Python

Koncepce jazyka

Python je skriptovací jazyk. Byl navržen tak, aby byl jednoduchý, rychlý a šetřil programátorovi co nejvíce práce. V Pythonu se nepíší dlouhé programy ale spíše kratší skripty. Slouží k rychlému vývoji aplikací v dynamických podmínkách. Proto je ideální pro tvorbu webů, či GUI. Je interpretovaný a nezávislý na platformě. Jako skriptovací jazyk je poměrně high-level a slouží také jako lepidlo pro další komponenty. Python je řádově pomalejší než jazyk C, ale spousta jeho build-in knihoven je v C napsaná. Pokud tedy pracujeme s nimi (například práce s texty), dosahujeme často stejné rychlosti jako v C. Pokud ale píšeme větší program čistě v Pythonu, nevyhneme se pomalejším konceptům jazyka jako je například objektový návrh (kvůli reflexi), či GC overhead.

Pomalým částem jazyka se dá vyhnout pomocí například knihoven, kdy mohu napsat knihovnu v jazyce C, či C++ s kritickou částí programu a připojit ji ke svému Python skriptu. Zároveň Python podporuje i propojení s jinými jazyky jako Java, C/C++. Takto se dá Python bez problémů, či ztrátě na výkonu, využít i pro větší projekty. Navíc má automatickou správu paměti a konzistentní vzhled vzhledem k pravidlům syntaxe (odstazování). Také podporuje Duck Typing – namísto toho, abych kontroloval správný typ proměnné pouze kontroluji, že proměnná podporuje operace, co od ní požaduji.

Datové typy, proměnné

Python nemá explicitně určované typy. Typy se určují za překladu. To je navíc podpořeno tím, že Python je čistě objektový jazyk. Všechno v Pythonu je objekt. Tedy všechny proměnné jsou reference na objekty a tedy jen ukazatele do paměti. Operátor == porovnává hodnoty referencí do hloubky. Tedy, mám-li dvoudimenzionální pole, porovná každou pozici. Pro porovnání referencí slouží operátor *is*. Na operátor *is* se nedá ale vždy spolehnout. Python dělá občas optimalizace při tvorbě objektů. Pokud napíšeme literál *1*, Python vytvoří v paměti objekt, který má v sobě uloženou hodnotu 1 a vrátí nám referenci na něj. Pokud poté použiji literál *1* znovu, čekal bych, že se vytvoří nový objekt a tedy i nová reference. Python ale pro úsporu paměti nový objekt nevytvoří, ale vrátí referenci na předešlý objekt 1. Tedy výraz:

$$a = 1$$
$$b = 1$$
$$a\ is\ b\ \#True$$

Vrátí hodnotu *True*. Na to se ale také nemůžeme spolehnout. Python tyto optimalizace dělá jen pro daný rozsah hodnot a pro speciální případy. Tedy například:

$$a = 1500$$
$$b = 1500$$
$$a\ is\ b\ \#False$$
$$1500\ is\ 1500\ \#True$$

Python je case-sensitive. Základní typy proměnných:

- None (= False)
- Boolean
- Integer (neomezeně dlouhý) (0 = False, ostatní = True)
- Float
- String – nativní podpora Unicode, násobení, slicing,
- Object

String se chová jako n-tice Stringů (znaků). Podporuje spoustu funkcí pro usnadnění zpracování a výpisu textů.

Životnost proměnných je dána lokální funkcí, nadřazenou funkcí, globálním prostředím, či modulem. Jelikož nedeklarujeme proměnné, vyvstává problém odkazování k proměnným z vyšších bloků. K tomu slouží použití pojmenování například jménem modulů, či klíčová slova *global* a *nonlocal.*

Seznamy, n-tice, slovníky

Seznam je v podstatě list referencí na objekty. Může obsahovat jakékoliv typy objektů – ukládá si na ně pouze reference. Seznam je mutable (upravovatelný, měnitelný) objekt ke kterému se dá přistupovat pomocí hranatých závorek a předaného indexu. Podporuje řadu funkcí jako konkatenaci (+), délku seznamu (len), kopírování, apod. Pokud chci zkopírovat seznam a použiji *a = b*, dostanu pouze mělkou kopii objektu. Tedy ukazatele na objekty zůstanou stejné. Pokud chci udělat i kopii uložených objektů, je potřeba použít modul *copy* a operaci *deepcopy*. Seznamy podporují řadu dalších funkcí a umožňují tak přistupovat k seznamu jako k zásobníku, či frontě.

Množiny (sety) jsou podobné jako seznamy. Jsou upravovatelné a mohou obsahovat jakékoliv typy objektů. Každý objekt (jeho hodnota) se však v množině smí objevit jen jednou. Existuje také frozenset, který nelze měnit. Množiny mají oproti seznamům výhody například v rychlejším prohledávání, či v možnosti použití množinových operací (sjednocení, průnik, rozdíl, xor).

N-tice, nebo-li tuple, je immutable (neměnitelný) seznam. Prvky nelze přidávat, ani odebírat. Deklaruje se většinou pomocí kulatých závorek, nejsou ale povinné (speciální příklad je při vytvoření jednoprvkového tuplu). Díky neměnnosti je lze použít jako hashovatelné objekty.

Slovníky jsou asociativní seznamy, kde neindexujeme čísly (ikdyž také můžeme) ale hashovatelnými objekty, nebo-li klíči. Klíč může být cokoliv, co lze hashovat. Musí být tedy immutable, abychom si neměnili hash pod rukama. Přistupujeme k prvkům pomocí hranatic. Můžeme je číst i měnit, či mazat. Pokud použijeme klasické iterování pomocí for cyklu, iterujeme klíči. Speciální operátor *in* testuje přítomnost klíče ve slovníku.

Řetězce

Mm nevím co víc dodat... Spousta funkcí ... Slicing je zajímavý. (str[start:end:step] – může mít záporné hodnoty (počítá se od konce)). Může na něj být nahlíženo jako na kód pomocí funkce *exec*.

Soubory

open, write, close, read, readline, readlines; for line in f; with open(...) as f. V Pythonu funguje automatická správa paměti. Máme jistotu, že se soubor zavře, když ho přestaneme používat. Nevíme ale kdy. Proto bychom měli vždy volat na konci práce se souborem funkci *close*. Můžeme také používat keyword *with ... as ...*, který funguje podobně jako *using* v jazyce C# a zavře soubor za nás automaticky, jakmile opustíme jeho blok.

Řídící konstrukce

Podmínky:

- If
- Elif
- Else

Neexistuje switch. *while* a *for* cyklus mohou obsahovat i výraz *else*, který se zavolá, pokud upustíme cyklus klasickým způsobem (bez použití *break*). Python obsahuje řadu operátorů:

- +
- -
- *
- /
- //
- **
- %
- and
- or
- not
- >
- <
- <=
- >=
- ==
- !=
- in
- not in

Lze je spojovat dohromady v podmínkách (0 < a < 5).

Operátory lze přetěžovat v objektech pomocí speciální *Magic* funkcí (__*add*__). Nejdříve se hledá funkce v prvním operandu. Poté v druhém. Pokud se nenajde ani v jednom, dostaneme výjimku.

Definice funkcí

Funkce nedefinují návratovou hodnotu. Mají tvar:

$$\begin{aligned}&\text{def}\, funkce(arg1, arg2):\\ &\quad stmnt1\\ &\quad \ldots.\end{aligned}$$

Pokud chceme prázdnou funkci, použijeme klíčové slovo *pass*. U argumentů také neurčujeme typ, pouze specifikujeme jméno. Můžeme také specifikovat odzadu defaultní hodnotu. Také můžeme přijímat jako parametry funkcí proměnlivý počet prvků. To lze pomocí označení posledního argumentu hvězdičkou. Takovýto poslední argument bude poté obsahovat pole naplněné předanými parametry nad rámec klasického počtu parametrů. Pokud chceme dostat pojmenované proměnné v podobě slovníku, použijeme dvě hvězdičky. Pokud ve funkci nevrátíme hodnotu pomocí výrazu *return*, je návratová hodnota *None*.

Moduly

Moduly jsou jako knihovny. Obsahují zapouzřené soubory funkcí, či objektů. Jakýkoliv skript může být modulem, pokud splníme určité požadavky na adresářovou strukturu a připojíme *setup* skript, který pomůže ostatním programátorům náš modul nainstalovat. Jméno modulu je poté stejné jako název souboru, ve kterém se funkcionalita nachází. Custom moduly se ukládají do adresáře Lib. Základní moduly v Pythonu jsou:

- builtins
- copy
- os – práce s platformou
- sys – systémové funkce a konstanty
- time
- re - regexp
- string
- random
- ...

$$\begin{aligned}&import\ modul\\ &from\ modul\ import\ functionality\\ &from\ modul\ import\ *\end{aligned}$$

Příkaz *dir* vypíše obsah modulu. Příkaz *reload* znovu načte modul pro novější verzi funkcionalit.

OOP v Pythonu

Všechno je objekt a vše je reflexivní. Nové třídy, či jejich atributy, lze definovat, či změnit, kdykoliv za běhu. V Pythonu neexistují privátní proměnné ani funkce. Lze definovat proměnnou jako slabě privátní tak, že před ní dáme podtržítko. To je ale spíše jen doporučení pro programátory, že by k této proměnné neměli přistupovat. Poté lze definovat proměnnou jako silně privátní pomocí dvou podtržítek. Následek je takový, že se funkci, či proměnné, vygeneruje nové jméno ve tvaru _Třída__jméno a programátor má tedy opět možnost přistupovat k proměnné. Filosofie tohoto přístupu je ta, že jsme všichni dospěli programátoři a odpovídáme si za svoje problémy, které by mohli vzniknout nevhodnou manipulací. Každý objekt má skryté vlastnosti. K těmto vlastnostem lze přistupovat pomocí *Magic* funkcí. Například proměnné, které objekt v sobě drží, nemusí být deklarovány předem. Namísto toho má v sobě každý objekt slovník a ukládá si držené hodnoty do něj. Slovník ovšem zabírá místo a lze tuto funkcionalitu obejít přímým výpisem proměnných, který má objekt podporovat. Přístup k proměnným poté probíhá pomocí tečkové notace, či pomocí instance.__dict__[„jméno proměnné“].

Tento návrh ovšem nedovoluje dělat read-only atributy objektů, statické funkce, či funkce tříd, které Python také podporuje. K tomu slouží dekorátory funkcí. Pomocí dekorátoru můžeme určit, zda definovaná funkce je statická, patřící třídě, property, či setter.

V Pythonu neexistují Interfacy. Důvod je podporovaný Duck Typing, který podporuje používat cokoliv na cokoliv, dokud to dává smysl. Pokud chceme zajistit, aby některé objekty podporovaly jisté funkce (spíše z dokumentačního hlediska), můžeme nadefinovat abstraktní třídy s abstraktními metodami a odvodit naši třídu od této abstraktní.

Python také podporuje vícenásobnou dědičnost. To s sebou nese řadu problémů a proběhl poměrně rozsáhlý vývoj této problematiky v historii jazyka. Abychom se vyhnuli kolizím jmen, doporučuje se používat dvojpodtržítkové názvy proměnných. Jelikož je Python interpretovaný jazyk, pokud zavoláme například nějakou funkci, která je stejně definována na více místech, projde se objekt vzhledem k MRO a vybere se první definice, na kterou interpret narazí. MRO = Method resolution order. MRO se vytvoří pomocí linearizace závislostí. Vps to funguje tak, že se jde DFS zleva doprava a bere se vždy poslední výskyt. Pokud se překladač nemůže rozhodnout, zda má upřednostnit jednu třídu nebo druhou:

$$class\ A: pass$$
$$class\ B(A): pass$$
$$class\ C(A, B): pass$$

Zde chceme nejdříve projít třídu C, poté A a potom B a až po B třídu A? Překladač neví co si má myslet a jestli upřednostnit B před A. A vyhodí tedy výjimku. Pokud bychom nadefinovali:

$$class\ C(B, A): pass$$

Proběhne překlad v pořádku, protože je jasně vidět, že B má přednost před A.

Python má automatickou správu paměti pomoci GarbageCollectoru. Každý objekt si počítá reference a pokud dosáhne počet jeho referencí na hodnotu 0, GC zavolá jeho destruktor (metoda __*del*__) a objekt odstraní. Počet referencí se zvyšuje a snižuje automaticky u bloků, či kontejnerů nebo jej můžeme ručně vyvolat pomocí klíčového slova *del*. Může se stát, že objekty mají cyklické reference mezi sebou. GC má ve skutečnosti v sobě 3 vrstvy – takzvané generace. Pokud se překročí množství paměti v první generaci proběhne clean-up a GC automaticky odstraní i všechny objekty který mají navzájem mezi sebou cyklické reference, ale zvenčí už nejsou dostupné. Pokud nějaké objekty přežijí, jsou přesunuty do generace 2. Ta má taky threshold a funguje obdobně. Obejkty, které se dostanou do generace 3 většinou přežívají do konce běhu programu. Python tak zastává politiku, že většina objektů umírá mladá.

Výjimky

Výjimky jsou součástí snad každého moderního jazyka. Python má řadu výjimek na všechno možné. Lze je ošetřovat blokem *try: ... except ...: ... else: ... finally: .....* V *try* bloku je ošetřený kód. *except* očekává jako parametr výjimku kterou má odchytit. Pokud nedostane žádný parametr, odchytává všechny výjimky. V jeho bloku následuje kód, který se má provést, pokud výjimka nastane. Následuje nepovinný blok *else*, který se vykoná, pokud žádná výjimka nenastala. A nakonec můžeme přidat i blok *finally* který se provede v každém případě. Ten se hodí například na uvolnění paměti. V Pythonu funguje politika, že žádná výjimka by neměla být tichá. Tedy vše by se mělo hlásit, či logovat nebo nějak zpracovat. *except* blok může vyhodit další výjimky. Odchytávání výjimek probíhá zhruba stejně jako v ostatních jazycích. Prohledá se lokální blok, zda obsahuje *except* statement a pokud ano, tak zda do něj patří naše výjimka. Pokud ano, tak prohledávání skončí a provede se *except* blok. Pokud ne, zahodíme vršek zásobníku a pokračujeme v prohledávání o úroveň níže.