nenapíšete na dané téma několik programů, jedná se pravděpodobně skutečně pouze o pocit.
Abyste nebyli ve zbytečném pokušení kontroly obcházet, nedovolenou spolupráci budeme relativně přísně trestat. Za každý prohřešek Vám budou strženy 4 tvrdé a 4 měkké body (celkem ekvivalent jednoho klasifikačního stupně) a opsaná kontrola bude vynulována (vždy jako celek – celý domácí úkol nebo celá týdenní příprava, bez ohledu na to, jak velká část byla opsaná). Opíšete-li tedy přípravu do cvičení, a v dotčeném bloku máte pouze 3 úspěšně odevzdané přípravy, znamená to automatické hodnocení X. Totéž platí o domácím úkolu, nemáte-li alespoň dva body z vnitrosemestrálky. Opisování u zkoušky nebo vnitrosemestrálky bude penalizováno 8 tvrdými a 8 měkkými body (dva stupně) a vynulováním bodového zisku (v případě zkoušky to samozřejmě znamená hodnocení F v dotčeném termínu).
To, jestli jste příklad řešili společně, nebo jej někdo vyřešil samostatně, a poté poskytl své řešení někomu dalšímu, není pro účely kontroly opisování důležité. Všechny „verze" řešení odvozené ze společného základu budou penalizovány stejně. Taktéž zveřejnění řešení budeme chápat jako pokus o podvod, a budeme jej trestat, bez ohledu na to, jestli někdo stejné řešení odevzdá, nebo nikoliv.
Podotýkáme ještě, že kontrola opisování nespadá do desetidenní lhůty pro hodnocení průběžných kontrol. Budeme se samozřejmě snažit opisování kontrolovat co nejdříve, ale odevzdáte-li opsaný příklad, můžete být penalizováni kdykoliv (tedy i dodatečně, až do konce zkouškového období).

Část B: Želví grafika

Tato kapitola je náplní cvičení v prvním týdnu semestru, a jejím smyslem je seznámit Vás s organizací cvičení, se studijními materiály (tedy zejména touto sbírkou), s programovacím prostředím PyCharm a se základními elementy syntaxe jazyka Python. Zároveň Vám připomeneme (nebo ukážeme) základy algoritmizace pomocí tzv. želví grafiky. Jednotlivé kapitoly sbírky obsahují 4 druhy příkladů: první sada jsou tzv. ukázky – jedná se o komentované řešení nějakého problému, které Vám ilustruje použití konstrukcí, které v daném týdnu budeme ve cvičení potřebovat. Tyto ukázky nenahrazují přednášku, přestože s ní mají určitý překryv – slouží k jejímu doplnění delšími, komentovanými ukázkami použití, které můžete využít jako inspiraci při řešení příkladů z ostatních částí. Tato kapitola obsahuje pět ukázek:

1. `square.py` – kreslení čtverce přímo a pomocí cyklu
2. `hexagon.py` – použití podprogramu
3. `boxes.py` – podprogramy s parametry
4. `isosceles.py` – použití proměnné
5. `flower.py` – podmíněné provádění kódu

Druhá část kapitoly obsahuje řešené úlohy: dostanete zadání, které byste si měli jako rozcvičku vyřešit sami. Ke každé takové úloze je ale přiloženo vzorové řešení (v souboru `uloha.sol.py`), do kterého můžete nahlédnout – buď v případě, že se zaseknete, nebo ke srovnání se svým vlastním řešením.

1. `pentagon.py` – pravidelný pětiúhelník
2. `right.py` – pravoúhlý trojúhelník (parametrický)

Třetí (hlavní) blok tvoří neřešené příklady. S výjimkou této „nulté" kapitoly musíte tři z těchto příkladů vyřešit a odevzdat předem – dva z nich nejpozději první čtvrtek a jeden v sobotu po odpovídající přednášce (tzn. v týdnu předcházejícímu tomu, ve kterém se tato kapitola bude řešit ve cvičení[3]). Následující cvičení si můžete vyřešit dopředu také – tento týden je to ale výjimečně dobrovolné.

1. `polygon.py` – pravidelný n-úhelník
2. `fence.py` – plot pomocí cyklu
3. `heartbeat.py` – EKG pomocí cyklu
4. `spiral.py` – spirála
5. `diamond.py` – kreslení stylizovaného diamantu
6. `circle.py` – kreslení kružnic
7. `trapezoid.py` – rovnoramenný lichoběžník

Poslední část tvoří tzv. „hvězdičkové" úlohy. Jedná se o těžší úlohy, které byste nicméně měli být schopni řešit. Jednu z nich můžete v dalších týdnech odevzdat místo dvou „běžných" příkladů z předchozího bloku.

1. `pizza.py` – kreslení kruhové výseče
2. `koch.py` – Kochova vločka
3. `hilbert.py` – Hilbertova křivka

[3] Na online konzultacích.

Část B.1: Ukázky

B.1.1 **[square]** Smyslem první ukázky je předvést základní „příkazy" (procedury – tento pojem si přesněji vysvětlíme v dalších ukázkách) pro kreslení obrázků. Tyto procedury ovládají „želvu", která se pohybuje po plátně a kreslí přitom čáru. Procedura `forward` želvě poručí, aby se posunula o danou vzdálenost vpřed (a nakreslila u toho úsečku ze své původní polohy do své nové polohy). Procedury `left` a `right` nic nekreslí, pouze želvou otočí o daný úhel (zadaný v stupních) doleva, resp. doprava.
Dovolíme-li želvě vracet se „po vlastních stopách", stačí nám tyto 3 procedury na vykreslení libovolného spojitého obrazce. Pro začátek zkusíme nakreslit čtverec:

```
def square():
```

Čtverec lze nakreslit jednoduše jako 4 navazující úsečky stejné délky, přičemž každé dvě po sobě jdoucí svírají pravý úhel.

```
    forward(100)
    right(90)
    forward(100)
    right(90)
    forward(100)
    right(90)
    forward(100)
```

Předchozí definice `square` nás ale příliš neuspokojuje: k čemu máme počítač, když jsme museli každý krok explicitně popsat? Zejména je na první pohled vidět, že příkazy se opakují. Jistě by bylo dobré, abychom mohli počítači sdělit, že má nějakou akci provést 4×, místo abychom ji zapsali 4× pod sebe – to je v podstatě základní mechanismus, kterým nám počítač šetří práci.

```
def square_loop():
```

Základní formou tzv. cyklu (angl. loop) je příkaz „proveď akci `n` krát", který se v Pythonu zapisuje jako `for i in range(n)` – v našem případě bude `n = 4`:

```
    for i in range(4):
```

Následuje tzv. tělo cyklu, které je tvořeno (odsazeným) seznamem příkazů, které se budou opakovat.

```
        forward(100)
        right(90)
```

Pozorný čtenář si jistě všiml, že definice `square` a `square_loop` nejsou

zcela ekvivalentní: ta druhá obsahuje jedno použití procedury `right` navíc. Pro tuto chvíli je nám to jedno, protože není-li volání `right` následováno žádným použitím `forward`, nebude mít na výsledný obrázek dopad. Nicméně obecně toto neplatí a je potřeba si na podobné okrajové případy dávat pozor.
Následuje definice `main`, smyslem které je demonstrovat funkčnost dříve definovaných `square` a `square_loop`.

```
def main():  # demo
```

Nejprve necháme želvu vykreslit čtverec „naivním“ způsobem, bez použití cyklu (první z definic výše).

```
    square()
```

Dále želvu požádáme, aby se přesunula na jiné místo plátna, aniž by nakreslila čáru: tento kus kódu pro nás není příliš podstatný, jeho smyslem je pouze vykreslit dva obrázky na jedno plátno, abychom je mohli lehce srovnat.

```
    penup()
    setheading(0)
    forward(200)
    pendown()
```

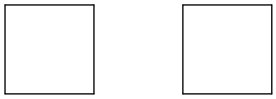
Na novém místě plátna požádáme želvu o vykreslení čtverce druhou metodou (cyklem). Jestli jsme se nespletli, budou oba obrázky identické.

```
    square_loop()
```

Příkazem (procedurou) `done` želvě oznámíme, že máme vše vykresleno a program má vyčkat na ukončení uživatelem.

```
    done()
```

Výstup testů by měl vypadat přibližně takto:



B.1.2 [hexagon] V této ukázce sestrojíme „segmentovaný“ šestiúhelník složením z 6 pootočených rovnostranných trojúhelníků. Smyslem je ukázat, že část výpočtu si můžeme pojmenovat, a poté ji s výhodou využít jako stavební kámen něčeho složitějšího. V tomto případě se vybízí pojmenovat si právě vykreslení onoho rovnostranného trojúhelníku:

```
def triangle():
    for i in range(3):
        forward(100)
        left(120)
```

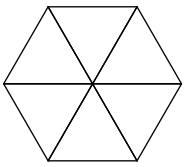
To, co jsme právě udělali, se obecně jmenuje definice podprogramu. V tomto případě se jedná konkrétně o proceduru, totiž podprogram, kterého smyslem je provést nějaké akce (vedlejší efekty). V našem případě je tedy `triangle` procedurou pro vykreslení rovnostranného trojúhelníku. Naše nově definovaná procedura `triangle` je k nerozeznání od těch zabudovaných (knihovních), které známe z předchozí ukázky: `left`, `forward` a pod.

```
def hexagon():
    for i in range(6):
        triangle()
        left(360 / 6)
```

Teď již víme, že `main` je také procedura, tedy podprogram, kterého smyslem je vykonat posloupnost akcí (typicky dalších procedur).

```
def main():  # demo
    hexagon()
    done()
```

Výstup testů by měl vypadat přibližně takto:



B.1.3 [boxes] Procedura, kterou jsme definovali v předchozí ukázce, totiž taková, která provede fixní (pokaždé stejnou) posloupnost akcí, není příliš zajímavá. Naštěstí lze procedury parametrizovat. Podobně jako u knihovních procedur `forward` nebo `left` si můžeme sami definovat proceduru, které pak při použití předáme nějaké číslo (obecněji hodnotu). Konkrétní předaná hodnota pak bude mít vliv na chování takto definované procedury.
Zde si definujeme proceduru `square`, která se nápadně podobá na proceduru `square_loop` z první ukázky, s jedním rozdílem: délka strany již není pevně daná, ale je nyní proceduře předána jako parametr.

```
def square(size):
    for i in range(4):
        left(90)
        forward(size)
```

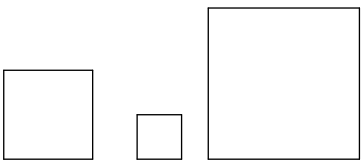
Takto definovanou proceduru můžeme opět používat zcela analogicky k těm zabudovaným – nyní včetně předání parametru, který diktuje, jak velký čtverec si přejeme vykreslit.

```
def main():  # demo
    square(100)
```

Připomínáme, že následující tři příkazy slouží pouze k přesunu želvy na jinou pozici na plátně.

```
    penup()
    forward(100)
    pendown()

    square(50)

    penup()
    forward(200)
    pendown()

    square(170)

    done()
```

Výstup testů by měl vypadat přibližně takto:



B.1.4 [isosceles] Doposud jsme se nezabývali otázkou, odkud pochází definice procedur `left`, `forward` apod. Protože ale v této ukázce budeme potřebovat další knihovní podprogramy, je čas zmínit existenci příkazu `import`. Tím oznámíme interpretu Pythonu, že hodláme využívat podprogramy z externích modulů. V tomto kurzu se omezíme na moduly ze standardní knihovny, totiž takové, které jsou dodávány s každým interpretem jazyka Python.
Pro úplnost dodáme, že modul je sbírka vzájemně souvisejících, znovupoužitelných podprogramů (a případně i složitějších artefaktů, kterými se ale nebudeme v tomto kurzu příliš zabývat).

```
from turtle import forward, left, right, penup, pendown, done,
setheading
```

Krom procedur pro práci se želvou budeme v tomto příkladu potřebovat několik matematických funkcí:

- odmocninu, realizovanou podprogramem `sqrt`,
- převod stupňů na radiány, realizovaný podprogramem `radians`,
- goniometrickou funkci tangens, realizovanou podprogramem `tan`.

Podprogramům, které realizují výpočet nějaké hodnoty na základě hodnot svých parametrů, budeme říkat čisté funkce, z důvodu jejich podobnosti s funkcemi z matematiky. Podprogramy `sqrt`, `radians` a `tan` jsou tedy v tomto smyslu (čistými) funkcemi.

```
from math import sqrt, radians, tan
```

Krom použití funkcí si v této ukázce předvedeme také použití proměnných. V nejjednodušším smyslu je proměnná pouze pojmenováním nějaké vypočtené hodnoty – takto je budeme nyní používat. Složitější případy použití proměnných (zejména přiřazení) si necháme na příští týden.
Obrázek, který budeme kreslit, je rovnoramenný trojúhelník, zadaný délkou základny a úhlem (v stupních) mezi základnou a ramenem.

```
def isosceles(base, angle):
```

První hodnotou, kterou si pojmenujeme (uložíme do proměnné) bude polovina základny: rovnoramenný trojúhelník si totiž pomyslně rozdělíme na dva stejné (pouze zrcadlově otočené) pravoúhlé trojúhelníky s odvěsnami `height` (výška) a `half_base` (polovina základny).

```
    half_base = base / 2
```

Protože trojúhelník máme zadaný základnou a přilehlým úhlem, potřebujeme vypočítat délku ramene. To se nejsnadněji provede pomocí už zmíněného pomyslného pravoúhlého trojúhelníku. Na výpočet délky ramene použijeme Pythagorovu větu, ale nejprve potřebujeme znát výšku (druhou z odvěsen pomyslného trojúhelníku). Protože máme úhel zadaný v stupních, musíme ho nejprve převést na radiány, pak jednoduše použijeme funkci tangens, která udává poměr odvěsen v pravoúhlém trojúhelníku (protilehlá k přilehlé). Výšku získáme jednoduchou úpravou definičního výrazu.

```
    height = half_base * tan(radians(angle))
```

Konečně můžeme přistoupit k výpočtu délky ramene:

```
    side = sqrt(height ** 2 + half_base ** 2)
```

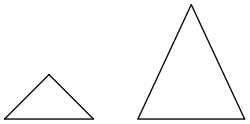
Nyní máme vše, co k vykreslení potřebujeme. Nejprve nakreslíme základnu, poté želvu otočíme o vedlejší úhel k `angle` (tak, aby úhel sevřený základnou a ramenem, které budeme kreslit jako další byl `angle`). Vrcholový úhel je daný vztahem `180 - 2 * angle`, nicméně opět potřebujeme želvu otočit o příslušný vedlejší úhel (hodnotu `2 * angle` dostaneme opět jednoduchou úpravou). Nakonec vykreslíme druhé rameno, a želva se tím vrátí do výchozí pozice.

```
    forward(base)
    left(180 - angle)
    forward(side)
    left(2 * angle)
    forward(side)
```

Abychom ověřili, že program pracuje správně, vykreslíme si dva různé trojúhelníky.

```
def main():  # demo
    isosceles(100, 45)

    penup()
    setheading(0)
    forward(150)
    pendown()

    isosceles(120, 65)
    done()
```

Výstup testů by měl vypadat přibližně takto:



B.1.5 [`flower`] Tato (pro tento týden poslední) ukázka předvede použití příkazu `if`, který slouží k podmíněnému vykonání nějaké akce. Nejprve si ale definujeme pomocnou proceduru `triangle`, která by nás již neměla překvapit: vykresluje tupoúhlý, rovnoramenný trojúhelník, který bude sloužit jako lupínek květiny. Důležitou vlastností této procedury je, že zachová pozici i orientaci želvy.

```
def triangle():
    forward(100)
    right(165)
    forward(52)
    right(30)
    forward(52)
    right(165)
```

Vykreslíme nyní stylizovanou květinu, které ale chybí některé lupínky: konkrétně ty, jejichž pořadové číslo je dělitelné třemi nebo pěti. Květinu budeme vykreslovat v cyklu, jak už je zvykem. To, čím se tato ukázka liší od předchozích, je, že samotná posloupnost akcí, které se v těle cyklu provedou, se bude iteraci od iterace lišit. Parametr nám zadává původní počet lupínků (kolik by jich bylo, kdyby žádný nechyběl).

```
def flower(petals):
    for i in range(petals):
```

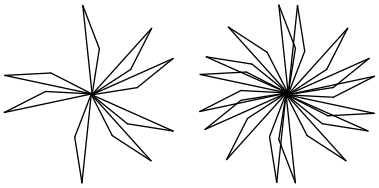
Podmínku zapisujeme klíčovým slovem `if`, následovaným výrazem, který se vyhodnotí na booleovskou hodnotu (tzn. `True` nebo `False`) a za dvojtečkou seznamem příkazů, které se provedou pouze, vyhodnotil-li se předaný výraz na hodnotu `True` (tzn. byl pravdivý).
V tomto případě se dotazujeme, zda má indexová proměnná `i` nenulový zbytek po dělení jak číslem 3 tak číslem 5: znamená to, že ani jeden z nich není dělitelem. Všimněte si, že podmínku pro „chybějící“ lupínek jsme negovali: lupínek vykreslíme, je-li tato (negovaná) podmínka splněna, tedy bude chybět v případě, že byla splněna původní podmínka ze zadání.
Budete-li srovnávat zápis programu s obrázkem, který kreslí, je důležité si uvědomit, že první index je 0 (a je tedy dělitelný například i 3), nultý lupínek bude tedy chybět. Kdyby nechyběl, „ukazoval“ by směrem doprava.

```
        if i % 3 != 0 and i % 5 != 0:
            triangle()
```

Bez ohledu na to, zda jsme lupínek vykreslili nebo nikoliv, musíme se pootočit k vykreslení (nebo přeskočení) dalšího lupínku: tento příkaz se provede v každé iteraci. Protože se pootočíme doprava, lupínky vykreslujeme ve směru hodinových ručiček (přičemž nultý by ukazoval 3 hodiny) – ve stejném směru, kterým ukazují vrcholy trojúhelníků, které lupínky reprezentují.

```
        right(360 / petals)

def main():  # demo
    flower(15)

    penup()
    setheading(0)
    forward(220)
    pendown()

    flower(30)
    done()
```

Výstup testů by měl vypadat přibližně takto:

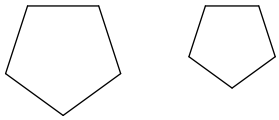


Část B.2: Rozcvička

B.2.1 [pentagon] Implementujte proceduru pentagon, která vykreslí pravidelný pětiúhelník se stranami o délce side pixelů.

```
def pentagon(side):
    pass
```

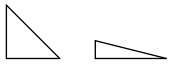
Výstup testů by měl vypadat přibližně takto:



B.2.2 [right] Implementujte proceduru right_triangle, která vykreslí pravoúhlý trojúhelník s odvěsnami o délkách side_a a side_b. Můžou se vám hodit funkce z modulu math.

```
def right_triangle(side_a, side_b):
    pass
```

Výstup testů by měl vypadat přibližně takto:



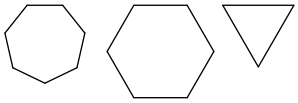
Část B.3: Příklady

B.3.1 [polygon] Zobecněte řešení z příkladu pentagon tak, abyste byli schopni vykreslit libovolný pravidelný mnohoúhelník. Toto obecné řešení implementujte jako proceduru polygon s parametry:

- sides je počet stran kresleného mnohoúhelníku, a
- length je délka každé z nich.

```
def polygon(sides, length):
    pass
```

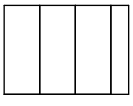
Výstup testů by měl vypadat přibližně takto:



B.3.2 [fence] Napište program, který nakreslí „plot“ o délce length pixelů, složený z prken (obdélníků) o šířce plank_width a výšce plank_height. Přesahuje-li poslední prkno požadovanou délku plotu, ořežte jej tak, aby měl plot přesně délku length. Zamyslete se nad rozdělením vykreslování do několika samostatných procedur. Při kreslení se vám také může hodit while cyklus.

```
def fence(length, plank_width, plank_height):
    pass
```

Výstup testů by měl vypadat přibližně takto:



B.3.3 [heartbeat] Implementujte proceduru heartbeat, která vykreslí stylizovanou křivku EKG. Parametr iterations udává počet tepů, které procedura vykreslí. Zbylé parametry zadávají amplitudu základního úderu a periodu slabšího úderu. Slabší úder má poloviční amplitudu. Například při periodě 3 bude mít sníženou amplitudu každý třetí úder, počínaje prvním.

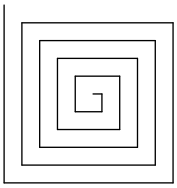
```
def heartbeat(amplitude, period, iterations):
    pass
```

Výstup testů by měl vypadat přibližně takto:

B.3.4 [spiral] Implementujte proceduru spiral, která vykreslí čtyřhrannou spirálu s rounds otočeními (počet otočení říká, kolik hran musíme překročit, vydáme-li se ze středu spirály po přímce libovolným směrem). Parametr step pak udává počet pixelů, o který se hrany postupně prodlužují.

```
def spiral(rounds, step):
    pass
```

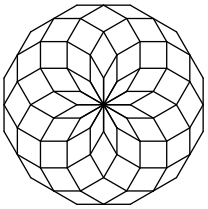
Výstup testů by měl vypadat přibližně takto:



B.3.5 [diamond] Napište proceduru pro vykreslení stylizovaného diamantu. Tento se skládá z mnohoúhelníků, které jsou vůči sobě natočené o vhodně zvolený malý úhel (takový, aby byl výsledný obrazec pravidelný). Každý mnohoúhelník má sides stran o délce length pixelů.

```
def diamond(sides, length):
    pass
```

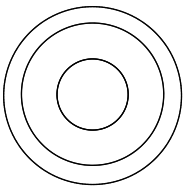
Výstup testů by měl vypadat přibližně takto:



B.3.6 [circle] Pomocí procedury pro mnohoúhelníky si nejprve zkuste vykreslit kružnici. Poté napište proceduru pro vykreslení kružnice o zadaném poloměru radius. (Nápověda: srovnejte obvod kružnice a pravidelného n-úhelníku). Kružnici nakreslete tak, aby její střed ležel v bodě, ve kterém byla želva před použitím procedury circle. Pro vypnutí a zapnutí kreslení použijte procedury penup a pendown. Po dokreslení kružnice vraťte želvu zpět do jejího středu.

```
def circle(radius):
    pass
```

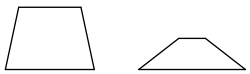
Výstup testů by měl vypadat přibližně takto:



B.3.7 [trapezoid] Nakreslete rovnoramenný lichoběžník s délkami základen base_length a top_length a výškou height (lichoběžník je čtyřhran s jednou dvojicí rovnoběžných stran – základen – spojených rameny, které jsou obecně různoběžné).

```
def trapezoid(base_length, top_length, height):
    pass
```

Výstup testů by měl vypadat přibližně takto:

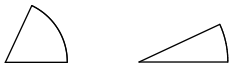


Část B.4: Bonusy

B.4.1 [pizza] Nakreslete kruhovou výseč („dílek pizzy“) se středovým úhlem zadaným (v stupních) parametrem `angle` a délkou strany `side`.

```
def pizza(side, angle):
    pass
```

Výstup testů by měl vypadat přibližně takto:



B.4.2 [koch] Pozor! Tento a následující příklad jsou založeny na rekurzi, kterou budeme probírat až na konci kurzu. Nemusíte si tedy lámat hlavu, pokud je neumíte vyřešit.
Nakreslete Kochovu vločku, která má stranu o délce `size`. Parametr `depth` udává kolikrát se má provést dělení strany vločky. Konstrukce začíná rovnostranným trojúhelníkem, přičemž vločka vzniká opakovanou aplikací následovného postupu na všechny úsečky, které v daném okamžiku tvoří obrazec:

1. vybranou stranu rozdělte na třetiny a prostřední část odstraňte,
2. nad prostřední částí sestrojte rovnostranný trojúhelník bez základny: danou stranu jste tak nahradili sekvencí 4 úseček: 2 zbývající krajní třetiny původní strany a 2 ramena přidaného trojúhelníku,

Daná iterace končí rozdělením poslední úsečky, která vznikla v iteraci předchozí. Proveďte celkem `depth` iterací. Testy vykreslují vločku hloubky dělení (počet iterací) 0 až 3.

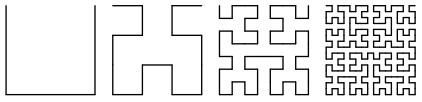
```
def koch_snowflake(size, depth):
    pass
```

Výstup testů by měl vypadat přibližně takto:

B.4.3 [hilbert] Nakreslete Hilbertovu křivku se stranou délky `size` a počtem dělení `iterations`. Hilbertova křivka vzniká, podobně jako Kochova vločka, opakovaným dělením stávajícího obrazce na zmenšené kopie sebe sama. Podrobnější návod, jak křivku nakreslit (na papír), naleznete na adrese https://is.muni.cz/go/9fh9k4.

```
def hilbert(size, iterations):
    pass
```

Výstup testů by měl vypadat přibližně takto:



Část B.5: Řešení

B.5.1 [pentagon.sol]

```
def pentagon(side):
    for i in range(5):
        forward(side)
        right(360 / 5)
```

B.5.2 [right.sol]

```
def right_triangle(side_a, side_b):
    hypotenuse = sqrt(side_a ** 2 + side_b ** 2)
    beta = degrees(asin(side_b / hypotenuse))

    forward(side_a)
    left(180 - beta)
    forward(hypotenuse)
    left(180 - (90 - beta))
    forward(side_b)
```

Část 1: If, cykly, proměnné

První kapitola sbírky slouží k procvičení látky z první přednášky – tento princip bude v platnosti celý semestr. Dva příklady přípravy v tomto týdnu musíte odevzdat do čtvrtka 8.10. a jeden další do soboty 10.10. Připomínáme, že musíte v každém čtyřtýdenním bloku splnit minimálně 3 přípravy ze 4. Detailněji jsou pravidla popsána v části A.
Tento týden se budeme zabývat zejména tzv. tokem řízení (anglicky control flow) – téma, které jsme načali už v nultém týdnu. Jedná se zejména o konstrukci podmíněného vykonání kódu (příkaz `if`) a o konstrukce pro opakované spuštění sekvence příkazů (příkazy `for`, `while`). V menší míře se budeme zabývat také proměnnými – pojmenovanými místy v paměti, do kterých lze ukládat hodnoty pro pozdější (případně vícenásobné) použití.
V ukázkách si na příkladech vysvětlíme již zmiňované základní konstrukce (teorii již znáte z přednášky). Ukázky označené znakem ⋆ jsou náročnější – pravděpodobně se u nich budete muset více soustředit. Nepovede-li se Vám takovou ukázku rozluštit napoprvé, zkuste ji na pár dnů odložit, a vrátit se k ní později (poté, co se Vám látka pro daný týden více rozležela v hlavě a již jste si vyřešili pár příkladů).

1. `is_triangle.py` – návratové hodnoty podprogramů, funkce
2. `sum.py` – použití indexů v cyklech
3. `fibonacci.py` – přepis matematické posloupnosti do algoritmu
4. `cycle.py` – použití podmíněného příkazu
5. `converge.py` ⋆ – výběr podposloupnosti

Dále máte k dispozici několik řešených příkladů, na kterých si můžete nové konstrukce procvičit:

1. `divisors.py` – zjištění počtu dělitelů čísla použitím cyklu
2. `powers.py` – součet po sobě jdoucích k-tých mocnin
3. `multiples.py` – počítání násobků

Hlavní náplní jsou samozřejmě neřešené úkoly. Ty, které jste si vybrali k odevzdání, vypracujte zcela samostatně, u těch zbývajících můžete pracovat způsobem, který Vám nejvíce vyhovuje: samostatně, probrat myšlenku se spolužáky, ale naprogramovat každý sám, dokonce si můžete vzájemně pomáhat i se samotným zápisem kódu. Ujistěte se ale, že v žádném případě neodevzdáváte příklad, se kterým Vám někdo pomáhal, a nepomáhejte spolužákům s příklady, které sami hodláte odevzdat!

1. `even.py` – součet sudých mocnin
2. `sequence.py` – n-té číslo posloupnosti s parametry
3. `nested.py` – vnořené posloupnosti
4. `prime.py` – kontrola prvočíselnosti
5. `pythagorean.py` – největší pythagorejská trojice
6. `geometry.py` – predikáty trojúhelníkových vlastností
7. `fibsum.py` – suma sudých členů Fibonacciho posloupnosti

8. next.py – výpočet následujícího většího násobku
9. coins.py – minimální počet mincí pro hodnotu

Na závěr několik bonusových (hvězdičkových) příkladů:

1. fibfibsum.py – použití posloupnosti k indexaci
2. is_abundant.py – vlastnosti čísel a jejich dělitelů
3. amicable.py – vlastnosti dvojic čísel

Část 1.1: Ukázky

1.1.1 [is_triangle] Abychom demonstrovali zápis a použití (čistých) funkcí a tedy i návratových hodnot, zadefinujeme si jednoduchou funkci se třemi parametry: délkami stran, které můžou (ale nemusí) zadávat trojúhelník. Výsledkem je pravdivostní hodnota (True nebo False), která říká, zda zadaná trojice délek stran skutečně popisuje přípustný trojúhelník. Funkcím, které nemají vedlejší efekty (tj. čistým), a kterých výsledkem je pravdivostní hodnota, říkáme predikáty.
Funkce, stejně jako procedury, definujeme klíčovým slovem def, za kterým následuje název funkce. Názvy (a později v semestru i typové anotace) parametrů píšeme do závorek za název funkce a oddělujeme je čárkami. V tomto kontextu mluvíme o formálních parametrech – v těle funkce se chovají jako proměnné, do kterých jsou přiřazeny hodnoty tzv. skutečných parametrů – těch, které jsou funkci předány při jejím použití (viz také níže). Řádek ukončíme dvojtečkou a pokračujeme tělem funkce: seznamem příkazů, které se při jejím použití (zavolání) vykonají.

```
def is_triangle(a, b, c):
```

Vykonávání funkce je (korektně) ukončeno buď dojdou-li příkazy k vykonání (dojdeme „na konec“), nebo vykonáním příkazu return. Chceme-li, aby funkce poskytla svému volajícímu nějaký výsledek, musíme použít příkaz return, kterému tuto výslednou hodnotu předáme. Výsledek můžeme zapsat jako libovolný výraz (zejména tedy nemusí být uložen v proměnné).
Všimněte si, že v tomto případě je výsledkem funkce logická konjunkce (použití operátoru and) tří podvýrazů, kde každý popisuje jednu variantu tzv. trojúhelníkové nerovnosti. Za zmínku zde stojí i konkrétní zápis těchto variant – první konjunkt je zapsán v abecedním pořadí, a každý další vznikl tzv. cyklickou záměnou předchozího, tzn. náhradami a → b, b → c a c → a.

```
    return (a + b > c) and (b + c > a) and (c + a > b)
```

Procedura main je součástí každého příkladu, a obsahuje jednoduché (základní) testy, které ověří, že jste naprogramovali zhruba to, co se očekávalo. Procházející testy nezaručují, že je Vaše řešení správné! U příkladů jsou testy pouze v kostrách (nachystaných zdrojových souborech .py): v HTML a PDF verzi sbírky je budeme zobrazovat jen v ukázkách jako je tato.

```
def main():  # demo
```

V tomto příkladu stojí za povšimnutí i samotný zápis testů (je důležité, abyste je uměli přečíst): příkaz assert ověří, že výraz, který mu předáváme, se vyhodnotí na hodnotu True, a pokud tomu tak není, program okamžitě ukončí s chybou.
Krom použití příkazu assert si všimněte i zápisu tzv. volání funkce (neboli jejího použití): volání funkce je výraz, který začíná jménem příslušné funkce, které je následováno závorkami, do kterých uvádíme (skutečné) hodnoty parametrů funkce. Závorky mohou být prázdné, ale nelze je vynechat.

```
    assert is_triangle(3, 4, 5)
    assert is_triangle(1, 1, 1)
    assert not is_triangle(1, 1, 3)
    assert not is_triangle(2, 3, 1)
```

1.1.2 [sum] Uvažme posloupnost $a_n = n^n$, a posloupnost jejích částečných součtů $s_n = \sum_{i=1}^{n} a_i = \sum_{i=1}^{n} i^i$. Ujistěte se, že těmto definicím rozumíte: neznáte-li například definici operátoru $\sum$ (suma), můžete se s výhodou obrátit na Wikipedii. Pro jistotu uvádíme několik členů obou těchto posloupností:

$$a_1 = 1^1 = 1$$
$$a_2 = 2^2 = 4$$
$$a_3 = 3^3 = 27$$

$$s_1 = \sum_{i=1}^{1} i^i = 1^1 = 1$$
$$s_2 = \sum_{i=1}^{2} i^i = 1^1 + 2^2 = 1 + 4 = 5$$
$$s_3 = \sum_{i=1}^{3} i^i = 1^1 + 2^2 + 3^3 = 1 + 4 + 27 = 32$$

Naším úkolem bude nyní naprogramovat v Pythonu (čistou) funkci nth_element(n), která počítá příslušné a_n, a (opět čistou) funkci partial_sum(n), která počítá příslušné s_n. První funkce je přímočará, stačí nám znát zabudovaný operátor mocnění ** a zápis definice funkce:

```
def nth_element(n):
    return n ** n
```

Výpočet partial_sum(n) bude nicméně o něco složitější: operátor suma sčítá řadu čísel, jejichž počet je dán rozdílem mezi jeho horním a dolním indexem. Objeví-li se v některém indexu proměnná, počet sečtených členů bude typicky záviset na hodnotě této proměnné.
Jak již jistě víte z přednášky, v situaci, kdy potřebujeme opakovaně provádět příkazy (a zejména není-li počet opakování konstanta) použijeme cyklus. Nejjednodušší formou cyklu je příkaz „opakuj n-krát“, který v Pythonu zapisujeme for i in range(n).
Krom hodnoty n je zde důležitá ještě proměnná i: obecně se jedná o tzv. proměnnou cyklu. Tato proměnná má k tělu cyklu podobný vztah, jako má parametr funkce k tělu funkce: před každým provedením těla (tzv. iterací) se do i přiřadí nová hodnota (jaká přesně hodnota to bude záleží na konkrétní formě cyklu).
V tomto případě – cyklus tvaru for i in range(n) – se do i přiřadí pořadové číslo iterace, a samotnou proměnnou i pak nazýváme indexovou proměnnou. Ve většině programovacích jazyků (a Python není výjimkou) se indexuje od 0, tzn. v první iteraci je i = 0, ve druhé i = 1, atd., konečně v poslední iteraci je i = n - 1. Nyní můžeme konečně přistoupit k definici funkce partial_sum(n):

```
def partial_sum(n):
```

Jako první krok si zavedeme proměnnou, do které budeme postupně přičítat jednotlivé hodnoty a_i – takové proměnné říkáme střadač nebo akumulátor (angl. accumulator).

```
    result = 0
```

Následuje samotný cyklus, který v každé iteraci do akumulátoru result přičte příslušnou hodnotu a_i. Protože indexová proměnná i je číslována od 0, ale hodnoty a_i jsou číslovány od 1, vypočteme hodnotu a_i jako nth_element(i + 1):

```
    for i in range(n):
        result += nth_element(i + 1)
```

Po skončení cyklu je v akumulátoru požadovaná suma $s_n = \sum_{i=1}^{n} a_i$. Pro každé i v rozmezí 0 až n - 1 (včetně) bylo provedeno tělo cyklu, a v result je tedy uložen součet nth_element(0 + 1) + nth_element(1 + 1) + ... + nth_element(n - 1 + 1), neboli nth_element(1) + nth_element(2) + ... + nth_element(n).

```
    return result

def main():  # demo
    assert partial_sum(1) == 1
    assert partial_sum(2) == 5
    assert partial_sum(3) == 32
    assert partial_sum(4) == 288
    assert partial_sum(7) == 873612
    assert partial_sum(15) == 449317984130199828
```

1.1.3 [fibonacci] (Čistá) funkce `fib` počítá `n`-tý prvek tzv. Fibonacciho posloupnosti, dané předpisem: $f(1) = f(2) = 1, f(n) = f(n-1) + f(n-2)$ – každý prvek této posloupnosti je tedy součtem předchozích dvou (s výjimkou prvních dvou, které jsou pevně dané).
Zkusíte-li si posloupnost napsat na papír (1, 1, 2, 3, 5, ...), zřejmě zjistíte, že nejjednodušší způsob jak to udělat, je sečíst vždy poslední dvě už napsaná čísla a výsledek připsat na konec vznikajícího seznamu. Na dřívější čísla se už nemusíme znovu dívat: pro výpočet dalšího prvku potřebujeme vidět právě dva předchozí prvky. Můžete tedy vzít gumu, a po připsání jednoho čísla na konec smazat jedno číslo ze začátku – ani s tímto opatřením nebudete mít s výpočtem žádný problém. Na papíře budou v každém momentě 2 nebo 3 čísla, podle toho, kde se ve výpočtu nacházíte.
Tuto myšlenku využijeme pro zápis algoritmu: budeme potřebovat dvě proměnné, které budou reprezentovat ony dvě „naposled zapsaná" čísla na konci posloupnosti (protože někdy máme ale na papíře čísla 3, budeme ve skutečnosti občas potřebovat ještě jednu – dočasnou – proměnnou).
Protože postup výpočtu sleduje fixní seznam kroků, který se dokola opakuje, použijeme navíc cyklus.

```
def fib(n):
```

Proměnná `a` reprezentuje předposlední a proměnná `b` poslední vypočtené Fibonacciho číslo. Na začátku jsme na papír napsali dvě jedničky – jedná se o ony pevně dané první dva prvky posloupnosti.

```
    a = 1
    b = 1
```

Zatím jsme „vypočítali" první a druhé Fibonacciho číslo. Zajímá-li nás `n`-té číslo, musíme připsat dalších `n - 2` čísel, aby platilo, že poslední číslo je to, které nás zajímá. V každé iteraci následujícího cyklu provedeme výpočet jednoho dalšího čísla (a umazání prvního čísla).

```
    for i in range(n - 2):
```

Do nové (dočasné) proměnné `c` si vypočteme další Fibonacciho číslo. Po tomto příkazu bude proměnná `a` obsahovat třetí číslo od konce aktuálně „zapsaného" seznamu, proměnná `b` číslo předposlední a proměnná `c` číslo poslední. Jsme nyní v situaci, kdy si pamatujeme zároveň 3 čísla.

```
        c = a + b
```

„Zapomenutí" prvního čísla realizujeme tak, že „nové" poslední dvě čísla (nyní `b` a `c`) uložíme do proměnných `a` a `b`. Hodnotou uloženou v (dočasné) proměnné `c` se nebudeme dále zabývat – v další iteraci cyklu proměnnou `c` přepíšeme novou dočasnou hodnotou. Zamyslete se, zda je pořadí následujících dvou příkazů důležité, a proč.

```
        a = b
        b = c
```

Jak jsme zmínili na začátku, proměnná `b` reprezentuje poslední vypočtené Fibonacciho číslo (s výjimkou krátkého okamžiku uprostřed cyklu). Protože jsme vypočetli právě `n` čísel, poslední z vypočtených čísel je `n`-té, a tedy proměnná `b` obsahuje kýžený výsledek funkce `fib`.

```
    return b

def main():  # demo
    assert fib(1) == 1
    assert fib(2) == 1
    assert fib(3) == 2
    assert fib(5) == 5
    assert fib(9) == 34
    assert fib(11) == 89
    assert fib(20) == 6765
    assert fib(40) == 102334155
```

1.1.4 [cycle] Uvažujme posloupnost definovanou jako $a_1 = 1, a_{n+1} = a_n \diamond n$, kde $\diamond$ se cyklicky vybírá z $+, \cdot, -$. Prvních 5 prvků této posloupnosti (zařazené v OEIS jako A047908) je:

$$
\begin{aligned}
a_1 &= 1 \\
a_2 &= a_1 + 1 = 2 \\
a_3 &= a_2 \cdot 2 = 4 \\
a_4 &= a_3 - 3 = 1 \\
a_5 &= a_4 + 4 = 5
\end{aligned}
$$

Naším úkolem bude napsat (čistou) funkci, která vyčíslí `n`-tý prvek této posloupnosti:

```
def cycle(n):
```

Protože budeme chtít použít cyklus `while`, musíme si indexovou proměnnou explicitně zavést:

```
    i = 1
```

K výpočtu a_i potřebujeme znát hodnotu a_{i-1}, proto si aktuální hodnotu a_i uložíme do proměnné `a_i` (podobně jako jsme k výpočtu Fibonacciho posloupnosti potřebovali poslední dva prvky). V další iteraci (poté, co se zvýší indexová proměnná `i`) budeme mít v `a_i` chvíli hodnotu a_{i-1}, kterou využijeme pro výpočet (nové) hodnoty a_i.

```
    a_i = 1
```

Cyklus `while`, jak jistě víte z přednášky, provádí své tělo tak dlouho, dokud platí podmínka cyklu. V tomto případě tedy budeme cyklus opakovat dokud platí `i < n`:

```
    while i < n:
```

Nyní se musíme rozhodnout, který operátor použít pro výpočet další hodnoty `a_i`. Protože cyklicky vybíráme ze 3 možností, můžeme se rozhodnout dle zbytku po dělení indexu `i` třemi: v první, čtvrté, sedmé atd. iteraci použijeme operátor `+`, v druhé, páté, ... operátor `*` a konečně ve třetí, šesté, ... operátor `-`:

```
        if i % 3 == 1:
            a_i = a_i + i
        elif i % 3 == 2:
            a_i = a_i * i
        else:  # i % 3 == 0
            a_i = a_i - i

        i += 1
```

V každé iteraci cyklu zvyšujeme indexovou proměnnou `i` o jedna, a před cyklem platilo $i \leq n$. Po cyklu musí tedy nutně platit `i == n`, a protože zároveň po každé iteraci platí, že `a_i` obsahuje hodnotu a_i, musí také platit, že po ukončení cyklu je v proměnné `a_i` uložena hodnota a_n.

```
    return a_i

def main():  # demo
    assert cycle(1) == 1
    assert cycle(2) == 2
```

```
    assert cycle(3) == 4
    assert cycle(4) == 1
    assert cycle(5) == 5
    assert cycle(6) == 25
    assert cycle(7) == 19
    assert cycle(8) == 26
```

1.1.5 [converge] ⋆ Každá omezená posloupnost – tedy taková, která nabývá hodnoty pouze z nějakého konečného intervalu – má tzv. konvergentní podposloupnost. Co tyto termíny přesně znamenají nás nemusí trápit (více se dozvíte v matematické analýze): nám bude stačit intuice. Podposloupnost je posloupnost, která vznikne „přeskočením" některých prvků původní posloupnosti (zde je B podposloupnost sestávající z lichých prvků posloupnosti A):

$$A \to 1, 2, 3, 4, 5, \ldots$$
$$B \to 1, 3, 5, \ldots$$

Konvergentní posloupnost je pak taková, že se její prvky postupně blíží nějaké konkrétní hodnotě (tzv. limitě L) – přibližně platí, že čím větší index i, tím je vzdálenost $|L - a_i|$ menší.
Naším úkolem bude takovou konvergentní podposloupnost najít: začneme omezenou posloupností $a_i = \sin(i)$ a budeme budovat konvergentní podposloupnost B s prvky b_j. Jak na to?
První pozorování je, že se stačí zabývat kladnými hodnotami a_i. Dále pak stačí zabezpečit, aby platilo $b_{j+1} \leq b_j$. Při výběru hodnoty b_1 máme určitou volnost, ale je výhodné zvolit $a_1 = b_1 = \sin(1)$. Zapišme nyní funkci `convergent(n)`, které výsledkem bude hodnota b_n:

```
def convergent(n):
```

Pro samotný výpočet budeme potřebovat dva indexy: index `i` náleží posloupnosti A (čísluje tedy prvky a_i) zatímco index `j` náleží posloupnosti B (čísluje prvky b_j).

```
    i = 1
    j = 1
```

Navíc si potřebujeme pamatovat poslední nalezenou hodnotu b_j – proměnná `last` bude vždy (opět s výjimkou krátkého okamžiku mezi dvěma sousedními příkazy uvnitř cyklu) obsahovat `j`-tou hodnotu posloupnosti B (kde `j` značí hodnotu proměnné `j` zavedené výše). Vzpomeňte si také, že $a_1 = b_1$.

```
    last = sin(i)
```

Následuje samotný cyklus, který bude hledat hodnotu b_j. Tento bude postupně procházet prvky a_i posloupnosti A. Vždy, když nalezneme nové a_i, pro které platí $a_i \leq b_j$ – kde b_j je uloženo v proměnné `last` – můžeme toto a_i přidat do posloupnosti B, jako b_{j+1}, a odpovídajícím způsobem upravit proměnné `j` a `last`. V programu zapisujeme a_i jako `sin(i)`.

```
    while j < n:
        i += 1
        if sin(i) > 0 and sin(i) <= last:
            j += 1
            last = sin(i)
```

Po ukončení cyklu platí `j == n` (před cyklem platilo `j <= n`, cyklus ukončíme jakmile přestane platit `j < n` a zároveň hodnotu `j` v každé iteraci zvýšíme nejvýše o 1). Protože v každém kroku platí, že proměnná `last` obsahuje prvek b_j a nyní zároveň platí `j = n`, celkem dostáváme, že po ukončení cyklu je v proměnné `last` uložena hodnota b_n.

```
    return last

def main():  # demo
    assert convergent(1) == sin(1)
    assert convergent(2) == sin(3)
    assert convergent(3) == sin(44)
```

Krom obvyklých konkrétních případů, které testujeme výše, můžeme ověřovat i vlastnosti námi implementovaných funkcí. Například níže kontrolujeme monotónnost (posloupnost je nestoupající) a omezenost zespodu (nulou). Tyto dvě vlastnosti dohromady zaručují, že posloupnost je konvergentní: samozřejmě, v konečném čase lze takto ověřit pouze konečný počet případů, a testy nám tedy ani jednu ze zmiňovaných tří vlastností nemohou zaručit.

```
    for i in range(5):
        assert convergent(i + 1) <= convergent(i)
        assert convergent(i) > 0
```

Část 1.2: Rozcvička

1.2.1 [divisors] Napište funkci, která vrátí počet různých kladných dělitelů kladného celého čísla `number` (např. číslo 12 je dělitelné 1, 2, 3, 4, 6 a 12 – výsledek `divisors(12)` bude tedy 6.

```
def divisors(number):
    pass
```

1.2.2 [powers] Napište funkci, která spočítá sumu prvních `n` `k`-tých mocnin kladných po sobě jdoucích čísel, tzn. sumu $s_n = \sum_{i=1}^{n} a_i$, kde i-tý člen $a_i = i^k$.

```
def powers(n, k):
    pass
```

1.2.3 [multiples] Napište funkci `sum_of_multiples` s parametrem `n`, která spočítá sumu kladných čísel a_i, kde $a_i \leq$ `n` a zároveň $3|a_i$ nebo $5|a_i$ (t.j. každé a_i je dělitelné třemi nebo pěti). Například pro `n = 10` je očekávaný výsledek $33 = 3 + 5 + 6 + 9 + 10$.

```
def sum_of_multiples(n):
    pass
```

Část 1.3: Příklady

1.3.1 [even] Uvažujme posloupnost a_i druhých mocnin sudých čísel $a_i = 4i^2$. Napište funkci, která vrátí sumu prvních `n` členů této posloupnosti $s_n = \sum_{i=1}^{n} a_i = \sum_{i=1}^{n} 4i^2$.

```
def even(n):
    pass
```

1.3.2 [sequence] Napište (čistou) funkci `sequence`, která spočítá hodnotu členu a_n níže popsané posloupnosti, kde `n` je první parametr této funkce. První člen posloupnosti, a_0, je zadán parametrem `initial`, každý další člen je pak určen sumou $a_j = \sum_{i=1}^{k} (-1)^i \cdot i \cdot a_{j-1}$, kde `k` je druhým parametrem funkce `sequence`. Například pro parametry `k = 3` a `initial = 2` jsou první 3 členy posloupnosti:

$$a_0 = 2$$
$$a_1 = \sum_{i=1}^{3} (-1)^i \cdot i \cdot a_0 = -a_0 + 2a_0 - 3a_0 = -2 + 4 - 6 = -4$$
$$a_2 = \sum_{i=1}^{3} (-1)^i \cdot i \cdot a_1 = -a_1 + 2a_1 - 3a_1 = 4 - 8 + 12 = 8$$

Očekávaný výsledek pro volání `sequence(2, 3, 2)` je tedy 8.

```
def sequence(n, k, initial):
    pass
```

1.3.3 [nested] Napište funkci `nested`, která spočítá `n`-tý člen posloup-

nosti (počítáno od 0), která vznikne napojením postupně se prodlužujících prefixů přirozených čísel.
Nechť A_i je posloupnost čísel 1 až i:

$$A_1 \to 1$$
$$A_2 \to 1, 2$$
$$A_3 \to 1, 2, 3$$
$$A_4 \to 1, 2, 3, 4$$
$$A_5 \to 1, 2, 3, 4, 5$$

Hledaná posloupnost B vznikne napojením posloupností A_1, A_2, A_3 ... (do nekonečna) za sebe:

$$B \to 1, 1, 2, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4, 5, 1, 2, \ldots$$

Vaším úkolem je najít n-tý prvek posloupnosti B.

```
def nested(n):
    pass
```

Dále napište funkci nested_sum, která spočítá sumu prvních n členů této posloupnosti.

```
def nested_sum(n):
    pass
```

1.3.4 [prime] Napište funkci, která ověří, zda je číslo number prvočíslo.

```
def is_prime(number):
    pass
```

1.3.5 [pythagorean] Napište funkci largest_triple, která najde pythagorejskou trojici (a, b, c) – totiž takovou, že a, b a c jsou přirozená čísla a platí $a^2 + b^2 = c^2$ (tzn. tvoří pravoúhlý trojúhelník). Hledáme trojici, která:

1. má největší možný součet $a + b + c$,
2. hodnoty a, b jsou menší než max_side.

Výsledkem funkce bude součet $a + b + c$, tedy největší možný obvod pravoúhlého trojúhelníku, jsou-li obě jeho odvěsny kratší než max_side.

```
def largest_triple(max_side):
    pass
```

1.3.6 [geometry] Napište predikát (tj. čistou funkci, která vrací pravdivostní hodnotu – boolean), který je pravdivý, je-li možno vytvořit pravoúhlý trojúhelník ze stran o délkách zadaných kladnými přirozenými čísly a, b a c.

```
def is_right(a, b, c):
    pass
```

Dále napište predikát, který je pravdivý, popisují-li parametry a, b a c rovnostranný trojúhelník.

```
def is_equilateral(a, b, c):
    pass
```

Konečně napište predikát, který je pravdivý, popisují-li parametry a, b a c rovnoramenný trojúhelník.

```
def is_isosceles(a, b, c):
    pass
```

1.3.7 [fibsum] Napište funkci, která spočítá sumu prvních n sudých členů Fibonacciho posloupnosti (tj. členů, které jsou sudé, nikoliv těch, které mají sudé indexy). Například volání fibsum(3) = 44 = 2 + 8 + 34.

```
def fibsum(n):
    pass
```

1.3.8 [next] Napište funkci, která pro zadané číslo number najde nejbližší větší číslo, které je násobkem čísla k.

```
def next_multiple(number, k):
    pass
```

Dále napište funkci, která pro zadané číslo number najde nejbližší větší prvočíslo.

```
def next_prime(number):
    pass
```

1.3.9 [coins] Uvažme, že chceme přesně zaplatit sumu value českými mincemi (denominace 1, 2, 5, 10, 20 a 50 korun). Spočtěte, kolik nejméně mincí potřebujeme.

```
def coins(value):
    pass
```

Část 1.4: Bonusy

1.4.1 [fibfibsum] Nechť A je Fibonacciho posloupnost s členy a_n a B je posloupnost taková, že má na i-té pozici a_i-tý prvek posloupnosti A, tj. prvek s indexem a_i (nikoliv prvek s indexem i). Napište funkci, která sečte prvních count prvků posloupnosti B (t.j. ty prvky posloupnosti A, kterých indexy jsou po sobě jdoucí Fibonacciho čísla).
Například fibfibsum(6) se vypočte takto:

$$a_1 + a_1 + a_2 + a_3 + a_5 + a_8 = 1 + 1 + 1 + 2 + 5 + 21 = 31$$

```
def fibfibsum(count):
    pass
```

1.4.2 [is_abundant] Napište predikát is_abundant, který je pravdivý, pokud je číslo number abundantní, t.j. je menší, než součet jeho vlastních dělitelů (všech dělitelů s výjimkou sebe sama).

```
def is_abundant(number):
    pass
```

1.4.3 [amicable] Napište predikát, který určí, jsou-li 2 čísla spřátelená (amicable). Spřátelená čísla jsou dvě přirozená čísla taková, že součet všech kladných dělitelů jednoho čísla (kromě čísla samotného) se rovná druhému číslu, a naopak – součet všech dělitelů druhého čísla (kromě něho samotného) se rovná prvnímu.

```
def amicable(a, b):
    pass
```

Část 1.5: Řešení

1.5.1 [divisors.sol]

```
def divisors(number):
    count = 1  # number always divides itself
    divisor = 1
    maximal = number // 2
    while divisor <= maximal:
        if number % divisor == 0:
            count += 1
        divisor += 1
    return count
```

1.5.2 [powers.sol]

```
def powers(n, k):
    result = 0
    for i in range(n):
```

```
            result += pow(i + 1, k)
        return result
```

1.5.3 [multiples.sol]

```
def sum_of_multiples(n):
    result = 0
    for i in range(n + 1):
        if i % 3 == 0 or i % 5 == 0:
            result += i
    return result
```

Část 2: Číselné algoritmy

Tento týden pokračujeme v programování s čísly (první setkání se složitějšími datovými typy nás čeká příští týden). Tentokrát si naprogramujeme řadu jednoduchých algoritmů, které si vystačí s konstrukcemi, které již známe: cykly `for` a `while`, podmíněnými příkazy `if`, proměnnými, a definicemi čistých funkcí. Významnější roli budou hrát i čísla s plovoucí desetinnou čárkou – typ `float`.
To, co bude tento týden nové je, že algoritmy, které budeme programovat, budou mít složitější strukturu, budou používat více proměnných a budou se typicky více větvit. V tomto týdnu byste si tedy z cvičení měli odnést základní dovednosti algoritmizace a v tomto kontextu si procvičit použití a zápis konstrukcí, které znáte z prvních dvou přednášek.
V neposlední řadě dojde tento týden i na základy dekompozice: některé algoritmy, které budeme programovat, bude vhodné rozložit na podprogramy. Podobně jako minulý týden, budeme tento týden pracovat pouze s čistými funkcemi: kdykoliv v příkladech pro tento týden zmíníme funkci, myslíme tím implicitně funkci čistou.
Rádi bychom ještě zmínili, že i v případě, že již znáte datový typ řetězec (`str`), není vhodné jej v této kapitole k ničemu používat. Řešení, která používají řetězce, nemusí být přijata.

1. `descending.py` – n-tá cifra čísla
2. `comb.py` – kombinační čísla, `for` a `while`
3. `triangle.py` – řešení trojúhelníků (desetinná čísla)

Řešené příklady:

1. `palindrom.py` – je číslo palindrom?
2. `digits.py` – počet cifer v posloupnosti
3. `fridays.py` – počet pátků 13. v zadaném roce

Základní úlohy:

1. `digit_sum.py` – variace na ciferný součet
2. `joined.py` – posloupnost čísel
3. `savings.py` – úročení a inflace
4. `delete.py` – umazávání cifer z čísla
5. `maximum.py` – lokální maximum na intervalu
6. `credit.py` – ověření korektnosti čísla platební karty
7. `cards.py` – visa, mastercard
8. `fraction.py` – převod na řetězcový zlomek
9. `workdays.py` – počet pracovních dnů v roce

Bonusové úlohy:

1. `bisection.py` – aproximace kořenů

Část 2.A: Poziční číselné soustavy

K zápisu čísel v západní civilizaci běžně používáme desítkovou soustavu. Desítková soustava je jednou z mnoha tzv. pozičních číselných soustav, při kterých je se hodnota čísla odvíjí od toho, na jaké pozici stojí jaká číslice. Hodnotu čísla získáme tak, že pozice číslujeme od nuly zprava, hodnotu každé číslice násobíme základem umocněným na pozici a výsledky sečteme.
V desítkové soustavě tedy nejpravější číslici násobíme $10^0 = 1$, druhou číslici zprava násobíme $10^1 = 10$, třetí zprava $10^2 = 100$ atd.
Můžeme ovšem za základ vzít i jiné číslo než je desítka. Třeba ve trojkové soustavě násobíme číslice zprava hodnotami 1, 3, 9, 27, … v sedmičkové soustavě násobíme číslice zprava hodnotami 1, 7, 49, 343.
To, že daný zápis je myšlen v soustavě s jiným základem než 10, typicky v matematice značíme uzávorkováním a dolním indexem. Například $(321)_7$ je zápis čísla 162, protože $3 \cdot 49 + 2 \cdot 7 + 1 = 162$.
Důležité je si uvědomit, že čísla (jako abstraktní pojem pro počet) jsou úplně nezávislá na zvolené reprezentaci. Pokud bychom se vyvinuli jinak a neměli deset prstů, ale třeba jen osm, tak by nám desítková soustava připadala bizarní a osmičková jako zcela přirozená. (A mimochodem, v historii se taky používala soustava dvanáctková nebo šedesátková – zbytek té historie vidíme např. na současném systému pro měření času.)
Hlavní myšlenkou zde je to, že $(101)_2 = 5$, tedy jde o totéž číslo, jen jinak zapsané.
V Pythonu máme standardně možnost používat tyto soustavy:

- desítkovou (používáme číslice `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9` a zápis čísel nezačíná žádným speciálním prefixem),
- dvojkou (používáme číslice `0`, `1` a zápis čísel začíná `0b`),
- osmičkovou (používáme číslice `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7` a zápis čísel začíná prefixem `0o`),
- šestnáctkovou (používáme číslice `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `a`, `b`, `c`, `d`, `e`, `f` a zápis čísel začíná prefixem `0x`).

Tedy např. číslo `0o321` je číslo $(321)_8 = 209$. Totéž číslo se taky dá v Pythonu zapsat jako `209` nebo `0xd1` nebo `0b11010001`, ale pořád je to stejné číslo, jak dokládá i skutečnost, že výraz `0b11010001 == 209` se vyhodnotí na `True`.

Část 2.1: Ukázky

2.1.1 [descending] V této ukázce si naprogramujeme jednoduchý algoritmus, který pracuje s desítkovým rozvojem celého čísla: konkrétně se budeme ptát, zda jsou v desítkovém zápisu daného čísla jednotlivé cifry uspořádané sestupně (uvažujeme pořadí od nejvýznamnější, tzn. nejlevější, cifry).
Protože chceme pracovat s ciframi, jeví se jako rozumné zadefinovat si pomocnou funkci, která nám vrátí konkrétní cifru. Desítkový rozvoj (celého) čísla n, které má m desítkových cifer, lze zapsat:

$$n = \sum_{i=0}^{m} a_i \cdot 10^i$$

kde každé pro každé a_i platí $0 \leq a_i \leq 9$. Za povšimnutí stojí i to, že dle zde použité definice má nejméně významná cifra („jednotky“) index 0. Chceme-li nalézt k-tou cifru, můžeme postupovat následovně: nejprve n vydělíme číslem 10^k – pohled na pravou stranu výše uvedené rovnosti nám rychle napoví, že členy, u kterých je mocnina desítky menší než k ze sumy úplně zmizí a člen a_k se stane nejnižším (rozmyslete si, jak vypadá člen kde $i = k$):

$$n/10^k = \sum_{i=k}^{m-k} a_i \cdot 10^{i-k}$$

Zbývá učinit následovné pozorování: protože nás zajímá hodnota a_k, a protože každé jiné a_i se v rozvoji $n/10^k$ objevuje vynásobeno nějakou kladnou mocninou desítky, můžeme s výhodou použít operaci zbytku po dělení (modulo, operátor `%`): tímto se zbavíme všech ostatních členů (formálněji: zbytek po dělení člena $a_i \cdot 10^{i-k}$ desíti je 0 pro každé $i > k$):

$$n/10^k \equiv a_k \pmod{10}$$

Tímto je vysvětlena na pohled velice jednoduchá funkce `get_digit`:

```
def get_digit(number, k):
    return (number // 10 ** k) % 10
```

Následující funkce pracuje na stejném principu: každé dělení desíti odstraní jednu cifru (jeden člen sumy, která definuje desítkový rozvoj). Počet provedených iterací si udržujeme v čítači `count`.

```
def count_digits(number):
    count = 0
    while number > 0:
        count += 1
        number = number // 10
    return count
```

Funkce `get_digit` a `count_digits` nám už umožní popsat náš původní problém přirozeným způsobem: pro každou dvojici cifer ověříme, že jsou ve správném pořadí. Protože cifry jsou při procházení zleva očíslovány sestupně, musíme si dát pozor, v jakém pořadí ony dvě srovnávané cifry následují.

```
def is_descending(number):
```

Dvojic cifer je o jednu méně, než cifer samotných: dvojciferné číslo má jednu dvojici cifer, trojciferné dvě, atd., proto musíme od výsledku `count_digits` odečíst jedničku.

```
    for k in range(count_digits(number) - 1):
```

Označme a_i cifry čísla `number`: volání funkce `get_digit(number, i)` tedy vrací hodnotu a_i. Cifra s indexem `k + 1` je nalevo od cifry s indexem `k`: mají-li být tedy cifry uspořádány sestupně zleva doprava, musí pro každou dvojici platit $a_{k+1} \geq a_k$. Protože kontrolujeme, že tato podmínka platí pro každou dvojici, jakmile nalezneme nějakou, která ji porušuje (proto v podmínce níže naleznete negaci „chtěné“ vlastnosti), víme, že celkový výsledek je `False`, a vykonávání funkce ukončíme příkazem `return` (na ostatní dvojice se už není potřeba dívat).

```
        if get_digit(number, k + 1) < get_digit(number, k):
            return False
```

V cyklu výše jsme zkontrolovali každou dvojici cifer: kdyby některá porušila kýženou vlastnost (cifry jsou uspořádané sestupně), spustil by se příkaz `return` a funkce by byla ukončena. Proto, dojdeme-li až sem, víme, že vlastnost platila pro každou dvojici cifer, a tedy platí i pro číslo jako celek.

```
    return True
```

Zbývá pouze ověřit, že jsme v implementaci neudělali chybu.

```
def main():  # demo
    assert is_descending(7)
    assert is_descending(321)
    assert is_descending(33222111)
    assert is_descending(9999)
    assert is_descending(7741)
    assert not is_descending(123)
    assert not is_descending(332233)
    assert not is_descending(774101)
```

2.1.2 [comb] V této ukázce se zaměříme na ekvivalenci `for` a `while` cyklů. Podíváme se přitom na kombinační čísla, definovaná jako:

$$(n\,|\,k) = n!/(k! \cdot (n-k)!)$$

kde $k \leq n$. Samozřejmě, mohli bychom počítat kombinační čísla přímo z definice, navíc v modulu `math` je již k dispozici funkce `factorial`, takže bychom se v zápisu obešli úplně bez cyklů. Nicméně jednoduché pozorování nám (resp. programu, který bude výpočet provádět) může ušetřit významné množství práce. Jak jistě víte, faktoriál je definován takto:

$$n! = \prod_{i=1}^{n} i$$

A tedy:

$$n!/k! = \prod_{i=1}^{n} i / \prod_{i=1}^{k} i = \prod_{i=k+1}^{n} i$$

Navíc, abychom měli zaručeno, že skutečně práci ušetříme, můžeme tento trik aplikovat na větší z k nebo $n-k$.

```
def comb_for(n, k):
```

Nejprve zjistíme, které z k resp. $n-k$ je menší: vzhledem k symetrii definice vůči těmto dvěma hodnotám můžeme případně k nahradit hodnotou $n-k$, aniž bychom změnili výsledek: platí $(n\,|\,k) = (n\,|\,n-k)$.

```
    if k < n - k:
        k = n - k
```

Dále chceme vynásobit všechna čísla mezi k a n (nicméně k samotné chceme přeskočit, zatímco n chceme zahrnout):

```
    numerator = 1

    for i in range(k + 1, n + 1):
        numerator *= i

    return numerator / factorial(n - k)
```

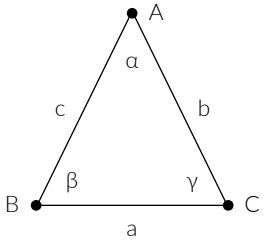
Nyní ekvivalentní definice pomocí cyklu `while`:

```
def comb_while(n, k):

    if k < n - k:
        k = n - k

    numerator = 1
    i = k + 1

    while i <= n:
        numerator *= i
        i += 1

    return numerator / factorial(n - k)
```

Kontrolu správnosti tentokrát provedeme trochu jinak: nebudeme kontrolovat předem vypočtené hodnoty, které bychom napsali do programu jako konstanty, jak jsme to většinou dělali doteď. Místo toho ověříme, že naše implementace dává stejný výsledek, jako výpočet přímo z definice. Díky tomu můžeme kontrolovat výrazně více případů, aniž bychom se takříkajíc upsali k smrti.

```
def main():  # demo
    for n in range(1, 50):
        for k in range(1, n):
            naive = factorial(n) / (factorial(k) * factorial(n - k))
            assert comb_for(n, k) == naive
            assert comb_while(n, k) == naive
```

2.1.3 [triangle] V této ukázce si napíšeme program, který bude počítat obvod trojúhelníka, který ale může být zadaný různými způsoby: tři strany, 2 strany a sevřený úhel, dva úhly a libovolná strana. Strany budeme značit a, b, c; úhel mezi a a b bude γ (`gamma`), mezi b a c bude α (`alpha`) a konečně mezi c a a je úhel β (`beta`):



Nejjednodušší je samozřejmě výpočet obvodu pro trojúhelník zadaný třemi stranami:

```
def perimeter_sss(a, b, c):
    return a + b + c
```

Následuje trojúhelník zadaný dvěma stranami a sevřeným úhlem, kdy získáme délku třetí strany použitím kosinové věty.

```
def perimeter_sas(a, gamma, b):
    c = sqrt(a ** 2 + b ** 2 - 2 * a * b * cos(radians(gamma)))
    return perimeter_sss(a, b, c)
```

Dále vyřešíme případ jedné strany a dvou jí přilehlých úhlů, kdy použijeme naopak větu sinovou.

```
def perimeter_asa(alpha, c, beta):
    gamma = radians(180 - alpha - beta)
    alpha = radians(alpha)
    beta = radians(beta)
    a = c * sin(alpha) / sin(gamma)
    b = c * sin(beta) / sin(gamma)
    return perimeter_sss(a, b, c)
```

Poslední případ, který budeme řešit, jsou dva úhly a strana přilehlá pouze druhému z nich. Tento případ lehce převedeme na předchozí.

```
def perimeter_aas(alpha, gamma, c):
    return perimeter_asa(alpha, c, 180 - alpha - gamma)
```

Tím končí samotná implementace, nyní přistoupíme k jejímu testování. Asi si uvědomujete, že v předchozím byl relativně velký prostor k překlepům a záměnám stran nebo úhlů. Proto budeme testovat důkladněji: než bylo dosud obvyklé – budeme postupovat podobně, jako v předchozí ukázce. Nejprve si implementujeme 2 pomocné funkce, které z popisu pomocí 3 délek stran vypočtou dva různé úhly:

```
def get_alpha(a, b, c):
    return acos((b ** 2 + c ** 2 - a ** 2) / (2 * b * c)) * 180 / pi

def get_beta(a, b, c):
    return acos((a ** 2 + c ** 2 - b ** 2) / (2 * a * c)) * 180 / pi
```

Pro samotnou kontrolu funkcí z rodiny `perimeter_*` si definujeme pomocnou proceduru, která pracuje s obecným trojúhelníkem, zadaným délkami stran.

```
def check_triangle(a, b, c):
    alpha = get_alpha(a, b, c)
    beta = get_beta(a, b, c)
    gamma = 180 - alpha - beta
    assert isclose(perimeter_sss(a, b, c), a + b + c)
    assert isclose(perimeter_sas(a, gamma, b), a + b + c)
    assert isclose(perimeter_sas(b, alpha, c), a + b + c)
    assert isclose(perimeter_sas(c, beta, a), a + b + c)
    assert isclose(perimeter_asa(alpha, b, gamma), a + b + c)
    assert isclose(perimeter_asa(beta, a, gamma), a + b + c)
    assert isclose(perimeter_asa(alpha, c, beta), a + b + c)
```

Na tomto místě si všimněte, že na číslech s plovoucí desetinnou čárkou (typ `float`) nepoužíváme běžnou rovnost `==`. Problém je, že výpočty tohoto typu mají omezenou přesnost: vypočteme-li stejnou hodnotu (v matematickém smyslu) dvěma různými postupy (označme výsledky jako `x` a `y`), může sice platit `x == y`, ale stejně dobře může také nastat `x != y`. To, co by mělo platit pokaždé je, že hodnoty `x` a `y` jsou si „blízko“ – tzn. že, až na chybu způsobenou nepřesností, jsou stejné. Žel, co přesně znamená „blízko“ není přesně definované a záleží od konkrétního výpočtu. Nám bude stačit výchozí definice funkce `isclose` z modulu `math`, která funguje dobře ve většině situací.

Zbývá proceduru `check_triangle` zavolat na vhodně zvolené trojúhelníky. Strany `a` a `b` můžeme volit libovolně:

```
def main():  # demo
    for a in range(1, 6):
        for b in range(1, 6):
```

stranu `c` pak ale musíme zvolit tak, aby byla splněna trojúhelníková nerovnost (jinak budou funkce `perimeter_*` zcela oprávněně počítat nesmysly):

```
            for c in range(max(abs(a - b) + 1, 1), a + b):
                check_triangle(a, b, c)
```

Na závěr si ještě demonstrujeme případ, kdy je řešení trojúhelníku skutečně nepřesné, totiž že výsledek, který obdržíme různými způsoby, může být skutečně různý.

```
    alpha = get_alpha(3, 4, 5)
    beta = get_beta(3, 4, 5)
    assert isclose(perimeter_asa(alpha, 5, beta), 12)
    assert perimeter_asa(alpha, 5, beta) != 12
    assert perimeter_sas(3, 90, 4) == 12
```

Část 2.2: Rozcvička

2.2.1 **[palindrome]** Napište predikát, který ověří, zda je číslo `number` palindrom, zapíšeme-li jej v desítkové soustavě. Palindrom se vyznačuje tím, že je stejný při čtení zleva i zprava.

```
def is_palindrome(number):
    pass
```

2.2.2 **[digits]** Napište funkci `count_digit_in_sequence`, která spočte kolikrát se cifra `digit` vyskytuje v číslech v rozmezí od čísla `low` po číslo `high` včetně. Například cifra 1 se na intervalu od 0 po 13 vyskytuje šestkrát, konkrétně v číslech: 1 10 11 12 13.

```
def count_digit_in_sequence(digit, low, high):
    pass
```

2.2.3 **[fridays]** Napište funkci, která spočítá počet pátků 13. v daném roce `year`. Parametr `day_of_week` udává den v týdnu, na který v daném roce padne 1. leden. Dny v týdnu mají hodnoty 0–6, počínaje pondělím s hodnotou 0.

```
def fridays(year, day_of_week):
    pass
```

Část 2.3: Příklady

2.3.1 **[digit_sum]** Implementujte funkci `power_digit_sum`, která vrátí „speciální“ ciferný součet čísla `number`, který se od běžného ciferného součtu liší tím, že každou cifru před přičtením umocníme na číslo její pozice. Pozice číslujeme zleva, přičemž první má číslo 1. Vstupem funkce `power_digit_sum` bude libovolné nezáporné celé číslo, na výstupu se očekává celé číslo. Výpočet budeme provádět v číselné soustavě se základem 7.
Příklad: Číslo 1234 zapíšeme v sedmičkové soustavě jako $(3412)_7$ – skutečně, $3 \cdot 7^3 + 4 \cdot 7^2 + 1 \cdot 7^1 + 2 \cdot 7^0 = 1029 + 196 + 7 + 2 = 1234$. Proto

`power_digit_sum(1234)` získáme jako $3^1 + 4^2 + 1^3 + 2^4 = 36$.

```
def power_digit_sum(number):
    pass
```

2.3.2 [`joined`] Napište funkci, která vytvoří číslo zřetězením `count` po sobě jdoucích kladných čísel počínaje zadaným číslem `start`. Tyto čísla zřetězte vyjádřená v binární soustavě. Například volání `joined(1, 3)` zřetězí sekvenci $1_2 = 1$, $(10)_2 = 2$ a $(11)_2 = 3$ vrátí číslo $(11011)_2 = 27$. V Pythonu lze binární čísla přímo zapisovat v tomto tvaru: `0b11011` (podobně lze stejné číslo zapsat v šestnáctkové soustavě zápisem `0x1b` nebo osmičkové jako `0o33`).

```
def joined(start, count):
    pass
```

2.3.3 [`savings`] Vaším úkolem bude spočítat, kolik následujících let Vám vydrží úspory o hodnotě `savings` v bance. Na konci každého roku Vám banka úročí obnos na účtu úrokovou sazbou `interest_rate` (zadanou v procentech). Dále, abyste pokryli své životní náklady, na začátku každého roku vyberete z účtu obnos `withdraw`, který se každým rokem zvyšuje o inflaci `inflation` (opět zadanou v procentech). Vybíraný obnos se po započítání inflace zaokrouhluje dolů na celá čísla. Úroková sazba a inflace jsou konstantní a meziročně se nemění. Po zúročení banka celkovou částku zaokrouhluje dolů na celá čísla.
Příklad: při počátečním obnosu 100000 korun, ročních výdajích 42000 korun, úrokové sazbě 3,2 % a inflaci 1,5 % bude po prvním roce na účtu $(100000 - 42000) \cdot 1.032 = 59856$. Další rok se výdaje zvýší o inflaci na $42000 \cdot 1.015 = 42630$.

```
def savings_years(savings, interest_rate, inflation, withdraw):
    pass
```

2.3.4 [`delete`] Napište funkci `delete_to_maximal`, která pro dané číslo `number` najde největší možné číslo, které lze získat smazáním jedné desítkové cifry.

```
def delete_to_maximal(number):
    pass
```

Napište funkci `delete_k_to_maximal`, která pro dané číslo `number` najde největší možné číslo, které lze získat smazáním (vynecháním) `k` desítkových cifer.

```
def delete_k_to_maximal(number, k):
    pass
```

2.3.5 [`maximum`] Napište funkci, která najde celé číslo `x`, které leží mezi hodnotami `low` a `high` (včetně), a pro které vrátí funkce `poly` maximální hodnotu (tzn. libovolné x takové, že pro všechny x' platí $f(x) \geq f(x')$, kde f je funkce, kterou počítá podprogram `poly`).

```
def poly(x):
    return 10 + 30 * x - 15 * x ** 3 + x ** 5

def maximum(low, high):
    pass
```

2.3.6 [`credit`] Napište predikát, který ověří, zda je číslo korektní číslo platební karty. Číslo platební karty ověříte podle Luhnova algoritmu:

1. zdvojnásobte hodnotu každé druhé cifry zprava; je-li výsledek větší než 9, odečtěte od něj hodnotu 9,
2. sečtěte všechna takto získaná čísla a cifry na lichých pozicích (kromě první cifry zprava, která slouží jako kontrolní součet),
3. je-li potřeba přičíst přesně hodnotu kontrolní cifry, aby byl součet dělitelný 10, je číslo platné.

Například pro číslo 28316 je kontrolní cifra 6 a součet je: $2 + (2 \cdot 8 - 9) + 3 + 2 \cdot 1 = 2 + 7 + 3 + 2 = 14$. K výsledku je potřeba přičíst 6, aby byl násobkem 10, což je přesně hodnota kontrolní cifry. Číslo karty je tedy platné.

```
def is_valid_card(number):
    pass
```

2.3.7 [`cards`] Napište predikát `is_visa`, který je pravdivý, reprezentuje-li číslo `number` platné číslo platební karty VISA, tj. začíná cifrou 4, má 13, 16, nebo 19 cifer a zároveň je platným číslem platební karty (viz příklad `credit`).

```
def is_visa(number):
    pass
```

Dále napište predikát `is_mastercard`, který je pravdivý, reprezentuje-li číslo `number` platné číslo platební karty MasterCard, tj. začíná prefixem 50–55, nebo 22100–27209, má 16 cifer a zároveň je platným číslem platební karty.

```
def is_mastercard(number):
    pass
```

2.3.8 [`fraction`] V této úloze bude Vaším úkolem získat hodnotu `index`-tého koeficientu řetězového zlomku pro racionální číslo s čitatelem `nom` a jmenovatelem `denom`.
Řetězový zlomek je forma reprezentace čísla jako součet celého čísla a_0 a převrácené hodnoty jiného čísla, které opět reprezentujeme součtem celého čísla a_1 a další převrácené hodnoty. Celá čísla a_n postupně tvoří řadu koeficientů řetězového zlomku.
Například řetězový zlomek $4 + (1/(2 + 1/(6 + (1/7))))$ reprezentuje číslo $415/93$ a jeho koeficienty jsou 4, 2, 6 a 7.
Koeficienty řetězového zlomku pro číslo n můžete získat iterativním postupem:

1. Rozdělte číslo n na jeho celočíselnou část p a zlomkovou část q. Číslo p přímo udává první koeficient posloupnosti, tzn. a_0, zbytek koeficientů je odvozen od q (viz další krok). Posloupnost má tedy tvar: $p; a_1, a_2, a_3, \ldots$.

2. Pro získání dalšího koeficientu opakujte 1. krok s převrácenou hodnotou zlomkové části $(1/q)$.

```
def continued_fraction(nom, denom, index):
    pass
```

2.3.9 [`workdays`] Napište funkci, která zjistí, kolik bude pracovních dnů v roce `year`, přičemž parametr `day_of_week` udává, na který den v týdnu v tomto roce padne první leden. Dny v týdnu mají hodnoty 0–6 počínaje pondělím s hodnotou 0.
České státní svátky jsou:

datum	svátek
1.1.	Den obnovy samostatného českého státu
—	Velký pátek
—	Velikonoční pondělí
1.5.	Svátek práce
8.5.	Den vítězství
5.7.	Den slovanských věrozvěstů Cyrila a Metoděje
6.7.	Den upálení mistra Jana Husa
28.9.	Den české státnosti
28.10.	Den vzniku samostatného československého státu
17.11.	Den boje za svobodu a demokracii
24.12.	Štědrý den
25.12.	1. svátek vánoční
26.12.	2. svátek vánoční

```
def workdays(year, day_of_week):
    pass
```

Část 2.4: Bonusy

2.4.1 [bisection] Napište funkci bisection, která aproximuje kořen spojité funkce f (předané parametrem fun) s chybou menší než epsilon na zadaném intervalu od low po high včetně. Algoritmus bisekce předpokládá, že v zadaném intervalu se nachází právě jedno řešení.
Při hledání řešení postupujte následovně:

1. spočtěte hodnotu funkce pro bod uprostřed intervalu, a je-li výsledek v rozsahu povolené chyby, vraťte tento bod,
2. jinak spočtěte hodnoty funkce v hraničních bodech intervalu a zjistěte, ve které polovině má funkce kořen,
3. opakujte výpočet s vybranou polovinou jako s novým intervalem.

Chybu e spočtete v bodě x jako $e = |f(x)|$.
Poznámka: funkci předanou parametrem můžete v Pythonu normálně volat jako libovolnou jinou funkci.

```
def bisection(fun, low, high, eps):
    pass

def fun_a(x):
    return x ** 2 - 3

def fun_b(x):
    return x ** 3 - x - 1

def fun_c(x):
    return sqrt(x) / x - x ** 3 + 5
```

Část 2.5: Řešení

2.5.1 [palindrome.sol]

```
def reverse(number):
    result = 0
    while number > 0:
        digit = number % 10
        result = result * 10 + digit
        number = number // 10
    return result

def is_palindrome(number):
    return number == reverse(number)
```

2.5.2 [digits.sol]

```
def count_digit_in_sequence(digit, low, high):
    count = 0
    if low == 0 and digit == 0:
        count += 1
    for number in range(low, high + 1):
        while number > 0:
            if digit == number % 10:
                count += 1
            number = number // 10
    return count
```

2.5.3 [fridays.sol]

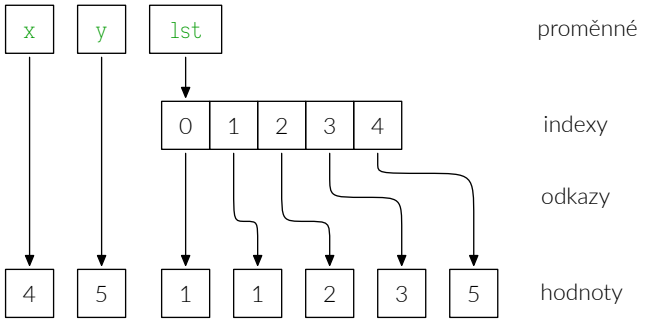
```
def is_leap(year):
    if (year % 400) == 0:
        return True
    if (year % 4) == 0 and not (year % 100) == 0:
        return True
    return False

def days_per_month(year, month):
    if month == 2:
        return 29 if is_leap(year) else 28
    if month == 4 or month == 6 or month == 9 or month == 11:
        return 30
    return 31

def is_friday(day_of_week):
    return day_of_week == 4

def fridays(year, day_of_week):
    count = 0
    for month in range(1, 13):
        days = days_per_month(year, month)
        for day in range(1, days + 1):
            if is_friday(day_of_week) and day == 13:
                count += 1
            day_of_week = (day_of_week + 1) % 7
    return count
```

Část 3: Seznamy a n-tice

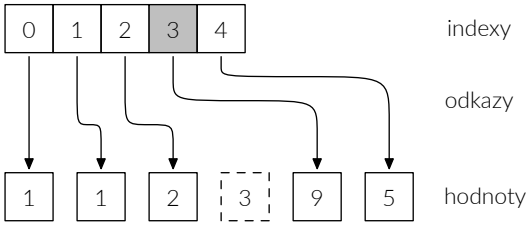
Tento týden se budeme poprvé zabývat složenými datovými typy, konkrétně těmi, které reprezentují sekvence: seznamy a uspořádanými n-ticemi. Prozatím jsme se setkali pouze s hodnotami tzv. skalárních typů: zejména int, float, bool. Použití těchto datových typů nám umožňovalo pamatovat si fixní množství dat: například při výpočtu n-tého prvku Fibonacciho posloupnosti jsme si potřebovali pamatovat tři čísla, které jsme měli uložené ve třech proměnných. To, co nám ale hodnoty tohoto charakteru neumožňovaly, bylo například zapamatovat si všechny dosud spočtené prvky. Zkuste se zamyslet, co by se stalo, kdybychom chtěli vyčíslit n-tý prvek posloupnosti zadané třeba takto (v OEIS nalezne pod číslem A165552):

$$a_1 = 1$$
$$a_n = \sum_{k=1}^{n-1} d(k,n) \cdot a_k$$

kde $d(k,n) = k$ když k dělí n a 0 jinak. Tady už nestačí pamatovat si poslední dva prvky – co je horší, nestačí nám žádný konstantní počet proměnných: potřebujeme jich tolik, kolikátý prvek chceme spočítat. To je přesně situace, kdy lze použít sekvenční datový typ: hodnota sekvenčního typu se skládá z libovolného počtu jiných hodnot, očíslovaných po sobě jdoucími celými čísly. Číslu, které popisuje pozici „vnitřní“ hodnoty, říkáme index, a podobně jak tomu bylo s indexovými proměnnými, první prvek má číslo (index) 0.
V Pythonu existují dva základní sekvenční typy: první je uspořádaná n-tice (anglicky tuple, případně n-tuple), ten druhý pak seznam (anglicky list). Hodnoty obou těchto typů mají vnitřní strukturu – vzpomeňte si, že proměnné váží hodnoty ke jménům: sekvence obdobně váže hodnoty k indexům (celým číslům). Seznam a n-tice se tedy chovají podobně, jako bychom měli proměnné pojmenované lst[0], lst[1], lst[2], atd. K těmto pomyslným proměnným můžeme navíc přistupovat nepřímo: jako index můžeme použít nejen konstantu, ale libovolné jiné číslo v programu – klidně třeba hodnotu proměnné, nebo i výraz, např. lst[i] nebo lst[i + 1].



Obdoba použití proměnné (např. ve výrazu `x + 1`, který se vyhodnotí na 5) je indexace seznamu, např. `lst[0]` se vyhodnotí na 1, `lst[2] + 1` se vyhodnotí na 3, atp. Výraz `lst[x]` se vyhodnotí na 5.
Máme-li hodnotu typu seznam, můžeme navíc měnit na kterou hodnotu ten-který index odkazuje, a tato změna odkazu je zcela analogická přiřazení do proměnné. Toto vnitřní přiřazení zapisujeme podobně jako to běžné, např. `lst[3] = 9`, a má obdobný efekt (na obrázku je již pouze hodnota typu seznam z proměnné `lst`):



Uspořádaná n-tice se pak od seznamu liší zejména tím, že nemá vnitřní přiřazení: přiřazení hodnot indexům je tedy pevně dané při vytvoření n-tice a nelze jej již dále v programu měnit. Zároveň do n-tice nelze po jejím vzniku přidávat nové indexy (těm by totiž bylo potřeba přiřadit hodnoty, a to v n-tici nelze).
Použití seznamů a n-tic si dále demonstrujeme na několika ukázkách:

1. `statistics.py` – iterace a indexace seznamů
2. `fibonacci.py` – konstrukce nového seznamu
3. `sequence.py` – výpočet výše uvedené posloupnosti
4. `points.py` – práce s n-ticemi a seznamy n-tic
5. `rotate.py` – mutace (vnitřní přiřazení) na seznamech

Řešené příklady:

1. `predicate.py` – predikáty na seznamech
2. `explosion.py` – filtrování seznamu podle kritéria

Základní úlohy:

1. `cartesian.py` – výpočet kartézského součinu
2. `concat.py` – spojování vnořených seznamů
3. `fraction.py` – vyhodnocení řetězového zlomku
4. `histogram.py` – četnost hodnot ve vstupním seznamu
5. `line_length.py` – délka lomené čáry
6. `merge.py` – sloučení dvou uspořádaných seznamů
7. `quiz.py` – vyhodnocení multiple-choice testu
8. `cellular.py` – jednoduché buněčné automaty
9. `rectangles.py` – překryv obdélníků v zadaném seznamu
10. `numbers.py` – převod číselných soustav
11. `right_cellular.py` – buněčný automat in situ

Bonusové úlohy:

1. `partition.py` – přerozdělení seznamu podle velikosti

Část 3.1: Ukázky

3.1.1 [`statistics`] Tato ukázka demonstruje základní použití seznamů: zejména jejich indexaci a iteraci. Oba tyto koncepty si demonstrujeme na výpočtu jednoduchých statistik nad prvky předem daného seznamu: průměru, mediánu a směrodatné odchylky.

Jako první statistiku vypočteme průměr, který získáme jako podíl součtu všech prvků vstupního seznamu a jeho délky. Protože obě tyto operace jsou v Pythonu zabudované, je definice velice jednoduchá:

```
def average(data):
    return sum(data) / len(data)
```

Protože indexace je v určitém smyslu jednodušší než iterace, budeme pokračovat výpočtem mediánu: medián je hodnota, která se objeví v uspořádaném souboru čísel uprostřed. Tento soubor dat budeme reprezentovat neprázdným seznamem:

```
def median(data):
```

První problém, který musíme vyřešit, je ona uspořádanost: naštěstí je to v Pythonu problém velice jednoduchý, protože máme k dispozici vestavěnou (čistou) funkci `sorted`, která vytvoří ze vstupního seznamu nový, vzestupně uspořádaný:

```
    data = sorted(data)
```

Zde je možná vhodné připomenout, že přiřazení do proměnné mění vazbu – to, na kterou hodnotu proměnná odkazuje – a nemá vliv na samotnou hodnotu. Vstupní seznam tedy není tímto přiřazením nijak dotčen, a funkce `median` je funkcí čistou.
Dalším krokem je výpočet indexu, na kterém nalezneme medián: tady nastávají dvě možnosti: buď je seznam liché, nebo sudé délky. Délku seznamu zjistíme vestavěnou (čistou) funkcí `len`:

```
    if len(data) % 2 == 1:
```

Případ liché délky je jednodušší, proto jej vyřešíme první. V tomto případě existuje skutečný prostřední prvek, a my pouze vrátíme jeho hodnotu. Celočíselné dělení dvěma nám dá právě ten správný index – přesvědčte se o tom!

```
        return data[len(data) // 2]
    else:
```

V opačném případě je seznam sudé délky (prázdný seznam neuvažujeme, nevyhovuje vstupní podmínce). Běžná definice mediánu v tomto případě říká, že výsledkem má být aritmetický průměr obou „prostředních“ hodnot (těch, které jsou nejblíže pomyslnému středu, který se nachází přesně mezi nimi).

```
        return (data[len(data) // 2] + data[len(data) // 2 - 1]) / 2
```

Poslední a nejsložitější statistikou je tzv. směrodatná odchylka s. Tuto spočítáme jako odmocninu tzv. rozptylu s^2, který je popsaný následovným vztahem (n je počet prvků, x_i jsou jednotlivé prvky a m je průměr):

$$s^2 = 1/(n-1)\sum_{i=1}^{n}(x_i - m)^2$$

```
def stddev(data):
```

Pro výpočet jednotlivých členů budeme potřebovat průměr, který již máme implementovaný výše. Dále si nachystáme střadač (akumulátor), do kterého sečteme jednotlivé kvadratické odchylky $(x_i - m)^2$:

```
    mean = average(data)
    square_error_sum = 0
```

Chceme-li pro každý prvek seznamu provést nějakou akci nebo výpočet, použijeme k tomu cyklus. Mohli bychom samozřejmě použít konstrukce, které již známe: indexovou proměnnou, cyklus tvaru `for i in range(n)`, funkci `len` a indexaci seznamu `data`. V případě, že ale indexovou proměnnou nepotřebujeme k ničemu jinému, než indexaci jednoho seznamu, lze použít mnohem úspornější a čitelnější zápis:

```
    for x_i in data:
```

V těle takovéhoto cyklu máme v proměnné `x_i` uloženy přímo hodnoty ze seznamu `data`, nemusíme tedy vůbec indexovat.

```
        square_error_sum += (x_i - mean) ** 2
```

Protože rozptyl (variance) je vlastně střední (průměrná) kvadratická odchylka s drobnou korekcí, vypočteme…

```
    variance = square_error_sum / (len(data) - 1)
```

… a celkový výsledek získáme jako odmocninu rozptylu:

```
    return sqrt(variance)
```

Konečně funkčnost ověříme na několika jednoduchých příkladech.

3.1.2 **[fibonacci]** V této ukázce si demonstrujeme vytváření seznamu, který bude výstupem (čisté) funkce `fib`. Seznam bude obsahovat prvních `n` členů Fibonacciho posloupnosti, které vypočteme už známým postupem (viz též `fibonacci.py` z části 1).

```
def fib(n):
```

Seznam budeme budovat v cyklu. Proměnné `a` a `b` již nebudeme potřebovat, protože máme k dispozici celý seznam, bylo by tedy nehospodárné pamatovat si dva prvky ještě jednou, a to jak z pohledu využití paměti (i když v tomto případě by to nebyl velký prohřešek), ale zejména z pohledu čitelnosti programu. Většinou je nežádoucí uchovávat stejnou informaci na více místech, není potom často jasné, jsou-li obě „místa" plně ekvivalentní, a pokud ano, tak že se chybou v programu nemůžou rozejít.

```
    out = [1, 1]

    for i in range(n - 2):
```

Pro výpočet dalšího Fibonacciho čísla využijeme zápis pro indexování seznamu od konce: je-li použitý index záporný, automaticky se k němu přičte délka indexovaného seznamu, tzn. `out[-2]` je totéž jako `out[len(out) - 2]`. Rozmyslete si, že tento výraz skutečně popisuje předposlední prvek seznamu `out`!

```
        value = out[-1] + out[-2]
```

Přidání na konec existujícího seznamu provedeme voláním metody `append`. Metody jsou podprogramy, které často leží někde mezi procedurou a čistou funkcí (nicméně i metody můžou být čisté, a naopak můžou mít i charakter procedury). Mají navíc ale jednu speciální vlastnost, v podobě význačného parametru, který píšeme při volání před jejich jméno. Následovné volání `append` má tedy dva parametry – `out` a `value`.

```
        out.append(value)
```

Nyní stojíme před drobným problémem: mohlo se stát, že volající si vyžádal méně než dva prvky posloupnosti, ale my jsme pro pohodlí výpočtu do seznamu vložili první dvě hodnoty. Jedna možnost řešení byla hned na začátku funkce ověřit, zda není `n` nula nebo jedna, a rovnou vrátit příslušný seznam (`[]` nebo `[1]`). My tento problém místo toho využijeme, abychom si ukázali, jak ze stávajícího seznamu hodnoty navíc odstranit. Rozmyslete si, že tělo cyklu se provede skutečně právě jednou, je-li `n = 1` a dvakrát, je-li `n = 0`. Metoda `pop` (bez dalších parametrů) odstraní ze seznamu poslední prvek.

```
    while len(out) > n:
        out.pop()

    return out
```

Jako obvykle, program zakončíme několika testy, abychom se ujistili, že námi implementovaná funkce pracuje (aspoň v některých případech) správně.

```
def main():  # demo
    assert fib(0) == []
    assert fib(1) == [1]
    assert fib(2) == [1, 1]
    assert fib(3) == [1, 1, 2]
    assert fib(5) == [1, 1, 2, 3, 5]
    assert fib(9) == [1, 1, 2, 3, 5, 8, 13, 21, 34]
```

3.1.3 **[sequence]** V předchozích dvou ukázkách byl seznam vstupem nebo výstupem funkce. Nyní se podíváme na funkci, která má na vstupu i výstupu pouze jediné číslo, ale seznam využije pro svůj výpočet. Vrátíme se k výpočtu n-tého prvku posloupnosti, podobně jak tomu bylo v příkladech z první části. Vyčíslovat budeme posloupnost, se kterou jsme se setkali v úvodu:

$$a_1 = 1$$
$$a_n = \sum_{k=1}^{n-1} d(k,n) \cdot a_k$$

kde $d(k,n) = k$ když k dělí n a 0 jinak. Implementace bude formou čisté funkce.

```
def sequence(position):
```

Podobně jako při výpočtu `fib` v předchozí ukázce si vytvoříme proměnnou, ve které budeme mít uložen dosud vypočtený prefix posloupnosti. V tomto případě to ale není proto, abychom jej mohli použít jako návratovou hodnotu, ale čistě pro naše interní účely.

```
    seq = [1]
```

Do seznamu `seq` budeme v cyklu přidávat nové prvky posloupnosti, v každé iteraci jeden. Potřebujeme provést `n - 1` iterací (jeden prvek už v seznamu máme). Nabízí se dvě možnosti: `for` cyklus, podobně jako v předchozím, nebo `while` cyklus. Protože potřebujeme indexovat od 1, není `for` cyklus příliš pohodlný, navíc u `while` cyklu je na pohled zřejmé, že má správný počet iterací, přikloníme se k této variantě:

```
    while len(seq) < position:
```

Do proměnné `n` si uložíme index právě počítaného prvku (číslováno od 1).

```
        n = len(seq) + 1
```

Nyní potřebujeme vypočítat hodnotu, kterou přidáme na konec seznamu. Nachystáme si střadač `sum`, ve kterém budeme počítat definiční sumu, a indexovou proměnnou `k` (která bude indexovat už vypočtené hodnoty počínaje první s indexem 1).

```
        sum = 0
        k = 1
```

Samotný výpočet sumy provedeme opět v cyklu.

```
        while k < n:
            if n % k == 0:
                sum += k * seq[k - 1]
            k += 1
```

V proměnné `sum` máme nyní další prvek posloupnosti, který si přidáme do seznamu `seq` a pokračujeme další iterací.

```
        seq.append(sum)
```

Seznam `seq` byl čistě pomocný – umožnil nám provést výpočet. Výsledkem funkce je ale jediné číslo, totiž `position`-tý prvek posloupnosti. Ten nalezneme na indexu `position - 1` (seznamy indexujeme od nuly, první prvek je tedy na indexu 0, atd.).

```
    return seq[position - 1]
```

Hodnoty pro testy pochází z databáze OEIS.

```
def main():  # demo
    from_oeis = [1, 1, 1, 3, 1, 6, 1, 15, 4, 8, 1, 54, 1, 10, 9,
                 135, 1, 78, 1, 100, 11, 14, 1, 822, 6, 16, 40]
    for i in range(len(from_oeis)):
        assert sequence(i + 1) == from_oeis[i]
```

3.1.4 [points] Uspořádané n-tice jsou v Pythonu velmi podobné seznamům: lze je indexovat a iterovat, ptát se na jejich délku funkcí len, ale také například vytvářet (n+m)-tice spojením n-tice s m-ticí. Jak jsme již zmiňovali v úvodu, zásadní rozdíl je, že n-tice nemá vnitřní přiřazení a nelze ji tedy po vytvoření měnit.

Ve skutečnosti ale n-tice používáme v programech výrazně jinak než seznamy, přestože mají velmi podobnou strukturu a operace. V typickém použití obsahuje seznam pouze jeden typ hodnot, ale počet hodnot je variabilní. N-tice se chovají opačně: je běžné, že obsahují hodnoty různých typů (ale všechny n-tice daného určení mají na stejném indexu stejný typ) a mají fixní počet položek.

Tento princip si demonstrujeme na příkladu, kde budeme pracovat s barevnými body v rovině. Body budeme reprezentovat jako trojice (souřadnice x, souřadnice y, barva). Každá n-tice, která reprezentuje bod, bude mít právě tuto strukturu, a bude mít vždy 3 složky (budeme tedy mluvit o trojicích). Navíc bude platit, že první dvě složky budou vždy čísla, a třetí složka bude vždy řetězec.

V principu můžeme k těmto složkám přistupovat indexací, ale existuje i mnohem lepší zápis – rozbalení n-tice do proměnných. Srovnejte si zápis `x, y, colour = point`, kde dále pracujeme se jmény x, y a colour, oproti point[0] a point[1] pro souřadnice a point[2] pro barvu. Pro srovnání si můžete v tomto příkladu přepsat všechny rozbalení trojic na indexaci a zvážit, co se Vám lépe čte.

Jako první si definujeme jednoduchou (čistou) funkci, která spočte Euklidovskou vzdálenost dvou bodů (která samozřejmě nezávisí na jejich barvě).

Poznámka: použití _ jako názvu proměnné není z pohledu Pythonu ničím zvláštním, jedná se o identifikátor jako kterýkoliv jiný. Nicméně jeho použitím indikujeme budoucím čtenářům, že hodnotu této proměnné nehodláme používat, a domluvou se tedy jedná o zástupný symbol.

```
def distance(a, b):
    a_x, a_y, _ = a
    b_x, b_y, _ = b
    return sqrt((a_x - b_x) ** 2 + (a_y - b_y) ** 2)
```

Dále si definujeme funkci, která v neprázdném seznamu najde barvu „nejlevějšího" bodu (takového, který má nejmenší x-ovou souřadnici).

```
def leftmost_colour(points):
    x_min, _, result = points[0]

    for x, _, colour in points:
        if x < x_min:
            x_min = x
            result = colour

    return result
```

Dále si definujeme čistou funkci, která dostane jako parametry seznam bodů points a barvu colour, a jejím výsledkem bude bod, který se nachází v těžišti soustavy bodů dané barvy (a který bude stejné barvy). Vstupní podmínkou je, že points obsahuje aspoň jeden bod barvy colour.

```
def center_of_gravity(points, colour):
    total_x = 0
    total_y = 0
    count = 0
    for p_x, p_y, p_colour in points:
        if colour == p_colour:
            total_x += p_x
            total_y += p_y
            count += 1

    return (total_x / count, total_y / count, colour)
```

Jako poslední si definujeme (opět čistou) funkci, která spočítá průměrnou vzdálenost bodů různé barvy. Vstupní podmínkou je, že seznam points musí obsahovat aspoň dva různobarevné body.

```
def average_nonmatching_distance(points):
    total = 0
    pairs = 0

    for i in range(len(points)):
        for j in range(i):
            _, _, i_colour = points[i]
            _, _, j_colour = points[j]
            if i_colour != j_colour:
                total += distance(points[i], points[j])
                pairs += 1

    return total / pairs
```

Testy jsou tentokrát rozsáhlejší, protože jsme definovali větší počet funkcí. Pro úsporu horizontálního místa některé testy používají lokální aliasy pro funkce, např. dist = average_nonmatching_distance – takové přiřazení znamená, že dist je (lokální) synonymum pro average_nonmatching_distance.

```
def main():  # demo
    test_distance()
    test_leftmost_colour()
    test_center_of_gravity()
    test_average_nonmatching_distance()

def test_average_nonmatching_distance():
    r00 = (0, 0, "red")
    r10 = (1, 0, "red")
    b20 = (2, 0, "blue")
    b10 = (1, 0, "blue")
    g30 = (3, 0, "green")
    y20 = (2, 0, "yellow")
    w40 = (4, 0, "white")
    dist = average_nonmatching_distance

    assert dist([r00, b20]) == 2
    assert dist([b10, r00, b20]) == 1.5
    assert dist([r00, b20, b10, g30]) == 1.8
    assert dist([r00, b20, g30]) == 2
    assert dist([r00, b20, b10, r10]) == 1
    assert dist([r00, b10, g30, y20, w40]) == 2

def test_center_of_gravity():
    r00 = (0, 0, "red")
    r22 = (2, 2, "red")
    b20 = (2, 0, "blue")
    b02 = (0, 2, "blue")
    cog = center_of_gravity

    assert cog([r00], "red") == (0, 0, "red")
    assert cog([r00, r22], "red") == (1, 1, "red")
    assert cog([b20, b02], "blue") == (1, 1, "blue")
    assert cog([r00, b02, b20, r22], "red") == (1, 1, "red")
    assert cog([r00, b02, b20, r22], "blue") == (1, 1, "blue")

    g68 = (6, 8, "green")
    g00 = (0, 0, "green")
    g64 = (6, 4, "green")
    g86 = (8, 6, "green")
    green = [g68, g00, g64, g86]

    assert cog([g68, g00, g64], "green") == (4, 4, "green")
```

```
    assert cog(green, "green") == (5, 4.5, "green")
    assert cog([r22, b20] + green, "green") == (5, 4.5, "green")

def test_leftmost_colour():
    p1 = (0, 0, "white")
    p2 = (-2, 15, "red")
    p3 = (13, -15, "yellow")
    p4 = (0, 1, "black")

    assert leftmost_colour([p1]) == "white"
    assert leftmost_colour([p3]) == "yellow"
    assert leftmost_colour([p1, p3]) == "white"
    assert leftmost_colour([p1, p3, p4, p2]) == "red"
    assert leftmost_colour([p1, p4]) == "white"
    assert leftmost_colour([p3, p4]) == "black"

def test_distance():
    p1 = (0, 0, "white")
    p2 = (1, 0, "red")

    assert distance(p1, (0, -1, "red")) == 1
    assert distance(p2, p1) == 1
    assert distance(p1, p2) == 1
    assert distance(p1, (2, 0, "black")) == 2
    assert distance(p1, (3, 4, "black")) == 5
    assert distance((-3, -4, "black"), p1) == 5
```

3.1.5 [rotate] V poslední ukázce pro tento týden se budeme zabývat vnitřním přiřazením, tzn. změnou samotné hodnoty typu seznam (změnou vnitřních vazeb indexů na hodnoty). Po delší době tedy budeme implementovat proceduru (podprogram, kterého hlavním smyslem je provést nějakou akci – v tomto případě pozměnit existující hodnotu). Tato procedura provede rotaci seznamu (na místě) o zadaný počet prvků. Např. rotací seznamu `[1, 2, 3, 4]`:

- o jedna doprava dostaneme seznam `[4, 1, 2, 3]`,
- o dva doprava seznam `[3, 4, 1, 2]`,
- o dva doleva tentýž seznam `[3, 4, 1, 2]` a konečně,
- o jedna doleva seznam `[2, 3, 4, 1]`.

Směr rotace určíme dle znaménka: kladná čísla budou rotovat doprava, záporná doleva.
Možností, jak „in situ" rotaci seznamu implementovat je několik, my si ukážeme dvě. První je konceptuálně nejjednodušší, ale nepříliš efektivní: jako základní operaci používá posuv o jedna doleva nebo doprava. Každá rotace o jedničku musí projít celý seznam, posuvy o větší počet prvků budou tedy procházet celý seznam mnohokrát – proto je tato implementace neefektivní.

```
def rotate_naive(lst, amount):
    while amount != 0:
        if amount < 0:
```

Posuv doleva implementujeme tak, že ze seznamu odstraníme první prvek (volání metody `pop(0)`), který vzápětí přidáme na konec (volání `append`).

```
            lst.append(lst.pop(0))
            amount += 1
        else:
```

Posuv doprava je analogický, ale odstraníme prvek z konce (volání `pop`) a metodou `insert` jej vložíme na začátek.

```
            lst.insert(0, lst.pop())
            amount -= 1
```

Jiná možnost je prvky rovnou posouvat na správné místo v seznamu (použitím vnitřního přiřazení), musíme si ale pamatovat prvky, které takto přepisujeme, a to až do doby, než je můžeme samotné přesunout na jejich cílovou pozici. Takových prvků může být najednou až tolik, jaká je velikost posuvu. Každý prvek ale přesouváme nejvýše jednou (bez ohledu na velikost posuvu), celkový počet operací je tedy výrazně menší než v předchozí implementaci.

```
def rotate_smart(lst, amount):
```

Pro jednoduchost implementujeme pouze posuvy doprava – posuvy doleva by byly analogické. Díky tomu je tato implementace při rotacích doleva méně efektivní (malé otočení doleva je totéž jako velké otočení doprava). V proměnné `backup` si budeme pamatovat ty prvky, které budeme v nejbližší době ukládat na své cílové pozice (po prvních `amount` přesunech zde budou uloženy právě ty prvky, které aktuálně v `lst` dočasně chybí).

```
    amount = amount % len(lst)
    backup = lst[0:amount]

    for i in range(len(lst)):
```

Do `target` spočteme cílové políčko pro další přesun, a prvek zde umístěný prohodíme s příslušným prvkem v seznamu `backup`. Na pozici `i % amount` seznamu `backup` se nachází prvek, který byl v původním seznamu na pozici `i`, a tedy je to ten prvek, který potřebujeme umístit do `lst[target]`. Jejich prohozením se do `backup[i % amount]` dostane prvek, který byl v původním seznamu na pozici `target` (tj. `i + amount`) a tedy se k němu vrátíme po dalších `amount` iteracích (`(i + amount) % amount == i % amount`).

```
        target = (i + amount) % len(lst)
        displaced = backup[i % amount]
        backup[i % amount] = lst[target]
        lst[target] = displaced
```

Protože máme dvě implementace stejné funkce, testy si parametrizujeme konkrétní implementací, aby nám stačilo napsat je jednou. Za parametr `rotate` se postupně doplní `rotate_naive` a `rotate_smart`.

```
def check_rotate(rotate):
    lst = [1, 2, 3, 4]
    rotate(lst, 1)
    assert lst == [4, 1, 2, 3]
    rotate(lst, -1)
    assert lst == [1, 2, 3, 4]
    rotate(lst, -2)
    assert lst == [3, 4, 1, 2]
    rotate(lst, -2)
    assert lst == [1, 2, 3, 4]
    lst.append(5)
    rotate(lst, 3)
    assert lst == [3, 4, 5, 1, 2]

def main():  # demo
    check_rotate(rotate_naive)
    check_rotate(rotate_smart)
```

Část 3.2: Rozcvička

3.2.1 [predicates] Napište predikát `all_greater_than`, který je pravdivý, právě když jsou všechna čísla v seznamu `sequence` větší než `n`.

```
def all_greater_than(sequence, n):
    pass
```

Dále napište predikát `any_even`, který je pravdivý, je-li v seznamu `sequence` aspoň jedno sudé číslo.

```
def any_even(sequence):
    pass
```

3.2.2 [explosion] Napište (čistou) funkci `survivors`, ktorá ze vstupního seznamu `objects` spočítá nový seznam, který bude obsahovat všechny prvky z `objects`, které jsou dostatečně vzdálené (dále než `radius`) od

bodu `center`.
Můžete si představit, že funkce implementuje herní mechaniku, kdy v bodě `center` nastala exploze tvaru koule, která zničila vše uvnitř poloměru `radius`, a funkce `survivors` vrátí všechny objekty, které explozi přežily.
Prvky parametru `objects` a parametr `center` jsou uspořádané trojice, které reprezentují body v prostoru.

```
def distance(a, b):
    pass

def survivors(objects, center, radius):
    pass
```

Část 3.3: Příklady

3.3.1 [`cartesian`] Napište funkci, která vrátí kartézský součin seznamů `x` a `y`, jako nový seznam dvojic.

```
def cartesian(x, y):
    pass
```

3.3.2 [`concat`] Napište funkci, která zploští seznam seznamů do jednoho nového seznamu tak, že vnořené seznamy pospojuje za sebe.

```
def concat(lists):
    pass
```

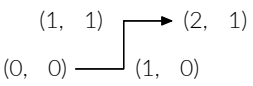
3.3.3 [`fraction`] Stejně jako v `02/fraction.py` budete v této úloze pracovat s řetězovým zlomkem. Tentokrát implementujeme převod opačným směrem, na vstupu bude seznam koeficientů řetězového zlomku, a výstupem bude zlomek klasický.
Naprogramujte tedy čistou funkci `continued_fraction`, ktorá dostane jako parametr seznam koeficientů a vrátí zlomek ve tvaru `(numerator, denominator)`.

```
def continued_fraction(coefficients):
    pass
```

3.3.4 [`histogram`] Napište (čistou) funkci, která pro zadaný seznam nezáporných čísel `data` vrátí nový seznam obsahující dvojice – číslo a jeho četnost. Výstupní seznam musí být seřazený vzestupně dle první složky. Můžete předpokládat, že v `data` se nachází pouze celá čísla z rozsahu [0, 100] (včetně).

```
def histogram(data):
    pass
```

3.3.5 [`line_length`] Napište čistou funkci, která dostane na vstup seznam bodů v rovině (tj. seznam dvojic čísel) a vrátí délku lomené čáry, která těmito body prochází (tzn. takové, která vznikne spojením každých dvou sousedních bodů seznamu úsečkou).
Například seznam `[(0, 0), (1, 0), (1, 1), (2, 1)]` definuje tuto lomenou čáru:



složenou ze tří segmentů (úseček) velikosti 1. Její délka je 3.

```
def point_distance(a, b):
    pass

def length(points):
    pass
```

3.3.6 [`merge`] Naprogramujte (čistou) funkci, která ze dvou vzestupně seřazených seznamů čísel `a`, `b` vytvoří nový vzestupně seřazený seznam, který bude obsahovat všechny prvky z `a` i `b`. Nezapomeňte, že nesmíte modifikovat vstupní seznamy (jinak by funkce nebyla čistá).
Pokuste se funkci naprogramovat efektivně.
Pozor: Při řešení nepoužívejte zabudované funkce pro řazení (funkce `merge` je mimo jiné pomocná funkce algoritmu merge sort, bylo by tedy absurdní zde sekvenci celou řadit). Řešení postavené na zabudované řadící funkci nebude uznáno jako příprava.

```
def merge(a, b):
    pass
```

3.3.7 [`quiz`] Naprogramujte funkci `mark_points`, která spočítá počet bodů, které student získal v multiple-choice testu. Vypracované řešení je reprezentováno parametrem `solution`, kterého prvky odpovídají možnostem, které student označil (tzn. je-li `solution[0]` rovno 2, odpověď na první otázku byla 2). Správné odpovědi jsou v parametru `answers` jako seznam dvojic, kde pozice v seznamu odpovídá číslu otázky, a dvojice je ve formě (správná odpověď, body).

```
def mark_points(answers, solution):
    pass
```

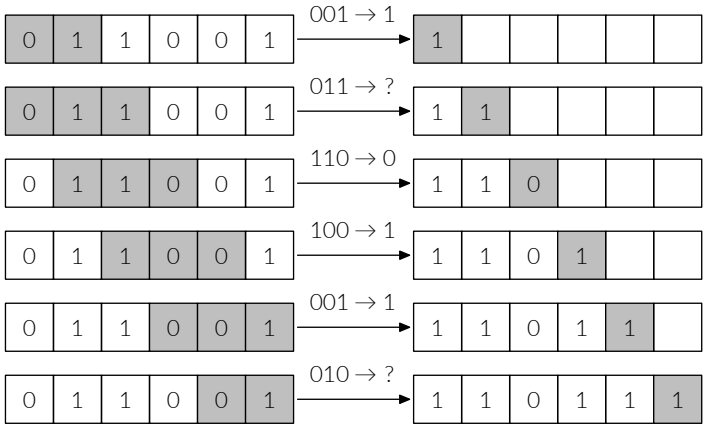
3.3.8 [`cellular`] Napište (čistou) funkci, která simuluje jeden krok výpočtu jednorozměrného buněčného automatu (cellular automaton). My se omezíme na binární (buňky nabývají hodnot 0 a 1) jednorozměrný automat s konečným stavem: stav takového automatu je seznam jedniček a nul, například:

0	1	1	1	0	0	1

Protože obecný automat tohoto typu je stále relativně složitý, budeme implementovat automat s fixní sadou pravidel:

`old[i - 1]`	`old[i]`	`old[i + 1]`	`new[i]`
0	0	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Pravidla určují, jakou hodnotu bude mít buňka v následujícím stavu, v závislosti na několika okolních buňkách stavu nynějšího (konkrétní indexy viz tabulka). Neexistuje-li pro danou vstupní kombinaci pravidlo, do nového stavu přepíšeme stávající hodnotu buňky. Na krajích stavu interpretujeme chybějící políčko vždy jako nulu.
Výpočet s touto sadou pravidel tedy funguje takto:

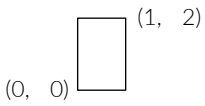


Na vstupu dostanete stav (konfiguraci) `state`, výstupem funkce je nový seznam, který obsahuje stav vzniklý aplikací výše uvedených pravidel na `state`.

```
def cellular_step(state):
    pass
```

3.3.9 [`rectangles`] Napište (čistou) funkci, která jako parametr dostane

seznam obdélníků a vrátí seznam obdélníků, které se překrývají s nějakým jiným. Obdélník samotný je reprezentovaný dvěma body (levým dolním a pravým horním rohem, a má nenulovou výšku i šířku). Obdélníky budeme zapisovat jako dvojice dvojic – `((0, 0), (1, 2))` například reprezentuje tento obdélník:



Mohl by se Vám hodit predikát, který je pravdivý, když se dva obdélníky překrývají:

```
def has_overlap(a, b):
    pass

def filter_overlapping(rectangles):
    pass
```

3.3.10 [numbers] V této úloze naprogramujeme trojici (čistých) funkcí, které slouží pro práci s číselnými soustavami. Reprezentaci čísla v nějaké číselné soustavě budeme ukládat jako dvojici `(base, digits)`, kde `base` je hodnota typu `int`, která reprezentuje základ soustavy, a `digits` je seznam cifer v této soustavě, kde každý prvek je hodnota typu `int`, která spadá do rozsahu [0, `base` - 1]. Index seznamu `digits` odpovídá příslušné mocnině `base`. Například:

- `(10, [2, 9])` je zápis v desítkové soustavě a interpretujeme jej jako `2 * 1 + 9 * 10`, co odpovídá číslu `92`
- `(7, [2, 1])` je zápis v sedmičkové soustavě a kóduje `2 * 1 + 1 * 7 = 9`

První funkce implementuje převod čísla `number` do ciferné reprezentace v soustavě se základem `base`:

```
def to_digits(number, base):
    pass
```

Další funkce provádí převod opačným směrem, z ciferné reprezentace `digits` vytvoří hodnotu typu `int`:

```
def from_digits(digits):
    pass
```

Konečně funkce `convert_digits` převede ciferný zápis z jedné soustavy do jiné soustavy. Nápověda: tato funkce je velmi jednoduchá.

```
def convert_digits(digits, base):
    pass
```

3.3.11 [right_cellular] Podobně jako v `cellular.py` budeme v této úloze pracovat s 1D buněčným automatem. Místo výpočtu nové konfigurace do nového seznamu ale budeme modifikovat stávající seznam. Toto samozřejmě nelze při použití stejných pravidel: v době vyhodnocování `i`-té buňky by již byla buňka s indexem `i - 1` přepsaná novou hodnotou. Proto použijeme pravidlo, které se dívá jen doprava:

old[i]	old[i + 1]	old[i + 2]	new[i]
1	0	0	0
0	1	0	1
0	1	1	1
1	0	1	0
1	1	1	0

Na rozdíl od předchozích příkladů, budeme v tomto implementovat proceduru: `cellular_in_situ` nebude hodnotu vracet, místo toho bude editovat seznam, který dostala jako parametr (viz též úvod k tomuto týdnu).

```
def cellular_in_situ(state):
    pass
```

Část 3.4: Bonusy

3.4.1 [partition] Naprogramujte proceduru `partition`, která na vstup dostane seznam čísel `data` a platný index `idx`. Pro pohodlnost hodnotu `data[idx]` nazveme `pivot`.
Procedura přeuspořádá seznam tak, že přesune prvky menší než `pivot` před `pivot` a prvky větší než `pivot` za `pivot`.
Po transformaci bude tedy seznam pomyslně rozdělen na tři části:

- čísla menší než `pivot`
- pivot
- čísla větší než `pivot`

Relativní pořadí prvků v první a poslední části není definováno, takže oba následovné výsledky pro seznam `[3, 4, 1, 2, 0]` a index `0` jsou správné: `[1, 0, 2, 3, 4]` nebo `[1, 2, 0, 3, 4]`.
Pozor: Při řešení nepoužívejte zabudované funkce pro řazení (funkce `partition` je mimo jiné pomocná funkce algoritmu quicksort, bylo by tedy absurdní zde sekvenci celou řadit). Řešení postavené na zabudované řadící funkci nebude uznáno jako příprava.

```
def partition(data, idx):
    pass
```

Část 3.5: Řešení

3.5.1 [predicates.sol]

```
def all_greater_than(sequence, n):
    out = True
    for x in sequence:
        out = out and x > n
    return out

def any_even(sequence):
    for x in sequence:
        if x % 2 == 0:
            return True
    return False
```

3.5.2 [explosion.sol]

```
def distance(a, b):
    x1, y1, z1 = a
    x2, y2, z2 = b
    return sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2 + (z1 - z2) ** 2)

def survivors(objects, center, radius):
    out = []
    for obj in objects:
        if distance(obj, center) > radius:
            out.append(obj)
    return out
```

Část 4: Řetězce

Tento týden se budeme zabývat řetězci – jedná se o datový typ, který se podobá na seznam (a na rozdíl od n-tic se i podobně používá), a kterého smyslem je reprezentovat v počítači text. Rozdíly oproti obecnému seznamu jsou dva: řetězec obsahuje vždy znaky (a nikdy hodnoty jiných

typů) a řetězec nemá vnitřní přiřazení, a tedy jednou postavený řetězec již nelze měnit. První z těchto vlastností je univerzální, druhá je spíše specialita Pythonu.
Podobně jako seznamy lze řetězce indexovat a iterovat. Druhou specialitou Pythonu je, že neexistuje samostatný typ pro znak, tento je reprezentován jednopísmenným řetězcem. To vede k zvláštní definici indexace (a iterace), kdy výsledkem indexace řetězce je opět řetězec (tento výsledný řetězec je vždy jednopísmenný).
Krom operací, které plynou z toho, že se jedná o sekvenční typ, řetězce navíc poskytují řadu operací, které jsou specifické pro sekvence znaků: převod na malá/velká písmena, rozčlenění podle oddělovače, sloučení seznamu řetězců do jednoho pomocí oddělovače atd.
Krom řetězců, které tvoří hlavní náplň tohoto týdne, budeme tento týden pokračovat i v práci se seznamy, zejména seznamy seznamů které tvoří dvourozměrné datové struktury (tabulky, matice, atp.).
Základní operace na řetězcích a práci s jednoduchým dvourozměrným seznamem si předvedeme v několika ukázkách:

1. `hamming.py` – základní operace s řetězci
2. `string_sum.py` – `split`, numerické hodnoty znaků
3. `power_table.py` – dvojrozměrný seznam

Jako obvykle následují řešené příklady:

1. `atbash.py` – jednoduchá substituční šifra
2. `palindrome.py` – řetězce, které se čtou stejně zleva i zprava

Základní sada příkladů:

1. `ipv4.py` – validace IP adres (protokol IP verze 4)
2. `isbn.py` – validace ISBN (mezinárodní systém číslování knih)
3. `person_id.py` – validace rodných čísel
4. `vigenere.py` – trochu složitější substituční šifra
5. `swap_columns.py` – prohození sloupců v tabulce
6. `polybius.py` – další příklad na šifrování, tentokrát s čísly
7. `divisors.py` – tabulace počtu společných dělitelů
8. `edge_detection.py` – hledání hran v obrázku

A na závěr dvojice bonusových úloh:

1. `mean_filter.py` – čištění obrázku od lokálních artefaktů
2. `pattern.py` – hledání vzorku v řetězci

Část 4.1: Ukázky

4.1.1 **[hamming]** V této ukázce implementujeme (čistou) funkci, která spočítá tzv. Hammingovu vzdálenost dvou řetězců, která určuje, v kolika znacích se odlišují. Pro jednoduchost budeme uvažovat pouze řetězce stejné délky. Velikost písmen budeme ignorovat (tzn. vzdálenost `hamming('a', 'A')` bude 0).

```
def hamming(s1, s2):
```

Doteď jsme to sice nedělali, ale zde se velice žádá ověřit vstupní podmínku, totiž že řetězce jsou stejně dlouhé.

```
    assert len(s1) == len(s2)
```

Asi nejjednodušší způsob, jak ignorovat velikost písmen je upravit si oba řetězce tak, aby obsahovaly pouze jednu velikost písmen (v tomto případě volíme velká písmena). Protože řetězce jsou vnitřně neměnné (nemají vnitřní přiřazení), jejich metody jsou čisté funkce a tedy volání `upper` vytvoří nový řetězec a ten původní zůstane beze změny. Přiřazením pouze změníme význam lokálních jmen `s1` a `s2`.

```
    s1 = s1.upper()
    s2 = s2.upper()
```

Pro dosud vypočtenou vzdálenost si vytvoříme střadač (akumulátor), do kterého budeme přičítat každou odlišnou dvojici znaků, kterou v řetězcích `s1` a `s2` potkáme.

```
    distance = 0
```

V cyklu spočítáme pozice, na kterých se řetězce liší. Použijeme indexovou proměnnou, protože v těle cyklu potřebujeme přistupovat na stejnou pozici dvou různých řetězců. Všimněte si, že řetězce indexujeme stejně jako seznamy nebo n-tice.

```
    for i in range(len(s1)):
        if s1[i] != s2[i]:
            distance += 1
```

V tuto chvíli máme již v `distance` napočítaný výsledek, který vrátíme volajícímu.

```
    return distance

def main():  # demo
    assert hamming("Python", "python") == 0
    assert hamming("AbCd", "aBcD") == 0
    assert hamming("string", "string") == 0
    assert hamming("aabcd", "abbcd") == 1
    assert hamming("abcd", "ghef") == 4
    assert hamming("Abcd", "bbcd") == 1
    assert hamming("gHefgh", "ghefkl") == 2
```

4.1.2 **[string_sum]** V této ukázce naprogramujeme funkci, která na vstupu dostane řetězec, který obsahuje dvě fráze oddělená čárkou. Výstupem bude nová fráze, které bude obsahovat „součet“ těchto dvou vstupních frází.
Sčítání definujeme po písmenech takto (minuska je malé písmeno, verzálka velké):

- mezera + mezera = mezera
- mezera + x-té písmeno = x-té písmeno (velikost se zachová)
- x-tá minuska + y-tá minuska = $(x + y)$-tá minuska
- x-tá verzálka + y-tá verzálka = $(x + y)$-tá verzálka
- x-tá minuska + y-tá verzálka nebo naopak = $(x + y)$-tá verzálka

Nejprve si definujeme několik pomocných funkcí na sčítání a odečítání písmenek. Musíme si zde dát pozor, protože jednotlivá písmena jsou v Pythonu reprezentována jako jednopísmenné řetězce, a mají tedy definovánu operaci sčítání (operátor `+`), která ale funguje jako zřetězení. Abychom mohli sčítat numerické hodnoty znaků, potřebujeme zabudovanou (čistou) funkci `ord`, která ze znaku vyrobí hodnotu typu `int`. Jaká čísla jsou kterým znakům přidělena nás nemusí příliš zajímat, důležité je, že velká i malá písmena jsou uspořádána abecedně (každé zvlášť, tzn. od jiného indexu).
Na to, abychom zjistili zda se jedná o minusku nebo verzálku můžeme využít predikátovou metodu `isupper` resp. `islower`.

```
def add(a, b):
    return chr(ord(a) + b)

def sub(a, b):
    return ord(a) - ord(b)

def index(a):
    if a.isupper():
        return sub(a, 'A')
    if a.islower():
        return sub(a, 'a')

def string_sum(words):
```

Nejprve musíme vstupní řetězec rozdělit podle znaku čárka na dvě fráze. K tomu můžeme využít metodu `split`, která daný řetězec rozdělí na seznam podřetězců, které byly v původním odděleny dodaným oddělovačem (opět řetězec). Vstupní podmínkou funkce je, že vstupní řetězec obsahuje právě jednu čárku, a tedy výsledek volání `split` vytvoří vždy dvouprvkový seznam. Proto můžeme využít syntaxe „rozbalení“ seznamu (které funguje stejně jako to, které známe u n-tic). Tento

přístup má výhodu mimo jiné v tom, že bude-li vstupní podmínka porušena, program na tomto místě okamžitě skončí s chybou. Proměnná result slouží jako zobecněný střadač – budeme v ní budovat výsledný řetězec.

```
    top, bottom = words.split(",")
    result = ""
```

Nyní řetězce zarovnáme na společnou délku. Není to nejefektivnější možnost, ale je zřejmé co se děje a v dalším již můžeme předpokládat, že jsou řetězce stejné délky, čím se kód znatelně zjednoduší.

```
    while len(top) < len(bottom):
        top += " "
    while len(top) > len(bottom):
        bottom += " "
```

Nyní zbývá samotné sčítání, pro které jsme si definovali pomocné funkce výše, a stačí tedy vyřešit jednotlivé případy popsané v zadání.

```
    for i in range(len(top)):
        if top[i] == ' ':
            result += bottom[i]
        elif bottom[i] == ' ':
            result += top[i]
        elif top[i].isupper() or bottom[i].isupper():
            result += add('A', index(top[i]) + index(bottom[i]))
        else:
            result += add('a', index(top[i]) + index(bottom[i]))

    return result
```

Nakonec na několika vstupech ověříme, že jsme v implementaci neudělali chybu.

```
def main():  # demo
    assert string_sum("xy,") == "xy"
    assert string_sum("xy,a") == "xy"
    assert string_sum("xy,b") == "yy"
    assert string_sum("xy,bb") == "yz"
    assert string_sum("Xy,bb") == "Yz"
    assert string_sum("xy,Bb") == "Yz"
    assert string_sum("xY,BB") == "YZ"
```

4.1.3 [power_table] Tato ukázka demonstruje velice jednoduchý program pro tvorbu dvourozměrného (obdélníkového) seznamu. Čistá funkce power_table vytvoří tabulku přirozených (kladných) mocnin nezáporných čísel, kde vždy v jednom řádku bude tatáž mocnina různých čísel, a v daném sloupci budou různé mocniny stejného čísla. Tabulka by měla vypadat zhruba takto (tabulka je zobrazena tak, že vnitřní seznamy jsou řádky – v prvním sloupci jsou indexy vnějšího a v záhlaví indexy vnitřního seznamu).

index	0	1	2	3	4	5
0	1	1	1	1	1	1
1	1	2	3	4	5	6
2	1	4	9	16	25	36
3	1	8	27	64	125	216

```
def power_table(max_pow, max_num):
```

Pro zjednodušení cyklu, který bude tabulku vyplňovat, si dopředu vytvoříme obdélníkový seznam a vyplníme jej nulami. Tzv. intensionální zápis seznamů znáte z přednášky: na prvním místě je popis hodnoty, která bude do seznamu vložena, poté následuje specifikace opakování, která používá syntaxi odvozenou od for cyklu. Celý vnitřní seznam je pro ten vnější onou specifikací hodnoty.

```
    result = [[0 for i in range(max_num)] for j in range(max_pow +
1)]
```

Nyní stačí tabulku vyplnit jednoduchým vnořeným for cyklem. Nesmíme zapomenout na posuv mezi přirozenými čísly (první je 1) a indexy (první je 0).

```
    for row in range(max_pow + 1):
        for col in range(max_num):
            result[row][col] = (col + 1) ** row
    return result
```

Není-li výpočet jednotlivé buňky příliš složitý, můžeme celý seznam zapsat intensionálně přímo:

```
def power_table_alt(max_pow, max_num):
    return [[(col + 1) ** row
             for col in range(max_num)]
             for row in range(max_pow + 1)]
```

Následují pomocné funkce pro testování a samotné testy.

```
def check_size(tab, rows, cols):
    assert len(tab) == rows
    for row in tab:
        assert len(row) == cols

def check_power_table(tabulate):
    check_size(tabulate(1, 1), 2, 1)
    check_size(tabulate(3, 2), 4, 2)
    assert tabulate(1, 1)[0][0] == 1
    assert tabulate(1, 1)[1][0] == 1
    assert tabulate(3, 3)[3][2] == 27
    assert tabulate(4, 3)[4][2] == 81
```

Část 4.2: Rozcvička

4.2.1 [atbash] Atbash je jednoduchá substituční šifra, kterou vynalezli Hebrejci (její název je odvozen od písmen hebrejské abecedy). Zašifrování písmene touto šifrou probíhá tak, že se spočítá vzdálenost daného písmena od prvního písmena používané abecedy a toto písmeno je nahrazené písmenem, které se nachází ve stejné vzdálenosti od konce abecedy.
V této úloze budeme jako abecedu používat velká písmena anglické abecedy: zašifrujeme-li písmeno C výsledkem bude X, protože C je třetím písmenem abecedy a X je třetí od konce.
Abychom dosáhli větší uživatelské přívětivosti, povolíme na vstupu i malá písmena a mezery – malá písmena budeme šifrovat obdobně jako velká, a šifrování bude velikost písmen zachovávat (malá písmena se šifrují na malá a velká na velká). Mezery šifrování i dešifrování zachová nezměněné.
Napište čistou funkci, která na vstupu dostane text (řetězec), zašifruje ho šifrou atbash výše popsaným způsobem, a výsledek vrátí opět jako řetězec.

```
def atbash_encrypt(text):
    pass
```

Dále napište čistou funkci, která na vstupu dostane řetězec představující text zašifrovaný šifrou atbash, tento text dešifruje, a výsledek vrátí jako řetězec.

```
def atbash_decrypt(text):
    pass
```

4.2.2 [palindrome] Napište predikát, kterého hodnota bude True, pokud v parametru dostane řetězec, který je palindrom, False jinak. Řetězec je palindrom, když jej můžeme přečíst v obou směrech (zprava doleva i zleva doprava) a jeho význam sa nezmění. Palindromy jsou například „oko“, „kajak“ a „madam“.
U řetězcových palindromů nehledíme na to, zda jsou jednotlivá pís-

mena velká nebo malá, tudíž palindromem je i slovo "Anna". Podobně nebereme v úvahu mezery – „jelenovi pivo nelej“ je tedy rovněž palindrom. Pro zjednodušení budeme ale v této úloze považovat znaky lišící se pouze diakritikou za různé.

```
def is_palindrome(text):
    pass
```

Část 4.3: Příklady

4.3.1 [ipv4] V této úloze se budeme zabývat adresami protokolu IP verze 4, které sestávají ze 4 čísel oddělených tečkami, například `192.0.2.0` (více informací o IPv4 naleznete například na Wikipedii). Adresy budeme reprezentovat řetězci.
Napište predikát, kterého hodnota bude `True`, představuje-li jeho parametr validní IPv4 adresu. Daná IPv4 adresa je validní právě tehdy, když je tvořená čtyřmi dekadickými čísly od 0 až 255 (včetně) oddělenými tečkou (pro jednoduchost v této úloze připouštíme pouze kanonický tvar IPv4 adres).

```
def ipv4_validate(address):
    pass
```

Dále napište čistou funkci, která vypočte číselnou hodnotu dané adresy. Konverze IPv4 adresy na její číselnou hodnotu je podobná konverzi binárního zápisu čísla na dekadický s tím rozdílem, že u IPv4 adresy pracujeme se základem 256. Hodnota adresy `192.0.2.0` je tedy $192 \cdot 256^3 + 0 \cdot 256^2 + 2 \cdot 256^1 + 0 \cdot 256^0 = 3\,221\,225\,984$. Můžete počítat s tím, že vstupem bude vždy validní IPv4 adresa ve výše popsaném kanonickém tvaru.

```
def ipv4_value(address):
    pass
```

4.3.2 [isbn] V této úloze se budeme zabývat systémem číslování knih ISBN, konkrétně variantami ISBN-10 (deseticifernou) a ISBN-13 (třináctcifernou). Obě tyto varianty ISBN obsahují ve své poslední cifře informaci, která slouží k jejich validaci, zatímco ostatní cifry reprezentují vybrané atributy dané publikace.
Napište predikát, jehož hodnota bude `True`, představuje-li hodnota parametru `isbn` korektně utvořené ISBN-10. Parametr bude do funkce předáván jako řetězec.
ISBN-10 se skládá z devíti desítkových cifer, které popisujících danou publikaci a poslední desáté cifry, jejíž hodnota může být 0 až 10 (kde hodnotu 10 zapisujeme jako velké X). Validní ISBN musí dále splnit následující podmínku:
Očíslujeme-li jednotlivé cifry daného ISBN zprava doleva, každou z cifer vynásobíme jejím pořadovým číslem, a výsledky sečteme, výsledný součet musí být dělitelný jedenácti. (Zkuste si rozmyslet, proč potřebujeme u poslední cifry povolit hodnotu 10.)
Příkladem validního ISBN-10 je 007462542X: $10 \cdot 0 + 9 \cdot 0 + 8 \cdot 7 + 7 \cdot 4 + 6 \cdot 6 + 5 \cdot 2 + 4 \cdot 5 + 3 \cdot 4 + 2 \cdot 2 + 1 \cdot 10 = 176$ a zároveň $176 = 11 \cdot 16$.

```
def isbn10_validator(isbn):
    pass
```

Dále napište podobný predikát, který ověří korektnost ISBN-13. Kontrolní součet provedeme takto:

1. očíslujeme jednotlivé cifry daného ISBN zprava doleva,
2. každou z cifer vynásobíme 3 je-li její pořadové číslo sudé
3. výsledky sečteme a ověříme, že výsledek je dělitelný deseti.

Rozmyslete si, proč u této varianty není potřeba povolit „X“.
Příkladem validního ISBN-13 je 9780716703440 protože $9 + 7 \cdot 3 + 8 + 0 \cdot 3 + 7 + 1 \cdot 3 + 6 + 7 \cdot 3 + 0 + 3 \cdot 3 + 4 + 4 \cdot 3 + 0 = 100$ a zároveň $100 = 10 \cdot 10$.

```
def isbn13_validator(isbn):
    pass
```

4.3.3 [person_id] Napište predikát, který validuje rodná čísla. Rodné číslo budeme v parametru předávat jako řetězec a pro zjednodušení budeme uvažovat pouze rodné čísla v současně přidělované formě: `RRMMDD/XXXX`, kde:

- `RR` představuje poslední dvojčíslí roku narození,
- `MM` měsíc narození, jedná-li se o rodné číslo muže a měsíc narození zvýšený o 50 jedná-li se o rodné číslo ženy,
- `DD` den narození, a konečně
- `XXXX` je unikátní čtyřčíslí, které odlišuje dvě osoby stejného pohlaví a data narození.

Například 065101 je prvních šest číslic rodného čísla dívky narozené 1. ledna 2006.
Každé platné rodné číslo musí být navíc po vynechání znaku / beze zbytku dělitelné jedenácti. Rodná čísla v uvedeném formátu začaly platit v roce 1954, proto hodnoty `RR` větší než 53 označují roky 20. století a ty s hodnotou menší než 54 označují roky 21. století. Nezapomeňte se zamyslet, jaká další omezení na platné rodné číslo plynou z významu jednotlivých dvojčíslí.

```
def person_id_validator(id_candidate):
    pass
```

4.3.4 [vigenere] Z přednášky již znáte Caesarovu šifru. Vigenèrova šifra funguje podobně, ale klíčem je řetězec (slovo), nikoliv pouze jediné číslo z rozsahu 0–26.
Šifrujeme (i dešifrujeme) tak, že pod vstupní text si podepíšeme klíč, který zopakujeme tolikrát, aby byl stejně dlouhý jako text. Každou takovou dvojici pak použijeme jako vstup pro Caesarovou šifru (písmeno klíče udává posuv — A = 0, B = 1, ...).
Jako příklad zašifrujeme slovo „PYTHON“ klíčem „BCD“: klíč nejprve prodloužíme opakováním na „BCDBCD“, poté spočítáme:

- `caesar('P', 1) = 'Q'`
- `caesar('Y', 2) = 'A'`
- `caesar('T', 3) = 'W'`
- `caesar('H', 1) = 'I'`
- `caesar('O', 2) = 'Q'`
- `caesar('N', 3) = 'Q'`

Napište (čistou) funkci, která na vstupu dostane text k zašifrování a klíč. Obojí budou řetězce libovolné délky, přičemž klíč je neprázdný a tvořený pouze písmeny anglické abecedy a text písmeny anglické abecedy a mezerami. Výstupem bude daný text zašifrovaný pomocí daného klíče. Pro zjednodušení budeme vracet zašifrovaný text ve velkých písmenech.

```
def vigenere_encrypt(text, key):
    pass
```

Dále napište (opět čistou) funkci, která na vstupu dostane zašifrovaný text a odpovídající klíč (obojí ve stejném formátu, jako parametry předešlé funkce) a vrátí dešifrovaný text. Pro zjednodušení budeme dešifrovaný text opět vracet ve velkých písmenech.

```
def vigenere_decrypt(text, key):
    pass
```

4.3.5 [swap_columns] Implementujte proceduru, která v parametru dostane:

- neprázdný dvourozměrný seznam (ve tvaru obdélníku: délky všech vnitřních seznamů jsou stejné), který reprezentuje tabulku, ve které vnitřní seznamy tvoří řádky, a dále
- seznam dvojic čísel sloupců (můžete se spolehnou, že daná čísla

vždy označují existující sloupce dané tabulky), které v dané tabulce vymění.

Například po použití této procedury se seznamem dvojic sloupců `[(1, 2)]` na seznam `[[1, 2, 3], [4, 5, 6], [7, 8, 9]]` tento změní na `[[1, 3, 2], [4, 6, 5], [7, 9, 8]]`.

```
def swap_columns(table, cols_to_swap):
    pass
```

4.3.6 [polybius] Polybiův čtverec je metoda kódováni, známa již ve starověkém Řecku, založená na převádění zprávy z původní abecedy do jiné abecedy o méně znacích. Abecedu původních zpráv uspořádáme do čtverce (pokud počet znaků abecedy není druhou mocninou, typicky zůstávají ve čtverci příslušný počet prázdných pozicí) a jednotlivé znaky zprávy pak kódujeme dvojicí souřadnic pozice, na které se daný znak nachází ve vzniklém čtverci. Konkrétně kódovací čtverec vytváříme tak, že do nejmenšího čtverce, do kterého se daná abeceda vejde, postupně, začínaje z levého horního rohu, vpisuje popořadě po řádcích jednotlivé znaky abecedy. Vzniklý čtverec pak v obou směrech indexujeme od jedničky, přičemž začínáme z levého horního rohu.
V této úloze budeme jako abecedu zpráv používat písmena velké anglické abecedy. Těchto písmen je 26, podle výše popsaného postupu bychom tedy vytvářeli čtverec o straně velikosti 6 a zůstalo by nám 10 volných pozic. Toto se u této abecedy obvykle řeší tak, že písmena `I` a `J` sdílí společnou pozici a dostáváme tak čtverec se stranou o velikosti 5 (volné pozice v něm nezůstanou žádné). Tento obvyklý postup budeme implementovat i my. Písmena `I` a `J` se tedy budou kódovat stejně. U dekódování je pak už snadné z kontextu poznat, které z těchto písmen bylo v původní zprávě (program bude příslušnou pozici vždy dekódovat na `I`). Při šifrování budeme krom velkých písmen anglické abecedy akceptovat i malá a zakódujeme je jakoby se jednalo o velké, při dekódováni ale budeme používat pouze velká písmena.
Například `E` se bude kódovat jako `15` poněvadž se nachází v prvním řádku a pátém sloupci, `I` a `J` se budou kódovat na `24` a třeba `T` na `44`. Kód slova `EJT` pak vypadá takto: `15 24 44`.
Napište (čistou) funkci, která na vstupu dostane slovo (řetězec) skládající se z velkých a malých písmen anglické abecedy a vrátí tento text zašifrován pomocí Polybiova čtverce jako řetězec, ve kterém budou kódy jednotlivých písmen odděleny mezerou.

```
def polybius_encode(word):
    pass
```

Dále napište čistou funkci, která na vstupu dostane řetězec reprezentující šifru vzniklou výše popsaným způsobem a vrátí původní slovo jako řetězec.

```
def polybius_decode(code):
    pass
```

4.3.7 [divisors] Napište čistou funkci, která na vstupu dostane dvě celá kladná čísla `rows` a `cols` a vrátí tabulku (dvourozměrný seznam) o `rows` řádcích a `cols` sloupcích. V buňce v řádku `y` a sloupci `x` bude počet společných dělitelů čísel `x` a `y`. Levý horní roh má souřadnice `x = y = 1`.
Například pro vstup `rows = 4`, `cols = 2` dostaneme tabulku `[[1, 1], [1, 2], [1, 1], [1, 2]]`.

```
def common_divisors(rows, cols):
    pass
```

4.3.8 [edge_detection] Jednou ze základních metod digitálního zpracování obrazu je detekce hran. V tomto kontextu hranou chápeme oblast s ostrým kontrastem: nemusí jít nutně o hranice mezi objekty, ale třeba také změny textury, rozhraní mezi světlem a tmou, atd. Hrany se v obrázcích detekují na základě velké změny jasu sousedních pixelů (více o informací naleznete například ve Wikipedii pod heslem „Edge detection“).
V této úloze si vyzkoušíte zjednodušený způsob, jak nalézt všechny hrany v daném obrázku, a vytvořit nový obrázek, který obsahuje pouze nalezené hrany.
Napište (čistou) funkci, která na vstupu dostane obrázek reprezentovaný obdélníkovým seznamem seznamů (délky všech vnitřních seznamů jsou stejné) celých čísel a vrátí nový obrázek stejné velikosti, který obsahuje pouze hrany původního obrázku. Konkrétně pixely ve výsledném obrázku, kde na vstupu detekujeme hranu, budou mít hodnotu 1 a všechny ostatní hodnotu 0.
Funkce bude brát krom vstupního obrázku ještě číselný parametr, který určí, o kolik se dva pixely musí nejméně lišit, abychom je označili za hranu. Konkrétně pak za součást nějaké hrany považujeme každý pixel, který se liší od některého ze svých čtyř sousedů alespoň o tuto hodnotu.

```
def find_edges(image, threshold):
    pass
```

Část 4.4: Bonusy

4.4.1 [mean_filter] Mean filter je běžný filtr sloužící na odstranění drobných vad z obrázku. Funguje tak, že lokálně přílišně se odlišující pixely považuje za chybné a napravuje je přiblížením jejich hodnoty hodnotám okolních pixelů. To realizuje tak, že každý pixel nahradí průměrem jeho původní hodnoty s hodnotami okolních pixelů.
Napište proceduru, která v parametru dostane obrázek reprezentován obdélníkovým dvourozměrným seznamem (délky všech vnitřních seznamů jsou stejné) a tento obrázek upraví aplikací mean filtru. Nové hodnoty jednotlivých pixelů přesněji spočítá tak, že zaokrouhlí průměr hodnot daného pixelu a všech jeho osmi sousedních pixelů, přičemž za sousední považujeme všechny pixely, které se ho dotýkají stranou nebo rohem. (Pro zaokrouhlování použijte vestavěnou funkci `round`.)

```
def mean_filter(image):
    pass
```

4.4.2 [pattern] Napište čistou funkci, která dostane na vstupu dva řetězce, `text` a `pattern` (vzor) a vrátí pozici prvního výskytu vzoru v textu, pokud se v něm nachází, -1 jinak. Text se skládá pouze z velkých a malých písmen anglické abecedy a mezer a vzor může navíc obsahovat znaky `?`, `[`, `]`, `-`, kterých význam je následovný:

- `?` představuje libovolný znak textu, zatímco
- `[a-c]` představuje libovolné z písmen mezi `a` a `c` včetně, tj. libovolné z písmen `a`, `b` nebo `c`.

Speciální znaky se ve vzoru nemůžou vykytovat v žádném jiném kontextu, než ve výše popsaném.
Například vzor `y[l-u]h[m-q]?` se v textu `Python` nachází jednou a to na pozici 1 a vzor "python" se v daném textu nenachází vůbec (naše funkce by vrátila -1).

```
def find_pattern(text, pattern):
    pass
```

Část 4.5: Řešení

4.5.1 [atbash.sol]

```
def atbash_encrypt_decrypt_char(character):
    if not character.isalpha():
        return character
    return chr(ord("Z") - (ord(character) - ord("A")))

def atbash_encrypt(text):
    result = ""
    upper_text = text.upper()
```

```
    for i in range(len(upper_text)):
        coded = atbash_encrypt_decrypt_char(upper_text[i])
        if text[i] == upper_text[i]:
            result += coded
        else:
            result += coded.lower()
    return result

def atbash_decrypt(text):
    return atbash_encrypt(text)
```

4.5.2 [palindrome.sol]

```
def is_palindrome(text):
    text = text.upper()
    words = text.split()
    text = "".join(words)
    for i in range(len(text) // 2):
        if text[i] != text[- i - 1]:
            return False
    return True
```

Část 5: Testování a typy

Tento týden se zaměříme na korektnost (správnost) programů – zejména nás bude zajímat nástroje, které nám pomohou psát programy bez chyb. K dispozici máme dvě základní kategorie takových nástrojů:

1. statické, totiž takové, které analyzují program aniž by jej spouštěli – pracují podobně jako například flake8, který již znáte,
2. dynamické, které kontrolují, zda program pracuje správně během samotného provádění programu.

Tyto dva přístupy ke kontrole správnosti programu reprezentují určitým způsobem opačné kompromisy. Dynamické nástroje jsou velice přesné (umožňují kontrolovat prakticky libovolné, i velmi složité, vlastnosti), ale nemůžou nám zaručit, že program se bude za všech okolností chovat správně. Taková kontrola je často velmi časově náročná, protože abychom si ověřili správnost programu, musíme jej testovat: opakovaně spouštět s různými vstupy.
Statická kontrola je naopak méně přesná (umožňuje nám kontrolovat pouze jednoduché vlastnosti programu), ale je rychlá (program není potřeba spouštět) a může být bezpečná (tzn. některé statické kontroly můžou zaručit, že určitý typ chyby v programu nikdy za běhu nenastane).
V kategorii statických nástrojů jsou pro nás zajímavé zejména typové anotace, které lze kontrolovat programem mypy. V tomto předmětu máme již zkušenost s dynamickou typovou kontrolou, kdy pokus například o sečtení čísla a řetězce vede na běhovou chybu, tzn. program v momentě, kdy se takovou operaci pokusí provést, havaruje s výjimkou TypeError. Typové anotace a statická typová kontrola nám umožní většině podobných chyb předejít, aniž bychom museli program spustit (natož důkladně testovat).
Z těch dynamických jsou pro nás přístupná zejména dynamická tvrzení, která zapisujeme již známým klíčovým slovem assert. Dynamická tvrzení nám zejména umožňují formalizovat a automaticky, při každém volání, kontrolovat vstupní a výstupní podmínky funkcí (podprogramů). Můžeme je také použít k zápisu a ověření dalších podmínek, o kterých jsme přesvědčeni, že musí v daném místě programu za každých okolností platit.
V obou případech (typové anotace a dynamická tvrzení) musíme do programu přidat dodatečné informace, které netvoří přímo součást výpočetní části programu (tzn. nepopisují samotné kroky výpočtu). Mohlo by se na první pohled zdát, že přidávat tyto „přebytečné“ prvky do programu je práce navíc, která nás bude při programování leda zdržovat. Trochu hlubší analýza ale odhalí, že počáteční zápis programu tvoří jen zlomek celkového času, který programováním strávíme – ladění a údržba typicky zaberou času mnohem víc. Investice do anotací se většinou v těchto návazných fázích vývoje programu velmi rychle vrátí.
Anotace plní 3 základní funkce:

1. nutí nás hlouběji se zamyslet o chování programu – často si uvědomíme chybu už v čase, kdy uvažujeme jaké použít anotace,
2. umožňují použití automatických nástrojů pro kontrolu správnosti, čím detekují chyby, které nám v prvním bodě přecijen proklouznou,
3. slouží jako dokumentace, jak pro programátory, kteří naše funkce chtějí použít, tak pro pozdější úpravy a opravy v samotném kódu.

Tento týden si práci s anotacemi (zejména těmi typovými) nacvičíme na příkladech. Nejprve ale jejich použití demonstrujeme v několika ukázkách:

1. shapes.py – typové anotace
2. barcode.py – vstupní a výstupní podmínky (1. část)
3. ean.py – vstupní a výstupní podmínky (2. část)

Řešené příklady:

1. trivial.py – základní typování
2. clocks.py – čtení času
3. fridays.py – typování

Základní úlohy:

1. database.py – typování
2. points.py – typování seznamů
3. course.py – kombinace typování
4. triangle.py – volitelné argumenty
5. gps.py – čtení GPS souřadnic
6. doctor.py – zanořené seznamy

Bonusové úlohy:

1. parse_time.py – čtení data a času

Část 5.1: Ukázky

5.1.1 [shapes] V tomto příkladu budeme počítat základní vlastnosti geometrických objektů, které budeme popisovat n-ticemi (zejména čísel). Příklad slouží k seznámení s typovou anotací parametrů a návratových hodnot podprogramů (funkcí).
Jak již víte z přednášky, anotace základních typů (int, float, str, atp.) se zapisuje přímo názvem typu, zatímco anotace složených typů vyžaduje definice typů z modulu typing. Jedná se zejména o seznamy (v anotacích zapisujeme jako List[element], kde element je typová anotace platná pro každý prvek seznamu) a n-tice (zapisujeme jako Tuple[x, y, z] – tento zápis značí trojici, kde x, y a z jsou postupně typové anotace pro první, druhou a třetí složku n-tice). Konečně případy, kdy potřebujeme otypovat hodnotu, která je typu type, ale nemusí nutně existovat (může být v některých případech None), použijeme anotaci Optional[type].

```
from typing import List, Tuple, Optional
```

Jako první si definujeme čistou funkci pro výpočet obsahu kruhu (anglicky disc), která má jediný parametr typu float a jejíž výsledkem je opět číslo typu float. Tím, že tyto skutečnosti zapíšeme do programu jako anotace de-facto deklarujeme vstupní a výstupní podmínky funkce: vstupní podmínkou je, že skutečná hodnota předávaného parametru je typu float, zatímco výstupní je, že návratová hodnota je též typu float. Pro jistotu připomínáme, že za splnění vstupní podmínky zodpovídá volající, zatímco za splnění výstupní podmínky zodpovídá volaná funkce.
Program mypy nám pro takto anotovanou funkci zaručí dvě věci: jednak,

že omylem funkci nezavoláme se špatným typem parametru (neporušíme vstupní podmínku na typy), třeba s hodnotou typu řetězec. Dále pak kontroluje, že v těle funkce neporušujeme výstupní podmínku – návratová hodnota je číslo typu `float` (nevrátíme omylem v žádném příkazu `return` ve funkci třeba řetězec, nebo `None`). K provedení této kontroly není potřeba program spouštět.

```
def disc_area(radius: float) -> float:
    return pi * radius ** 2
```

Zatímco pro popis kruhu nám stačí jediné číslo, pro popis obdélníku již potřebujeme čísla dvě, výšku a šířku. Máme dvě možnosti: můžeme potřebné hodnoty předat jako dva samostatné parametry, nebo můžeme obě hodnoty zabalit do n-tice (dvojice). Druhý přístup je lepší v případě, kdybychom potřeboval vytvořit třeba seznam obdélníků (to bude i náš případ). Proto zvolíme přístup s dvojicí čísel. Někdy má smysl složitější typy pojmenovat, a protože s obdélníky budeme pracovat na více místech, zavedeme si pro typ dvojice čísel jméno `Rectangle`:

```
Rectangle = Tuple[float, float]
```

Nyní již můžeme přistoupit k samotné definici (opět čisté) funkce pro výpočet plochy obdélníku. Výsledkem bude opět číslo.

```
def rectangle_area(dimensions: Rectangle) -> float:
    width, height = dimensions
    return width * height
```

Elipsa reprezentuje podobný případ, kdy potřebujeme k jejímu popisu dvě čísla, tentokrát délky jejích dvou poloos. Všimněte si, že typ popisující elipsu je identický s typem pro obdélník. S tím jsou spojeny určité problémy, které si objasníme níže. Protože elipsami se nebudeme dále zabývat, nebudeme tentokrát typ pojmenovávat.

```
def ellipse_area(semiaxes: Tuple[float, float]) -> float:
    major, minor = semiaxes
    return pi * major * minor
```

Abychom demonstrovali i nehomogenní n-tice (tj. takové, které mají složky různých typů), zadefinujeme si ještě pravidelný n-úhelník, který zadáme hlavním poloměrem (tzn. vzdáleností vrcholu od středu) a počtem vrcholů (který je na rozdíl od poloměru celočíselný).

```
def polygon_area(polygon: Tuple[float, int]) -> float:
    radius, vertices = polygon
    half_angle = pi / vertices
    half_side = sin(half_angle) * radius
    minor_radius = cos(half_angle) * radius
    return vertices * minor_radius * half_side
```

Nyní si definujeme funkci, která budou pracovat s trochu složitějšími typy: vstupem bude seznam barevných obdélníků a jedna vybraná barva, výsledkem bude celková plocha dané barvy. Pro barvu (reprezentovanou řetězcem) si zavedeme typové synonymum: to je typicky vhodné v případech, kdy se příslušný typ objevuje jako složka n-tice. Uvažte rozdíl mezi čitelností typové anotace `Tuple[Tuple[int, int], str]` vs. `Tuple[Rectangle, Colour]`.

```
Colour = str

def coloured_area(rectangles: List[Tuple[Rectangle, Colour]],
                  selected_colour: str) -> float:
```

Na tomto místě musíme `mypy` trochu pomoct, protože literál `0` lze interpretovat jako celé i jako desetinné číslo, přičemž výchozí interpretace je celočíselná. V podstatě máme dvě možnosti: můžeme literál zapsat jako `0.0`, čím nejednoznačnost odstraníme, nebo přidáme typovou anotaci i proměnné (střadači) `area`. Taková anotace se zapisuje na levou stranu přiřazení a syntakticky je stejná jako anotace parametru.

```
    area: float = 0
```

Cyklus pro sečtení ploch se už od zápisu, na který jsme zvyklí, nijak neliší. Stojí nicméně za zmínku, že `mypy` za nás kontroluje krom správného volání funkce `rectangle_area` také to, že srovnáváme hodnoty stejných (obecněji kompatibilních) typů – kdybychom omylem srovnali třeba řetězec (barvu) a obdélník (třeba proto, že jsme zaměnili pořadí `rect` a `colour` při rozbalování hodnoty typu `Tuple[Rectangle, Colour]`), `mypy` by nás na tuto chybu upozornilo.

```
    for rect, colour in rectangles:
        if colour == selected_colour:
            area += rectangle_area(rect)
    return area
```

Dále napíšeme funkci, která ze seznamu obdélníků vybere ten s největší plochou, existuje-li takový právě jeden. Je zde vidět, že návratový typ může být, podobně jako typy parametrů, složitější – připomínáme, že `Optional[type]` znamená, že hodnota může být buď typu `type` nebo `None` (vzpomeňte si také, že `Rectangle` je synonymum pro `Tuple[float, float]`).

```
def largest_rectangle(rectangles: List[Rectangle]) \
        -> Optional[Rectangle]:

    if len(rectangles) == 0:
        return None

    largest = rectangles[0]
    count = 0

    for r in rectangles:
        if isclose(rectangle_area(r), rectangle_area(largest)):
            count += 1
        elif rectangle_area(r) > rectangle_area(largest):
            count = 1
            largest = r

    return largest if count == 1 else None
```

Konečně napíšeme funkci, která ze seznamu obdélníků vybere ty, které mají plochu stejnou nebo větší, než je průměrná plocha celého vstupního seznamu (který musí být neprázdný).

```
def large_rectangles(rectangles: List[Rectangle]) \
        -> List[Rectangle]:
    total = sum([rectangle_area(r) for r in rectangles])
    average = total / len(rectangles)
    result = []
    for r in rectangles:
        if rectangle_area(r) >= average:
            result.append(r)
    return result
```

Nyní zbývá pouze popsané funkce otestovat:

```
def main() -> None:  # demo
    unit_rectangle = (1, 1)
    assert isclose(rectangle_area(unit_rectangle), 1)
    assert isclose(rectangle_area((2, 2)), 4)
    assert isclose(polygon_area((sqrt(2), 4)), 4)
    assert isclose(polygon_area((1, 6)), 2.5980762113533)
    assert isclose(ellipse_area((1, 1)), 3.1415926535898)
    assert isclose(ellipse_area((2, 6)), 37.699111843078)
    assert isclose(ellipse_area((12.532, 8.4444)), 332.4597362298)
```

Na začátku jsme zmiňovali, že elipsu a obdélník reprezentujeme stejným typem, a že by to mohlo vést k určitým problémům. Samozřejmě, nemůže se stát nic horšího, než co by se stalo, kdybychom anotace nepoužili vůbec, nicméně musíme si zároveň uvědomit, že typové anotace nejsou všemožné, a ani před něčím, co napohled vypadá jako typová chyba nás nemusí ochránit. Uvažte následující (zakomentovaný) příkaz – protože `unit_rectangle` je typu `Tuple[float, float]` a funkce

`ellipse_area` očekává parametr téhož typu, je z pohledu `mypy` takové volání v pořádku. Přesto je zřejmé, že takovéto použití nebylo zamýšleno, a téměř s jistotou povede k chybě v programu. Tuto konkrétní situaci lze lépe řešit použitím složených datových typů, které si ukážeme přespříští týden.

```
    pass  # assert ellipse_area(unit_rectangle) == 1

    red, green, blue = ("red", "green", "blue")
    red_1 = ((1, 1), red)
    red_2 = ((5, 6), red)
    green_1 = ((1, 1), green)
    green_2 = ((5, 6), green)
    blue_1 = ((2, 3), blue)
    assert isclose(coloured_area([red_1, green_1], red), 1)
    assert isclose(coloured_area([red_1, red_2], red), 31)
    assert isclose(coloured_area([red_1, green_2, blue_1], blue), 6)
    assert isclose(coloured_area([red_1, green_1], blue), 0)
    assert largest_rectangle([]) is None
    assert largest_rectangle([(1, 1), (4, 3), (6, 2)]) is None
    assert largest_rectangle([(5, 5), (4, 3), (1, 1)]) == (5, 5)
    assert largest_rectangle([(12, 2), (10.2, 1.5)]) == (12, 2)
    r_1, r_2, r_3 = ((1, 3), (5, 5), (7, 2))
    assert large_rectangles([r_1, r_2, r_3]) == [r_2, r_3]
    assert large_rectangles([r_1, r_2]) == [r_2]
    assert large_rectangles([r_1, r_1]) == [r_1, r_1]
```

5.1.2 [barcode] Tato ukázka je první z dvojice, která demonstruje použití tvrzení (assertion) pro popis vstupních a výstupních podmínek. Nejprve si v rychlosti zopakujme trochu teorie.

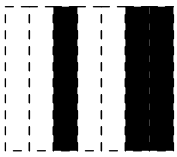
Velmi důležitá vlastnost tvrzení je, že ve správném (korektním) programu musí za všech okolností platit. Dojde-li k porušení některého tvrzení, program havaruje s chybou `AssertionError` a vždy se jedná o chybu v programu. Je-li tedy uživatel schopen programu předložit vstup, který způsobí, že program havaruje s chybou `AssertionError`, tento program je špatně.

Smyslem takovýchto tvrzení tedy není kontrola vstupu, nebo jiných okolností, které můžou selhat – naopak, slouží jako dokumentace a pomůcka k ladění: odhalit příčinu chybného chování programu je tím snazší, čím dříve si všimneme nějakou odchylku od chování očekávaného. Budeme-li důsledně kontrolovat vstupní a výstupní podmínky příkazy `assert`, je pravděpodobné, že chybu odchytíme brzo (program havaruje).

Naopak, budeme-li spoléhat na vlastní neomylnost (případně neomylnost kolegů), ale chyba se do programu přeci dostane, bude se pravděpodobně nekontrolovaně šířit – funkce, kterých vstupní podmínka nebyla splněna jednoduše vypočtou nesprávný výsledek, se kterým bude program nadále pracovat a produkovat další a další nesmyslné mezivýsledky. Výstup nebo chování programu bude nesprávné, ale bude velice obtížné a časově náročné poznat, ve kterém kroku výpočtu došlo k první chybě.

Nyní již můžeme přejít k ukázkovému programu: téma první části budou čárové kódy. V tomto modulu se budeme zabývat samotným kódováním sekvence černých a bílých pruhů, zatímco v části druhé (`ean.py`) se budeme zabývat již dekódovanými číselnými hodnotami.

Čárový kód sestává z řady pruhů (anglicky area), kde každý pruh může být černý nebo bílý. Pruhy zabírají celou výšku kódu a mají fixní šířku, přičemž na šířku se vždy dotýkají: dva sousední černé pruhy tvoří jednolitou plochu. Každá číslice je kódována do sedmi pruhů, třeba číslice 2 vypadá takto (v binárním zápisu 0010011; na obrázku je šířka jednoho pruhu přehnaná, skutečné pruhy jsou velmi úzké).



0 0 1 0 0 1 1

Každá číslice má 3 různá možná kódování, značená `L`, `R` a `G`, přičemž v kódech EAN-8, se kterými budeme pracovat, se objevují pouze kódování `L` a `R`, která jsou vzájemně inverzní: obrázek výše je v kódování `L`, odpovídající kódování `R` je následovné:

1 1 0 1 1 0 0

Čárové kódy standardu EAN mají 5 skupin pruhů:

- počáteční skupina, vždy 101,
- první polovina číslic (každá kódována do sedmi pruhů),
- středová dělící skupina, vždy 01010,
- druhá polovina číslic (opět po sedmi pruzích),
- koncová skupina, vždy 101.

Následuje kompletní příklad se dvěma číslicemi (2 a 2), první kódovanou v `L` a druhou v `R`. Pro odlišení jsou pruhy koncových a středové skupiny vybarveny světlejší barvou a místo 0 a 1 používají symboly _ a X:

X _ X 0 0 1 0 0 1 1 _ X _ X _ 1 1 0 1 1 0 0 X _ X

Jako první definujeme predikát `barcode_valid`, který bude kontrolovat platnost kódu (tzn. má-li požadovanou strukturu a správně zakódované číslice). Protože se jedná o relativně složitý predikát, některé kontroly oddělíme do samostatných funkcí (mnoho z nich navíc později využijeme při dekódování). Krom samotného čárového kódu má funkce parametry `digit_count` (počet očekávaných číslic kódu), `l_coding` je požadované kódování levé číselné části (`L` nebo `R`) a `r_coding` pravé číselné části.

```
def barcode_valid(barcode: str, digit_count: int,
                  l_coding: str, r_coding: str) -> bool:
```

Vstupní podmínky tohoto predikátu se dotýkají pouze pomocných parametrů. Zapíšeme je jako tvrzení na začátku těla:

```
    assert l_coding == 'L' or l_coding == 'R'
    assert r_coding == 'L' or r_coding == 'R'
    assert digit_count % 2 == 0
```

Pro lepší čitelnost kódu si pojmenujeme několik užitečných konstant.

```
    boundary_size = 3
    center_size = 5
    total_marker_size = 2 * boundary_size + center_size
```

Nejprve zkontrolujeme, má-li čárový kód správnou délku: musí obsahovat dvě krajové a jednu středovou skupinu a sudý počet pruhů, které kódují číslice.

```
    if len(barcode) < total_marker_size:
        return False  # not enough space for all required markers
    if (len(barcode) - total_marker_size) % 2 != 0:
        return False  # does not evenly split into halves

    half_width = barcode_half_width(barcode)
    center_start = boundary_size + half_width
    center_end = center_start + center_size
```

Dále prověříme, že krajové a středová značka mají správné hodnoty.

```
    if barcode[:boundary_size] != '101':
```

```
        return False  # bad start marker
    if barcode[-boundary_size:] != '101':
        return False  # bad end marker
    if barcode[center_start:center_end] != '01010':
        return False
```

Nakonec zkontrolujeme, že má správně zakódované číslice. Zde uplatníme několik pomocných funkcí, kterých definice uvidíme později: (čistá) funkce barcode_digits z čárového kódu extrahuje dvě číslice-kódující oblasti, predikát barcode_valid_digits ověří, že vstupní číselná oblast správně kóduje číslice.

```
    if half_width % 7 != 0:
        return False
    if 2 * half_width // 7 != digit_count:
        return False

    left, right = barcode_digits(barcode)

    if not barcode_valid_digits(left, l_coding):
        return False
    if not barcode_valid_digits(right, r_coding):
        return False

    return True
```

Pomocná funkce pro výpočet délky jedné ze dvou číslicových oblastí čárového kódu, v počtu pruhů. Vstupní podmínkou je správná délka kódu (taková, aby se dal správně rozdělit na příslušné oblasti). Vstupní podmínku opět zapíšeme pomocí příkazů assert.

```
def barcode_half_width(barcode: str) -> int:
    assert len(barcode) >= 11
    assert (len(barcode) - 11) % 2 == 0
    return (len(barcode) - 11) // 2
```

Jak již bylo zmíněno, funkce barcode_digits extrahuje z čárového kódu dvě číselné oblasti. Potřebné vstupní podmínky již kontroluje pomocná funkce barcode_half_width kterou hned na začátku voláme, nebudeme je tedy ve funkci barcode_digits opakovat.

```
def barcode_digits(barcode: str) -> Tuple[str, str]:
    half_width = barcode_half_width(barcode)
    left = barcode[3:3 + half_width]
    right = barcode[8 + half_width:-3]
    return (left, right)
```

Dále potřebujeme být schopni kódovat a dekódovat jednotlivé číslice, k čemu nám poslouží následující dvojice funkcí. V druhém parametru zadáváme, které kódování číslic požadujeme (L nebo R). V kódovací funkci je vstupní podmínkou jednak správnost druhého parametru, ale také to, že digit je skutečně jediná číslice.

```
def barcode_encode_digit(digit: int, coding: str) -> str:
    assert 0 <= digit <= 9
    assert coding == 'L' or coding == 'R'

    codes = ['0001101', '0011001', '0010011', '0111101', '0100011',
             '0110001', '0101111', '0111011', '0110111', '0001011']

    code = ''
    for area in codes[digit]:
        if coding == 'L':
            code += area
        if coding == 'R':
            code += '1' if area == '0' else '0'
    return code
```

Dekódování číslic provedeme „hrubou silou“ (lze to i lépe, ale pro tuto chvíli k tomu úplně nemáme ty správné jazykové prostředky). Vstupní podmínkou je, že v code je správný počet pruhů (7). Nepovede-li se číslici v zadaném kódování přečíst, funkce vrátí None.

```
def barcode_decode_digit(code: str, coding: str) -> Optional[int]:
    assert len(code) == 7
    for digit in range(10):
        if barcode_encode_digit(digit, coding) == code:
            return digit
    return None
```

Nyní jsme již připraveni definovat predikát, který bude kontrolovat správné kódování dané číselné oblasti. Jednak musí ověřit správnou délku. Jestli délka vyhovuje, opakovaným použitím funkce barcode_decode_digit se pokusíme jednotlivé číslice přečíst – selže-li tato funkce na některé skupině sedmi pruhů, je kód neplatný.

```
def barcode_valid_digits(areas: str, coding: str) -> bool:
    if len(areas) % 7 != 0:
        return False
    while len(areas) > 0:
        if barcode_decode_digit(areas[0:7], coding) is None:
            return False
        areas = areas[7:]
    return True
```

Konečně můžeme přistoupit k samotnému kódování a dekódování číselných oblastí čárového kódu. Dekódovat lze pouze platnou číselnou oblast, vstupní podmínkou je tedy pravdivost predikátu barcode_valid_digits. Je tedy odpovědnost volajícího špatné čárové kódy zamítnout před pokusem o jejich dekódování (lze k tomu využít třeba právě predikátu barcode_valid_digits, není-li platnost zaručena jinak).

```
def barcode_decode(areas: str, coding: str) -> str:
    assert barcode_valid_digits(areas, coding)
    result = ''
    while len(areas) > 0:
        digit = barcode_decode_digit(areas[:7], coding)
        areas = areas[7:]
```

Protože v areas je uložena platná číselná oblast, musí se nám povést každou jednotlivou číslici dekódovat.

```
        assert digit is not None
        result += str(digit)
    return result
```

Zbývá poslední funkce, která ze zadaných číslic vytvoří číselnou oblast čárového kódu. Vstupní podmínkou je zde to, že vstupní řetězec se skládá pouze z číslic. Všimněte si, že podmínku kontrolujeme postupně, vždy předtím, než se pokusíme danou číslici zakódovat.
Výstupní podmínkou je, že jsme vytvořili platnou číselnou oblast. Vzpomeňte si, že výstupní podmínka je (v případě čisté funkce) vlastnost návratové hodnoty, kterou funkce sama zaručuje. Výstupní podmínku zapisujeme jako tvrzení (assert) před návratem z funkce.

```
def barcode_encode(digits: str, coding: str) -> str:
    result = ''
    for digit in digits:
        assert digit.isdigit()
        result += barcode_encode_digit(int(digit), coding)
    assert barcode_valid_digits(result, coding)
    return result

def main() -> None:  # demo
    assert not barcode_valid('111', 0, 'L', 'L')
    assert barcode_valid('10101010101', 0, 'L', 'L')
    code_27_good = '101' + '0010011' + '01010' + '0111011' + '101'
    code_27_bad1 = '101' + '0010011' + '01010' + '0101011' + '101'
    code_27_bad2 = '101' + '0010111' + '01010' + '0111011' + '101'
    code_1337_good = '101' + '0011001' + '0111101' + '01010' \
        + '1000010' + '1000100' + '101'
    assert barcode_valid(code_27_good, 2, 'L', 'L')
    assert barcode_valid(code_1337_good, 4, 'L', 'R')
    assert not barcode_valid(code_27_bad1, 2, 'L', 'L')
```

```
    assert not barcode_valid(code_27_bad2, 2, 'L', 'L')
    code_27_l, code_27_r = barcode_digits(code_27_good)
    assert code_27_l == '0010011'
    assert code_27_r == '0111011'
    assert barcode_decode(code_27_l, 'L') == '2'
    assert barcode_decode(code_27_r, 'L') == '7'
    assert barcode_encode('13', 'L') == '00110010111101'
    assert barcode_encode('37', 'R') == '10000101000100'
```

5.1.3 [ean] European Article Number (EAN) je systém číslování výrobků, který pravděpodobně znáte z čárových kódů v supermarketech. EAN funguje podobně jako ISBN, se kterým jste minulý týden pracovali v příkladu 04/isbn.py, nicméně neomezuje se na knihy. V této ukázce budeme pokračovat v používání tvrzení (assert) pro popis vstupních a výstupních podmínek funkcí. Protože budeme chtít převádět číselné kódy na čárové a obráceně, využijeme funkce pro práci s čárovými kódy, které jsme definovali v předchozí ukázce.

```
from barcode import barcode_valid, barcode_decode, barcode_encode, \
    barcode_digits
```

Podobně jako v případě ISBN budeme EAN reprezentovat jako řetězec. Jako první si zadefinujeme predikát, který bude rozhodovat, jedná-li se o platný EAN: postup je podobný jako pro ISBN, poslední cifra je kontrolní. EAN existuje v několika délkách, ale algoritmus pro jejich kontrolu je vždy stejný: proto dostane náš predikát krom samotného EAN jako parametr i očekávanou délku kódu. Tento predikát samotný nemá žádné vstupní podmínky.

```
def ean_valid(ean: str, length: int) -> bool:
    checksum = 0
    odd = length % 2 == 1
    if len(ean) != length:
        return False
    for digit in ean:
        if not digit.isdigit():
            return False
        checksum += int(digit) * ean_digit_weight(odd)
        odd = not odd

    return checksum % 10 == 0
```

Pomocná funkce, která popisuje váhy jednotlivých číslic v EAN kódu (pro účely výpočtu kontrolní číslice).

```
def ean_digit_weight(odd: bool) -> int:
    return 1 if odd else 3
```

Další funkce, kterou budeme definovat, slouží k vytvoření platného EAN-13 kódu z jednotlivých komponent: prefixu GS1 (zjednodušeně odpovídá zemi výrobce), kódu výrobce (který je minimálně pěticiferný) a kódu samotného výrobku. Vstupní podmínky odpovídají omezením na jednotlivé komponenty. Celková délka kódu bez kontrolního součtu musí být 12 cifer. Funkce komponenty zkombinuje a přidá kontrolní cifru. Výstupní podmínkou je, že jsme vytvořili platný třináctimístný EAN kód (kontrolujeme ji těsně před návratem z funkce).

```
def generate_ean(gs1: str, manufacturer: str, product: str) -> str:
    assert len(gs1) == 3
    assert len(manufacturer) >= 5
    assert len(gs1) + len(manufacturer) + len(product) == 12

    ean = gs1 + manufacturer + product
    odd = True
    check = 0
    for digit in ean:
        check += int(digit) * ean_digit_weight(odd)
        odd = not odd
    check = 10 - check % 10

    ean = gs1 + manufacturer + product + str(check)
    assert ean_valid(ean, 13)
    return ean
```

Následují dvě funkce pro konverzi mezi číselným a čárovým kódem. První dostane na vstupu platnou číselnou reprezentaci EAN-8 (tuto vstupní podmínku kontroluje první příkaz assert). Výstupní podmínkou naopak je, že funkce vytvoří platný čárový kód – tuto kontrolujeme, jak je obvyklé, těsně před návratem.

```
def ean8_to_barcode(ean: str) -> str:
    assert ean_valid(ean, 8)
    left = barcode_encode(ean[0:4], 'L')
    right = barcode_encode(ean[4:8], 'R')

    barcode = '101' + left + '01010' + right + '101'
    assert barcode_valid(barcode, 8, 'L', 'R')
    return barcode
```

Poslední funkce v tomto souboru slouží pro opačnou konverzi: z čárového kódu vytvoří číselnou reprezentaci. Vstupní podmínkou je, že čárový kód je platný a kóduje 8 číslic; toto díky predikátu barcode_valid lehce ověříme. Nicméně si musíme dát pozor na výstupní podmínku: mohlo by se zdát, že analogicky k předchozímu případu by bylo rozumné požadovat platnost číselného EAN.
Není tomu tak: byla-li splněna vstupní podmínka (čárový kód barcode je platný), funkce musí svoji výstupní podmínku vždy splnit. Musíme si ale uvědomit, že existují platné osmičíslicové čárové kódy, které nekódují platný EAN-8. Proto je výstupní podmínka platnosti EAN kódu příliš silná – nedokážeme ji zabezpečit.
Jako vhodné řešení se jeví v případě, kdy na vstupu dostaneme čárový kód reprezentující neplatný EAN, vrátit hodnotu None: výstupní podmínku tak zeslabíme jen minimálně. Bude vždy platit, že výstupem je buď platný EAN-8 (a to vždy, když je to možné), nebo hodnota None (pouze v případech, kdy vstup reprezentoval neplatný EAN-8). Ze zápisu návratové hodnoty je zřejmé, že tato výstupní podmínka je splněna, nemá tedy smysl ji dodatečně kontrolovat příkazem assert.

```
def barcode_to_ean8(barcode: str) -> Optional[str]:
    assert barcode_valid(barcode, 8, 'L', 'R')
    left, right = barcode_digits(barcode)
    ean = barcode_decode(left, 'L') + barcode_decode(right, 'R')
    if not ean_valid(ean, 8):
        return None
    return ean

def main() -> None:  # demo
    assert ean_valid("12345670", 8)
    assert ean_valid("1122334455666", 13)
    assert not ean_valid("12345674", 8)
    assert not ean_valid("1122334455664", 13)
    assert generate_ean("123", "123212", "123") == "1231232121235"
    assert generate_ean("444", "12345", "1111") == "4441234511119"
    assert ean8_to_barcode("12345670") == \
           "10100110010010011011110101000110" \
           "10101001110101000010001001110010101"
    assert ean8_to_barcode("11112228") == \
           "10100110010011001001100100110010101" \
           "01101100110110011011001001000101"
    assert barcode_to_ean8("10100110010010011011110101000110"
           "10101001110101000010001001110010101") == "12345670"
    assert barcode_to_ean8("10100110010011001001100100110010101"
           "01101100110110011011001001000101") == "11112228"
    assert barcode_to_ean8("10100110010011001001100100110010101"
           "01101100110110011011001110010101") is None
```

Část 5.2: Rozcvička

5.2.1 [trivial] Otypujte následující funkce tak, aby prošla typová kontrola s přiloženými testy.

Funkce degrees konvertuje radiány na stupně.

```
def degrees(radians):
    return (radians * 180) / pi
```

Funkce to_list rozdělí řetězec podle čárek do seznamu.

```
def to_list(raw):
    return [s.strip() for s in raw.split(',')]
```

Funkce diagonal vytvoří seznam obsahující prvky na diagonále matice matrix.

```
def diagonal(matrix):
    diag = []
    for i in range(len(matrix)):
        diag.append(matrix[i][i])
    return diag
```

Funkci with_id je v parametru elements předán seznam dvojic (celočíselný klíč, řetězec). Funkce najde prvek s klíčem id a vrátí odpovídající řetězec.

```
def with_id(elements, id):
    for element_id, val in elements:
        if id == element_id:
            return val
    return None
```

Funkce format_student naformátuje informace o studentovi do řetězce. Student je popsán trojicí: učo, jméno a volitelný atribut rok ukončení studia.

```
def format_student(student):
    uco, name, graduated = student
    if graduated:
        return str(uco) + ", " + name + ", " + str(graduated)
    return str(uco) + ", " + name
```

Predikát is_increasing je pravdivý, pokud je seznam celých čísel seq rostoucí.

```
def is_increasing(seq):
    for i in range(1, len(seq)):
        if seq[i - 1] >= seq[i]:
            return False
    return True
```

5.2.2 [clocks] Napište a otypujte funkci parse_time, která z řetězce ve 24-hodinovém formátu hodiny:minuty přečte čas do dvojice celých čísel (hodiny, minuty). Není-li vstupní řetězec zadaného formátu nebo neobsahuje platný čas, funkce vrátí None.

```
def parse_time(input_string):
    pass
```

5.2.3 [fridays] Otypujte následující implementaci příkladu 02/fridays.py.

```
def is_leap(year):
    if (year % 400) == 0:
        return True
    if (year % 4) == 0 and not (year % 100) == 0:
        return True
    return False

def days_per_month(year, month):
    if month == 2:
        return 29 if is_leap(year) else 28
    if month == 4 or month == 6 or month == 9 or month == 11:
        return 30
    return 31

def is_friday(day_of_week):
    return day_of_week == 4

def fridays(year, day_of_week):
    count = 0
    for month in range(1, 13):
        days = days_per_month(year, month)
        for day in range(1, days + 1):
            if is_friday(day_of_week) and day == 13:
                count += 1
            day_of_week = (day_of_week + 1) % 7
    return count
```

Část 5.3: Příklady

5.3.1 [database] V této úloze budete pracovat databázovou s tabulkou. Tabulka je dvojice složená z hlavičky a seznamu záznamů. Hlavička obsahuje seznam názvů sloupců. Jeden záznam je tvořen seznamem hodnot pro jednotlivé sloupce tabulky (pro jednoduchost uvažujeme jenom hodnoty typu řetězec). Ne všechny hodnoty v záznamech musí být vyplněny – v tom případě mají hodnotu None.
Vaším úkolem bude nyní otypovat a implementovat následující funkce.
Funkce header vrátí hlavičku tabulky table.

```
def get_header(table):
    pass
```

Funkce records vrátí seznam záznamů z tabulky table.

```
def get_records(table):
    pass
```

Procedura add_record přidá záznam record na konec tabulky table. Můžete předpokládat, že záznam record bude mít stejný počet sloupců jako tabulka.

```
def add_record(record, table):
    pass
```

Predikát is_complete je pravdivý, neobsahuje-li tabulka table žádnou hodnotu None.

```
def is_complete(table):
    pass
```

Funkce index_of_column vrátí index sloupce se jménem name. Můžete předpokládat, že sloupec s jménem name se v tabulce nachází. První sloupec má index 0.

```
def index_of_column(name, header):
    pass
```

Funkce values vrátí seznam platných hodnot (tzn. takových, které nejsou None) v sloupci se jménem name. Můžete předpokládat, že sloupec se jménem name se v tabulce nachází.

```
def values(name, table):
    pass
```

Procedura drop_column smaže sloupec se jménem name z tabulky table. Můžete předpokládat, že sloupec se jménem name se v tabulce nachází.

```
def drop_column(name, table):
    pass
```

5.3.2 [points] Vraťme se k ukázkovému příkladu 03/points.py, kde Vám byly představeny n-tice. Při takto komplikovaných typech je vhodné funkce otypovat, jak pro čitelnost, tak pro jednodušší hledání chyb. Vaším úkolem bude nyní otypovat funkce i testovací procedury a případně proměnné, tak, aby Vám prošla typová kontrola. Doporučujeme si zavést typové aliasy pro opakující se jednoznačně pojmenova-

telné typy.
Funkce `distance` spočte Euklidovskou vzdálenost dvou bodů `a` a `b`.

```
def distance(a, b):
    a_x, a_y, _ = a
    b_x, b_y, _ = b
    return sqrt((a_x - b_x) ** 2 + (a_y - b_y) ** 2)
```

Funkce `leftmost_colour` v neprázdném seznamu bodů najde barvu „nejlevějšího“ bodu (takového, který má nejmenší x-ovou souřadnici).

```
def leftmost_colour(points):
    x_min, _, result = points[0]

    for x, _, colour in points:
        if x < x_min:
            x_min = x
            result = colour

    return result
```

Dále funkce `center_of_gravity` dostane jako parametry seznam bodů `points` a barvu `colour`; jejím výsledkem bude bod, který se nachází v těžišti soustavy bodů dané barvy (a který bude stejné barvy). Vstupní podmínkou je, že `points` obsahuje alespoň jeden bod barvy `colour`.

```
def center_of_gravity(points, colour):
    total_x = 0
    total_y = 0
    count = 0
    for p_x, p_y, p_colour in points:
        if colour == p_colour:
            total_x += p_x
            total_y += p_y
            count += 1

    return (total_x / count, total_y / count, colour)
```

Jako poslední si definujeme funkci `average_nonmatching_distance`, která spočítá průměrnou vzdálenost bodů různé barvy. Vstupní podmínkou je, že seznam `points` musí obsahovat alespoň dva různobarevné body.

```
def average_nonmatching_distance(points):
    total = 0
    pairs = 0

    for i in range(len(points)):
        for j in range(i):
            _, _, i_colour = points[i]
            _, _, j_colour = points[j]
            if i_colour != j_colour:
                total += distance(points[i], points[j])
                pairs += 1

    return total / pairs
```

5.3.3 [course] V této úloze bude Vaším úkolem implementovat a otypovat následující funkce, které implementují dotazy na školní kurzy. Kurz je reprezentován seznamem dvojic (student, známka), přičemž student je trojice (učo, jméno, semestr) a známka je řetězec z rozsahu `A` až `F`.
Funkce `failed` vrátí seznam studentů kurzu `course`, kteří z něj mají známku `F`.

```
def failed(course):
    pass
```

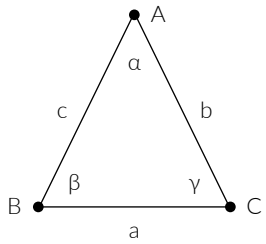
Funkce `count_passed` vrátí počet studentů, kteří úspěšně ukončili kurz `course`, tedy z něj nemají známku `F`. Parametr `semester` je volitelný: je-li specifikován (není `None`), funkce vrátí počet úspěšných studentů v daném semestru, jinak vrátí počet všech úspěšných studentů.

```
def count_passed(course, semester):
    pass
```

Funkce `student_grade` vrátí známku studenta s učem `uco`. Pokud takový student v kurzu `course` není, vrací `None`.

```
def student_grade(uco, course):
    pass
```

5.3.4 [triangle] V této úloze bude Vaším úkolem rozšířit a otypovat implementaci z ukázky `02/triangle.py`.
Strany trojúhelníku značíme a, b, c. Úhel mezi a a b je γ (`gamma`), mezi `b` a `c` je α (`alpha`) a mezi c a a je úhel β (`beta`):



1. Prvním úkolem bude implementovat obecnou funkci `perimeter`, která má volitelné parametry tří stran a tří úhlů trojúhelníku. Je-li to možné z předaných parametrů, funkce spočítá obvod trojúhelníku jednou z metod SSS, ASA, SAS, jinak vrátí `None`.

2. Druhým úkolem bude otypovat zbytek pomocných funkcí tak, aby Vám prošla typová kontrola. Typ funkce `perimeter` neměňte.

```
def perimeter(a: Optional[float],
              b: Optional[float],
              c: Optional[float],
              alpha: Optional[float],
              beta: Optional[float],
              gamma: Optional[float]) -> Optional[float]:
    pass
```

Funkce `perimeter_sss` spočte obvod trojúhelníku zadaného třemi stranami.

```
def perimeter_sss(a, b, c):
    return a + b + c
```

Funkce `perimeter_sas` spočte obvod trojúhelníku zadaného dvěma stranami a nimi sevřeným úhlem.

```
def perimeter_sas(a, angle, b):
    c = sqrt(a ** 2 + b ** 2 - 2 * a * b * cos(radians(angle)))
    return perimeter_sss(a, b, c)
```

Funkce `perimeter_asa` spočte obvod trojúhelníku zadaného stranou a jí přilehlých úhlů.

```
def perimeter_asa(alpha, c, beta):
    gamma = radians(180 - alpha - beta)
    alpha = radians(alpha)
    beta = radians(beta)
    a = c * sin(alpha) / sin(gamma)
    b = c * sin(beta) / sin(gamma)
    return perimeter_sss(a, b, c)
```

5.3.5 [gps] Napište a otypujte funkci `parse_gps`, která přečte GPS souřadnice ze vstupního řetězce `raw`. Očekávaný vstup je ve formátu `lat=X;lon=Y` kde `X` a `Y` jsou čísla. Není-li vstup není v tomto formátu, funkce vrací `None`, jinak vrátí dvojici s číselnými hodnotami zeměpisné šířky (latitude) a délky (longitude). Nespadají-li hodnoty do platného rozsahu souřadnic, funkce vrací `None`: zeměpisná šířka je v rozmezí -90 až 90 a délka v rozmezí -180 až 180.

```
def parse_gps(raw):
```

```
    pass
```

Dále napište a otypujte funkci `parse_gps_stream`, která přečte seznam GPS souřadnic a vrátí seznam dvojic s číselnými hodnotami souřadnic. Souřadnice na vstupu jsou každá na vlastním řádku. Nekóduje-li kterýkoliv řádek GPS souřadnici, funkce vrátí `None`.

```
def parse_gps_stream(raw):
    pass
```

5.3.6 [doctor] V této úloze bude Vaším úkolem implementovat funkce pracující se seznamem pacientů `patients` u lékaře. Každý pacient má záznam (dvojici), který obsahuje jeho unikátní identifikátor a seznam návštěv s výsledky. Návštěva je reprezentovaná čtveřicí – rokem, kdy pacient navštívil lékaře, a naměřenými hodnotami: pulz, systolický a diastolický tlak. Seznam návštěv pacienta je uspořádaný vzestupně od nejstarší. Můžete předpokládat, že každý pacient má alespoň jeden záznam.
Vaším prvním úkolem bude implementovat a otypovat funkci `missing_visits`, která zjistí, kteří pacienti nebyli na prohlídce od roku `year`. Jako výsledek vraťte seznam identifikátorů pacientů.

```
def missing_visits(year, patients):
    pass
```

Dále napište a otypujte funkci `patient_reports`, která vrátí seznam zpráv o pacientech. Zpráva o pacientovi je čtveřice, která obsahuje záznam o jeho nejvyšším doposud naměřeném pulzu a pro každou měřenou hodnotu informaci, zda se měření dané hodnoty v jednotlivých letech konzistentně zvyšují (`True` nebo `False`).
Například zpráva o pacientovi `(1, [(2015, 91, 120, 80), (2018, 89, 125, 82), (2020, 93, 120, 88)])` je `(93, False, False, True)`.

```
def patient_reports(patients):
    pass
```

Část 5.4: Bonusy

5.4.1 [parse_time] Napište a otypujte funkci `parse_datetime`, která z řetězce ve formátu `den/měsíc/rok hodiny:minuty` přečte datum a čas. Návratovou hodnotou funkce bude dvojice `(datum, čas)`, kde `datum` je trojice ve tvaru `(den, měsíc, rok)` a `čas` je dvojice `(hodiny, minuty)`. Všech pět položek je typu `int`. Není-li vstupní řetězec v zadaném formátu, nebo neobsahuje platné datum a čas, funkce vrátí `None`.

```
def parse_datetime(input_string):
    pass
```

Část 5.5: Řešení

5.5.1 [trivial.sol]

```
def degrees(radians: float) -> float:
    return (radians * 180) / pi

def diagonal(lst: List[List[int]]) -> List[int]:
    diag = []
    for i in range(len(lst)):
        diag.append(lst[i][i])
    return diag

def to_list(raw: str) -> List[str]:
    return [s.strip() for s in raw.split(',')]

Element = Tuple[int, str]

def with_id(elements: List[Element], id: int) -> Optional[str]:
    for element_id, val in elements:
        if id == element_id:
            return val
    return None

Student = Tuple[int, str, Optional[int]]

def format_student(student: Student) -> str:
    uco, name, graduated = student
    if graduated:
        return str(uco) + ", " + name + ", " + str(graduated)
    return str(uco) + ", " + name

def is_increasing(seq: List[int]) -> bool:
    for i in range(1, len(seq)):
        if seq[i - 1] >= seq[i]:
            return False
    return True
```

5.5.2 [clocks.sol]

```
Time = Tuple[int, int]
Range = Tuple[int, int]

def parse_int(raw: str, range: Optional[Range]) -> Optional[int]:
    if not raw.isdigit():
        return None

    value = int(raw)
    if not range:
        return value
    low, high = range
    if low <= value <= high:
        return value
    return None

def parse_time(input_string: str) -> Optional[Time]:
    raw = input_string.split(":")
    if len(raw) != 2:
        return None

    hours = parse_int(raw[0], (0, 23))
    minutes = parse_int(raw[1], (0, 59))
    if hours is None or minutes is None:
        return None
    return (hours, minutes)
```

5.5.3 [fridays.sol]

```
Day = int
Year = int
Month = int

def is_leap(year: Year) -> bool:
    if (year % 400) == 0:
        return True
    if (year % 4) == 0 and not (year % 100) == 0:
        return True
    return False

def days_per_month(year: Year, month: Month) -> int:
    if month == 2:
        return 29 if is_leap(year) else 28
    if month == 4 or month == 6 or month == 9 or month == 11:
        return 30
    return 31

def is_friday(day_of_week: Day) -> bool:
    return day_of_week == 4

def fridays(year: Year, day_of_week: Day) -> int:
    count = 0
    for month in range(1, 13):
        days = days_per_month(year, month)
        for day in range(1, days + 1):
```

```
        if is_friday(day_of_week) and day == 13:
            count += 1
        day_of_week = (day_of_week + 1) % 7
    return count
```

Část 6: Datové struktury

V této kapitole se budeme opět zabývat zabudovanými datovými strukturami: z třetí kapitoly již známe seznam a n-tici, tento týden přibudou zásobník (stack), fronta (queue), slovník (dictionary) a množina (set). Ukázky:

1. `hills.py` – použití zásobníku k sledování nadmořské výšky
2. `digram.py` – použití slovníku pro frekvenční analýzu
3. `closure.py` – práce s množinami čísel
4. `factory.py` ⋆ – výrobní fronta

Řešené příklady:

1. `brackets.py` – kontrola uzávorkování s více druhy závorek
2. `symmetric.py` – kontrola symetričnosti relace
3. `morse.py` – překlad z/do Morseovy abecedy

Neřešené příklady:

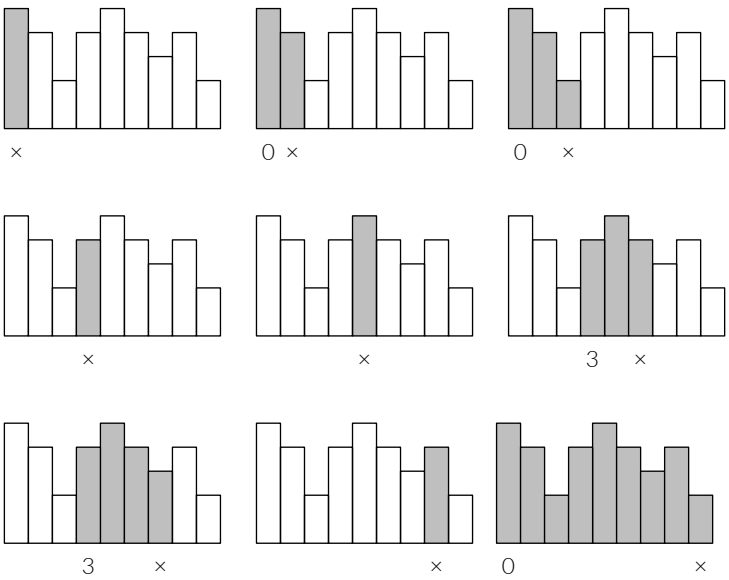
1. `rpn.py` – vyhodnocení výrazů v postfixovém zápisu
2. `transitive.py` – tranzitivní relace
3. `b_happy.py` – rozhodování iterativně zadané vlastnosti čísel
4. `flood.py` – vyplňování jednobarevné plochy v rastrovém obrázku
5. `histogram.py` – statistické zpracování jednorozměrného signálu
6. `alchemy.py` – výroba substancí podle sady pravidel
7. `shortest.py` – procházení jednoduchého bludiště
8. `fixpoint.py` – hledání pevného bodu množinové funkce

Bonusové (těžší) příklady:

1. `connectivity.py` – spojitost sítě MHD
2. `lakes.py` – jezírka v krajině

Část 6.1: Ukázky

6.1.1 [`hills`] Uvažme následovný problém: na vstupu máme výškový profil trasy, a zajímá nás, jak dlouho jsme se pohybovali ve výšce aspoň takové, v jaké jsme teď. Zajímavé hodnoty budeme samozřejmě dostávat pouze na sestupu. Například (aktuální pozici budeme značit symbolem × a odpovídající úsek vyšší nadmořské výšky vybarvíme):



Definujeme tedy čistou funkci `hills`, která dostane na vstupu seznam výšek (celých čísel) a které výsledkem bude stejně dlouhý seznam indexů, které odpovídají vždy prvnímu vybarvenému sloupci v ilustraci výše.

```
def hills(heights: List[int]) -> List[int]:
```

V proměnné `stack` budeme udržovat zásobník, který bude obsahovat indexy všech předchozích vrcholů, které jsou nižší než ten aktuální. Do proměnné `indices` budeme počítat výsledný seznam indexů.

```
    stack: List[int] = []
    indices: List[int] = []
    for i in range(len(heights)):
        while len(stack) > 0 and heights[stack[-1]] >= heights[i]:
            stack.pop()
        if len(stack) == 0:
            indices.append(0)
        else:
            indices.append(stack[-1] + 1)
        stack.append(i)
    return indices
```

Funkčnost ověříme na několika příkladech (seznam `example` odpovídá obrázku výše).

```
def main() -> None:  # demo
    assert hills([1, 2, 3]) == [0, 1, 2]
    assert hills([3, 2, 1]) == [0, 0, 0]
    assert hills([1, 2, 1]) == [0, 1, 0]
    assert hills([2, 2, 2]) == [0, 0, 0]
    assert hills([1, 2, 3, 2]) == [0, 1, 2, 1]
    assert hills([1, 3, 2, 3]) == [0, 1, 1, 3]
    assert hills([3, 1, 3, 2]) == [0, 0, 2, 2]
    example = [4, 3, 1, 3, 4, 3, 2, 3, 1]
    assert hills(example) == [0, 0, 0, 3, 4, 3, 3, 7, 0]
```

6.1.2 [`digram`] V této ukázce demonstrujeme použití slovníku (`dict`) k jednoduché frekvenční analýze textu, konkrétně výpočtu relativní frekvence digramů, kde:

- digram je dvojice (po sobě jdoucích) písmen,
- relativní frekvencí myslíme poměr počtu výskytů daného digramu vůči celkovému počtu digramů.

Implementace bude čistá funkce, která na vstupu dostane řetězec a výstupem bude slovník, kde klíče budou dvojpísmenné řetězce a hodnoty budou typu `float` v rozsahu 0–1 takové, že součet všech hodnot v slovníku bude 1.

```
def digram_frequency(text: str) -> Dict[str, float]:
    freq: Dict[str, float] = {}
```

Nejprve potřebujeme zjistit samotné počty výskytů jednotlivých digramů, které v dalším cyklu normalizujeme. Procházíme všechny dvojpísmenné podřetězce, ale dále se zabýváme pouze těmi, které sestávají ze dvou písmen (tzn. mezery, interpunkci, číslice atd. ignorujeme). Slovník indexujeme klíčem – indexace funguje podobně jako u seznamu, s tím rozdílem, že indexy nemusí být celá čísla ani nemusí tvořit souvislou řadu. Pokus o indexaci klíčem, který ve slovníku zatím neexistuje je chyba. Zde stojí za připomenutí, že operátor `x += y` je obvykle zkratka pro `x = x + y`, a tedy `dict[key] += y` se chová jako `dict[key] = dict[key] + y`, tedy pokus o indexaci klíčem `key` na pravé straně proběhne dříve, než samotné vložení klíče vnitřním přiřazením.

```
    for i in range(len(text) - 1):
        digram = text[i:i + 2].lower()
        if digram.isalpha():
```

Na ověření přítomnosti klíče ve slovníku můžeme použít operátor `in` resp. `not in`.

```
        if digram not in freq:
            freq[digram] = 0
        freq[digram] += 1
```

Metoda `values` nám umožňuje chovat se k hodnotám uloženým ve slovníku jako k sekvenci (klíče zapomene). Sečíst všechny hodnoty ve slovníku je tedy velmi jednoduché.

```
    total = sum(freq.values())
```

Metoda `items` naopak ze slovníku vyrobí sekvenci dvojic (klíč, hodnota) – tuto sekvenci lze iterovat (stejně jako seznam) a jednotlivé dvojice lze rozbalit do proměnných.

```
    for key, value in freq.items():
        freq[key] = value / total

    return freq
```

Jako obvykle si ověříme, že funkce `digram_frequency` se na jednoduchých vstupech chová tak, jak čekáme:

```
def main() -> None:  # demo
    freq = digram_frequency("lorem ipsum dolor sit amet")
    assert isclose(freq['lo'], 2 / 17)
    assert isclose(freq['or'], 2 / 17)
    assert isclose(freq['am'], 1 / 17)
    assert 'rs' not in freq
    assert ' i' not in freq

    freq = digram_frequency("hrad Veveří nehoří")
    assert isclose(freq['ve'], 2 / 13)
    assert isclose(freq['ří'], 2 / 13)
    assert isclose(freq['ne'], 1 / 13)

    freq = digram_frequency("ехал Грека через реку")
    assert isclose(freq['пе'], 3 / 14)
    assert isclose(freq['ек'], 2 / 14)
```

6.1.3 [closure] V této ukázce se budeme zabývat datovým typem množina. Stejně jako u seznamů, slovníků a podobně se jedná o složený typ, který má prvky. Množina má některé vlastnosti společné jak se seznamem – obsahuje pouze prvky, ale nikoliv klíče, tak se slovníkem – podobně jako klíče ve slovníku, hodnoty v množině můžou být přítomny nejvýše jednou. Od seznamu se liší mimo jiné tím, že množinu nelze indexovat (pouze iterovat).

Krom omezení na výskyt každého prvku nejvýše jednou poskytuje množina efektivní test na přítomnost prvku (podobně, jako slovník poskytuje efektivní test na přítomnost klíče). Chceme-li zjistit, objevuje-li se nějaká hodnota v běžném seznamu, strávíme tím čas, který je přímo úměrný počtu prvků tohoto seznamu. Naopak v množině lze očekávat, že čas potřebný pro zjištění přítomnosti na počtu prvků v množině vůbec nezávisí: trvá přibližně stejně dlouho nalézt prvek v množině o deseti prvcích i v množině o deseti milionech prvků (takto to funguje v Pythonu – tato operace má očekávanou konstantní složitost; některé jiné jazyky poskytují datový typ množina, kde čas potřebný k zjištění přítomnosti prvku závisí na tom, kolik řádů má číslo popisující její velikost – mluvíme pak o tzv. logaritmické složitosti).

Uvažme zobrazení $f : A \times A \to A$ kde $A \subseteq \mathbb{Z}$ a f je zadané tabulkou (slovníkem, kde klíč je dvojice čísel a hodnota je číslo – rozmyslete si, že takový slovník skutečně reprezentuje tabulku, budou-li ve slovníku přítomny všechny potřebné dvojice). Například logickou spojku and lze podobnou tabulkou reprezentovat takto (budeme-li reprezentovat True číslem 1 a False číslem 0):

	0	1
0	0	0
1	0	1

Jako slovník bychom stejnou tabulku zapsali takto:

```
{(0, 0): 0, (0, 1): 0,
 (1, 0): 0, (1, 1): 1}.
```

Zobrazení f budeme říkat operace a budeme jej popisovat následujícím typem:

```
Operation = Dict[Tuple[int, int], int]
```

Na vstupu tedy dostaneme tabulku, která reprezentuje f a množinu čísel $B \subseteq A$. Naším úkolem bude nalézt nejmenší množinu čísel C takovou, že:

- $B \subseteq C$, tedy C obsahuje všechny zadané prvky,
- pro každé $(x, y) \in B \times B$ platí $f(x, y) \in B$ – říkáme, že množina C je uzavřena na operaci f.

```
def closure(set_b: Set[int], operation_f: Operation) -> Set[int]:
```

Jak budeme postupovat? Množinu C budeme budovat postupně: začneme tím, že do C vložíme všechny prvky z B:

```
    set_c = set_b.copy()
```

Dále budeme procházet všechny dvojice ze součinu $C \times C$, a nalezneme-li takovou, že její obraz ještě v množině C není, přidáme jej tam. Toto ale nemůžeme udělat přímo: přidat prvek do množiny, kterou právě iterujeme je zakázáno (protože by bylo těžké zaručit, aby byla iterace konzistentní – tzn. aby se nestalo, že v iteraci uvidíme některé, ale ne všechny, nové prvky).

Proto si napíšeme pomocnou funkci `find_missing`, která najde chybějící prvky a vrátí je jako množinu. Stojíme před dvěma problémy: po přidání nových prvků musíme celou proceduru opakovat, protože vznikly nové dvojice. Tento problém vyřešíme tak, že budeme funkci `find_missing` volat opakovaně, tak dlouho, dokud bude nalézat nové prvky.

Druhý problém je, že tento postup není příliš efektivní: rádi bychom se vyhnuli procházení dvojic, které jsme již kontrolovali. To sice samozřejmě lze, ale značně by nám to zkomplikovalo kód, proto tentokrát ušetříme práci sobě (a nějakou tím přiděláme počítači).

```
    to_add = find_missing(set_c, operation_f)

    while len(to_add) != 0:
        for x in to_add:
            set_c.add(x)
        to_add = find_missing(set_c, operation_f)

    return set_c
```

Pomocná (čistá) funkce `find_missing` je velmi jednoduchá: projde všechny dvojice z $C \times C$ (tedy součinu množiny `set_c` se sebou samou), a zobrazí-li se tato dvojice na prvek, který v `set_c` zatím není, přidá ho do své návratové hodnoty.

```
def find_missing(set_c: Set[int], operation_f: Operation) \
        -> Set[int]:
    result: Set[int] = set()

    for x in set_c:
        for y in set_c:
            to_add = operation_f[(x, y)]
            if to_add not in set_c:
                result.add(to_add)

    return result
```

Zbývá otestovat, že funkce `closure` se chová jak čekáme.

```
def main() -> None:  # demo
    op_and = {(0, 0): 0, (0, 1): 0, (1, 0): 0, (1, 1): 1}
    op_xor = {(0, 0): 0, (1, 0): 1, (0, 1): 1, (1, 1): 0}
    set_false = set([0])
    set_true = set([1])
    set_both = set([0, 1])

    assert closure(set_false, op_and) == set_false
    assert closure(set_true, op_and) == set_true
    assert closure(set_both, op_and) == set_both
    assert closure(set_false, op_xor) == set_false
    assert closure(set_true, op_xor) == set_both

    add_mod4 = {(0, 0): 0, (0, 1): 1, (0, 2): 2, (0, 3): 3,
                (1, 0): 1, (1, 1): 2, (1, 2): 3, (1, 3): 0,
                (2, 0): 2, (2, 1): 3, (2, 2): 0, (2, 3): 1,
                (3, 0): 3, (3, 1): 0, (3, 2): 1, (3, 3): 2}

    assert closure(set([0]), add_mod4) == set([0])
    assert closure(set([1]), add_mod4) == set([0, 1, 2, 3])
    assert closure(set([2]), add_mod4) == set([0, 2])
    assert closure(set([3]), add_mod4) == set([0, 1, 2, 3])
    assert closure(set([0, 2]), add_mod4) == set([0, 2])
```

6.1.4 [factory] ⋆ V této ukázce si implementujeme jednoduchou „továrnu“ která vyrábí různé typy výrobků. Výrobek má název a seznam součástek (jiných výrobků), které k jeho výrobě potřebujeme. Továrna má sklad, kde má součástky uloženy (v různém množství), a frontu požadavků na výrobu. Mít mnoho součástek na skladě je drahé, proto se snažíme skladové zásoby minimalizovat.
Nejprve si zadefinujeme potřebné typy, které budou popisovat stav továrny. Popis výrobků: každému výrobku (popsanému názvem) přiřadíme slovník, který popisuje jaké součástky výroba vyžaduje a v jakém množství.

```
Products = Dict[str, Dict[str, int]]
```

Datový typ `Warehouse` popisuje aktuální skladové zásoby: kolik kusů kterého výrobku máme uskladněno. Hodnoty zde uložené jsou vždy nezáporné.

```
Warehouse = Dict[str, int]
```

Pracovní fronta `Queue` popisuje jaké výrobky máme naplánované k výrobě, zatímco slovník `InFlight` popisuje jak se projeví zpracování aktuální fronty na skladových zásobách: počítá nové výrobky naplánované k výrobě (kladná čísla), ale také rezervované součástky potřebné pro jejich výrobu (záporná čísla). Uvažme výrobek X, kterého výroba vyžaduje dvě součástky Y a jednu součástku Z. Když přidáme sedm kusů výrobku X do fronty – `('X', 7)` – musíme zároveň upravit slovník `InFlight`: hodnotu klíče `'X'` zvýšíme o 7, hodnotu klíče `'Y'` snížíme o 14 a hodnotu klíče `'Z'` snížíme o 7.

```
Queue = Deque[Tuple[str, int]]
InFlight = Dict[str, int]
```

Tím je popis továrny kompletní: samotná továrna má tedy 4 složky: popis výrobků a jejich součástek, skladové zásoby, pracovní frontu, a sumarizaci efektu fronty na skladové zásoby.

```
Factory = Tuple[Products, Warehouse, Queue, InFlight]
```

Pro zjednodušení procedur, které budou s továrnou pracovat, si nejprve zadefinujeme jednoduchou funkci, která vytvoří továrnu ve výchozím stavu: skladové zásoby nastaví na 0 a stejně tak inicializuje slovník `in_flight`. Nebudeme tedy později muset řešit situaci, že nějaký klíč není v některém slovníku vůbec přítomen.

```
def setup_factory(dependencies: Products) -> Factory:
    warehouse: Warehouse = {}
    in_flight: InFlight = {}
    queue: Queue = Deque()
    for product, _ in dependencies.items():
        warehouse[product] = 0
        in_flight[product] = 0
    return (dependencies, warehouse, queue, in_flight)
```

Nyní můžeme přistoupit k definici procedur, které budou stav továrny měnit: první procedura přidá požadavek na výrobu do fronty (a zároveň upraví slovník `InFlight`).

```
def queue_product(factory: Factory, product: str,
                  product_count: int) -> None:
    dependencies, warehouse, queue, in_flight = factory
```

Do fronty nejprve vložíme chybějící komponenty: tyto potřebujeme vyrobit předtím, než můžeme přistoupit k výrobě samotného výrobku `product`. Vkládat je budeme v takovém množství, abychom přesně doplnili stávající skladové zásoby na potřebný počet.

```
    for component, count in dependencies[product].items():
        have = warehouse[component] + in_flight[component]
        need = product_count * count
        if need > have:
            queue_product(factory, component, need - have)
        in_flight[component] -= need
        assert warehouse[component] + in_flight[component] >= 0
```

Poté můžeme vložit samotný požadavek na příslušný počet vyžádaných výrobků.

```
    queue.append((product, product_count))
    in_flight[product] += product_count
```

Kdykoliv během výroby může přijít požadavek na odeslání části skladových zásob jinam (buď zákazníkovi, nebo do jiné továrny). Kdykoliv se tak stane, hrozí, že nebudeme mít dostatek skladových zásob k výrobě některého již naplánovaného výrobku. Přeskládat frontu by bylo ale nákladné a nepraktické, proto chybějící komponenty zařadíme na konec fronty a s případným nedostatkem se vypořádáme později.

```
def ship_product(factory: Factory, product: str,
                 count: int) -> bool:
    _, warehouse, _, in_flight = factory
    if warehouse[product] < count:
        return False
    warehouse[product] -= count
    expected_total = warehouse[product] + in_flight[product]
    if expected_total < 0:
        queue_product(factory, product, -expected_total)
    return True
```

Poslední procedura zpracuje nejstarší požadavek ve frontě: použije skladové zásoby komponent k výrobě požadovaného počtu výrobku. Je-li fronta prázdná, procedura vrátí `False` a nic neudělá.
Přes veškeré naše úsilí se ale může stát, že na skladě není dostatek komponent, protože některé mohly být od vložení požadavku do fronty odeslány ze skladu pryč. V takové situaci vyrobíme tolik kusů, kolik nám skladové zásoby dovolují a zbytek vložíme na konec fronty.
Všimněte si také práci se slovníkem `in_flight`: protože jsme z fronty odstranili celý požadavek na n kusů, musíme upravit `in_flight` tak, aby odpovídal změně stavu fronty, bez ohledu na to, kolik kusů produktu se nakonec povede vyrobit. Naproti tomu změna ve stavu skladových zásob musí odrážet skutečně vyrobený počet kusů.

```
def build_product(factory: Factory) -> bool:
    dependencies, warehouse, queue, in_flight = factory
    if len(queue) == 0:
        return False

    product, product_count = queue.popleft()
```

```
    in_flight[product] -= product_count
    can_build = product_count

    for component, count in dependencies[product].items():
        can_build = min(can_build, warehouse[component] // count)
    for component, count in dependencies[product].items():
        warehouse[component] -= can_build * count
        in_flight[component] += product_count * count

    warehouse[product] += can_build
    if can_build < product_count:
        queue_product(factory, product, product_count - can_build)

    return True
```

Zbývá implementaci továrny otestovat. Za povšimnutí stojí, že procedury, zejména takové, které pracují se složitějším stavem, se testují o něco hůře, než čisté funkce.

```
def main() -> None:  # demo
    factory = setup_factory({'algorithm': {'idea': 1, 'coffee': 3},
                             'theorem': {'coffee': 7, 'goat': 1},
                             'coffee': {'beans': 1, 'water': 1},
                             'beans': {}, 'water': {},
                             'goat': {}, 'idea': {}})
    _, warehouse, _, in_flight = factory

    assert not build_product(factory)
    queue_product(factory, 'theorem', 3)
    assert warehouse['theorem'] == 0
    assert warehouse['coffee'] == 0
    assert warehouse['goat'] == 0
    assert in_flight['algorithm'] == 0
    assert in_flight['coffee'] == 0
    assert in_flight['theorem'] == 3

    for i in ['water', 'beans', 'goat', 'coffee']:
        assert build_product(factory)
    assert warehouse['goat'] == 3
    assert warehouse['coffee'] == 21
    assert in_flight['goat'] == -3
    assert in_flight['coffee'] == -21

    assert build_product(factory)
    assert warehouse['theorem'] == 3
    assert warehouse['goat'] == 0
    assert warehouse['coffee'] == 0
    assert in_flight['goat'] == 0
    assert in_flight['coffee'] == 0

    queue_product(factory, 'idea', 2)
    queue_product(factory, 'algorithm', 3)
    assert not ship_product(factory, 'idea', 1)
    assert build_product(factory)  # the two ideas
    assert ship_product(factory, 'idea', 1)

    for i in ['water', 'beans', 'coffee', 'idea']:
        assert build_product(factory)
    assert warehouse['idea'] == 2
    assert warehouse['coffee'] == 9
    assert build_product(factory)  # first batch of algorithms
    assert warehouse['coffee'] == 3
    assert warehouse['idea'] == 0
    assert warehouse['algorithm'] == 2
    assert build_product(factory)  # missing idea
    assert warehouse['idea'] == 1
    assert build_product(factory)  # algorithm
    assert warehouse['algorithm'] == 3
    assert warehouse['idea'] == 0
```

Část 6.2: Rozcvička

6.2.1 **[brackets]** Řetězec je korektně uzávorkován právě tehdy, když buďto neobsahuje závorky žádné, nebo pro každou závorku v něm existuje taková opačná závorka, že řetězec uvnitř tohoto páru závorek je korektně uzávorkován.
V této úloze budeme uvažovat následující typy závorek:

- kulaté závorky: otevírací `(` a uzavírací `)`,
- hranaté závorky `[`, `]`, a konečně
- složené závorky `{`, `}`.

Každá otevírací závorka musí být spárována s právě jednou uzavírací. Žádné další znaky za závorky nepovažujeme.
Například:

- řetězec `abc` je korektně uzávorkován, protože žádné závorky neobsahuje,
- řetězec `()` je správně uzávorkován, protože `(` lze spárovat s `)` tak, že vnitřní řetězec je prázdný a tedy dobře uzávorkovaný,
- `[()]` je také korektně uzávorkovaný protože `[` lze spárovat tak, že vnitřní řetězec je `()` o kterém již víme, že je správně uzávorkovaný, a konečně
- `([)]` korektně uzávorkován není, protože k `(` obsahuje jediný možný pár, ale vzniklý vnitřní řetězec `[` nemá pro `[` žádnou uzavírací závorku.

Napište predikát, kterého výsledek bude `True`, dostane-li v parametru korektně uzávorkovaný řetězec, `False` jinak.

```
def correct_parentheses(text: str) -> bool:
    pass
```

6.2.2 **[symmetric]** Jak jistě víte, binární relací nad danou množinou A je každá množina dvojic prvků z množiny A, tzn. relace nad A je podmnožina kartézského součinu $A \times A$. Daná relace se pak nazývá symetrická, platí-li pro všechny dvojice (a, b) z této relace, že se v relaci zároveň nachází i dvojice (b, a). V této úloze budeme pracovat s relacemi nad celými čísly.
Napište predikát, kterého hodnota bude `True` dostane-li v parametru symetrickou relaci, `False` jinak.

```
def is_symmetric(relation):
    pass
```

6.2.3 **[morse]** Morseova abeceda je způsob jednoduchého kódování písmen latinské abecedy používán v telegrafii. Jednotlivá písmena se v ní kódují následovně: A: `.-`, B: `-...`, C: `-.-.`, D: `-..`, E: `.`, F: `..-.`, G: `--.`, H: `....`, I: `..`, J: `.---`, K: `-.-`, L: `.-..`, M: `--`, N: `-.`, O: `---`, P: `.--.`, Q: `--.-`, R: `.-.`, S: `...`, T: `-`, U: `..-`, V: `...-`, W: `.--`, X: `-..-`, Y: `-.--`, Z: `--..`
Napište (čistou) funkci, která na vstupu dostane text složený z velkých písmen latinské abecedy (kterých kódy jsou popsány výše) a mezer, a vrátí tento text zakódovaný do Morseovy abecedy, přičemž mezi kódy jednotlivých písmen budou mezery a na místech, kde byly mezery v původním textu budou mezery ve výsledném kódu tři.

```
def morse_encode(text: str) -> str:
    pass
```

Dále napište (čistou) funkci, která na vstupu dostane text zakódovaný v Morseově abecedě (postupem popsaným výše) a vrátí původní text, ze kterého tento kód vznikl.

```
def morse_decode(code: str) -> str:
    pass
```

Část 6.3: Příklady

6.3.1 [rpn] Napište (čistou) funkci, která na vstupu dostane:

- neprázdný výraz `expr` složený z proměnných a z aritmetických operátorů, zapsaný v postfixové notaci, a
- slovník, přiřazující proměnným číselnou hodnotu (můžete se spolehnout, že všechny proměnné použité v daném výrazu jsou v tomto slovníku obsaženy),

a vrátí číslo, na které se daný výraz vyhodnotí. Mezi všemi operátory a proměnnými se v daném výrazu nachází mezera a povolené operátory jsou pouze `+` a `*`.

```
def rpn_eval(expr: str, variables: Dict[str, int]) -> int:
    pass
```

6.3.2 [transitive] Jak jsme již zmiňovali dříve, binární relací nad danou množinou je množina dvojic prvků z této množiny. Daná relace se pak nazývá tranzitivní, platí-li pro všechny dvojice (a, b), (b, c) z této relace, že se v relaci nachází i dvojice (a, c). V této úloze budeme opět pracovat s relacemi nad celými čísly.
Napište predikát, který rozhodne jestli je zadaná relace tranzitivní.

```
def is_transitive(relation: Set[Tuple[int, int]]) -> bool:
    pass
```

6.3.3 [b_happy] Dané přirozené číslo je b–šťastné platí-li, že nahradíme-li jej součtem druhých mocnin jeho cifer, vyjádřených v poziční soustavě se základem `b`, a tento postup budeme dále opakovat na takto vzniklém čísle, po konečném počtu kroků dostaneme číslo 1.
Například číslo 3 je 4–šťastné, protože $3 = (3)_4$; $3^2 = 9 = (21)_4$; $2^2+1^2 = 5 = (11)_4$; $1^2 + 1^2 = 2 = (2)_4$; $2^2 = 4 = (10)_4$; $1^2 + 0^2 = 1$.
Číslo 2 není 5–šťastné, protože $2 = (2)_5$; $2^2 = 4 = (4)_5$; $4^2 = 16 = (31)_5$; $3^2 + 1^2 = 10 = (20)_5$; $2^2 + 0^2 = 4$ a protože se nám ve výpočtu číslo 4 zopakovalo, nemůžeme již dojít k výsledku 1.
Napište predikát, který o číslu `number` rozhodne, je-li `base`-šťastné.

```
def is_b_happy(number: int, base: int) -> bool:
    pass
```

6.3.4 [flood] Flood fill je algoritmus z oblasti rastrové grafiky, který vyplní souvislou jednobarevnou plochy nějakou jinou barvou. Postupuje tak, že nejdříve na novou barvu obarví pozici, na které začíná, dále se pokusí obarvit její sousedy (pozice jiné, než cílové barvy se neobarvují), a podobně pokračuje se sousedy těchto sousedů, atd. Zastaví se, dojde-li na okraj obrázku, nebo narazí na pixel, který nemá žádné nové stejnobarevné sousedy.
Napište proceduru, která na vstupu dostane plochu reprezentovanou obdélníkovým seznamem seznamů (délky všech vnitřních seznamů jsou stejné), počáteční pozici (je zaručeno, že se bude jednat o platné souřadnice), a cílovou barvu, na kterou mají být vybrané pozice přebarveny.

```
Position = Tuple[int, int]
Area = List[List[int]]

def flood_fill(area: Area, start: Position, colour: int) -> None:
    pass
```

6.3.5 [histogram] Napište (čistou) funkci, která na vstupu dostane signál `signal` reprezentovaný seznamem celočíselných amplitud (vzorků). Výsledkem bude statistika tohoto signálu, kterou vytvoří následujícím způsobem:

1. funkce signál nejdříve očistí od všech vzorků s amplitudou větší než `max_amplitude` a menších než `min_amplitude`,
2. následně jej převzorkuje tak, že sloučí každých `bucket` vzorek (poslední vzorek může být nekompletní) do jednoho vypočtením jejich průměru, a jeho následným zaokrouhlením (pomocí vestavěné funkce `round`),
3. nakonec spočítá, kolikrát se v upraveném signálu objevují jednotlivé amplitudy, a vrátí slovník, kde klíč bude amplituda a hodnota bude počet jejích výskytů.

```
def histogram(data: List[int], max_amplitude: int,
              min_amplitude: int, bucket: int) -> Dict[int, int]:
    pass
```

6.3.6 [alchemy] V této úloze budete zjišťovat, je-li možné pomocí alchymie vyrobit požadovanou substanci. Vstupem je:

- množina substancí, které již máte k dispozici (máte-li už nějakou substanci, máte ji k dispozici v neomezeném množství),
- slovník, který určuje, jak lze z existujících substancí vytvářet nové: klíčem je substance kterou můžeme vytvořit, a hodnotou je seznam „vstupních“ substancí, které k výrobě potřebujeme,
- cílová substance, kterou se pokoušíme vyrobit.

Napište predikát, kterého hodnota bude `True`, lze-li z daných substancí podle daných pravidel vytvořit substanci požadovanou, `False` jinak.

```
def is_creatable(owned_substances: Set[str],
                 rules: Dict[str, Set[str]], wanted: str) -> bool:
    pass
```

6.3.7 [shortest] Představte si bludiště, které se skládá z chodníků a rozcestí. Každý chodník začíná buď vstupem do bludiště nebo na nějakém rozcestí, a je buď slepý, končí východem z bludiště, nebo vede na nové rozcestí. Každé rozcestí má tedy jeden „příchozí“ chodník, který je ten jediný, po kterém se na něj lze od vchodu dostat, a několik chodníků „odchozích“ po kterých lze pokračovat.
Jakkoliv budete bludištěm chodit, nemůže se tedy stát, že se vrátíte na rozcestí, kde už jste byli, aniž byste se vraceli po vlastních stopách.
Aby se nám s bludištěm lépe pracovalo, každé rozcestí má jméno (řetězec). Chodník je pak zadaný jako dvojice (odkud, kam). Hodnota „odkud“ může být buď řetězec `entry`, nebo název rozcestí. Hodnota „kam“ může být buď řetězec `blind`, řetězec `exit`, nebo název rozcestí. Z každého rozcestí vede aspoň jeden chodník. Bludiště je zadané jako seznam chodníků.
Vaším úkolem je zjistit, jakým nejmenším počtem chodníků musíme projít, abychom se dostali k nějakému východu. Vstup do bludiště je pouze jeden, východ je aspoň jeden.

```
def shortest_path(labyrinth: List[Tuple[str, str]]) -> int:
    pass
```

6.3.8 [fixpoint] Mějme funkci `f`, která pro dané celé číslo `a` vrátí množinu obsahující `a`, `a // 2` a `a // 7`. Použitím této funkce na množině pak míníme její použití na každém prvku dané množiny a následné sjednocení všech obdržených výsledků.
Napište (čistou) funkci, která na množinu ze svého argumentu použije `f`, dále použije `f` na obdržený výsledek a takto bude pokračovat až dojde do bodu, kdy se dalším použitím `f` daná množina už nezmění. Výsledkem bude počet aplikací `f` na množinu, které bylo potřeba provést, než se proces zastavil.
Například z množiny `{1, 5, 6}` vznikne první aplikací popsané funkce množina `{0, 1, 2, 3, 5, 6}`:

- hodnota `1` se zobrazila na `{1, 1 // 2 = 0, 1 // 7 = 0}`,
- hodnota `5` na `{5, 5 // 2 = 2, 5 // 7 = 0}`, a konečně
- hodnota `6` na `{6, 6 // 2 = 3, 6 // 7 = 0}`.

Po další aplikaci se už množina nijak nezmění, proto je výsledkem číslo jedna.

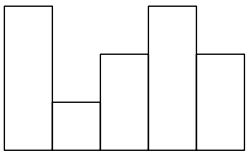
```
def fixpoint(starting_set: Set[int]) -> int:
    pass
```

Část 6.4: Bonusy

6.4.1 [connectivity] Uvažme městskou hromadnou dopravu, která má pojmenované zastávky, mezi kterými jezdí (pro nás anonymní) spoje. Spoje mají daný směr: není zaručeno, že jede-li spoj z *A* do *B*, jede i spoj z *B* do *A*. Dopravní síť budeme reprezentovat slovníkem, kde klíčem je nějaká zastávka *A*, a jemu příslušnou hodnotou je seznam zastávek, do kterých se lze z *A* dopravit bez dalšího zastavení.
Napište predikát, který rozhodne, je-li možné dostat se z libovolné zastávky na libovolnou jinou zastávku pouze použitím spojů ze zadaného slovníku.

```
def all_connected(stops: Dict[str, List[str]]) -> bool:
    pass
```

6.4.2 [lakes] Napište (čistou) funkci, která na vstupu dostane průřez krajiny a spočte, kolik vody se v dané krajině udrží, bude-li na ni neomezeně pršet. Krajina je reprezentována sekvencí celých nezáporných čísel, kde každé reprezentuje výšku jednoho úseku. Všechny úseky jsou stejně široké a mimo popsaný úsek krajiny je všude výška 0.
Například krajina `[3, 1, 2, 3, 2]` dokáže udržet 3 jednotky vody (mezi prvním a čtvrtým segmentem):




```
def lakes(land: List[int]) -> int:
    pass
```

Část 6.5: Řešení

6.5.1 [brackets.sol]

```
def correct_parentheses(text: str) -> bool:
    brackets = {"(": ")", "[": "]", "{": "}"}
    stack: List[str] = []
    for char in text:
        if char in brackets.keys():
            stack.append(char)  # push to the stack
        elif char in brackets.values():
            if len(stack) == 0:
                return False
            if brackets.get(stack.pop()) != char:
                return False
    return len(stack) == 0
```

6.5.2 [symmetric.sol]

```
def is_symmetric(relation: Set[Tuple[int, int]]) -> bool:
    for a, b in relation:
        if not (b, a) in relation:
            return False
    return True
```

6.5.3 [morse.sol]

```
def morse_encode(text: str) -> str:

    table = {"A": ".-", "B": "-...", "C": "-.-.", "D": "-..",
             "E": ".", "F": "..-.", "G": "--.", "H": "....",
             "I": "..", "J": ".---", "K": "-.-", "L": ".-..",
             "M": "--", "N": "-.", "O": "---", "P": ".--.",
             "Q": "--.-", "R": ".-.", "S": "...", "T": "-",
             "U": "..-", "V": "...-", "W": ".--", "X": "-..-",
             "Y": "-.--", "Z": "--.."}

    result = ""
    for char in text:
        result += table.get(char, " ")
        result += " "
    return result[:-1]

def morse_decode(code: str) -> str:

    table = {".-": "A", "-...": "B", "-.-.": "C", "-..": "D",
             ".": "E", "..-.": "F", "--.": "G", "....": "H",
             "..": "I", ".---": "J", "-.-": "K", ".-..": "L",
             "--": "M", "-.": "N", "---": "O", ".--.": "P",
             "--.-": "Q", ".-.": "R", "...": "S", "-": "T",
             "..-": "U", "...-": "V", ".--": "W", "-..-": "X",
             "-.--": "Y", "--..": "Z"}

    result = ""
    words = code.split("   ")
    for word in words:
        letters = word.split(" ")
        for letter in letters:
            result += table.get(letter, "")
        result += " "
    return result[:-1]
```