# IB002 — Pravidla a rady k programovacím úkolům

únor 2023

## Úvod a pravidla

Smyslem programovacích úkolů v tomto předmětu je procvičení základů algoritmizace, k čemuž stačí znalost základních konstrukcí zvoleného programovacího jazyka (Python). Protože některé výrazové prostředky jazyka Python obcházejí podstatu řešených problémů, omezujeme je v tomto předmětu následujícím způsobem:

**Použité knihovny.** Cílem implementační části předmětu je prakticky si vyzkoušet probírané algoritmy a datové struktury. Smyslem těchto úkolů tedy není hledání Pythonovských knihoven, které zadané problémy řeší. Každý úkol proto specifikuje standardní knihovny jazyka Python, které je při jeho řešení možné používat.

**Globální proměnné.** Jakékoli použití globálních proměnných je zakázáno. Za globální proměnné považujeme všechno, k čemu se pod stejným jménem dá přistoupit z libovolné funkce i mimo funkci a co je alespoň v jedné z funkcí nějak modifikováno. Je v pořádku používat globální *konstanty* (přiřadí se do nich jednou, mimo funkce, a dál se nemění). Parametry funkcí jsou lokální proměnné, stejně jako jakékoli proměnné, do nichž se uvnitř funkce přiřazuje a nejsou deklarovány pomocí `global`. Chcete-li zdůvodnění tohoto bodu, stačí zkusit na webu vyhledat „global variables are evil“. My jejich používání zakazujeme proto, že chceme, abyste uměli správně předávat hodnoty do funkcí (pomocí parametrů) a ven z nich (pomocí návratových hodnot), a to zejména při rekurzi.

Se zákazem globálních proměnných souvisí další omezení. Za prvé, v *implicitních* parametrech funkcí povolujeme pouze neměnné hodnoty (čísla, řetězce, pravdivostní hodnoty, ntice neměnných hodnot). Modifikovatelné hodnoty (seznamy apod.) v implicitních parametrech stejně většinou znamenají chybu; navíc se v podstatě jedná o globální stav (hodnota implicitního parametru se zachovává mezi jednotlivými voláními funkce).

Za druhé, nepovolujeme použití uzávěr (closures) – tedy „funkcí“ definovaných uvnitř jiných funkcí.[1] Důvody jsou stejné jako u zákazu globálních proměnných – zkušenosti ukazují, že mnozí studenti používají uzávěry zejména proto, aby tento zákaz obešli. Jako výše tedy platí – chcete-li si mezi funkcemi (nebo mezi voláními téže funkce) předávat nějaká data, používejte parametry a návratové hodnoty.

**Datové struktury.** Smyslem tohoto předmětu není pracovat s datovými strukturami, které už za vás implementoval někdo jiný. Naopak je naším cílem, abyste složeným datovým strukturám (stromy, hašovací tabulky, ...) porozuměli na úrovni jejich návrhu a implementace jejich operací. Ve většině úkolů se proto omezíme jen na některé vestavěné datové struktury Pythonu. Kromě těch

[1] Abychom byli zcela přesní, ne každá vnořená funkce v Pythonu je uzávěrou; uzávěry jsou jen ty, které někde ve svém těle používají lokální proměnné z vnější funkce. Na syntaktické úrovni je však tuto skutečnost obtížné rozlišit, proto se zákaz týká všech vnořených funkcí. Nepředstavuje to podstatný problém, protože každou (skutečnou) vnořenou funkci můžete kdykoli „vytáhnout ven“.

neměnných (čísla, řetězce, bool) budeme většinou používat pouze seznamy a ntice. K seznamům se přitom často budeme chovat jako k polím (ohledně složitosti práce se seznamy viz níže). Ostatní modifikovatelné datové struktury (množiny, slovníky, ...) budou povoleny jen tehdy, je-li to výslovně uvedeno v zadání.

**Výjimky.** V řešeních nepovolujeme používání výjimek (`try` / `raise`). Ze zkušeností víme, že studenti výjimky často zneužívají a snaží se jimi nahradit standardní tok řízení. Typickým příkladem je použití výjimky pro návrat z hluboko zanořené rekurzivní funkce, popřípadě vrácení hodnoty. Chceme, abyste se naučili o rekurzi uvažovat bez magického „vyskoč ven z libovolně zanořené rekurze".

**Ostatní.** Nepovolujeme použití funkcí `eval` / `exec` / `compile`. V minulých letech se občas studenti snažili ohnout některou z těchto funkcí až bizarním způsobem, např. jsme viděli řešení, kdy student potřeboval dojít na k-tý prvek zřetězeného seznamu a místo použití cyklu napsal něco jako `eval("node" + ".next" * k)`, což zajímavým způsobem selhalo. Tyto funkce jsou navíc potenciálně nebezpečné, takže byste se jim stejně měli pokud možno vyhýbat. V odevzdaných řešeních rovněž nepovolujeme použití příkazu `yield`. Tento příkaz jde vysoce nad rámec základních úvah nad návrhem a složitostí algoritmů, na které se v tomto předmětu chceme soustředit. Zcela mimo záběr předmětu je asynchronní programování (klíčové slovo `async`).

## Vyhodnocovací služba

Domácí úkoly se odevzdávají skrze odevzdávárny v ISu. Vyhodnocování se spouští každých cca 5 minut, přičemž proběhnou až dvě fáze testů. V první fázi, tzv. *syntax*, se zkontroluje, jestli je možné soubor s řešením načíst, jestli neobsahuje nějaké nepovolené syntaktické konstrukce apod. Rovněž se spouští nástroje `flake8` a `mypy`; jejich výstup je ovšem pouze informativní a neovlivňuje projití / selhání testu. Pokud test syntaxe selže, odevzdání se nepočítá. V opačném případě se spustí testy fáze *verity*, které testují korektnost a složitost vašeho řešení. Na tyto testy máte celkem pět pokusů, odevzdání nad tento limit už nebudou bodována. Pro účely bodového zisku se počítá nejlepší odevzdání. Úkoly smíte odevzdávat i po deadline; k takovýmto odevzdáním sice obdržíte výsledky testů, ale již bez bodového zisku.

Výsledky testů najdete v poznámkovém bloku. V případě, že neprošly testy korektnosti, je v tomto výstupu informace o tom, proč neprošly, a to včetně protipříkladu (příkladu vstupu, na kterém řešení dopadne chybně). Vyhodnocovací služba vám dá vždy jen jeden protipříklad pro každý test, který selhal. Tímto protipříkladem se můžete inspirovat a napsat si více vlastních testů, nebo jej použít k nasměrování úvah o korektnosti.

Toto se netýká testů složitosti – ty jen typicky zahlásí, že vaše řešení má pravděpodobně špatnou složitost. Protože testy složitosti jsou (už z principu) jen přibližné, může se vzácně stát, že je jejich výsledek falešně negativní. V takovém případě máte možnost se ozvat na diskusním fóru a nechat si řešení zkontrolovat člověkem. Používejte tuto možnost ale jen v případě, že jste si své řešení pořádně prošli a jste si jeho složitostí docela jisti.

## Složitost

V tomto předmětu nás kromě korektnosti často zajímá také asymptotická složitost algoritmů. Přitom je třeba si uvědomit, že i vestavěné funkce Pythonu a operace s vestavěnými datovými strukturami nejsou „zadarmo", ale samy o sobě mají rovněž nějakou časovou (a prostorovou) složitost. Toto uvědomění patří k jedním z cílů tohoto předmětu. Níže proto uvádíme stručný přehled typických operací a jejich složitostí.

Pythonovské seznamy (`list`) nejsou zřetězené seznamy, ale jsou implementovány pomocí tzv. dynamického pole (to je pole, které se může realokovat a zvětšovat). Složitosti operací se seznamy v Pythonu jsou následující:

Konstantní operace: $\Theta(1)$

- přístup k jednomu prvku `seznam[index]`,
- zjištění velikosti seznamu `len(seznam)`,
- mazání prvků z konce seznamu `seznam.pop()`,
- přidávání prvků na konec seznamu `seznam.append(prvek)`[2].

Lineární (k délce seznamu, není-li řečeno jinak) operace: $\Theta(n)$

- kopie seznamu `seznam[:]` nebo `seznam.copy()`,
- slice seznamu `seznam[from:to:step]` (lineární k počtu zkopírovaných prvků),
- hledání prvku v seznamu `prvek in seznam` nebo `seznam.index(prvek)`,
- jakékoli operace, které se v nejhorším případě nutně musí podívat na všechny prvky seznamu, jako jsou `min(seznam)`, `max(seznam)`, `sum(seznam)` apod.,
- smazání prvku ze seznamu `seznam.remove(prvek)`, `seznam.pop(index)` (v případě `pop`: lineární k počtu prvků od smazaného do konce),
- vložení prvku dovnitř seznamu `seznam.insert(index, prvek)` (lineární k počtu prvků od `index` do konce),
- iterování přes seznam `for prvek in seznam:`,
- zřetězení dvou seznamů `seznam1 + seznam2` (lineární k součtu délek obou seznamů),
- přiřetězení seznamu `seznam1 += seznam2` nebo `seznam1.extend(seznam2)` (lineární k délce druhého seznamu[1]).

Horší než lineární operace:

- řazení seznamu `sorted(seznam)` nebo `seznam.sort()` – $\Theta(n \log n)$.

Pythonovské množiny (`set`) a slovníky (`dict`) v domácích úkolech ani implementačních testech nepoužívejte, pokud to nebude explicitně povoleno v zadání. Jsou implementovány pomocí hashovacích tabulek. To mimochodem znamená, že složitost vkládání, vyhledávání a mazání prvků je sice v průměrném případě konstantní, ale v nejhorším případě lineární.

Pro zvídavé: Uvedené informace o složitosti Pythonovských konstrukcí se týkají implementace CPython, kterou používá vyhodnocovací služba a kterou velmi pravděpodobně používáte také. V tomto předmětu je považujeme za závazné. Podrobnější informace je možno najít na adrese: https://wiki.python.org/moin/TimeComplexity. Složitost, která nás zajímá, je zde uvedena jako „Amortized Worst Case“. Slovu „amortized“ zatím nemusíte věnovat pozornost[1].

## Rekurze v Pythonu

Ve zkratce: Pokud při vyhodnocení dostanete chybu `RecursionError`, pak váš kód buďto cyklí, nebo velmi neefektivním způsobem používá rekurzi.

Rekurze je při navrhování algoritmů velmi užitečný nástroj a řešení některých problémů bez rekurze může být zbytečně obtížné. Při implementaci v Pythonu je ale třeba dávat pozor na to, abychom se v rekurzi nezanořili příliš (to vede k chybě `RecursionError`).

[2]Tato složitost je pouze amortizovaná, ale tento detail si dovolíme v tomto předmětu ignorovat. Více se o amortizované složitosti dozvíte v navazujícím předmětu IV003.

Domácí úkoly a jejich testy jsou navržené tak, aby s rozumně použitou rekurzí nebyl problém. Za rozumně použitou rekurzi zde zejména považujeme rekurzivní procházení stromových datových struktur, rekurzi, která podstatným způsobem zmenšuje prohledávaný prostor (např. na polovinu při binárním vyhledávání v seznamu), a rekurzivní implementaci procházení do hloubky.

Pro pokročilejší: Pozor na to, že Python (schválně) neoptimalizuje tail rekurzi. Tail rekurzivní funkce je tedy v Pythonu lepší psát iterativně (i když v některých jiných jazycích by tento převod provedl kompilátor).

## Časté chyby a problémy

**Špatná složitost.** Nejčastějšími problémy v minulých letech byly používání řezů `seznam[od:do]` a hledání v seznamech `prvek in seznam`. Obě tyto operace mají lineární složitost.

**Vytváření objektů.** Objekty třídy `Node` vytváříme takto: `x = Node()`, ne takto: `x = Node`. Tím druhým příkazem jste třídu `Node` přiřadili do proměnné `x`, ale žádný nový objekt nevznikl.

**Chyby vyplývající z toho, že `1 == True`.** V Pythonu (podobně jako v některých jiných jazycích, ale zdaleka na všech) platí, že `True == 1` a `False == 0`. S tím souvisí řada chyb, pokud si z funkce chcete někdy vracet číslo a někdy pravdivostní hodnotu. Nejjednodušší řešení: Z funkce si můžete vracet více než jednu věc tak, že budete vracet dvojici (trojici, ntici) hodnot.

## Používání `is`/`==` v Pythonu

Zjednodušené pravidlo pro používání `is` a `==` v Pythonu zní: *Zapoměňte na `is` a `is not`, používejte `==` a `!=`. Chcete-li striktně dodržovat PEP8, používejte `is` (pouze) pro porovnávání s `None`.* Pokud vás zajímají detaily, čtěte dál; jinak můžete zbytek této části přeskočit.

Platí to, co bylo řečeno v IB111, tedy že `==` porovnává hodnoty, zatímco `is` porovnává odkazy. Nejjednodušeji je možné si to ilustrovat na Pythonovských seznamech:

```
a = [1, 2, 3]
b = [1, 2, 3]
c = b
```

V tuto chvíli platí `a == b`, `a == c`, `b == c`, `b is c`, ale `a is not b`, `a is not c`. Pro lepší pochopení můžete použít stránku http://pythontutor.com, kde jasně vidíte, že `b` a `c` se odkazují na tentýž seznam, zatímco `a` se odkazuje na jiný seznam.

Trochu matoucí může být, že `is` zdánlivě funguje i pro čísla. Nicméně si můžete snadno ověřit, že to funguje jen pro čísla dostatečně malá, máme-li tedy například `a = 10` a `b = 100`, pak (ve standardní implementaci CPython) platí `a * a is b`, ale `a * a * a is not a * b`. Důvodem je kešování malých čísel v Pythonu, což je implementační detail, na který se nemůžete spoléhat.

Zároveň platí, že vytvoříte-li si vlastní třídu (`class`), pak se na objektech této třídy `==` chová standardně stejně jako `is`. (Toto chování `==` se dá změnit napsáním speciální metody, ale to v tomto předmětu používat nebudeme.) Doporučení pro používání `==`/`is` tedy zní:

- Vždy pište `==`, pokud porovnáváte hodnoty, tj. čísla, řetězce, obsahy Pythonovských seznamů, slovníků, apod.
- Pište `==` nebo `is` (je to jedno), pokud porovnáváte objekty vlastních typů (definovaných pomocí `class`).
- Pokud chcete dodržovat PEP8, při porovnání s `None` používejte `is`.

Totéž samozřejmě platí i pro negované verze, tj. `!=`/`is not`.