Studentský matematicko-fyzikální časopis

M&M

Ročník XVII / Číslo 2

Milí kamarádi,

chtěli bychom vám všem poděkovat za všechna došlá řešení úloh i témat. Na vaše opravená řešení a výsledky se můžete těšit společně se třetím číslem našeho časopisu. V něm také už otiskneme nejzajímavější příspěvky k tématům. Zvláštní poděkování pak patří deseti odvážným, kteří se zapojili do tématu Mapování a účastnili se měření teploty.

V tomto čísle vám přinášíme zadání čtyř nových úloh a také druhou várku letošních témat. Nechybí ani pokračování seriálu o Pythonu, se kterým se letos budeme setkávat celý rok.

Někteří řešitelé dostanou toto číslo přímo na soustředění. Těm přejeme, ať se jim tam co nejvíce líbí. My se pro to pokusíme udělat maximum. A těm, na které tentokrát místo nezbylo, přejme, ať se jim daří při řešení. A snad se uvidíme při soustředění na jaře.

Pěkné chvíle strávené nejen při čtení našeho časopisu vám přejí

organizátoři

Termín odeslání druhé série: 29. 11. 2010

Zadání úloh

Úloha 2.1 – Ubrousky (4b)

Crrr... Jarek se líně převalil v posteli. „To už mám opravdu vstávat?“ Podíval se na hodiny, a zjistil, že mu za hodinu začíná škola. Nezbývalo než vstát a jít se nasnídat.

Během snídaně Jarka zaujaly dva čtvercové ubrousky ležící na stole, a tak si s nimi začal hrát. Dal je k sobě tak, aby se dotýkaly pouze jedním rohem, ten nazval A. Zbylé rohy jednoho ubrousku pak označil B_1, C_1 a D_1, druhého B_2, C_2 a D_2. A představoval si v duchu přímky B_1B_2, C_1C_2 a D_1D_2. Dokažte, že se mu tyto tři přímky vždy protnou v jednom bodě. A to bez ohledu na velikost a natočení ubrousků.

Úloha 2.2 – Lasery (4b)

Příjemně posilněn snídaní se Jarek vypravil na cestu do školy. Během cesty tramvají si pročítal své zápisky z fyziky. Zrovna se učili o laserech, a tak četl: Jednou z hlavních výhod laserů je, že jejich svazek může být velmi dobře rovnoběžný a rozchází se jen velmi málo. Díky tomu se také dá energie nasměrovat do velmi malého objemu. Žádný laserový paprsek ale nemůže být dokonale rovnoběžný.

Rozbíhavost se dá (složitě) spočítat z Maxwellových rovnic, nebo jednoduše z principu neurčitosti. Ten říká, že u žádné částice (třeba částice světla – fotonu) nemůžeme znát zároveň polohu i hybnost. Pro neurčitosti bude vždy platit

$\Delta x_i \Delta p_i \geq \frac{\hbar}{2}$, *kde* $\hbar = 1{,}054571 \cdot 10^{-34}\,\mathrm{m^2\,kg\,s^{-1}}$ *je redukovaná Planckova konstanta a* Δx_i *a* Δp_i *značí neurčitosti polohy a hybnosti ve směru* i*-té souřadné osy – princip neurčitosti platí pro každou osu zvlášť. A když je svazek úzký, známe jeho polohu v tom směru docela dobře, ale hybnost ...*

Jarka napadla při čtení následující otázka: Jak široký musí alespoň být laserový svazek (o dané vlnové délce), pokud urazil vzdálenost ze Země na Měsíc? Uměli byste mu na ni odpovědět?

Úloha 2.3 – Hledání v mlze (4b)

Když Jarek dorazil ke škole, nestačil se divit. Mimo školníka tam nepotkal vůbec nikoho. Podíval se pozorně na hodinky, a zjistil, že je teprve sedm. Přitom školu má až od osmi. To mu připadalo divné. Vybavoval si, že když ráno vstával, určitě na budíku bylo sedm hodin. Sedl si tedy do prázdné třídy a najednou se mu vybavil sen, který se mu v noci zdál.

Ve snu Jarek hledal v mlze ve čtverečkované síti o rozměrech $m \times n$ ztraceného kamaráda. Ten naopak hledal jeho. Mlha byla přitom tak velká, že bylo vždy vidět pouze na políčko, kde zrovna stál, a na sousední políčka. Hledali se tak, že vždy alespoň jeden z nich udělal krok do náhodně zvoleného směru (všechny možné pohyby měly stejnou pravděpodobnost) a rozhlédl se, jestli toho druhého nevidí.

Bylo by lepší, kdyby oba hledali zároveň, nebo kdyby jeden stál a druhý hledal? Jarek měl dost času, a tak na odpověď po chvíli přišel. Zkuste to i vy. Odhadněte nejdříve výsledek bez počítače. Pak napište program, který danou situaci nasimuluje, a udělejte pomocí něj statistiku, kterou nakonec nějak interpretujte. Závisí nějak výsledky na rozměrech čtverečkované sítě?

Úloha 2.4 – Karty (2b)

Konečně se začali trousit spolužáci. Mezi prvními přišel Bedřich, který si s sebou přinesl neobvyklý balíček karet a ukazoval všem trik, který vymyslel.

Bedřich má balíček 23 karet. Na každé z nich je na jedné straně kolečko a na druhé křížek. Přitom 14 karet je otočených nahoru křížkem, zbývajících 9 kolečkem. Bedřich tvrdí, že když mu zavážou oči a karty zamíchají (během míchání se nesmí žádná karta obrátit), dokáže je rozdělit na dvě hromádky tak, aby na každé z nich bylo stejně karet otočených křížkem. [1] Dokázali byste to i vy?

Najednou se z ničeho nic ozvalo dlouhé crrr... a začala hodina.

Jarek ale pozor nedával. Vzpomněl si na další sen. Byl na nějakém místě, kde byl porouchaný čas. A bylo ho potřeba opravit. Vybavoval si, že s pomocí nějakých mnichů se o to pokoušel... Ale už si nedokázal vzpomenout, jestli se mu to opravdu podařilo.

[1] Během rozdělování už Bedřich může karty libovolně otáčet.

Zadání témat

V tomto čísle vám přinášíme zadání dvou nových témat. Jednoho matematického a jednoho fyzikálního. Po jejich pročtení už budete znát všechna témata, která jsme pro vás letos připravili. Je tedy pravý čas si vybrat, jestli se vám některé líbí, a případně k němu poslat co nejzajímavější článek. Nehodnotí se jenom, k jak složitým nebo přesným výsledkům dojdete, ale i vaše kreativita a to, jakou otázku si dokážete v rámci tématu zformulovat. Nebojte se poslat cokoli, co vás během přemýšlení o tématu napadne a bude vám připadat třeba jen trochu zajímavé.

Téma 3 – Ztracen v lese

Ach né! Unesli Rikiho! Tomu se sice podařilo v nestřežený okamžik zlotřilým únoscům uprchnout, ale teď je úplně sám kdesi uprostřed hlubokého lesa! Potřebuje se co nejrychleji dostat ven, aby mohl o všem neprodleně informovat organizátory M&M! Pomůžete mu?

Věřím, že jste okamžitě vyhrkli „No samozřejmě!“, ale já vás uklidním. Tento příběh se nestal (zatím...). Jedná se pouze o nácvik krizových variant, do kterých se může Váš oblíbený lišák dostat. Riki nemá v hlavě jasno, jak by se měl v popsané situaci zachovat, tak spoléhá na vás :-).

Řekněme, že Riki zná celkovou *plochu* S lesa, ve kterém se nachází. Jak se má pohybovat, aby co nejdříve narazil na kraj lesa, i když bude mít tu největší smůlu? Jinak řečeno: Poskytněte Rikimu návod, aby měl jistotu, že nejpozději po čase t už bude z lesa určitě venku. Jak malé umíte zařídit t (v závislosti na S)? Pokud uznáte za vhodné, můžete o lese předpokládat, že „nemá díry“, nebo dokonce že je konvexní.

Zabývat se můžete i případem, kdy Riki zná nejen plochu, ale i *tvar* lesa. Například může vědět, že les je kruhový/čtvercový/hvězdicovitý/...Ovlivní to nějak jeho taktiku? Nezapomeňte, že stále neví, kde přesně se v lese nachází, ví jen to, jak les vypadá na mapě.

Riki je bystrý lišák, možná by se byl schopen i v lese zorientovat! Pokud by znal tvar lesa a navíc by věděl, kde je sever, jak rychle by uměl vybloudit? Zkuste úlohu řešit pro různé tvary lesa – začněte od těch jednoduchých.

A co kdyby Riki věděl, že přesně 1 km nějakým směrem od něj vede přímo skrz les cesta? Riki ví, jak se na ní dostat po projití (v nejhorším případě) $1 + 2 \cdot \pi$ kilometrů. Existuje i lepší strategie? Popište ji! Umíte se dostat pod sedm kilometrů? Pod šest a tři čtvrtě?

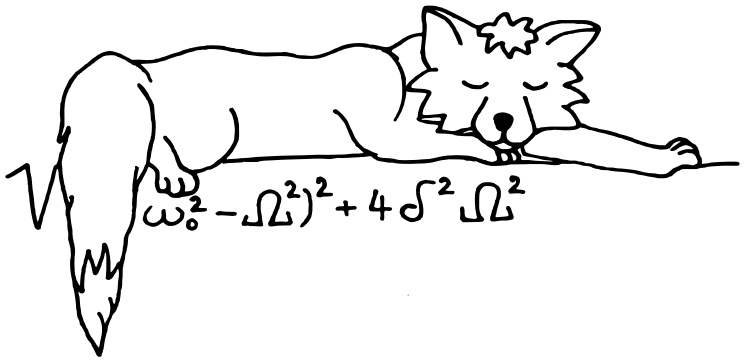
O žádném řešení nemusíte rozhodovat, zda je optimální (ačkoliv samozřejmě můžete). Důležité je Rikimu poradit aspoň nějak :-).

Téma 4 – Sjezdovka

Nebylo by to krásné rozjet se jen tak po sjezdovce, a nemuset pak už přibrždovat, protože ideální rychlost už se bude držet sama? Zkuste takovou sjezdovku navrhnout. Tj. jaký tvar by měla mít plocha, aby po ní mohlo těleso s konstantním třením f sjíždět dolů konstantní rychlostí, kterou už dosáhlo dřív.

Obloučky tedy zanedbáme. A jak by měla vypadat první část sjezdovky, na níž chcete té cílové rychlosti dosáhnot, a plynule přejít v pohodlnou jízdu konstantní rychlostí (tj. zrychlení musí být spojitá funkce polohy)? Jak by naopak měl vypadat ideální konec sjezdovky k pohodlnému zabrždění? Jaký profil je podle vás „nejpohodlnější“? Jaký musí být profil sjezdovky, abyste si naopak užili trochu zábavy, třeba aby se vaše rychlost měnila jako konst $+ \sin x$, nebo cokoli jiného vás napadne.

A co opačný pohled... Co s vaší rychlostí udělá malý skokánek uprostřed sjezdovky? Co dalšího byste mohli na sjezdovku umístit? Můžete řešit jak teoreticky, tak experimentálně či modelováním.



Seriál o Pythonu (II. díl)

V pokračování seriálu o Pythonu se prokoušeme skoro celým zbytkem syntaxe Pythonu, seznámíme se s některými pokročilejšími vlastnostmi (ty jsou označené *) a namíříme k jeho užitečným aplikacím. Na konci vás jako obvykle čeká několik úloh.

Praktické rady ohledně Pythonu pro Windows i Linux najdete stále na adrese `http://mam.mff.cuni.cz/python/`. Přibyly tam odkazy na online vyhodnocovače Pythonu, kde si můžete své jednoduché výtvory vyzkoušet bez instalace Pythonu.

Pokud byste měli se zprovozněním Pythonu stále problém, rádi vám zkusíme poradit, než se dostaneme k hraní si s konkrétními knihovnami.

Dodatky k minulému dílu

V minulém díle jsme zapomněli zmínit dvě podstatné vlastnosti `and` a `or` – tyto logické operátory vyhodnotí svůj první operand a pokud je z jeho hodnoty jasný výsledek (nepravda u `and`, pravda u `or`), rovnou vrátí jeho hodnotu. Tomuto se říká zkrácené vyhodnocování a většinou jen šetří čas, ale má význam v případech, kdy druhý parametr je funkce s dalšími efekty.

Druhou vlastností je to, že pokud vyjde první argument u `and` pravda (resp. nepravda u `or`), pak tyto vrátí rovnou svůj druhý argument. Pokud je to `bool`,

pak je to stejné jako kdyby se provedlo opravdové vyhodnocení konjunkce či disjunkce, ale pro jiné typy se toho dá dobře využít:

```
print(seznam or 'seznam je prazdny')
```

vypíše hodnotu `seznam` pokud je neprázdný, jinak zadaný řetězec.

Rovněž jsme vám zamlčeli bitové operátory `&`, `|`, `~` a `^`, které provádějí bitovou konjunkci, disjunkci, negaci a nonekvivalenci (xor).

Často se vám budou hodit též zkratkové operátory `+=`, `-=`, `*=`, `/=`, `&=`, `|=`, `^=` a poněkud obskurní `**=`. Například `a+=b` je totéž co `a=a+b`, ale je to mnohem kratší, pokud je místo `a` složitější výraz.

K `print` existuje i jednoduchá vstupní funkce `raw_input(vyzva)`, která vypíše (nepovinnou) výzvu, přečte jeden řádek a vrátí ho jako řetězec. Funkce `input` navíc vyhodnotí vstup jako kus Pythonu a je lepší se jí vyhnout. V Pythonu 3.0 se už `input` chová jako `raw_input` ve 2.x.

Další typy

Slovník (typ `dict`) je množina dvojic klíčů a hodnot, ale umí rychle vyhledávat hodnoty podle klíčů. Zapisuje se `d={k1:h1, k2:h2, k3:h3}`. Klíče mohou být libovolných neměnných typů (tedy skoro cokoliv kromě seznamu a slovníku), hodnoty mohou být cokoliv.
Hodnotu příslušející ke klíči získáte `d[klic]`, `klic in d` zjistí zda má slovník daný klíč, `dict.keys()` vrátí seznam klíčů, prvek se maže pomocí `d.pop(klic)`, vkládá a mění se `d[klic]=x`. Slovníky umí i sjednocení a další operace.

k-tice (typ `tuple`) se zapisují jako seznamy s kulatými závorkami místo hranatých, prázdná „0-tice“ je `()`, 1-tice se zapisuje `(42,)` a ne `(42)` (to je jen číslo). Indexují se stejně jako seznamy, ale na rozdíl od seznamů jsou neměnné a dají se používat jako klíče slovníků.

Množiny (typ `set`) jsou jako neuspořádané seznamy. Na rozdíl od seznamů mají ale rychlou operaci hledání. Vytváří se `m=set(seznam)`, test náležení se provádí pomocí `prvek in m`, `m.add(prvek)` vkládá, `m.remove(prvek)` maže. Může obsahovat stejné typy jako slovník klíče. Zvládá i rozdíly a sjednocení množin, ale na to určitě přijdete sami.

Bloky a scope

Tomu, co jste viděli jako tělo definované funkce v minulém díle, se říká *blok*. Blok je skupina stejně odsazených příkazů, může obsahovat prázdné řádky či komentáře a končí před prvním *méně* odsazeným řádkem, který není komentář. Bloky můžete vnořovat, vnořené bloky mají větší odsazení než blok vnější a neukončují ho.

Bloky se používají pro strukturování programu (viz následující sekce), ale vnořený blok můžete začít i jen tak.

Scope je označení pro oblast programu, kde je proměnná vidět pod konkrétním jménem. Každý soubor (modul), definice funkce nebo třídy (viz později) má svou sadu *lokálních proměnných* a tyto nejsou nikdy přímo vidět vně, zato jsou vidět ve vnořených definicích funkcí a tříd.

Lokální proměnná vznikne při přiřazení jménu, které ještě není v lokálních proměnných. Hranice bloků ve scope (na rozdíl od Pascalu a C) nehrají žádnou roli – proměnná vytvořená uvnitř `for` bude vidět i za koncem jeho bloku. Lokálním proměnným pro celý soubor (modul), tedy těm nastaveným na nejvyšší úrovni, se říká *globální*.

Co se ale stane, když máme globální proměnnou např. `i` a použijeme ji v definici funkce? Pokud její hodnotu pouze čteme (či voláme její metody), budeme pracovat s globální proměnnou `i`. Pokud do `i` přiřadíme, pak se vytvoří nová lokální a ke globálnímu `i` se už přímo nedostaneme. Toto umožňuje vytvořit si ve funkci proměnné bez ohledu na to, jak vypadá okolní kód.

Python poskytuje jednu záchrannou síť – pokud byste ve funkci `i` nejprve četli a pak do `i` přiřadili a vytvořili tím lokální, dojde k chybě. Pokud byste opravdu toto chtěli, použijte jiné jméno lokální proměnné.

Pro případ, že chcete do globální proměnné přiřadit, použijte příkaz `global i`, který vám globální `i` zpřístupní k přiřazování.

Větvení, podmínky a cykly

Program v Pythonu samozřejmě nemusí být jen pevnou posloupností příkazů. Pro větvení programu se používá `if`:

```
if podmínka :
        odsazený blok příkazů pro pravdu
else:
        odsazený blok příkazů pro nepravdu
```

Podmínka je typu `bool`, ale jak už jsme psali, lze použít skoro libovolný typ a pak jde o nenulovost, neprázdnost, ... Po vyhodnocení podmínky se provede buď první, nebo druhý blok příkazů. Část s `else:` je nepovinná. Nezapomeňte na dvojtečku!

Cyklus přes hodnoty seznamu se provádí pomocí `for`:

```
for řídící_proměnná in seznam :
        odsazený blok příkazů
```

Odsazený blok příkazů se provede jednou za každou položku seznamu, přičemž bude tzv. řídící proměnná nastavena postupně na první prvek seznamu, pak druhý, atd. Pokud byla v řídící proměnné předtím nějaká data, budou přepsána.

Python nemá for-cyklus jako Pascal či C. Pokud chcete, aby proměnná nabývala celočíselných hodnot z nějakého intervalu, použijte funkci `range(start, stop, krok)`. Ta vrací seznam čísel začínajících `start`, končící **před** `stop` a s krokem `krok`. Pokud vynecháte `krok`, bude 1. Pokud vynecháte i `start`, bude 0. Pokud ji použijete ve `for`-cyklu, nemusíte se bát, že by vám seznam velikosti 10^9 sežral paměť – v takovém případě se generuje postupně.[2]

[2] Takových „falešných líných polí“ v Pythonu potkáte více, dokonce si je můžete sami naprogramovat, ale to už je na seriál moc velká specialita.

Pokud neznáte předem počet opakování, použijte `while`:

`while` *podmínka* `:`
    odsazený blok příkazů

Zde se blok opakuje dokud je splněna podmínka. Ta se testuje vždy znovu před začátkem cyklu (i prvního).

V rámci cyklů `for` a `while` se občas hodí pomocné příkazy `break` a `continue`. `break` okamžitě přeruší cyklus a dál pokračuje za ním, `continue` přeruší aktuální cyklus a pokračuje následujícím cyklem od začátku bloku příkazů (u `for` s dalším prvkem seznamu).

Zběsilejší funkce*

Python v sobě má i několik užitečných myšlenek z funkcionálního programování. Jak jsme již psali, funkce, stejně jako cokoliv jiného, můžete přiřazovat a předávat jako parametry. Např. `def aplikuj(f,param): return f(param)` je zcela legitimní funkce.

Existuje několik užitečných funkcí pro práci se seznamy. `map(f, seznam)` aplikuje funkci `f` na každý prvek `seznam`u a vrátí seznam výsledků. Např. `map(math.sqrt, range(100))` vrátí seznam odmocnin čísel 0 až 99. Funkci `map` můžete předat i více seznamů, pak je funkce dostává najednou:

`map(f,a,b,c) == [f(a[0],b[0],c[0]), f(a[1],b[1],c[1]), ...]`

Funkce `filter(f, seznam)` vrátí seznam těch prvků, pro které `f(prvek)` vrátí pravdu. Např. `filter(bool, range(-5, 6))` vrátí jen nenulové prvky.

Pokud se vám nechce definovat např. funkce zvětšující o jedničku, můžete použít *lambda funkci*. `lambda x: x+1` je funkce jednoho parametru, který vrací zvětšený o jedničku. Toto můžete použít kdekoliv, kde byste použili jméno funkce, takže např. `(lambda x:x+1)(5)==6`, `f=lambda x:x+1; f(5)==6`, nebo `map(lambda x:x+1, [5,1])==[6,2]`. Lambda funkce více parametrů se píše jako `lambda x,y,z:x*(y+z)`. Tělo lambda funkce může být pouze jeden výraz.

Zpracování seznamů*

Ukázalo se, že se `map`, `filter` a `lambda` hodí často, ale hůře se čtou, proto převzal Python od Haskellu tzv. *list comprehensions*. Ty se zapisují jako

[*výraz* `for` *proměnná* `in` *seznam* [`if` *podminka*]].

Část `if` *podmínka* je nepovinná. Ze seznamu se vezmou všechny hodnoty, postupně se přiřadí do proměnné, a pokud splní podmínku, dosadí se do výrazu. Výsledek je seznam výsledků.

Můžete například napsat

```
[x**2 for x in range(10) if x%2==0],
```

což vyjde `[0, 4, 16, 36, 64]` a je kratší a intuitivnější zápis pro

```
map(lambda x:x**2, filter(lambda x:(x%2==0, range(10)))).
```

Volitelný počet parametrů a pojmenované parametry*

Některé funkce mají volitelné parametry s implicitní hodnotou (`range`, `open`), jiné zas mají neomezený počet parametrů (`str.join`, `map`). Jak to funguje a jak toho dosáhnout i u vlastních funkcí?

Při deklaraci funkce můžete posledním několika parametrům nastavit *implicitní hodnotu*. Pokud některé při voláni neuvedete, použije se tato hodnota:

```
def prevod_jednotek(x, z, do='m', presnost=6): ...
```

Při volání je možné napsat několik parametrů ve správném pořadí (jsou určené pozicí) a zbylé dopsat v libovolném pořadí s uvedením jejich jména, např.: `prevod_jednotek(4.2, presnost=1, z='ft')`. Tomuto mechanizmu se říká pojmenované parametry. Pokud chcete, aby vaše funkce přijímala i další pojmenované parametry, doplňte do seznamu parametrů např. `**parametry`, `parametry` pak bude slovník s pojmenovanými parametry navíc.

Pokud chcete vytvořit funkci, která bude mít libovolný počet parametrů, na konec seznamu parametrů dopište `*dalsi`, `dalsi` pak bude k-tice extra parametrů. Toto vše lze s jistými omezeními kombinovat – všechny pojmenované parametry musí být až za pozičními.

```
def f(x=0, y=0, *dalsi, **pojmen): return (x,y,dalsi,pojmen)
    f(1, 2, 3, 4, t=True) == (1, 2, (3, 4), {'t': True})
       f(humpf=0, y=42) == (0, 42, (), {'humpf': 0})
```

Pokud chcete nějaké funkci předat hotový slovník pojmenovaných parametrů či seznam extra pozičních parametrů, můžete do seznamu parametrů přidat `*dalsi` a `**pojmen`, např.:

```
f(1, 2, 3, *[4,5], t=0, **{'mam':'super'})
      == (1, 2, (3, 4, 5), {'mam': 'super', 't': 0})
```

Práce se soubory

Soubor na disku otevřete funkcí `f=open(cesta, mod)`. Cesta k souboru se zadává jako normální řetězec. Nezapomeňte, že zpětné lomítko se v Pythoních

řetězcích zapisuje `'\\'`. Mód je několik znaků (řetězec), kterým určíte, zda soubor otevíráte ke čtení (`'r'`), pro čtení i zápis (`'r+'`), pro připisování na konec (`'a'`) nebo pro přepsání (`'w'`, které vymaže obsah souboru pokud existoval). Přidané `'t'` označuje, že je soubor textový, `'b'` binární, ale rozdíl je jen při převádění konců řádků mezi Windows a Linuxem (to vás ale nemusí většinou trápit).

Soubor má typ `file` a můžete do něj psát pomocí `f.write(data)`. Pokud píšete do textového souboru, nezapomeňte na `'\n'` pro konec řádku. `f.read(pocet)` čte daný počet znaků (nebo celý soubor pokud počet vynecháte). Pokud čtete na konci souboru, `read` vrátí `""`. Soubor po sobě musíte zavřít funkcí `f.close()`.

Funkce `f.readline()` načte jeden řádek včetně znaku konce řádku (byl-li tam), `f.readlines()` vrátí všechny řádky souboru jako seznam řetězců. Funkce `readlines` je chytrá podobně jako `range` a nenačte ve skutečnosti soubor celý najednou, ale načítá řádky postupně jak je používáte.

Můžete použít soubor jako seznam, pak se bude chovat jako seznam řádků stejně jako u `readlines`. Hodí se to například ve `for`-smyčce, kdy chcete něco udělat pro každý řádek souboru.

Pokud pracujete s textovými soubory v kódování jiném než ASCII, Python ho může transparentně překládat za vás při čtení i zápisu. Tuto funkci jen naznačíme, podrobný výklad je nad možnosti seriálu.

V podstatě stačí použít místo `open` funkci `codecs.open`, která navíc přijímá parametr kódování `encoding`. Tedy například můžete použít

```
f=codecs.open(cesta, mod, encoding='utf8').
```

Užitečné moduly

Krom modulu `math`, který jsme viděli minule, zde zmíníme ještě několika dalších užitečných modulů.

Modul `sys` obsahuje spoustu informací o interpretu Pythonu. Zajímavá je funkce `sys.exit()`, která okamžitě ukončí program, seznam parametrů zadaných z příkazové řádky `sys.argv` (`sys.argv[0]` je vždy jméno programu tak, jak byl spuštěn) a standardní vstup, výstup a chybový výstup jako Pythoní soubory (`sys.stdin`, `sys.stdout` a `sys.stderr`, který se od `stdout` liší jen v detailech). `print` a `raw_input` používají právě tyto soubory.

Moduly `time`, `datetime` a `calendar` zacházejí s datem a časem.

`datetime.date`, `datetime.time` a `datetime.datetime` jsou datové typy pro uchovávání data a času. Ten se pak dá vypisovat, porovnávat, odčítat a tak podobně. Aktuální datum a čas zjistíte pomocí `datetime.date.today()` a `datetime.datetime.now()`. Všechny typy mají atributy jako `date.day` a `time.hour`.

Modul `time` poskytuje přímější přístup k datu a času přes `time.time()`, hlavně ale pak funkci `time.sleep(sec)`, která počká daný počet sekund (může být i reálné číslo). `time.clock()` vám poví, kolik sekund procesorového času už váš program spotřeboval.

Modul `calendar` vám poví, který den v týdnu bylo předposlední úterý posledního přechodného roku. Jeho prozkoumávání necháme na vás.

Modul `random` obsahuje (pseudo)náhodný generátor. `random.random()` vrací náhodné číslo z intervalu $< 0, 1)$, `random.randint(a,b)` vrací celé číslo z $\{a, \ldots b\}$, `random.choice(sekvence)` náhodně vybere z k-tice či seznamu. Modul obsahuje i výběrové funkce pro další zajímavá rozložení, např. Gaussovo.

Z dalších modulů, které by vás mohly na začátku zajímat, jmenujme třeba matematické `cmath` pro práci s komplexními čísly a `fractions` pro práci s racionálními čísly (zlomky), nebo moduly s efektivními datovými strukturami[3] `array` a `collections`.

K prozkoumávání modulů se ideálně hodí `ipython` popsaný na našem webu. Často mají shrnující nápovědu i samotné moduly (`math.__doc__` nebo `"math?"` v `ipython`-u). Standardní moduly mají vynikající nápovědu na stránkách Pythonu. Vše je dokonce anglicky![4]

Vlastní moduly*

Nakonec ještě uvedeme, jak se vytvářejí vlastní moduly. To je až příliš snadné – vytvoříte soubor `mujmodul.py` plný definic a příkazů a umístíte ho někde, kde ho Python najde, například do aktuálního adresáře. Při prvním `import mujmodul` se celý soubor spustí jako skript (může např. obsahovat inicializační, příkazy) a nadefinované funkce a globální proměnné budou k dispozici s prefixem `mujmodul`, např.: `mujmodul.thando('Tralala!')`.

Při dalších importech (pokud by třeba `mujmodul` používalo více použitých modulů) už se inicializace znovu neprovádí.

Python moduly hledá v adresářích v `sys.path`. Pokud chcete svůj modul umístit na místo, kde ho Python *vždy* najde, vyberte si některý z těchto adresářů nebo tam adresář se svými moduly můžete vždy před importem přidat:

```
import sys
sys.path.append('C:\\Tomasovo\\Python')
import mujmodul
```

„Správné“ vytváření vlastních modulů je na delší povídání. Na o něco kratší povídání je pak vytváření podmodulů a omezování toho, co je z modulu po exportu vidět.

Úloha 1.5 – Seriál o Pythonu (II. díl) (7b)

Všechny úlohy jsou řešitelné s tím, co jsme v seriálu zatím popsali, ale můžete používat i další standardní prostředky. Snažte se o co nejelegantnější řešení. Nějaké body samozřejmě získáte i pokud nám pošlete jen částečné řešení, za nadstandardní řešení můžete dostat bonus.

[3] Např. velké seznamy reálných čísel typu `list` jsou mnohem náročnější na paměť než totéž v C či Pascalu.

[4] Místo mnohem běžnějších jazyků, jako je čínština.

I.a (1b): Popište, co dělá následující program:

```
def fix(f,*p):return f(f,*p)
print(fix(lambda f,x:(x<1 and 1) or x*f(f, x-1), 10))
```

I.b (2b): Pomocí `fix` a lambda-funkcí (bez definic vlastních funkcí) spočtěte efektivně n-tý prvek Fibonacciho posloupnosti. Vyhněte se exponenciálnímu počtu volání.

II (2b): Seznamy v Pythonu mohou obsahovat cokoliv, včetně sebe sama. Např.:

```
a=['A','a']; b=[a,'B',a]; c=[b,'C',a,'c',b]; a.insert(1,c)
```

Vytvořte funkci, která takové zacyklené seznamy vypisuje. Nesmíte při tom použít žádnou Pythoní funkci, která by to udělala za vás, jako `str` či `repr`. Výstup by měl být co nejčitelnější, můžete tedy například vyzdvihnout hloubku zanoření odsazením. Bylo by navíc dobré si seznamy např. očíslovat a nějak naznačit, který už vypsaný seznam je ten vnořený.

III (2b): Napište program, který *transponuje* textový soubor jako matici, tedy prohodí znaky mezi pozicemi (řádek x, sloupec y) a (sloupec x, řádek y). Kde je to potřeba, vyplňte prázdná místa mezerami. Všechny mezery na koncích řádků odstraňte (i kdyby se jednalo o ty z původního souboru).

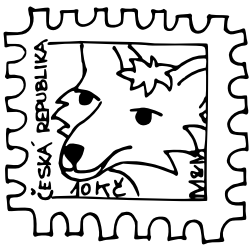
Jména vstupního a výstupního souboru ideálně přečtěte z příkazové řádky (ze `sys.argv`). Pokud jste machři, můžete ošetřit i kódování souboru jako UTF-8 a jiné. Pokud nevíte, o co jde, nemusíte se o to starat (předpokládejte 1 bajt = 1 znak), jen to možná nebude fungovat na soubory s češtinou.

Tom

Adresa redakce:

M&M, OVVP, UK MFF
Ke Karlovu 3
121 16 Praha 2

Telefon: +420 221 911 235
E-mail: `MaM@atrey.karlin.mff.cuni.cz`
WWW: `http://mam.mff.cuni.cz`



Časopis M&M je zastřešen Oddělením pro vnější vztahy a propagaci Univerzity Karlovy, Matematicko-fyzikální fakulty a vydáván za podpory středočeské pobočky Jednoty českých matematiků a fyziků.