadnější.)

Výběr programovacího jazyka (Praktické faktory)



Zdroj: https://youtu.be/fe2cO3nEmAg?si=UZx4W4z40HgTYaVL

Silné stránky Pythonu

Klíčové výhody (STRENGTHS) — Důvody, proč je Python tak oblíbený napříč obory.

- **Čitelnost a Údržba:**
  - Jednoduchá, jasná syntaxe a povinné odsazení (**PEP 8**) nutí psát „čistý kód“.
  - Kód je téměř jako pseudokód, což snižuje náklady na údržbu.
- **Nízká bariéra vstupu (Ideální pro školení):**
  - Python je výborný pro rychlé uchopení základů programování a algoritmizace.
- **Obrovský Ekosystém a Knihovny:**
  - Obrovská standardní knihovna („batteries included“).
  - Dostupnost modulů pro „Data Science“ (`NumPy`, `Pandas`), web (`Django`, `Flask`) a automatizaci.

Rychlost, s jakou dokážete v Pythonu realizovat nápad, je často jeho největší předností.

Slabé stránky Pythonu

Omezení a nevýhody (WEAKNESSES) — Kdy je potřeba zvážit jiný jazyk.

- **Rychlost běhu (Performance):**
  - Jako **interpretovaný** jazyk je v čistém kódu pomalejší než kompilované jazyky (C, C++, Go).
  - Pomalejší spouštění programu (tzv. „startup time“).
- **Omezení paralelismu (GIL):**
  - **Global Interpreter Lock (GIL)** brání plnému využití více procesorových jader standardního Pythonu.
- **Dynamické chyby za běhu (Runtime Errors):**
  - **Dynamické typování** (nemusíte deklarovat typ) je sice rychlé, ale chyby v typování se odhalí až při spuštění programu, ne při kompilaci.
- **Spotřeba paměti:**
  - Flexibilní a dynamická povaha objektů může vést k vyšší paměťové náročnosti.

Pro většinu firemních a akademických úloh však tyto nevýhody nejsou kritické, protože je lze obvykle obejít (např. voláním C knihoven).

Pracovní prostředí

Instalace Pythonu

Získání interpreta

Vždy stahujte a instalujte **aktuální stabilní verzi** Pythonu.

1. **Stažení:** Navštivte oficiální web `python.org`.
   - Vyberte instalátor odpovídající Vašemu operačnímu systému (Windows, macOS, Linux).
2. **Instalace (Windows):**
   - **Klíčový krok:** Při instalaci **zaškrtněte možnost** „Add Python to PATH“. To umožní spouštět Python z příkazového řádku odkudkoli.
   - Dokončete instalaci.
3. **Ověření:** Otevřete příkazový řádek (nebo terminál) a zadejte:

```
python --version
pip --version
```

4. Měli byste vidět nainstalované verze Pythonu a nástroje **pip** (správce balíčků).

Instalace Pythonu (Virtuální prostředí)

Proč používat virtuální prostředí (`venv`)?

Virtuální prostředí slouží k izolaci projektů a jejich závislostí.

- **Problém závislostí:** Různé projekty často vyžadují „různé verze stejné knihovny“.
  - Projekt A potřebuje `Requests 1.0` a Projekt B potřebuje `Requests 2.5`. Instalace jedné verze globálně by porušila druhou.
- **Řešení:** Virtuální prostředí (`venv`) vytvoří „izolovanou kopii“ Pythonu pro daný projekt.
  - Všechny balíčky instalované pomocí `pip` se uloží pouze do tohoto izolovaného adresáře.
  - Globální instalace Pythonu zůstane **čistá**.

Instalace Pythonu (Virtuální prostředí)

- **Základní příkazy:**
  1. Vytvoření: `python -m venv nazev_prostredi`
  2. Aktivace (Linux/macOS): `source nazev_prostredi/bin/activate`
  3. Aktivace (Windows): `nazev_prostredi\Scripts\activate`

Vždy začínejte nový projekt vytvořením a aktivací virtuálního prostředí.
Většina IDE (např. PyCharm) vytváří virtuální prostředí automaticky.

Práce v příkazovém řádku (Základy navigace)

Terminál a příkazový řádek

Terminál (shell) je „přímý přístup“ k operačnímu systému. Je nezbytný pro spouštění Pythonu a správu balíčků.

- **Aktuální adresář:**
  - `pwd` (Linux/macOS) nebo `cd` bez argumentů (Windows): Zjistí, kde se právě nacházíte.
- **Výpis obsahu:**
  - `ls` (Linux/macOS) nebo `dir` (Windows): Zobrazí seznam souborů a složek v aktuálním adresáři.
- **Změna adresáře (Move):**
  - `cd nazev_slozky`: Přesun do podsložky.
  - `cd ..`: Přesun o úroveň výš.
  - `cd /` nebo `cd ~` (Linux/macOS): Přesun do kořenového nebo domovského adresáře.

Doporučuje se pracovat v anglickém názvosloví cest a souborů bez mezer.

Práce v příkazovém řádku (Spouštění Pythonu)

Interakce s interpretem

Python se spouští pomocí příkazu `python` a má dva základní režimy spuštění.

- **Interaktivní režim (REPL):**
  - Slouží k rychlému „testování kódu“, ověření syntaxe nebo jednoduchým výpočtům.
  - **Spuštění:** Zadejte pouze příkaz `python` (nebo `python3`).
  - Dostanete se do prostředí s promptem `>>>`, kde můžete psát kód řádek po řádku.
  - **Ukončení:** Použijte `exit()` nebo `Ctrl+Z` (`Ctrl+D` na Linux/macOS).
- **Spuštění souboru (Skript):**
  - Slouží ke spuštění komplexních, předem uložených programů.
  - **Spuštění:** `python muj_skript.py`
  - Interpreter provede všechny příkazy v souboru od začátku do konce.
  - Skript se spustí v aktuálně **aktivním virtuálním prostředí**.

Správce balíčků PIP (Instalace)

Základní práce s knihovnami

PIP („Pip Installs Packages“) stahuje a umisťuje externí knihovny do Vašeho projektu.

- **Instalace aktuální verze:**
  - `pip install nazev_balicku`
  - Nainstaluje **nejnovější stabilní verzi** balíčku do aktivního virtuálního prostředí.
  - *Příklad:* `pip install requests`
- **Instalace specifické verze:**
  - `pip install nazev_balicku==verze`
  - Zajišťuje „stabilní a reprodukovatelné chování“ v projektech, kde je nutná konkrétní verze balíčku.
  - *Příklad:* `pip install django==4.2.0`
- **Aktualizace a odinstalace:**
  - `pip install --upgrade nazev_balicku`
  - `pip uninstall nazev_balicku`

Správce balíčků PIP (Závislosti)

Správa požadavků (`requirements.txt`)

Projekty musí být „reprodukovatelné“ – to znamená, že je musí jít spustit kdekoli se stejnými závislostmi.

- **Uložení závislostí:**
  - `pip freeze > requirements.txt`
  - **Klíčový příkaz.** Uloží „přesný seznam“ všech nainstalovaných balíčků a jejich verzí z virtuálního prostředí do souboru.
  - Tento soubor se typicky sdílí s kódem na GitHubu.
- **Hromadná instalace:**
  - `pip install -r requirements.txt`
  - Nainstaluje všechny balíčky (včetně přesných verzí) definované v souboru.
  - To je standardní postup, jak nastavit prostředí novému vývojáři.
- **Zobrazení informací:**
  - `pip list`: Vypíše všechny balíčky v aktivním prostředí.
  - `pip show nazev_balicku`: Zobrazí detaily o konkrétním balíčku.

Úprava programů (Editor vs. IDE)

Proč nepoužívat Notepad?

Pro psaní kódu je **obyčejný textový editor** (jako Notepad) naprosto nedostatečný.

- **Textový Editor (Simple Editor):**
  - Slouží pouze k zápisu textu.
  - Nezajistí „zvýraznění syntaxe“, nevidí chyby, nezná Python.
  - *Příklady:* Poznámkový blok (Windows), TextEdit (macOS).
- **Pokročilý Editor (VS Code, Sublime):**
  - Určen primárně pro kód.
  - Podporuje zvýraznění syntaxe, odsazování a „rozšíření“ pro konkrétní jazyky.
- **IDE (Integrated Development Environment):**
  - **Integrované vývojové prostředí** (např. PyCharm, Visual Studio).
  - Komplexní nástroj spojující editor s funkcemi pro ladění, testování a správu virtuálních prostředí.

Klíčové funkce IDE pro Python

Nástroje pro efektivní kódování

IDE a pokročilé editory „dělají práci za Vás“ tím, že předvídají a kontrolují.

- **Zvýraznění syntaxe (`Syntax Highlighting`):**
  - Různé části kódu (klíčová slova, proměnné, řetězce) jsou „barevně odlišeny“, což usnadňuje čtení a odhalování překlepů.
- **Automatické doplňování (`IntelliSense`):**
  - Editor Vám „navrhuje možné názvy funkcí a metod“ na základě toho, co píšete. Tím výrazně zrychluje psaní kódu.
- **Lintery a kontrola chyb (`Linting`):**
  - Kontrola kódu **v reálném čase** na chyby (např. chybějící dvojtečka) nebo na nedodržení stylu (PEP 8).
  - Chyby jsou obvykle podtrženy vlnovkou.
- **Ladění kódu (`Debugger`):**
  - Umožňuje „krokovat“ program po řádcích a sledovat, jak se mění hodnoty proměnných. Nezbytné pro hledání komplexních chyb.

*Doporučené nástroje pro Python: **PyCharm** (IDE), **VS Code** (pokročilý editor).*

Online nástroje (Rychlé testování)

Rychlé prototypování bez instalace

Online nástroje Vám umožní „rychle psát a spouštět kód“ bez nutnosti mít vše instalované na vlastním počítači.

- **Online Interpretry:**
  - Ideální pro testování „jednoduchých algoritmů“ nebo ověření syntaxe.
  - Nabízejí základní rozhraní pro kód, vstup a výstup.
  - *Příklady:* OnlineGDB, Python.org Shell.
- **Online IDE/Platformy (REPLs):**
  - Poskytují komplexnější prostředí pro „malé projekty“. Umožňují práci s více soubory.
  - Skvělé pro sdílení a spolupráci na jednoduchých úkolech.
  - *Příklady:* **Replit**, CodePen (pro JS/HTML, ale některé podporují Python).
- **Kdy je použít?**
  - Při **prvních krůčcích** s Pythonem, na cestách, nebo když nemáte přístup ke svému počítači.

Online nástroje (Jupyter Notebooks)

Jupyter Notebook a Google Colab

Jde o „interaktivní prostředí“, kde můžete kombinovat kód, výstupy a dokumentaci do jednoho dokumentu.

- **Charakteristika Notebooku:**
  - Kód je rozdělen do **bunēk** („cells“), které lze spouštět nezávisle na sobě.
  - Ideální pro **analýzu dat**, vizualizace a vytváření srozumitelných tutoriálů.
  - Výstupy („tabulky, grafy, text“) se zobrazují přímo pod spuštěnou buňkou.

Online nástroje (Jupyter Notebooks)

Jupyter Notebook a Google Colab

Jde o „interaktivní prostředí“, kde můžete kombinovat kód, výstupy a dokumentaci do jednoho dokumentu.

- **Jupyter Notebook / JupyterLab:**
  - Lze nainstalovat lokálně pomocí `pip` a spustit v prohlížeči.
  - Poskytuje plnou kontrolu nad prostředím.
- **Google Colaboratory (Colab):**
  - **Cloudová verze** Jupyter Notebooku od Google.
  - Běží kompletně online a často nabízí bezplatný přístup k výkonnému hardware (GPU/TPU).
  - Ideální pro studentské a akademické projekty.

Notebooky jsou v datové vědě de facto „standard pro sdílení“ analytické práce.

První kroky

Počítání a práce s čísly

Základní datové typy pro čísla

Python pracuje primárně se dvěma typy čísel: **celá čísla** (`int`) a **desetinná čísla** (`float`).

- **Celočíselný typ (`int`):**
  - Neomezená přesnost (využívá libovolný počet bytů).
  - Př. `10`, `-500`, `1000000`.
- **Desetinný typ (`float`):**
  - V Pythonu reprezentovaný jako „dvojitá přesnost“ (standard IEEE 754).
  - Př. `10.5`, `3.14159`, `2.0`.

Počítání a práce s čísly

- **Základní aritmetické operátory:**
  - Sčítání (`+`), Odčítání (`-`), Násobení (`*`).
  - Klasické dělení (`/`) – **vždy vrací float** (např. `5 / 2` vrací `2.5`).
  - Celočíselné dělení (`//`) – **zahodí desetinnou část** (např. `5 // 2` vrací `2`).
  - Zbytek po dělení (`%`, modulo) – (např. `5 % 2` vrací `1`).
  - Mocnina (`**`) – (např. `2 ** 3` vrací `8`).

Komunikace s uživatelem (vstup a výstup)

Základní interakce s programem

Každý program musí umět **zobrazit informace** a **přijímat data** od uživatele.

- **Vstup (Příjem dat):**
  - Používá se funkce `input(zprava)`.
  - Funkce **vždy vrací textový řetězec** (`str`), i když uživatel zadá číslo!
  - Je nutná „konverze typu“ (`casting`) na číslo, pokud chcete s daty počítat.
  - **Příklad konverze:**

```
1 age_str = input("Zadejte vek: ")
2 age = int(age_str) % Konverze na cele cislo
```

Komunikace s uživatelem (Vstup a Výstup)

Základní interakce s programem

Každý program musí umět **zobrazit informace** a **přijímat data** od uživatele.

- **Výstup (Zobrazení dat):**
  - Používá se funkce `print(argumenty)`.
  - Automaticky přidává „nový řádek“ po každém volání.
  - **Formátování (F-stringy):** Moderní způsob pro vkládání proměnných do textu (řetězce):

```
1 print(f"Vysledek je {10 + 5}")
```

Proměnné a hodnoty

Co je proměnná?

Proměnná je „jmenovka“, kterou umístíme na hodnotu v paměti. Hodnota je to, co se počítá a ukládá.

- **Operátor přiřazení:**
  - Používá se operátor `=` (rovná se).
  - Funguje zprava doleva: hodnota na pravé straně se „přiřadí“ proměnné na levé straně.
  - *Příklad:* `cena = 50`
- **Dynamické typování:**
  - V Pythonu nemusíte proměnným explicitně říkat, jaký typ dat budou držet („int“, „str“, „float“).
  - Typ proměnné je určen **hodnotou, kterou drží**.
  - *Příklad:* `x = 10` (x je `int`), později `x = 'text'` (x je `str`).

Proměnné a hodnoty

- **Pravidla pro pojmenování (PEP 8):**
  - Používejte malá písmena a slova oddělujte podtržítkem (`snake_case`).
  - Musí začínat písmenem nebo podtržítkem (nesmí číslem).
  - Vyhněte se rezervovaným slovům (`if`, `for`, `print`).
  - *Příklad:* `cena_polozky`, `jmeno_uzivatele`.

Díky dynamickému typování je Python flexibilní, ale vyžaduje od programátora pečlivost.

Práce s textem (Základy a operace)

Textový řetězec (`str`)

Textový řetězec je **sekvence znaků** a je jedním z nejpoužívanějších datových typů.

- **Definice řetězce:**
  - Řetězce se uzavírají do „jednoduchých“ (`'text'`) nebo „dvojitých“ (`"text"`) uvozovek.
  - Je dobrá praxe být **konzistentní** (např. používat primárně dvojité uvozovky).
  - **Více řádků:** Použijte trojité uvozovky (např. `"""text"""`).

Práce s textem (Základy a operace)

- **Základní operace:**
  - **Spojování (`+`):** Spojí dva řetězce dohromady.
    Příklad: `jmeno + ' ' + prijmeni`
  - **Násobení (`*`):** Vytvoří kopii řetězce tolikrát, kolikrát je násobena.
    Příklad: `'A' * 5` vrací `'AAAAA'`
- **Indexování (Přístup ke znakům):**
  - Znaky jsou číslovány **od nuly** zleva.
    Příklad: `'Python'[0]` vrací `'P'`.

Práce s textem (Klíčové metody)

Metody řetězců

Python nabízí „desítky vestavěných metod“, které slouží k transformaci a kontrole textových dat.

- **Změna velikosti písmen (Case conversion):**
  - `text.upper()`: Převede všechna písmena na velká.
  - `text.lower()`: Převede všechna písmena na malá.
  - `text.capitalize()`: První písmeno na velké, zbytek na malá.
- **Odstranění mezer (Whitespace):**
  - `text.strip()`: Odstraní „bílé znaky“ (mezery, tabulátory, nové řádky) ze začátku a konce řetězce.
- **Kontrola řetězce (Checking):**
  - `text.startswith("A")`: Vrací `True` nebo `False` (pravda/nepravda).
  - `text.isdigit()`: Vrací `True`, pokud řetězec obsahuje jen číslice.
- **Dělení (Splitting):**
  - `text.split(separator)`: Rozdělí řetězec na „seznam řetězců“ podle daného oddělovače.

*Důležité: Metody řetězce **nemění** původní řetězec, ale vrací jeho novou, upravenou kopii.*

Přiřazení a kopírování

Objekty a reference

Vše je objekt

V Pythonu je **vše objekt** (číslo, řetězec, funkce, třída). Proměnná je pouze „jmenovka“ nebo „reference“, která na objekt ukazuje.

- **Reference (jmenovka):**
  - Když napíšeme `a = 10`, `a` se stane referencí na objekt s hodnotou `10`.
  - Když poté napíšeme `b = a`, proměnná `b` neobsahuje novou kopii hodnoty `10`, ale odkazuje „na stejný objekt“ jako `a`.
- **ID objektu:**
  - Funkce `id(objekt)` vrací unikátní identifikátor objektu v paměti.
  - Pokud dvě proměnné odkazují na tentýž objekt, jejich `id()` bude stejné.
- **Demonstrace:**

```
1 a = 10
2 b = a
3 print(id(a)) # Napr.: 140730032992928
4 print(id(b)) # Stejne jako id(a)!
```

Uložení globálních proměnných (Scope)

Rozsah platnosti (Scope)

Proměnné mají omezený **rozsah platnosti**. Kód, který je mimo tento rozsah, proměnnou nevidí.

- **Lokální proměnné:**
  - Vytvořeny „uvnitř funkce“ a existují pouze po dobu běhu této funkce.
  - **Implicitní chování:** Když uvnitř funkce přiřazujete, vždy vytváříte novou **lokální** proměnnou (pokud již lokálně neexistuje).
- **Globální proměnné:**
  - Vytvořeny „mimo jakoukoliv funkci“ (na nejvyšší úrovni modulu).
  - **Čtení** globální proměnné uvnitř funkce je bezproblémové.
- **Klíčové slovo `global`:**
  - Pokud chcete **změnit hodnotu** existující globální proměnné uvnitř funkce, musíte použít klíčové slovo `global`.

Uložení globálních proměnných (Scope)

Příklad použití `global`:

```
1 cite_number = 0 # GLOBALNI
2
3 def cite():
4     global cite_number  # Chceme zmenit globalni promennou
5     cite_number = cite_number + 1
6     print(f"Citovano: {cite_number}x")
7
8 cite() # Vypise: Citovano: 1x
```

Používání globálních proměnných je obecně „dobré omezit“, aby byl kód čitelnější a předvídatelnější.

Význam operátoru přiřazení (=)

Přiřazení rovná se reference

Operátor = znamená **vytvořit novou referenci** na existující objekt. „Nekopíruje data“, pouze dává objektu novou jmenovku.

- **Imutabilní objekty (čísla, řetězce):**
  - Pokud změníte `a`, Python vytvoří nový objekt, a `a` se přesune. `b` bude stále ukazovat na starou hodnotu.
- **Mutabilní objekty (seznamy, slovníky):**
  - Zde je problém nejviditelnější! Objekty lze „měnit na místě“.
  - Když `seznam_A` přiřadíte `seznam_B` (`seznam_B = seznam_A`), obě jmenovky ukazují na **stejný seznam v paměti**.

Význam operátoru přiřazení (=)

Demonstrace mutability:

```
1 list_A = [1, 2, 3]
2 list_B = list_A  # Vytvoreni reference
3
4 list_B.append(4) # Zmena pres list_B
5
6 print(list_A)
7 # Vypis: [1, 2, 3, 4]
8 # list_A je take zmenen, protoze sdili data!
```

Pro skutečné kopírování mutabilních objektů je nutné použít metody jako `.copy()` *nebo modul* `copy`.

Kopírování objektů (mělké vs. hluboké)

Jak vytvořit skutečnou kopii dat?

Abychom se vyhnuli sdílení referencí, musíme data „explicitně zkopírovat“. Existují dva druhy kopírování.

- **Mělká kopie (`Shallow Copy`):**
  - Vytvoří se **nový kontejner** (nový seznam, nový slovník).
  - Obsah kontejneru se kopíruje, ale pokud kontejner obsahuje „jiné objekty“ (např. vnořené seznamy), zkopírují se pouze **jejich reference**.
  - **Použití:** Vestavěná metoda `.copy()` nebo list slicing `[:]`.
- **Hluboká kopie (`Deep Copy`):**
  - Vytvoří **nový kontejner** a rekurzivně zkopíruje „všechny vnořené objekty“.
  - Objekty v kopii a v originále jsou **zcela nezávislé**.
  - **Použití:** Je nutné importovat modul `copy` a použít funkci `copy.deepcopy()`.

Kopírování objektů (mělké vs. hluboké)

Příklad mělké kopie:

```
1 import copy
2 original = [1, [2, 3]]
3 shallow = original.copy() # Melka kopie
4
5 shallow[1].append(4) # Zmeni vnoreny seznam!
6 print(original) # Vypise: [1, [2, 3, 4]]
```

Hluboká kopie je pomalejší, ale jediná bezpečná pro kopírování složitých struktur s vnořenými mutabilními objekty.

Seznamy objektů (`list`)

Základní mutabilní datový typ

Seznam (`list`) je **uspořádaná, měnitelná (mutabilní) sekvence** různých objektů.

- **Vlastnosti seznamu:**
  - Vytváří se v „hranatých závorkách“ (např. `[1, "a", 3.14]`).
  - **Indexování:** Prvky jsou přístupné podle indexu od 0 (např. `muj_seznam[0]`).
  - **Mutabilita:** Můžete „měnit, přidávat a odebírat“ prvky po vytvoření seznamu.

Seznamy objektů (`list`) – klíčové metody

- **Klíčové metody pro manipulaci:**
  - `seznam.append(prvek)`: Přidá prvek **na konec** seznamu.
  - `seznam.insert(index, prvek)`: Vloží prvek na „specifický index“.
  - `seznam.pop()`: Odebere a **vrátí poslední prvek**.
  - `seznam.remove(hodnota)`: Odebere „první výskyt“ dané hodnoty.
- **Příklad manipulace:**

```
1 nakup = ["chleba", "mleko"]
2 nakup.append("vajicka") # ["chleba", "mleko", "vajicka"]
3 nakup.remove("chleba") # ["mleko", "vajicka"]
4 posledni = nakup.pop() # posledni = "vajicka"
```

Seznamy objektů (`list`) – indexování


```
1 colors = ['red', 'green', 'blue', 'yellow', 'black']
2 print(colors[0])  # red
3 print(colors[2])  # blue
```

Seznamy objektů (`list`) – negativní indexování


```
1 colors = ['red', 'green', 'blue', 'yellow', 'black']
2 print(colors[-1])  # black
3 print(colors[-3])  # blue
```

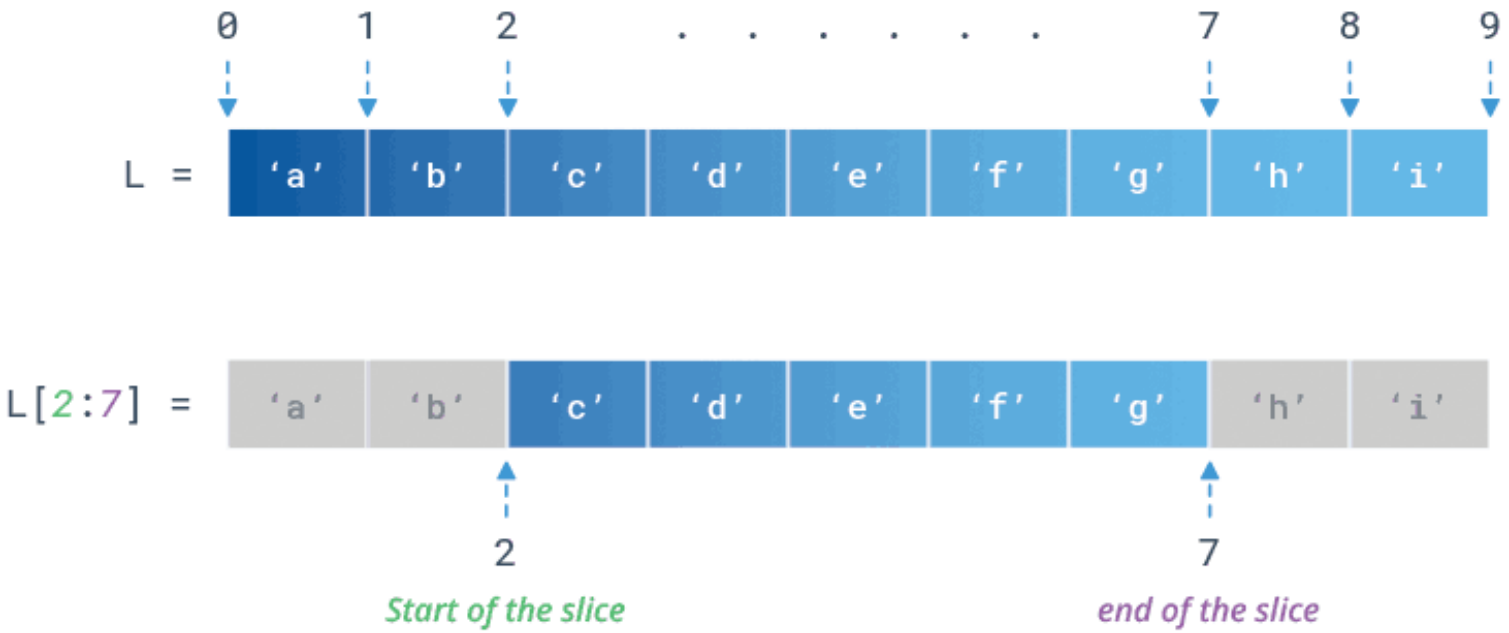
Slicing (Řezání sekvencí)

Výběr podsekvencí

Slicing je výkonný mechanismus pro výběr „souvislé části“ (podsekvence) z řetězce, seznamu nebo n-tice.

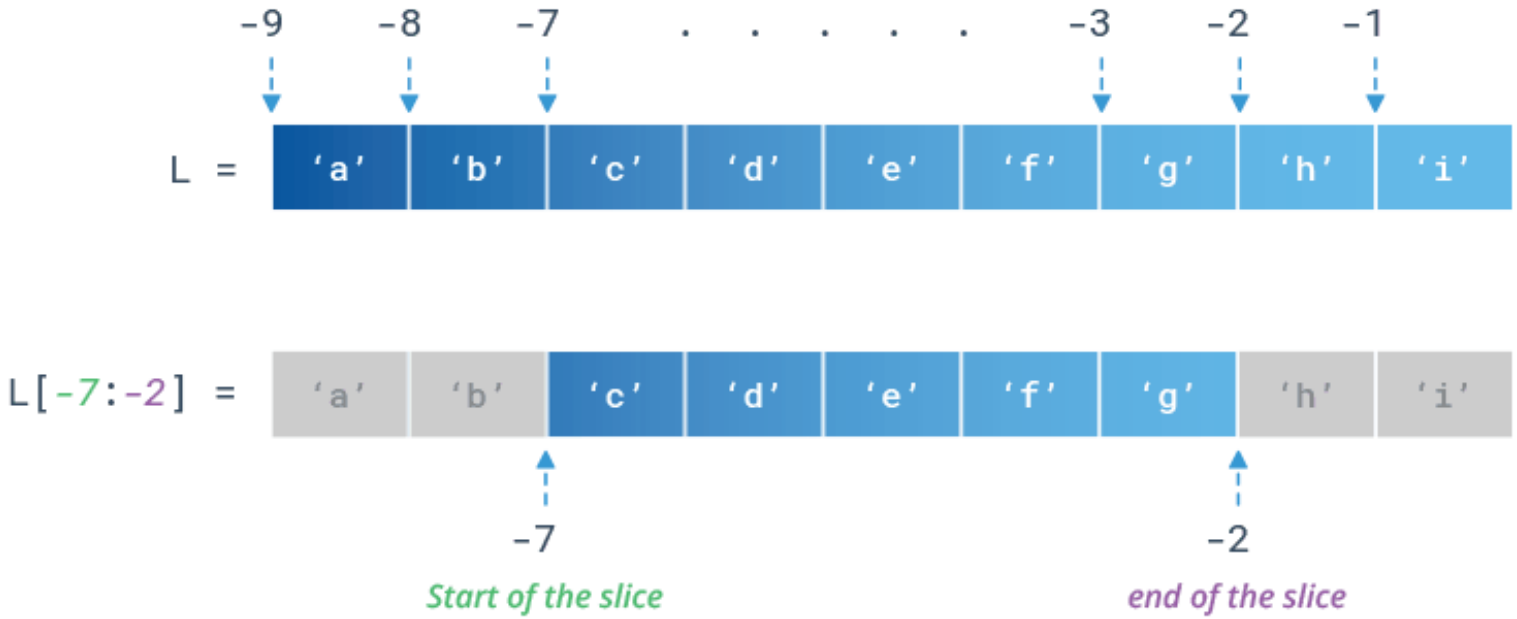
- **Základní syntaxe:**
  - `sekvence[START : STOP : KROK]`
  - **STOP index** je vždy „vyloučen“ (Python řeže po prvek před ním).
- **Důležité triky:**
  - Vynechání START/STOP znamená „od začátku“/„do konce“.
  - **Kopírování:** `seznam[:]` vytvoří „mělkou kopii“ celého seznamu.
  - **Obrácení:** `sekvence[::-1]` je rychlý trik pro „obrácení“ pořadí prvků.
- **Negativní indexování:**
  - `[-1]` odkazuje na „poslední prvek“.
  - `[-2]` na předposlední prvek, atd.

Slicing (Řezání sekvencí) – příklad



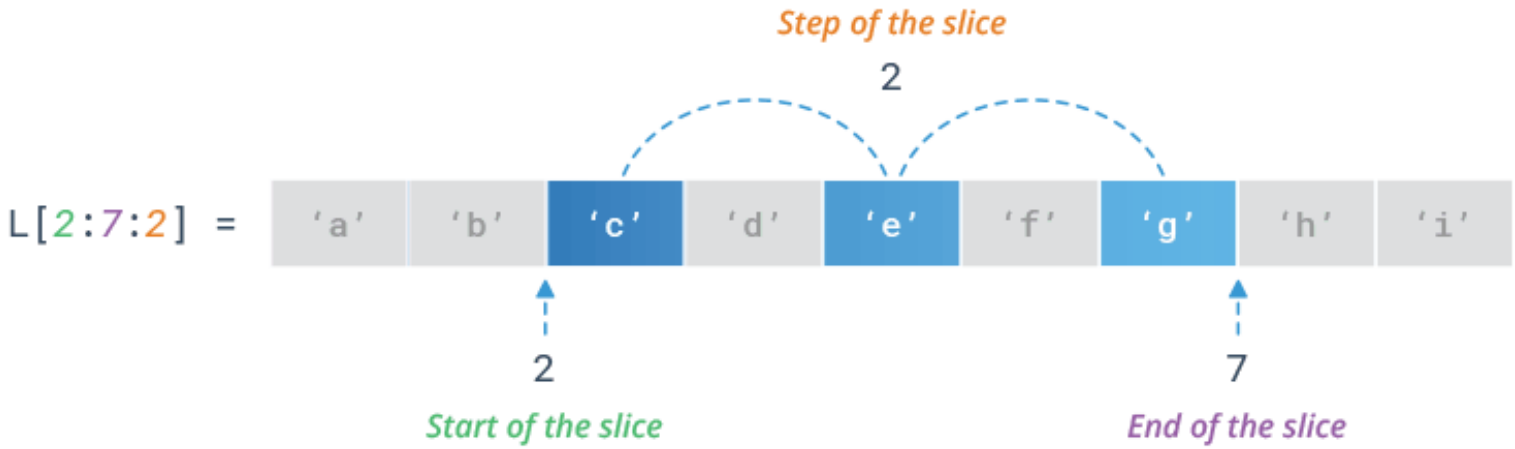

```
1 #      0    1    2    3    4    5    6    7    8    9
2 abc = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
3 print(abc[2:7])  # ['c', 'd', 'e', 'f', 'g']
4 print(abc[:5])   # ['a', 'b', 'c', 'd', 'e']
5 print(abc[6:])   # ['g', 'h', 'i']
6 print(abc[1:2])  # ['b']
```

Slicing (Řezání sekvencí) – příklad




```
#      0    1    2    3    4    5    6    7    8    9
abc = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
#     -9   -8   -7   -6   -5   -4   -3   -2   -1    0
print(abc[-7:-2])  # ['c', 'd', 'e', 'f', 'g']
print(abc[:-5])    # ['a', 'b', 'c', 'd']
print(abc[-6:])    # ['d', 'e', 'f', 'g', 'h', 'i']
print(abc[-2:-7])  # []
```

Slicing (Řezání sekvencí) – příklad




```
1 #      0    1    2    3    4    5    6    7    8    9
2 abc = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
3 #     -9   -8   -7   -6   -5   -4   -3   -2   -1    0
4 print(abc[2:7:2])   # ['c', 'e', 'g']
5 print(abc[::-1])    # ['i', 'h', 'g', 'f', 'e', 'd', 'c', 'b',
      'a']
6 print(abc[::-2])    # ['i', 'g', 'e', 'c', 'a']
```

Klíčové kolekce – `set` (množina)

Neuspořádaná kolekce unikátních prvků

Množina (`set`) je neuspořádaná, **mutabilní** kolekce, která „neobsahuje žádné duplicity“. Je ideální pro testování členství a množinové operace.

- **Vlastnosti:**
  - Vytváří se ve „složených závorkách“ (např. `{1, 5, 9}`) nebo pomocí konstruktoru `set()`.
  - Prvky množiny musí být **imutabilní** (nelze uložit seznam, lze uložit n-tici).
  - Přidání duplicitního prvku nemá žádný efekt.
- **Základní operace:**
  - `sada.add(prvek)`: Přidá jeden prvek.
  - `sada.remove(prvek)`: Odebere prvek (vyvolá chybu, pokud prvek neexistuje).
  - `sada.discard(prvek)`: Odebere prvek (bez chyby, pokud prvek neexistuje).

Klíčové kolekce – set (množina)

Množinové operace:

Operace	Symbol	Metoda
Sjednocení (Union)	|	`sada1.union(sada2)`
Průnik (Intersection)	&	`sada1.intersection(sada2)`
Rozdíl (Difference)	-	`sada1.difference(sada2)`

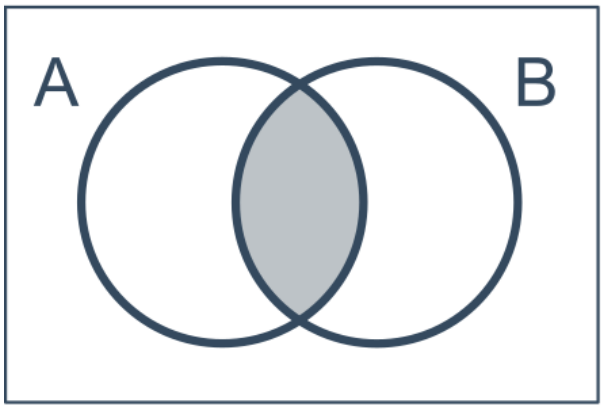
Příklad použití:

```
1 barvy = {"cervena", "modra", "zelena", "modra"}
2 print(len(barvy)) # Vypise: 3 (duplikat modra ignorovan)
3
4 nove_barvy = {"cervena", "zluta"}
5 spojeni = barvy | nove_barvy # Sjednoceni
```

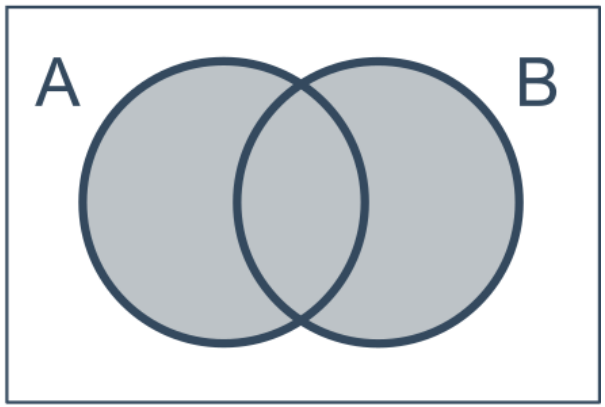
Sady jsou extrémně rychlé pro testování členství (`prvek in sada`).

Klíčové kolekce – set (množina)

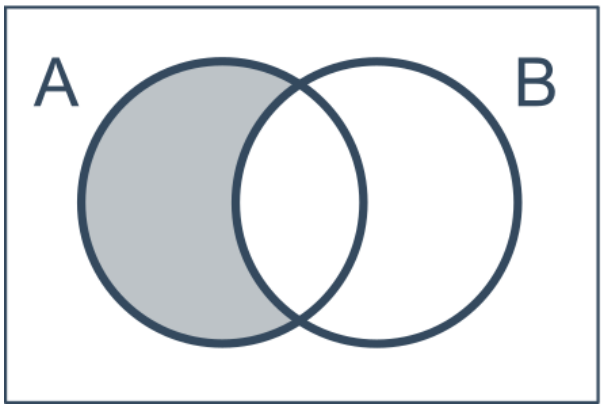
průnik



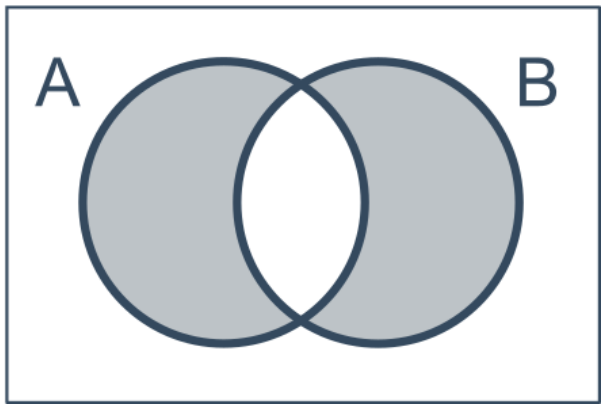
sjednocení



rozdíl



symetrický rozdíl



Klíčové kolekce (Slovníky a N-tice)

Výběr správné datové struktury

Volba mezi `list`, `tuple` a `dict` závisí na tom, zda data potřebujete „měnit“ a zda k nim přistupujete „podle indexu“ nebo „podle klíče“.

- **N-tice (`tuple`):**
  - Vytváří se v „kulatých závorkách“ (např. `(1, 2, 'c')`).
  - Je **Imutabilní** (neměnná). Je-li jednou vytvořena, nelze přidat, odebrat ani změnit prvek.
  - Ideální pro „bezpečný přenos dat“ (např. při vracení více hodnot z funkce).
- **Slovník (`dict`):**
  - Vytváří se ve „složených závorkách“ (např. `{'jmeno': 'Jan', 'vek': 30}`).
  - Je **Mutabilní** (měnitelný) a ukládá data ve formátu **Klíč: Hodnota**.
  - Přístup k hodnotám se děje pomocí „klíče“, nikoli číselného indexu.

Klíčové kolekce – příklad

Příklad Slovníku:

```
1 osoba = {"jmeno": "Petr", "povolani": "Dev"}
2 print(osoba["jmeno"]) # Vypise: Petr
3
4 osoba["povolani"] = "Manager" # Zmena hodnoty
5 osoba["mesto"] = "Brno" # Pridani noveho paru
```

Slovník je nejpoužívanější datová struktura pro „strukturovaná data“ v Pythonu.

Struktura kódu

Příkazy a odsazení (Indentation)

Python nemiluje závorky

Na rozdíl od mnoha jiných jazyků (C++, Java) Python nepoužívá závorky „{}“ k vymezení bloků kódu. Místo toho se spoléhá na **odsazení**.

- **Konec příkazu:**
  - Každý příkaz končí novým řádkem (není potřeba středník ;, pokud nepíšete více příkazů na jeden řádek – což je „špatná praxe“).
- **Bloky kódu (Code Blocks):**
  - Blok kódu (tělo funkce, cyklu, podmínky) začíná **dvojtečkou** (:).
  - Všechny příkazy v daném bloku musí být „odsazeny“ na stejnou úroveň.
- **Konvence PEP 8:**
  - Oficiální styl pro odsazení jsou **4 mezery**. (Nikdy nepoužívejte kombinaci mezer a tabulátorů – to vede k chybám!)

Příkazy a odsazení (Indentation)

Příklad struktury:

```
1 if vek >= 18:          # 1. Zacatek bloku s dvojteckou
2     print("Dospely") # 2. Odsazeno o 4 mezery
3     print("Vstup povolen") # 3. Stejne odsazeni
4 else:          # 4. Konec bloku, navrat na puvodni uroven
5     print("Neni dospely")
```

Špatné odsazení není jen nepořádek – je to „syntaktická chyba“ (`IndentationError`).

Podmínky (`if`, `elif`, `else`)

Podmíněné spouštění kódu

Podmínky řídí, „které bloky kódu“ se spustí na základě vyhodnocení logického výrazu.

- **Syntaxe:** Vždy začíná `if`, může následovat libovolný počet `elif` (else if) a končí volitelným `else`. Vše končí dvojtečkou.
- **Logické (Booleovské) hodnoty:** Podmínka musí být vyhodnocena jako `True` (pravda) nebo `False` (nepravda).
- **Relační operátory:** Porovnávají hodnoty.
  - Rovnost (`==`), Nerovnost (`!=`), Větší (`>`), Menší (`<`), Větší/rovno (`>=`), Menší/rovno (`<=`).
- **Logické operátory:** Spojují více podmínek.
  - `and`: Vrací `True` jen, když jsou „obě podmínky“ `True`.
  - `or`: Vrací `True`, když je „alespoň jedna podmínka“ `True`.
  - `not`: Obrací logickou hodnotu (`True` na `False` a naopak).

Podmínky (if, elif, else)

Příklad:

```
1 vek = 25
2 pohlavi = "Z"
3
4 if vek >= 18 and pohlavi == "M":
5     print("Muz a dospely")
6 elif vek >= 18 and pohlavi == "Z":
7     print("Zena a dospela")
8 else:
9     print("Malolety")
```

Instrukce `match` (Strukturní porovnávání)

Moderní nástroj pro podmíněné provedení (Python 3.10+)

Instrukce `match` umožňuje porovnat hodnotu s **mnoha různými strukturami** (vzorci) a spustit kód pouze při „přesné shodě vzoru“.

- **Syntaxe:** Začíná `match hodnota:` a pokračuje jedním nebo více bloky `case vzor:`.
- **Využití:**
  - Nahrazuje dlouhé a nepřehledné řetězce „`if`/`elif`“ pro testování hodnot (např. stavových kódů, typu příkazu).
  - Dokáže **rozbalit (unpack)** složité struktury (seznamy, n-tice) přímo ve vzoru case.
- **Speciální vzory:**
  - **Zástupný vzor (`_`):** Shoduje se s „čímkoli“ (funguje jako `else` nebo `default`).
  - **Vazebný vzor (`case nazev`):** Shoduje se s čímkoli a „uloží hodnotu“ do proměnné `nazev`.

Instrukce `match` (Strukturní porovnávání)

Příklad (Rozbalování seznamu):

```
1 prikaz = ["MOVE", 100, 50]
2
3 match prikaz:
4     case ["QUIT"]:
5         print("Ukonceni programu.")
6     case ["MOVE", x, y]: # Rozbali presne 3 prvky
7         print(f"Jdi na x={x}, y={y}")
8     case _:
9         print("Neznamy prikaz.")
```

Instrukce `match` výrazně zvyšuje „čitelnost“ kódu, který se zabývá zpracováním stavů a příkazů.

Cykly (`for`, `while`)

Opakované spouštění kódu

Cykly slouží k **automatickému opakování** bloku kódu, což eliminuje nutnost psát stejný kód vícekrát.

- **Cyklus `for`:**
  - Používá se pro **iteraci** (průchod) přes sekvence (seznamy, řetězce, `range()`).
  - `range(N)`: Generuje sekvenci čísel od 0 do $N - 1$.
  - *Příklad:* `for cislo in [1, 2, 3]: print(cislo)`

Cykly (`for`, `while`)

- **Cyklus `while`:**
  - Opakuje kód tak dlouho, dokud je „daná podmínka pravdivá“ (`True`).
  - Vyžaduje manuální změnu podmínky uvnitř cyklu, aby nedošlo k nekonečné smyčce.
  - *Příklad:* `while pocitadlo < 5: pocitadlo += 1`

Cykly (`for`, `while`)

- **Klíčová slova pro kontrolu cyklu:**
  - `break`: Okamžitě **ukončí celý cyklus**.
  - `continue`: Přeskočí „zbytek aktuální iterace“ a skočí na další opakování cyklu.

Ošetření chyb (`try`, `except`)

Výjimky a bezpečný kód

Když program narazí na neočekávanou situaci (např. dělení nulou, špatný vstup), vyvolá **výjimku** („Exception“). Musíme se naučit tyto chyby zachytávat.

- **Blok `try`:**
  - Kód, který může potenciálně selhat, je umístěn do bloku `try`.
- **Blok `except`:**
  - Pokud v bloku `try` dojde k chybě, řízení se přesune sem.
  - Můžete zachytit „konkrétní typ“ chyby (např. `ZeroDivisionError`) nebo jakoukoliv chybu (`except Exception as e:`).
- **Blok `finally`:**
  - Kód v tomto bloku se **vždy spustí**, bez ohledu na to, zda došlo k chybě, nebo ne.
    Ideální pro „úklid“ (např. zavření souborů).

Ošetření chyb (try, except)

Příklad ošetření vstupu:

```
try:
    cislo = int(input("Zadejte cislo: "))
    vysledek = 10 / cislo
except ValueError:
    print("Neni zadano cislo.")
except ZeroDivisionError:
    print("Nelze delit nulou.")
except Exception as e:
    print(f"Jina chyba: {e}")
finally:
    print("Pokus o vypocet dokoncen.")
```

Používání `try...except` je esenciální pro tvorbu „robustních“ (odolných) aplikací.

Organizace kódu

Psaní funkcí

Proč psát funkce?

Funkce pomáhají organizovat kód do **znovupoužitelných bloků**. Program pak funguje jako celek složený z „černých skříněk“.

- **Cíle funkcí:**
  - **Abstrakce:** Schovává složitost za jednoduchým názvem (např. `sin(x)`).
  - **Opakovaná použitelnost:** Kód se píše jen jednou, volá se mnohokrát.
  - **Čitelnost:** Usnadňuje pochopení toku programu.
- **Syntaxe a `def`:**
  - Funkce se definuje klíčovým slovem `def`, následuje název funkce a „závorky pro parametry“.
  - Jako vždy, tělo funkce je vymezeno **dvojtečkou** a **odsazením**.
- **Návratová hodnota (`return`):**
  - Příkaz `return` ukončí funkci a pošle „hodnotu zpět“ volajícímu kódu.
  - Pokud funkce nemá `return` (nebo ho má prázdný), automaticky vrací `None`.

Psaní funkcí

Příklad funkce:

```
1 def soucet_cisel(a, b): # a, b jsou parametry
2     vysledek = a + b
3     return vysledek     # Vraci hodnotu
4
5 # Volani funkce
6 x = soucet_cisel(5, 3)
7 print(x) # Vypise 8
```

Doporučení PEP 8: Název funkce by měl být malými písmeny oddělenými podtržítkem (`snake_case`).

Předávání parametrů

Způsoby volání funkcí

Argumenty mohou být předány buď podle **pozice** v definici funkce, nebo explicitně pomocí **klíčových slov**.

- **Poziční argumenty (`Positional`):**
  - Argumenty jsou přiřazeny parametrům v „pořadí, v jakém jsou uvedeny“ při volání.
  - *Příklad:* Pro `def info(jmeno, vek)` voláme: `info('Petr', 30)`.
- **Klíčové argumenty (`Keyword`):**
  - Explicitně uvedete název parametru, ke kterému argument patří. Pořadí pak „není důležité“.
  - *Příklad:* `info(vek=30, jmeno='Petr')`. Zlepšuje čitelnost kódu.
- **Výchozí hodnoty (`Default Arguments`):**
  - Umožňují, aby byly parametry volitelné. Při definici jim přiřadíte „výchozí hodnotu“.
  - Tyto argumenty se musí uvádět až **za pozičními argumenty**.

Předávání parametrů

Příklad výchozí hodnoty:

```
1 def tiskni_hlasku(text, hlasite=False):
2     if hlasite:
3         print(text.upper())
4     else:
5         print(text)
6
7 tiskni_hlasku("Ahoj") # hlasite=False
8 tiskni_hlasku("Ahoj", hlasite=True)
```

Důležité pravidlo: Všechny volitelné (defaultní) argumenty musí být definovány „za všemi povinnými“ (pozičními) argumenty.

Lokální proměnné (Rozsah platnosti)

LEGB pravidlo pro vyhledávání proměnných

Python má přísná pravidla, kde proměnnou hledat. Tento rozsah („scope“) je definován **LEGB** pravidlem.

Lokální proměnné (Rozsah platnosti)

- **Priorita vyhledávání:**
  1. **L**ocal (Lokální) — Uvnitř aktuální funkce.
  2. **E**nclosed (Vnořené) — Uvnitř obklopující funkce (pro vnořené funkce).
  3. **G**lobal (Globální) — Na nejvyšší úrovni modulu.
  4. **B**uilt-in (Vestavěné) — Předdefinované funkce a konstanty (`print`, `len`, `True`).

Lokální proměnné (Rozsah platnosti)

- **Lokální princip:**
  - Pokud proměnnou **vytvoříte přiřazením** uvnitř funkce, stává se automaticky **lokální**.
  - Po skončení funkce je tato lokální proměnná „zničena“ a nelze k ní přistupovat zvenčí.
- **Čtení vs. zápis:**
  - **Číst** globální proměnnou uvnitř funkce můžete.
  - **Měnit** (zapisovat) globální proměnnou uvnitř funkce můžete jen po použití klíčového slova `global`.

Lokální proměnné (Rozsah platnosti)

```
a = "Global"
```

Global Scope

```
def outer():
    a = "Enclosing"

    def inner():
        a = "Local"
        print("a - inner:", a)

    inner()
    print("a - outer:", a)

outer()
print("a - global:", a)
```

Local scope of inner()

Enclosing Scope

```
a - inner: Local
a - outer: Enclosing
a - global: Global
```

Lokální proměnné (Rozsah platnosti)

Příklad LEGB:

```
1 pi = 3.14159 # Globalni
2
3 def vypocet_plochy(r):
4     pi = 3.14  # LOKALNI (nove vytvorena)
5     plocha = pi * r**2
6     return plocha
7
8 vypocet_plochy(5)
9 print(pi) # Stale vypise 3.14159 (Globalni nebyla zmenena)
```

Doporučení: Snažte se „nepoužívat stejné názvy“ pro lokální a globální proměnné, aby nedošlo k záměně.

Dokumentace (Komentáře a Docstrings)

Dokumentace jako součást kódu

Kód by měl být sice „sám o sobě srozumitelný“, ale dokumentace zajišťuje rychlé pochopení komplexních logik.

- **Komentáře (#):**
  - Používá se symbol #. Měl by vysvětlovat **proč** se kód píše určitým způsobem, ne jen co dělá.
  - Ideální pro „krátká vysvětlení“ složitějších řádků kódu.
- **Docstrings (Dokumentační řetězce):**
  - Víceřádkový řetězec uzavřený v „trojitých uvozovkách“ (`"""..."""`).
  - Slouží k dokumentaci modulů, tříd, metod a **funkcí**.
  - Lze jej číst za běhu programu pomocí `funkce.__doc__` nebo v IDE.
- **Struktura Docstrings (Google/NumPy styl):**
  - Stručný popis funkce (1 řádek).
  - Sekce `Args`: Popis všech parametrů a jejich typů.
  - Sekce `Returns`: Popis návratové hodnoty a jejího typu.

Dokumentace (Komentáře a Docstrings)

- **Příklad Docstring:**

```python
def pricti(a: int, b: int) -> int:
    """Spocita soucet dvou cisel.

    Args:
        a: Prvni scitanec (cele cislo).
        b: Druhy scitanec (cele cislo).

    Returns:
        Soucet a a b jako cele cislo.
    """
    return a + b
```

Správně napsané Docstrings jsou základem pro automatické „generování API dokumentace“.

Objektově orientované programování

Instance a třídy (Plán vs. realizace)

Základní stavební kameny OOP

OOP umožňuje modelovat „reálné entity“ (např. auto, pes, účet) pomocí konceptu tříd a instancí.

- **Třída (`Class`):**
  - Je to **abstraktní plán**, šablona nebo návrh pro vytváření objektů.
  - Definuje „jaká data“ (atributy) a „jaké chování“ (metody) bude mít objekt tohoto typu.
  - *Příklad:* Třída `Auto` definuje, že auto má barvu a umí jet.
- **Instance / Objekt (`Instance`):**
  - Je to **konkrétní realizace** třídy.
  - Objekt má vlastní, „unikátní hodnoty“ pro definované atributy.
  - *Příklad:* Instance `moje_auto` je konkrétní červená Škoda.
- **Základní syntax a tvorba instance:**
  - Třída se definuje klíčovým slovem `class` (dle konvence s velkým počátečním písmenem).
  - Instance se vytváří jako volání funkce.

Instance a třídy — příklad

```
1 class Pes: # Plan: Trida
2     pass # Zatim prazdna
3
4 muj_pes = Pes()      # Realizace: Instance 1
5 tvuj_pes = Pes()     # Realizace: Instance 2
6
7 print(muj_pes) # Vypise: <__main__.Pes object at ...>
```

Každá instance „přiděluje paměť“ pro uložení svých vlastních, unikátních dat.

Datové atributy (Stav objektu)

Stav = Data

Datový atribut je proměnná, která drží data. Souhrn všech atributů objektu definuje jeho **aktuální stav**.

- **Přidávání atributů:**
  - V Pythonu lze atributy „dynamicky přidat“ k instanci i po jejím vytvoření pomocí tečkové notace (`instance.atribut = hodnota`).
  - Každá instance má „vlastní sadu atributů“, i když je vytvořena ze stejné třídy.
- **Čtení a změna stavu:**
  - K atributu se přistupuje pomocí `instance.nazev_atributu`.
  - Změna hodnoty atributu mění „stav daného objektu“.
- **Konvence pro vnitřní atributy:**
  - Atributy, které by měly být považovány za „interní“ (neměly by se měnit zvenčí), se často označují **jedním podtržítkem** (`_atribut`).

Datové atributy — příklad dynamického přidání

```python
class Auto:
    pass # Prazdna trida

auto1 = Auto()
auto2 = Auto()

# Dynamicke pridani stavu
auto1.barva = "cervena"
auto2.barva = "modra"

print(auto1.barva) # Vypise: cervena
print(auto2.barva) # Vypise: modra
```

Správný způsob inicializace atributů je uvnitř „konstruktoru“ (`__init__`), což probereme později.

Psaní metod (Chování objektu)

Metoda = Funkce + Stav

Metoda je funkce, která je definována uvnitř třídy a může „přistupovat a měnit stav“ (atributy) konkrétní instance.

- **Klíčový parametr `self`:**
  - Každá metoda instance musí mít jako svůj **první parametr** klíčové slovo `self`.
  - `self` je konvenční název pro „referenci“ na samotnou instanci (objekt), na které je metoda volána.
  - Slouží k přístupu k atributům objektu (např. `self.barva`, `self.jmeno`).
- **Volání metody:**
  - Metoda se volá pomocí „tečkové notace“ (`instance.nazev_metody(argumenty)`).
  - Python automaticky předá referenci na instanci jako první argument `self`.

Psaní metod — příklad

```
1  class Pes:
2      def __init__(self, jmeno): # Konstruktor
3          self.jmeno = jmeno     # Atribut
4
5      def stekej(self, hlasitost): # Metoda
6          # Pristup k atributu pres self
7          print(f"Baf, jmenuji se {self.jmeno}!")
8          print(f"Stekam na urovni {hlasitost}.")
9
10 muj_pes = Pes("Alik")
11 muj_pes.stekej(hlasitost=10)
12 # Vypise: Baf, jmenuji se Alik!
```

Bez parametru `self` *nemůže metoda „rozpoznat“, na které instanci má operovat.*

Základy polymorfismu (Mnoho tvarů)

Stejné rozhraní, různé implementace

Polymorfismus (z řečtiny: „mnohotvarost“) je schopnost různých objektů reagovat na **stejnou metodu nebo operátor** různým způsobem.

- **Princip:** Kód volá metodu na objektu, aniž by musel vědět, jakého je „přesně typu“. Postačí, že danou metodu implementuje.
- **Příklad 1: Operátor sčítání (`+`):**
  - Pro čísla provádí aritmetické sčítání (`1 + 1` vrací `2`).
  - Pro textové řetězce provádí (spojování) (`'A' + 'B'` vrací `'AB'`).
  - Operátor `+` je polymorfní.
- **Příklad 2: Vestavěná funkce `len()`:**
  - Pro seznam (`list`) vrací „počet prvků“.
  - Pro řetězec (`str`) vrací „počet znaků“.
  - Pro slovník (`dict`) vrací „počet párů klíč-hodnota“.
- **Role v OOP:** Polymorfismus zjednodušuje kód, protože nemusíte psát `if/elif` pro každý možný typ objektu.

Díky „Duck-typingu“ (viz později) je Python přirozeně polymorfní.

Jednoduchá dědičnost

Opakovaná použitelnost kódu

Dědičnost umožňuje jedné třídě („potomek“ / `Child`) převzít všechny atributy a metody od jiné třídy („rodič“ / `Parent`).

- **Syntaxe:** Třída dědí tak, že název rodičovské třídy je uveden „v závorce“ za názvem potomka při definici třídy.
- **Přepsání metody (`Overriding`):**
  - Třída potomku může **přepsat** (re-implementovat) metodu, která již existuje v rodičovské třídě.
  - Díky tomu může potomek reagovat na „stejné volání metody“ jiným, specifickým způsobem (viz polymorfismus).
- **Příkaz `super()`:**
  - Používá se pro „volání metody rodičovské třídy“ zevnitř třídy potomka.
    Bývá klíčový v konstruktorech (viz později).

Jednoduchá dědičnost — příklad

```
1  class Zvire:
2      def zvuk(self):
3          return "Neznamy zvuk"
4
5  class Pes(Zvire): # Dedi od Zvire
6      def zvuk(self): # Prepise metodu zvuk
7          return "Haf haf!"
8
9  pes = Pes()
10 zvire = Zvire()
11
12 print(pes.zvuk()) # Vypise: Haf haf!
13 print(zvire.zvuk()) # Vypise: Neznamy zvuk
```

Dědičnost je silný nástroj, ale je třeba jej používat s rozmyslem – vztah by měl být „je-typem“ (např. Pes je typem Zvířete).

Objektový model v Pythonu

Konstruktory (`__init__`)

Inicializace stavu objektu

Konstruktor je speciální metoda, která se volá **automaticky** hned po vytvoření nové instance. Slouží k nastavení „počátečního stavu“ objektu.

- **Metoda `__init__`:**
  - Je to standardní název pro konstruktor v Pythonu (tzv. „Dunder“ či „magická“ metoda).
  - Musí mít jako první parametr `self`.
  - Všechny parametry předané při vytváření instance (např. `Osoba("Jana", 30)`) jsou předány do této metody.
- **Role `__init__`:**
  - Umožňuje definovat **povinné atributy** již při vytváření objektu.
  - V těle metody přiřazujeme předané hodnoty k instančním atributům (např. `self.jmeno = jmeno`).
- **Alternativní metoda `__new__`:**
  - `__new__` je skutečná „továrna“ na objekt, volá se před `__init__`. V běžné praxi se téměř nemění.

Konstruktory (`__init__`) – příklad

```
1 class Osoba:
2     def __init__(self, jmeno, vek):
3         # Nastaveni stavu objektu
4         self.jmeno = jmeno
5         self.vek = vek
6
7 # Volani konstruktoru (nevolame __init__ primo!)
8 jana = Osoba("Jana", 30)
9 print(jana.jmeno) # Vypise: Jana
```

Pokud třída nedefinuje `__init__`, Python použije prázdný, „výchozí konstruktor“.

Speciální metody (Dunder metody)

Definování chování objektu

Speciální metody jsou metody začínající a končící „dvěma podtržítky“ (`__jmeno__`). Umožňují objektům interagovat s vestavěnými funkcemi a syntaxí.

- **Přetížení operátorů (Operator Overloading):**
  - Implementací `__add__` určíte, co se stane, když se objekty sečtou (`obj1 + obj2`).
  - Implementací `__len__` určíte, co vrátí funkce `len(obj)`.
- **Metoda `__str__`:**
  - Vrátí „člověku čitelnou“ textovou reprezentaci objektu (pro výstup uživateli, např. `print()`).
- **Metoda `__repr__`:**
  - Vrátí „jednoznačnou a technickou“ textovou reprezentaci objektu (pro vývojáře a ladění, ideálně by měla jít použít pro vytvoření objektu).

Speciální metody (Dunder metody) – příklad

Příklad `__str__` a `__repr__`:

```
1  class Bod:
2      def __init__(self, x, y):
3          self.x, self.y = x, y
4
5      def __str__(self):
6          return f"Bod je na pozici ({self.x}, {self.y})"
7
8      def __repr__(self):
9          return f"Bod({self.x}, {self.y})"
10
11 b = Bod(10, 20)
12 print(b) # Vola __str__: Bod je na pozici (10, 20)
```

Pokud není definováno `__str__`,

`print()` automaticky volá `__repr__`.

Dynamické vlastnosti (`@property`)

Metoda, která se tváří jako atribut

Vlastnost (`Property`) je způsob, jak volat metodu, aniž by kód musel používat závorky `()`. Umožňuje „řídit čtení, zápis a mazání“ atributů.

- **Dekorátor `@property`:**
  - Umístí se nad metodu, která by jinak byla getterem (čtenářem hodnoty).
  - K volání metody se přistupuje jako k „běžnému atributu“.
- **Účel (Getter):**
  - Vypočítává hodnotu za běhu (např. stáří z data narození).
  - Zajišťuje, že čtená hodnota je „vždy aktuální“.
- **Setter a validace:**
  - Pomocí dekorátorů `@nazev.setter` a `@nazev.deleter` můžete definovat, co se stane při „zápisu nebo mazání“ hodnoty atributu.
  - Zde probíhá klíčová **validace** (např. věk nesmí být záporný).

Dynamické vlastnosti (`@property`) – příklad

Příklad (Vypočítávaná vlastnost):

```
1  class Kruh:
2      def __init__(self, r):
3          self.polomer = r
4
5      @property # Cteni polomeru se tvari jako atribut
6      def plocha(self):
7          # Plocha se vypocita vzdy pri cteni
8          return 3.14159 * self.polomer**2
9
10 k = Kruh(r=5)
11 # Volano bez zavorek, jako atribut:
12 print(k.plocha)
```

Správné použití `@property` je základním nástrojem pro „zapouzdření“ (enkapsulaci) stavu objektu.

Duck-typing (Kachní typování)

Filozofie Pythonu: Nejde o typ, ale o chování

Duck-typing je princip definovaný slavnou frází: „Pokud to kváká jako kachna, chodí jako kachna a vypadá jako kachna, pak je to kachna.“

- **Praktický význam:**
  - V Pythonu se kód při práci s objekty **nedívá na jeho typ** (`class Pes` nebo `class Kocka`).
  - Dívá se pouze na to, zda objekt implementuje „požadovanou metodu“ nebo atribut.
  - Dva objekty z různých tříd jsou považovány za stejné, pokud mají **stejné rozhraní** (stejné názvy metod a stejné parametry).
- **Polymorfismus na steroidech:**
  - Umožňuje velmi flexibilní kód.
    Jakákoliv třída může být použita v cyklu `for`, pokud má správně implementovánu metodu `__iter__`.

Duck-typing – příklad

```
1  class Kacena:
2      def zvuk(self): return "Kvak!"
3
4  class Robot:
5      def zvuk(self): return "Pip!"
6
7  def dej_zvuk(zvire):
8      return zvire.zvuk() # Ocekavame metodu zvuk()
9
10 kacena = Kacena()
11 robot = Robot()
12
13 print(dej_zvuk(kacena)) # Funguje
14 print(dej_zvuk(robot))  # Funguje
15 # Funkci nezajima typ, jen existence metody!
```

Duck-typing je důvodem, proč je Python tak „flexibilní“ a proč se v něm snadno integrují různé komponenty.

Volání metod předka (`super()`)

Přístup k rodičovské třídě

Klíčové slovo `super()` poskytuje referenci na „rodičovskou třídu“ (případně na další třídu v dědičné hierarchii).

- **Primární použití:**
  - Slouží k **volání metody rodičovské třídy**, kterou potomek přepsal (overrode), aby se zajistilo, že se vykoná i původní logika.
  - Nejčastěji se používá k volání **konstruktoru** `__init__` rodičovské třídy.
- **Syntaxe `super()`:**
  - Ve většině případů se volá jednoduše: `super().metoda(argumenty)`.
  - Není potřeba explicitně předávat `self` nebo název rodičovské třídy.
- **Proč volat rodičovský konstruktor?**
  - Rodičovská třída je zodpovědná za inicializaci svých atributů. Zavoláním `super().__init__` zajistíte, že „atributy rodiče budou inicializovány“ ještě před inicializací atributů potomka.

Volání metod předka (`super()`) – příklad

```python
class Zvire:
    def __init__(self, jmeno):
        self.jmeno = jmeno

class Pes(Zvire):
    def __init__(self, jmeno, rasa):
        # 1. Zavolani konstruktoru rodice
        super().__init__(jmeno)
        # 2. Inicializace vlastnich atributu
        self.rasa = rasa

pes = Pes("Max", "labrador")
print(pes.jmeno) # Ma jmeno diky super()
```

Správné použití `super()` je klíčové pro správné fungování „složitějších hierarchií“ tříd a vícenásobné dědičnosti.

Standardní knihovna

Využívání hotových nástrojů (moduly a balíčky)

Modulární kód

Programování neznamená psát vše od nuly. Standardní knihovna poskytuje moduly pro „běžné úlohy“ (matematika, datum, sítě).

- **Modul:**
  - Je to **jeden soubor** s kódem Pythonu (`.py`). Obsahuje funkce, třídy a proměnné.
  - Slouží k logickému „seskupení souvisejících funkcí“.
- **Balíček (`Package`):**
  - Je to **složka** obsahující více modulů. Typicky má soubor `__init__.py` (v moderním Pythonu je volitelný).
- **Syntaxe `import`:**
  - `import modul`: Načte celý modul. K funkcím se přistupuje přes `modul.funkce()`.
  - `from modul import funkce`: Načte jen „konkrétní funkci“ nebo třídu. Lze volat přímo `funkce()`.

Využívání hotových nástrojů – příklad

Příklad modulu `math`:

```
1 import math
2 from math import pi # Import jen konstanty pi
3
4 # 1. Volani funkce z importovaneho modulu
5 print(math.sqrt(16)) # Vypise 4.0
6
7 # 2. Volani primo (protoze jsme pouzili 'from')
8 obvod = 2 * pi * 10
9 print(obvod)
```

Ukládání dat (Základy práce se soubory)

Zápis a čtení

Základní funkce `open()` slouží k otevření souboru. Je klíčové používat příkaz `with` pro „bezpečné zavření“ souboru.

- **Režimy otevření souboru:**
  - `'r'`: Čtení (výchozí).
  - `'w'`: Zápis. Přepíše obsah souboru.
  - `'a'`: Připojení (Append). Přidá data na konec.
- **Bezpečné použití `with`:**
  - Konstrukt `with open(...) as f:` automaticky zavře soubor, i když dojde k chybě. To je „nejlepší praxe“.
- **Zápis do souboru:**
  - Metoda `f.write(text)` zapíše text.

Ukládání dat – příklad

Příklad zápisu:

```
1 with open("data.txt", "w") as f:
2     f.write("Ahoj svete!")
3     f.write("\n")
4     f.write("Dalsi radek.")
5
6 # Soubor je zde jiz automaticky uzavren.
```

Pro čtení se používá `f.read()` (celý obsah) nebo `f.readlines()` (seznam řádků).

Ukládání dat (JSON, CSV)

Standardní formáty pro data

Pro ukládání složitějších struktur (seznamy, slovníky) se používají standardní formáty, které jsou „čitelné i pro jiné systémy“.

- **JSON (JavaScript Object Notation):**
  - Lehký formát, který přímo odpovídá „Python slovníkům a seznamům“.
  - Ideální pro **API komunikaci** a konfigurační soubory.
  - Modul `json`: Používá funkce `json.dump()` (zápis do souboru) a `json.load()` (načtení ze souboru).
- **CSV (Comma Separated Values):**
  - Formát pro „tabulková data“ (řádky a sloupce). Data jsou oddělena znakem (typicky čárkou nebo středníkem).
  - Ideální pro **export z databází** a tabulkových editorů.
  - Modul `csv`: Zajišťuje správné zacházení s uvozovkami a oddělovači.
- **Serializace objektů (Pickle):**
  - Modul `pickle` umí „ukládat celé Python objekty“, ale je určen pouze pro komunikaci mezi Python programy.

Práce s HTTP (API a knihovna `requests`)

Interakce s webovými službami

Většina moderních aplikací komunikuje s jinými službami přes web. To se děje pomocí protokolu **HTTP** a **API**.

- **API (Application Programming Interface):**
  - Sada pravidel, která definuje, jak mohou dvě softwarové komponenty „vzájemně komunikovat“ (např. jak získat data o počasí).
- **HTTP metody (Požadavky):**
  - `GET`: Používá se pro **čtení** dat (např. načtení webové stránky nebo dat z API).
  - `POST`: Používá se pro **odeslání** dat na server (např. odeslání formuláře nebo vytvoření nového zdroje).
- **Knihovna `requests`:**
  - Ačkoli není součástí Standardní knihovny, je „de facto standardem“ pro práci s HTTP požadavky v Pythonu.
  - Zjednodušuje odesílání požadavků a zpracování odpovědí (např. automatickou konverzí JSON dat).

Práce s HTTP (API a knihovna `requests`) – příklad

Příklad (GET požadavek):

```python
import requests

# Ziskani dat z verejneho API
odpoved = requests.get("https://jsonplaceholder.typicode.com/posts/1")

# Kontrola stavu (200 = OK)
if odpoved.status_code == 200:
    data = odpoved.json() # Konverze JSON na Python slovnik
    print(data['title'])
else:
    print("Chyba pri stahovani dat.")
```

Pro instalaci externích knihoven se používá nástroj „pip“ (např. `pip install requests`).

Diskuze a další zdroje

Kam dál? (Shrnutí a závěr)

Nejdůležitější koncepty pro začátek

Po tomto školení byste měli znát základy pro psaní „čistého, organizovaného a znovupoužitelného“ kódu v Pythonu.

- **Základy Pythonu:**
  - **Datové typy:** Rozumět rozdílu mezi `int`, `float`, `str` a `list`.
  - **Řízení toku:** Správné používání **odsazení**, `if`/`elif`/`else` a cyklů `for`/`while`.
  - **Organizace:** Umět psát a volat **funkce** (`def`, `return`).
- **Pokročilejší principy:**
  - **OOP:** Rozdíl mezi **třídou** a **instancí**, použití `self` a `__init__`.
  - **Reference:** Chápat, že `=` vytváří **reference**, nikoli kopie (u mutabilních typů).
  - **Nástroje:** Používání **virtuálních prostředí** a správce balíčků `pip`.

Kam dál? (Shrnutí a závěr)

- **Vaše další kroky:**
  - **Data Science:** Naučte se knihovny „`NumPy`“ a „`Pandas`“.
  - **Verzování:** Osvojte si `Git` a `GitHub` pro sledování změn.
  - **Web:** Začněte s mikroframeworkem „`Flask`“ nebo `Django`.

Doporučená dokumentace a zdroje

Nepřestávejte se učit!

Nejlepší zdroj informací je „oficiální dokumentace“ a komunitní projekty.

- **Oficiální dokumentace:**
  - `docs.python.org` – **Primární zdroj** pro reference a Standardní knihovnu.
  - `peps.python.org` – Vše o stylu a konvencích (zejména **PEP 8**).
- **Knihy a kurzy:**
  - Navazující kurz PYTH2 – Python - pokročilé programování.
  - Hledejte knihy zaměřené na Váš obor (Data Science, Web development).
- **Řešení problémů:**
  - `Stack Overflow` – Největší komunita pro programovací otázky a odpovědi.

Česká Python komunita

Zůstaňte ve spojení

Aktivní komunita je nejlepší cestou k získání pomoci a „zůstat v obraze“ s novinkami.

- **Konference a Srazy (Meetups):**
  - **PyCon CZ** – Česká národní Python konference.
  - **PyLadies CZ** – Kurzy a workshopy, skvělé pro začátečníky.
  - **Měsíční meetupy** – Konají se pravidelně ve větších městech (Praha, Brno, Ostrava).
- **Online zdroje:**
  - Skupiny na „LinkedIn“ nebo „Facebooku“ zaměřené na Python/Data Science.
  - České blogy a zpravodajské portály.

Hodnocení

Budeme velmi rádi, pokud nám poskytnete hodnocení tohoto kurzu.

VSTUP PRO STUDENTY

Kód kurzu PYTH1

Jméno

Příjmení

Odeslat

ictPRO

Děkuji za pozornost!