UNIVERZITA KONŠTANTÍNA FILOZOFA V NITRE
FAKULTA PRÍRODNÝCH VIED A INFORMATIKY

PROGRAMOVANIE V JAZYKU PYTHON

Vybrané kapitoly pre učiteľov informatiky

Gabriela Lovászová, Viera Michaličková, Nika Kvaššayová

NITRA 2022

PROGRAMOVANIE V JAZYKU PYTHON: VYBRANÉ KAPITOLY PRE UČITEĽOV INFORMATIKY

GABRIELA LOVÁSZOVÁ – VIERA MICHALIČKOVÁ – NIKA KVAŠŠAYOVÁ

2022

Programovanie v jazyku Python: Vybrané kapitoly pre učiteľov informatiky

Edícia Prírodovedec č. 797

Autori:

doc. RNDr. Gabriela Lovászová, PhD.

PaedDr. Viera Michaličková, PhD.

PaedDr. Nika Kvaššayová, PhD.

Recenzenti:

RNDr. Ján Skalka, PhD.

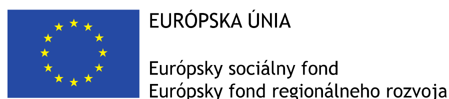
Mgr. Katarína Magdinová

(c) 2022 Univerzita Konštantína Filozofa v Nitre

Publikácia bola vytvorená v rámci projektu *„Skvalitňovanie praktickej prípravy budúcich pedagogických zamestnancov na UKF“* ITMS 312011Z815, ktorý sa realizuje vďaka podpore z Európskeho sociálneho fondu a Európskeho fondu regionálneho rozvoja v rámci Operačného programu Ľudské zdroje.

ISBN 978-80-558-1951-8

Partneri: http://www.minedu.gov.sk/, http://www.esf.gov.sk/



OPERAČNÝ PROGRAM
ĽUDSKÉ ZDROJE

OBSAH

Úvod 5
1 Údajové štruktúry v jazyku Python 6
1.1 Kedy je ten správny čas? 7
1.2 Vzťahy medzi osobami 13
1.3 Conwayova hra Život 19
1.4 Programujeme umenie 26
2 Funkcionálne techniky programovania v jazyku Python 32
2.1 Python ako multiparadigmový jazyk 33
2.2 Funkcionálny štýl 37
2.3 Funkcie vyššieho rádu 46
2.4 Zhrnutie 54
3 Programovanie hardvéru BBC micro:bit v jazyku MicroPython 55
3.1 Wearables – nositeľná elektronika 58
3.2 Nositeľné tričko 61
3.3 Odznak 64
3.4 Valentínsky odkaz 68
3.5 Nočná obloha 71
3.6 Čelenka 74
3.7 Disko neopixel LED topánky 76
3.8 Meranie ticha 77
3.9 Plyšová rozprávajúca hračka 78
Literatúra 79
Prílohy 80

ÚVOD

Programovací jazyk Python je v súčasnosti najpopulárnejším programovacím jazykom. Podľa indexu popularity programovacích jazykov, ktorý mesačne aktualizuje spoločnosť TIOBE ako tzv. TIOBE index, sa jazyk Python v rokoch 2021, 2022 udomácnil na prvej priečke indexu pred jazykmi C a Java (TIOBE, 2022). Okrem toho, že jazyk Python je populárny vo svete profesionálneho programovania, má aj vlastnosti, pre ktoré je dobrým kandidátom na jazyk vhodný na vyučovanie programovania:

- má voľne dostupný prekladač pre rôzne operačné systémy,
- je multiparadigmový, čo znamená, že umožňuje programovať viacerými štýlmi programovania,
- je interpretovaný, čo umožňuje interaktívne experimentovanie žiakov v príkazovom riadku,
- programy sú veľmi dobre čitateľné vďaka jednoduchej syntaxi a prehľadné vďaka odsadzovaniu vnorených častí kódu.

Z týchto dôvodov je jazyk Python odporúčaný ako programovací jazyk pre vyššie sekundárne vzdelávanie metodickým usmernením Štátneho pedagogického ústavu.

V tejto učebnici predstavujeme príklady problémov s komentovanými riešeniami z menej tradičných oblastí programovania v jazyku Python na strednej škole. V prvej kapitole sa zameriavame na prácu s údajovými štruktúrami. Ponúkame štyri príklady komplexných úloh so zaujímavým tematickým obsahom, v riešení ktorých sú použité dvojrozmerné údajové štruktúry. V druhej kapitole predstavujeme funkcionálny štýl programovania v jazyku Python. Všimli sme si pri práci s nadanými žiakmi, že vo svojich riešeniach často využívajú prvky funkcionálneho programovania, vďaka ktorým sú ich programy kratšie, hutnejšie. Preto sa tejto téme venujeme systematickejšie. V tretej kapitole sa zaoberáme témou programovania mikrokontrolérov BBC micro:bit, ktoré sa väčšinou používajú na základných školách a programujú pomocou blokového programovacieho jazyka. V tejto učebnici predstavujeme programovanie s využitím textového jazyka MicroPython na príkladoch projektov, ktorých výsledkom je nositeľná elektronika. Všetky prezentované námety na programovanie svojou obťažnosťou neprekračujú štandardy strednej školy, ale rozširujú tematické oblasti a techniky programovania.

Učebnica vznikla v rámci projektu *„Skvalitňovanie praktickej prípravy budúcich pedagogických zamestnancov na UKF*“. Cieľom je poskytnúť študijný materiál pre ďalšie sebavzdelávanie cvičným učiteľom, ktorí sú dôležitými partnermi univerzity v príprave budúcich učiteľov informatiky formou pedagogickej praxe. Učebnica je určená tiež študentom učiteľstva informatiky, má rozvíjať ich odborné vedomosti z programovania potrebné počas pedagogickej praxe na strednej škole. Publikáciu odporúčame tiež všetkým praktizujúcim učiteľom informatiky, ktorí cítia potrebu dopĺňať si svoje vedomosti v neustále a rýchlo sa rozvíjajúcom odbore informatika.

1 ÚDAJOVÉ ŠTRUKTÚRY V JAZYKU PYTHON

Programovací jazyk Python ponúka viacero užitočných údajových štruktúr (zoznamy, n-tice, množiny, slovníky). Stredoškoláci sa stretávajú predovšetkým s postupnosťami prvkov (čísel alebo reťazcov), ktoré ukladajú do jednorozmerných alebo dvojrozmerných zoznamov (matíc, tabuliek). V prvých úlohách obvykle trénujú základné operácie so zoznamami – vytvorenie prázdneho zoznamu, pridávanie prvkov do zoznamu a prístup k prvkom pomocou indexov. Úlohy sú typicky zamerané na prehľadávanie zoznamu s cieľom:

- zistiť, či alebo koľkokrát zoznam obsahuje prvok s danou vlastnosťou,
- vypočítať z hodnôt prvkov nejakú zaujímavú hodnotu (napr. najväčší prvok, aritmetický priemer),
- upraviť obsah pôvodného zoznamu podľa zadaných požiadaviek (napr. zmeniť hodnotu niektorých prvkov, usporiadať ich, vynechať niektoré prvky).

V nasledujúcich podkapitolách ponúkame čitateľovi námety na netradičné zadania, v ktorých sa zoznamy používajú pri riešení komplexnejších problémov. Zároveň poskytujú priestor na rozšírenie a prehĺbenie vedomostí z programovania v jazyku Python, ďalšie rozvíjanie programátorských zručností, analytického a kritického myslenia.

V podkapitole 1.1 je potrebné vstupné údaje reprezentovať zoznamom dvojíc (pristupovať k prvkom tejto dvojrozmernej štruktúry a usporiadať ich). Pri riešení problému tiež ukazujeme, ako môže vhodné predspracovanie vstupných údajov ovplyvniť efektívnosť riešenia.

V podkapitole 1.2 využívame zoznam zoznamov na uloženie informácie o neorientovanom grafe. V sérii úloh uvádzame žiakov do problematiky teórie grafov a grafových algoritmov. Pri porovnávaní rôznych implementácií grafu (ako matice susednosti a pomocou zoznamov susedov) majú žiaci príležitosť uvedomiť si, že záleží na tom, ako budeme abstraktnú štruktúru reprezentovať v programe.

V podkapitole 1.3 žiaci opäť využijú svoju skúsenosť s dvojrozmernou štruktúrou – zoznamom zoznamov a naprogramujú počítačovú simuláciu bunkového automatu. Zaujímavou súčasťou zadania je využitie zoznamu zoznamov na zapamätanie si siete štvorcov nakreslených na grafickom plátne. Vytvorená aplikácia má jednoduché grafické používateľské rozhranie, ktoré umožňuje krokovanie simulácie alebo jej sledovanie v prípade automatického riadenia pomocou časovača.

V podkapitole 1.4 využívame náhodu na generovanie umeleckého diela. Vytvorený obraz je nakreslený na grafickom plátne, v programe si ho opäť pamätáme v zozname zoznamov.

V závere každej podkapitoly uvádzame stručné zhrnutie a metodický komentár.

1.1 KEDY JE TEN SPRÁVNY ČAS?

Na festival sa majú dostaviť známe osobnosti (celebrity) z celého sveta. V zóne vyhradenej pre celebrity sa každý fanúšik môže z bezpečnostných dôvodov zdržať iba jednu hodinu. Organizátori vopred oznámili rozvrh príchodov a odchodov všetkých celebrít. Podľa neho sa môžeme rozhodnúť, kedy sa nám najviac oplatí prísť.

Tabuľka 1.1 obsahuje jeden konkrétny príklad vstupu. V predposlednom riadku vidíme, že cyklista Peter Sagan príde o šiestej a odíde o deviatej hodine. Časové intervaly, v ktorých sú celebrity prítomné, sú sprava otvorené. O deviatej by sme sa už s Petrom Saganom nestretli.

Tabuľka 1.1 Príklad vstupu

celebrita	príchod	odchod
Adele	6	8
Meryl Streep	7	8
Leonardo DiCaprio	5	9
Justin Timberlake	7	10
Peter Sagan	6	9
Ed Sheeran	9	10

ÚDAJOVÁ ŠTRUKTÚRA

Celebrít, ktoré prídu na festival, je veľa a o každej z nich si potrebujeme pamätať dvojicu čísel (čas príchodu a čas odchodu). K týmto údajom bude pri spracovaní praktické pristupovať pomocou indexov. Na uloženie vstupu je výhodné použiť zoznam dvojíc:

```
>>> intervaly = [(6, 8), (7, 8), (5, 9), (7, 10), (6, 9), (9, 10)]
```

Pripomeňme si najprv niektoré užitočné príkazy:

```
>>> len(intervaly)
6
>>> intervaly[0]
(6, 8)
>>> intervaly[2]
(5, 9)
>>> intervaly[-1]
(9, 10)
>>> intervaly[0][0]
6
>>> intervaly[0][1]
8
>>> intervaly[0][0] <= 5 < intervaly[0][1]
False
```

Zistili sme počet prvkov zoznamu (počet celebrít, ktoré plánujú prísť) a vypísali konkrétne prvky s indexami 0, 2 a -1 (t. j. posledný prvok). Keďže každý prvok zoznamu je dvojica, na prístup k jej prvkom je tiež potrebné použiť index. Vypísali sme začiatok a koniec intervalu s indexom 0. Overili sme si tiež, že o piatej hodine by sme túto celebritu nestretli.

V Pythone je možné ľahko overiť, s akým údajovým typom práve pracujeme. Vo výpise vidíme, že premenná *intervaly* je zoznam, jeho prvkami sú n-tice:

```
>>> type(intervaly)
<class 'list'>
>>> type(intervaly[0])
<class 'tuple'>
```

ÚLOHA 1

Predpokladajme, že v zozname *intervaly* máme uložené vstupné údaje. Aké informácie z nich vieme získať? Sformulujte niekoľko otázok, na ktoré by sa dalo odpovedať pomocou programu.

Riešenie

Napadnúť nám môžu rôzne otázky, napr. niektoré z uvedených:

a) Koľko celebrít príde?
b) Koľko celebrít bude prítomných o konkrétnej hodine?
c) Kedy príde/odíde prvá/posledná celebrita?
d) Koľko bude párty celebrít trvať?
e) Kedy mám prísť, aby som videl čo najviac celebrít?
f) Ktorá celebrita príde/odíde prvá/posledná?
g) Ktorá celebrita bude prítomná najkratšie/najdlhšie?
h) V akom poradí budú celebrity prichádzať?

Je zrejmé, že na posledné tri otázky (na rozdiel od ostatných) nám program nedokáže odpovedať. Mená celebrít sme si totiž neuložili. Ak to bude potrebné, môžeme namiesto zoznamu dvojíc použiť zoznam trojíc. Prvky v n-tici (ani v zozname) nemusia byť rovnakého typu. V našom prípade by trojicu tvoril jeden reťazec znakov a dve celé čísla:

```
intervaly = [('Adele', 6, 8), ('Meryl Streep', 7, 8),
             ('Leonardo DiCaprio', 5, 9), ('Justin Timberlake', 7, 10),
             ('Peter Sagan', 6, 9), ('Ed Sheeran', 9, 10)]
```

ÚLOHA 2

Napíšte program, v ktorom pre zadaný vstup zistíte odpovede na otázky a) a b) z Úlohy 1.

Riešenie

Vstupné údaje budeme pre jednoduchosť editovať priamo v zdrojovom kóde. Môžeme si vopred pripraviť viaceré zoznamy s rôznymi prvkami, ktoré budeme používať na testovacie účely:

```
# vstup
intervaly = [(6, 8), (7, 8), (5, 9), (7, 10), (6, 9), (9, 10)]

def pocet_pritomnych(intervaly, hodina):
    pocet = 0
    for x in intervaly:
        if x[0] <= hodina < x[1]:
            pocet += 1
    return pocet
```

```
# hlavný program
print(f'Spolu pride {len(intervaly)} celebrit.')
h = int(input('Hodina prichodu:'))
n = pocet_pritomnych(intervaly, h)
print(f'O {h}. hodine bude pritomnych {n} celebrit.')
```

Vo funkcii vidíme postupné spracovanie všetkých prvkov vstupného zoznamu. Pre každý interval overíme, či obsahuje hodnotu zadanú prostredníctvom druhého parametra.

ÚLOHA 3

O koľkej príde prvá celebrita a kedy odíde posledná z nich? Ako dlho bude celá akcia s celebritami trvať?

Riešenie

Odpovedať na prvú otázku znamená zistiť minimum zo všetkých príchodov a maximum zo všetkých odchodov uložených v prvkoch zoznamu *intervaly*:

```
z = intervaly[0][0]
k = intervaly[0][1]

for x in intervaly:
    z = min(z, x[0])
    k = max(k, x[1])

print(f'Prva celebrita pride o {z}. hodine.')
print(f'Posledna celebrita odide o {k}. hodine.')
print(f'Akcia potrva {k - z} hodin.')
```

Pomocné premenné *z* a *k* môžeme inicializovať na hodnoty príchodu a odchodu prečítané z nultého prvku vstupného zoznamu (príp. na iné rozumné hodnoty). Následne v cykle spracujeme všetky intervaly. Premenná *x* reprezentuje jeden prvok zoznamu, *x[0]* je prvý prvok tejto dvojice, *x[1]* je jej druhý prvok. Namiesto zápisov s príkazom vetvenia používame na priebežné aktualizovanie hodnôt premenných *z* a *k* funkcie *min()* a *max()*.

Uvedené riešenie nevyžaduje usporiadanie zoznamu. **A je možné vôbec takýto zoznam n-tíc usporiadať?**

USPORIADANIE ZOZNAMU N-TÍC

Pri práci so zoznamami máme k dispozícii metódu *sort()*. Ak ju zavoláme pre zoznamy čísel či reťazcov, prvky zoznamu sa usporiadajú podľa veľkosti, v prípade reťazcov lexikograficky (podľa abecedy). Nastavením voliteľného parametra môžeme upresniť smer usporiadania:

```
>>> a = [1, 5, 3, 2, 4]
>>> b = ['Zuzana', 'Maria', 'Andrej', 'Peter', 'Matus']
>>> a.sort()
>>> a
[1, 2, 3, 4, 5]
>>> a.sort(reverse=True)
>>> a
[5, 4, 3, 2, 1]
>>> b.sort()
```

```
>>> b
['Andrej', 'Maria', 'Matus', 'Peter', 'Zuzana']
```

Pre úplnosť pripomíname, že na usporiadanie prvkov zoznamu môžeme použiť aj funkciu *sorted()*. Zoznam v tomto prípade posielame do funkcie ako vstupný parameter. Funkcia vráti nový zoznam, v ktorom sú prvky usporiadané, pôvodný zoznam nijako nemení:

```
>>> a = [1, 5, 3, 2, 4]
>>> b = sorted(a)
>>> b
[1, 2, 3, 4, 5]
>>> a
[1, 5, 3, 2, 4]
```

Skúsme teraz usporiadať zoznam n-tíc:

```
>>> c = [(1, 5), (7, 8), (1, 4), (3, 5), (3, 1), (2, 10)]
>>> c.sort()
>>> c
[(1, 4), (1, 5), (2, 10), (3, 1), (3, 5), (7, 8)]
```

Z výpisu je zrejmé, že pri porovnávaní dvojíc prvkov sa rozhoduje najprv podľa prvého prvku n-tice. V prípade rovnosti sa pokračuje porovnávaním nasledujúcich prvkov n-tice.

Naša vlastná funkcia na usporiadanie prvkov zoznamu dvojíc by mohla vyzerať napr. takto (v ukážke sme naprogramovali algoritmus MinSort):

```
# vstup
intervaly = [(6, 8), (7, 8), (5, 9), (7, 10), (6, 9), (9, 10)]

def usporiadaj(c):
    for i in range(len(c) - 1):
        imin = i
        for j in range(i + 1, len(c)):
            if c[j][0] < c[imin][0]:
                imin = j
            elif c[j][0] == c[imin][0]:
                if c[j][1] < c[imin][1]:
                    imin = j
        c[imin], c[i] = c[i], c[imin]

# hlavný program
usporiadaj(intervaly) # alebo intervaly.sort()
print(intervaly)
print(f'prvy prichod: {intervaly[0][0]}']
print(f'posledny odchod: {intervaly[-1][1]'}
```

Vzorový program vypíše okrem usporiadaného zoznam aj hodinu príchodu prvej celebrity a čas odchodu poslednej celebrity. Tieto informácie sú po usporiadaní zoznamu k dispozícii v nultom a poslednom prvku.

ÚLOHA 4

Napíšte program, ktorý pre zadaný vstup zistí, **kedy bude prítomných najviac celebrít**. Keď chceme získať čo najviac podpisov alebo sa odfotiť s čo najväčším počtom celebrít, oplatí sa nám prísť práve v tomto okamihu.

Riešenie

K riešeniu tohto problému môžeme pristúpiť rôzne. Bez toho, aby sme intervaly vo vstupom zozname usporiadali, alebo s využitím tohto predspracovania.

V prvom, **pomalšom riešení**, pre každú celú hodinu z rozsahu trvania akcie (t. j. v časoch *z*, *z+1*, *z+2*, *z+3*, ..., *k-1*) overíme, koľko celebrít je aktuálne prítomných. Pri aktualizovaní hodnoty premennej *max* si zapamätáme aj súvisiaci časový údaj:

```
z, k = intervaly[0][0], intervaly[0][1]
for x in intervaly:
    z = min(z, x[0])
    k = max(k, x[1])

max = cas = -1
for i in range(z, k):
    n = 0
    for x in intervaly:
        if x[0] <= i < x[1]:
            n += 1
    if n > max:
        cas, max = i, n

print(f'V case {cas} bude pritomnych {max} celebrit.')
```

Aby sme sa vyhli opakovanému zisťovaniu, kto je práve prítomný, pozrime sa na časový úsek trvania akcie inak. V každom okamihu môže niekto prísť (vtedy počet prítomných rastie) alebo aj odísť (vtedy počet prítomných klesne). Údaje uložené v zozname *intervaly* **bude preto výhodné predspracovať** tak, aby sme mali k dispozícii informáciu o časoch a typoch udalostí, ktoré v jednotlivých časoch nastanú:

```
udalosti = []
for x in intervaly:
    udalosti.append((x[0], 'p'))
    udalosti.append((x[1], 'o'))
udalosti.sort()
```

Vytvorili sme nový zoznam *udalosti*. Z každého prvku pôvodného zoznamu vyberieme čas príchodu a čas odchodu. Ak ide o príchod, do zoznamu pridáme ako ďalšiu udalosť dvojicu *(x[0], 'p')*. Pre každý odchod zase vytvoríme dvojicu *(x[1], 'o')*. Vytvorený zoznam udalostí usporiadame. Porovnajme obsah pôvodného a nového zoznamu:

```
print(intervaly, udalosti)
```

Výstup na obrazovke:

```
[(6, 8), (7, 8), (5, 9), (7, 10), (6, 9), (9, 10)]
[(5, 'p'), (6, 'p'), (6, 'p'), (7, 'p'), (7, 'p'), (8, 'o'), (8, 'o'),
 (9, 'o'), (9, 'o'), (9, 'p'), (10, 'o'), (10, 'o')]
```

Vidíme, že napr. o deviatej dve celebrity odišli a jedna nová prišla. Vzhľadom na spôsob, akým sa zoznam n-tíc východiskovo usporadúva, odchody sú v poradí vždy pred príchodmi (pretože *'o'* < *'p'*), čo nám vyhovuje.

Usporiadaný zoznam udalosti už len spracujeme, napr. v samostatnej funkcii:

```
def max_pocet_celebrit(u):
    max, cas = -1, -1
    n = 0
    for x in u:
        if x[1] == 'o':                  # niekto práve odišiel
            n -= 1
        elif x[1] == 'p':                # niekto práve prišiel
            n += 1
        if n > max:
            max, cas = n, x[0]           # aktuálne maximum a príslušný čas
    return cas, max
```

Toto riešenie má aj inú výhodu: hranice intervalov na vstupe už nemusia byť celé čísla. Vo funkcii *max_pocet_celebrit()* totiž spracúvame zoznam po sebe idúcich udalostí a tie môžu nastať v ľubovoľnom okamihu, teda aj takom, ktorý je reprezentovaný desatinným číslom. Napr. pre vstupný zoznam:

```
intervaly=[(4.0,6.0),(4.5,7.0),(5.25,8.0),(5.0,6.0),(5.5, 8.0),(7.0,7.5)]
```

program oznámi:

```
V case 5.5 bude pritomnych 5 celebrit.
```

ZHRNUTIE

V tejto podkapitole sme vstupné údaje o celebritách uložili do zoznamu. Jeho prvky boli tiež štruktúrované, pracovali sme teda s dvojrozmernou údajovou štruktúrou (zoznamom n-tíc). Precvičili sme si prístup k údajom pomocou indexov. Naučili sme sa usporiadať zoznam s prvkami typu *tuple*. Videli sme tiež dva prístupy k riešeniu jedného problému. Predspracovanie vstupných údajov transformovaním pôvodného zoznamu na výhodnejší viedlo k rýchlejšiemu a univerzálnejšiemu riešeniu.

Vo vzorových programoch sme pracovali s jedným konkrétnym vstupným zoznamom. Pre žiakov bude zaujímavejšie, keď navrhnú vlastné zoznamy významných osobností z rôznych oblastí (hudba, film, veda, literatúra, história a pod.) a využijú ich následne na testovacie účely. V priebehu testovania programu by sme mali upriamiť pozornosť žiakov na rôzne extrémne prípady vstupov. Aký bude výstup programu v prípade, že je vstupný zoznam prázdny? Čo ak budú existovať viaceré časové okamihy s rovnakým počtom prítomných celebrít?

V riešení **Úlohy 2** sme napísali funkciu na zistenie počtu celebrít prítomných o konkrétnej hodine. Toto zadanie môžeme obmeniť novou požiadavkou - do funkcie budeme chcieť zadávať časový interval, v ktorom budeme prítomní na festivale. S koľkými celebritami sa stretneme?

V **Úlohe 4** sme vytvorili program, ktorý odporučí časový okamih s maximálnym počtom prítomných celebrít. Nie je to však jediné kritérium, podľa ktorého sa dá rozhodovať. Niektoré celebrity môžu byť pre nás zaujímavejšie ako iné. Túto informáciu je potrebné uviesť na vstupe. Nebudeme pracovať so zoznamom dvojíc ale trojíc. Okrem hraníc časového intervalu bude každá trojica obsahovať aj váhu prislúchajúcu príslušnej celebrite. Program bude hľadať časový okamih, v ktorom bude celková (príp. priemerná) váha prítomných celebrít najväčšia.

Vzorové riešenia všetkých úloh (vrátane obmien zadaní) uvádzame v prílohe.

1.2 VZŤAHY MEDZI OSOBAMI

Vstupný textový súbor obsahuje informáciu o vzájomných vzťahoch medzi *n* osobami očíslovanými číslami *0, 1, 2, ... , n-1*. V prvom riadku je uvedený počet osôb. Nasledujúce riadky obsahujú záznamy o tom, koho jednotlivé osoby poznajú. V príklade vstupného súboru nižšie vidíme, že osoba 0 pozná osobu 1 a osobu 2. Osoba 1 pozná osobu 0, 2 a 4. Osoba 2 pozná osoby 1 a 3. Osoba 3 nepozná nikoho. Osoba 4 pozná všetky ostatné osoby.

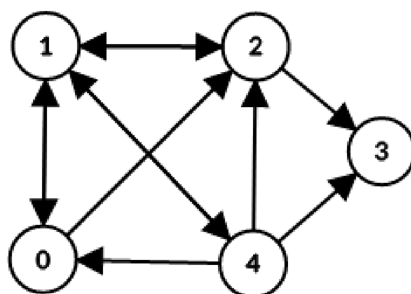
Príklad vstupného súboru *vstup.txt*:

```
5
0 1
0 2
1 0
1 2
1 4
2 1
2 3
4 0
4 1
4 2
4 3
```

Existuje taká osoba, ktorú poznajú všetci, ale ona nepozná nikoho z ostatných?

GRAF AKO ABSTRAKTNÝ MATEMATICKÝ MODEL

Čísla v textovom súbore sú opisom reálnej situácie, ktorej matematickým modelom je graf. Vrcholy grafu reprezentujú jednotlivé osoby. Hranami sú spojené tie vrcholy, medzi ktorými je príslušný vzťah. Vrcholy spojené hranou nazývame susedné. Ak osoba *u* pozná osobu *v*, hrana vychádza z vrcholu *u* a smeruje k vrcholu *v*. Takýto graf voláme orientovaný:



Obrázok 1.1 Orientovaný graf zo vzorového vstupného súboru vstup.txt

ÚLOHA 1

a) V zadaní hovoríme o vzťahoch medzi skutočnými osobami. Uveďte príklad sociálnej siete z online prostredia, v ktorej by ste vzťahy medzi používateľmi mohli znázorniť orientovaným grafom.

b) Aké iné situácie sa dajú znázorniť pomocou grafu?

<u>Riešenie</u>

a) Sociálnu sieť založená na tom, že používateľ sleduje príspevky niekoho iného (napr. Twitter, Instagram a pod.) modelujeme orientovaným grafom. Ak chceme pomocou grafu vizualizovať

sociálnu sieť založenú na priateľstve medzi používateľmi, čo je obojstranný vzťah (napr. v prípade Facebook-u), hrany nemajú orientáciu. Takýto graf voláme neorientovaný.

b) Graf je abstraktný matematický model a používa sa pri riešení praktických problémov z rôznych oblastí – v technických, prírodných aj spoločenských vedách (napr. medzi mestami vedú cesty, zariadenia v telekomunikačnej sieti sú prepojené káblami, tímy na športovom turnaji súperia vo vzájomných zápasoch, z jednej webovej stránky existuje odkaz na inú webovú stránku a pod.).

ÚLOHA 2

Grafické znázornenie informácie uloženej v súbore je názorné, pomáha nám porozumieť riešenému problému. V programe však musíme graf uložiť v pamäti počítača a to tak, aby sa sme ho mohli ľahko prehľadávať (efektívne niečo o grafe zistiť), príp. graf neskôr modifikovať (pridať/zrušiť vrchol či hranu). Ako by sme mohli graf implementovať v jazyku Python?

<u>Riešenie</u>

Potrebujeme si zapamätať, ktoré dvojice vrcholov sú spojené hranou. Túto informáciu môžeme uchovávať v tabuľke (v tzv. matici susednosti) s *n* riadkami a *n* stĺpcami. V jazyku Python maticu obvykle reprezentujeme zoznamom zoznamov:

	0	1	2	3	4
0	0	1	1	0	0
1	1	0	1	0	1
2	0	1	0	1	0
3	0	0	0	0	0
4	1	1	1	1	0

g: [[0, 1, 1, 0, 0], [1, 0, 1, 0, 1], [0, 1, 0, 1, 0], [0, 0, 0, 0, 0], [1, 1, 1, 1, 0]]

Obrázok 1.2 Matica susednosti ako zoznam zoznamov

Pracujeme teda s dvojrozmernou štruktúrou. Ak v grafe existuje hrana z vrcholu *u* do vrcholu *v*, v matici susednosti *g* bude mať prvok *g[u][v]* hodnotu *True*, inak *False*. Namiesto logických hodnôt sa často používajú aj celé čísla 1 a 0. V prípade neorientovaného grafu je zrejmé, že táto matica bude symetrická (prvky *g[u][v]* a *g[u][v]* budú mať rovnakú hodnotu pre všetky *u, v = 0, 1, ..., n-1*.)

ÚLOHA 2

Graf uložený vo vstupnom textovom súbore *vstup.txt* načítajte do programu. Použite maticu susednosti.

<u>Riešenie</u>

Namiesto logických hodnôt budeme do matice susednosti zapisovať celé čísla, výstup na obrazovku tak bude prehľadnejší. Súbor otvoríme na čítanie a spracujeme jednotlivé riadky.

Najskôr pripravíme prázdny zoznam, do ktorého postupne pridávame riadky matice susednosti. **Každý riadok je vytvorený osobitne** ako zoznam obsahujúci *n* prvkov s hodnotou *0*:

```
# spracovanie vstupného súboru
with open('vstup.txt') as f:
    # prečítame počet vrcholov
    n = int(f.readline())

    # pripravíme maticu typu n x n
    g = []
    for i in range(n):
        g.append([0] * n)

    # prečítame a uložíme informáciu o hranách
    for riadok in f:
        hrana = riadok.split()
        u = int(hrana[0])
        v = int(hrana[1])
        g[u][v] = 1
    # obsah matice vypíšeme na obrazovku
    print(g)
```

Maticu s *n* riadkami a *n* stĺpcami vieme vytvoriť aj kratším zápisom, **s využitím generátorovej notácie**:

```
g = [[0] * n for i in range(n)]
```

Každý riadok matice **musí** byť samostatný zoznam. Snaha o skrátenie zápisu na:

```
g = [[0] * n] * n
```

vedie k vytvoreniu takého **zoznamu, ktorého každý prvok je ten istý objekt** – jediný existujúci zoznam s *n* prvkami vytvorený pri vyhodnotení výrazu *[0] * n*. Ak by sme v takomto zozname zmenili hodnotu jedného z prvkov, napr. *g[0][1]*, vo výpise by sme zbadali, že rovnaká zmena nastala aj „vo všetkých ostatných riadkoch".

Výpis matice susednosti bude prehľadnejší s využitím pomocnej funkcie:

```
def vypis(g):
    for riadok in g:
        print(*riadok)
```

Operátor * zabezpečí „rozbalenie" hodnôt zoznamu riadok. Jednotlivé hodnoty budú vo výpise oddelené medzerou. Porovnajte:

```
print(g)
vypis(g)
```

Výstup:

```
[[0, 1, 1, 0, 0], [1, 0, 1, 0, 1], [0, 1, 0, 1, 0], [0, 0, 0, 0, 0], [1,
1, 1, 1, 0]]
0 1 1 0 0
1 0 1 0 1
0 1 0 1 0
0 0 0 0 0
1 1 1 1 0
```

V prvom prípade je vidno, že naozaj pracujeme so zoznamom zoznamov. Druhý výstup je prehľadnejší.

ÚLOHA 3

Doplňte program tak, aby vypísal odpoveď na otázku z úvodu podkapitoly. Budeme hľadať osobu, ktorú všetci poznajú, ale ona nepozná nikoho. Môže ísť opäť o celebritu, ktorá o existencii svojich fanúšikov ani netuší.

Riešenie

Napíšeme funkciu vracajúcu číslo osoby (index), ktorá spĺňa podmienku, alebo hodnotu -1, ak sa taká osoba v grafe nenachádza. Pre každý z vrcholov grafu najprv overíme, či sa v jeho riadku nachádza hodnota 1. Ak nie, znamená to, že táto osoba *u* nikoho nepozná. Zostáva skontrolovať, či osobu *u* poznajú všetci ostatní. Prezrieme preto celý stĺpec *u*. Teraz zužitkujeme rozhodnutie reprezentovať prítomnosť hrany hodnotou 1. Ak sčítame všetky prvky príslušného stĺpca a vyjde nám hodnota *n-1*, našli sme celebritu. Súčet je o jedna menší ako počet vrcholov, keďže fakt, že osoba pozná samu seba, neberieme do úvahy:

```
def celebrita(g):
    for u in range(len(g)):
        if 1 not in g[u]:
            s = 0
            for v in range(len(g)):
                s += g[v][u]
            if s == n-1:
                return u
    return -1
```

ÚLOHA 4

Sformujte iné otázky o načítanom grafe a vytvorte program, ktorý na ne odpovie.

Riešenie

Graf môžeme prehľadávať z rôznych dôvodov. Uvedieme niekoľko príkladov otázok (námetov na doplňujúce zadania):

a) Pozná sa každý s každým? (napr. žiaci jednej triedy). V matici sú v tomto prípade samé jednotky, s výnimkou prvkov na hlavnej diagonále.
b) Existuje osoba - cudzinec, ktorú nikto nepozná a ani ona nikoho nepozná? Ak existuje, tak v jej riadku a stĺpci budú len prvky s hodnotou 0.
c) Koľko párov osôb sa pozná navzájom? Pre takéto páry osôb platí, že *g[u][v] = g[v][u] == 1*. Dajte pozor, aby ste ten istý pár poznajúcich sa osôb nezapočítali dvakrát!
d) Ktorá z osôb má najviac priateľov (za priateľov považujeme takú dvojicu osôb, ktoré sa poznajú navzájom)?

ÚLOHA 5

V predchádzajúcich programoch sme graf implementovali ako **maticu susednosti** pomocou zoznamu zoznamov. K riadkom a stĺpcom takejto matice máme prístup pomocou indexov, overovanie existencie konkrétnej hrany, jej pridávanie či odstraňovanie, je preto veľmi efektívne. Ak je však hrán v grafe málo (graf je riedky), matica susednosti je zbytočne veľká (takmer všetky jej prvky majú hodnotu *False*, resp. *0*). Nevýhodou je tiež pomalé získavanie zoznamov susedných

vrcholov. Pre každý vrchol je potrebné kontrolovať, ktoré z prvkov príslušného riadku matice susednosti majú hodnotu *True*, resp. *1*.

Navrhnite inú, úspornejšiu implementáciu grafu. Využite ju v programe, v ktorom zistíte, ktorá z osôb má najviac priateľov (za priateľov považujeme takú dvojicu osôb, ktoré sa poznajú navzájom).

Riešenie

Pre každý vrchol si zapamätáme iba zoznam jeho susedov. V tomto **zozname zoznamov** tak nebudú mať všetky prvky nutne rovnakú dĺžku (predtým išlo o rovnako dlhé riadky matice). Ak niektorý z vrcholov s indexom *u* nemá žiadne susedné vrcholy, zoznam *g[u]* bude prázdny:

```
# spracovanie vstupného súboru
with open('vstup.txt') as f:
    # prečítame počet vrcholov
    n = int(f.readline())
    # pripravíme zoznam so zoznamami susedov jednotlivých vrcholov
    g = [[] for i in range(n)]
    for riadok in f:
        hrana = riadok.split()
        u = int(hrana[0])
        v = int(hrana[1])
        # ak uv je hrana, do zoznamu susedov vrcholu u pridáme vrchol v
        g[u].append(v)
```

Funkciu vypisujúcu graf na obrazovku môžeme upraviť takto:

```
def vypis(g):
    for u in range(len(g)):
        print(u,':', g[u])
```

Pre vzorový vstupný súbor získame výstup v prehľadnej forme (vidíme priamo zoznamy susedov jednotlivých vrcholov):

```
0 : [1, 2]
1 : [0, 2, 4]
2 : [1, 3]
3 : []
4 : [0, 1, 2, 3]
```

Napíšeme ešte funkciu, ktorá vráti číslo osoby s najväčším počtom priateľov. V prípade, že žiadny priatelia neexistujú, vráti hodnotu -1. Spolu s číslom osoby vrátime aj hodnotu nájdeného maxima. Pre každú osobu spracujeme jej zoznam susedných vrcholov a zistíme, či sa jedná o priateľa. Priebežne aktualizujeme hodnotu doteraz objaveného maxima:

```
def najviac_priatelov(g):
    osoba = max = -1
    for u in range(len(g)):
        pocet = 0
        for v in g[u]:
            if u in g[v]:   # overenie existencie hrany
                pocet += 1
        if pocet > max:
            max = pocet
            osoba = u
    return osoba, max
```

Pre vzorový vstup získame výsledok:

```
Osoba 1 ma 3 priatelov.
```

V zdrojovom kóde vidíme, že zápis overenia existencie hrany *uv* je vďaka operátoru *in* veľmi jednoduchý. V skutočnosti sa zoznam susedov *g[v]* prehľadáva, čo je zdĺhavejšie ako priamy prístup k prvkom pomocou indexov, ktorý sa používa v matici susednosti.

ZHRNUTIE

V tejto podkapitole predpokladáme, že žiaci už majú predchádzajúcu skúsenosť s údajovou štruktúrou zoznam a vedia spracúvať riadky textového súboru.

V **Úlohe 1** sa žiaci oboznamujú so základnou terminológiou z teórie grafov. Spoznávajú graf ako univerzálny matematický model, ktorý je užitočný v rôznych problémových situáciách. Rozdiel medzi orientovaným a neorientovaným grafom ilustrujeme na príklade sociálnych sietí, ktoré sú žiakom veľmi blízke. Tieto grafy sú neohodnotené. Pri uvádzaní ďalších príkladov z praxe môžeme spomenúť aj také grafy, ktorých hrany alebo vrcholy majú ohodnotenia (napr. dĺžky cestných úsekov, názvy miest a pod.).

V nasledujúcej úlohe sa zamýšľame nad tým, ako si informáciu o vrcholoch a hranách grafu, ktorá je zapísaná v textovom súbore, zapamätáme v programe. Žiaci by si mali najprv nakresliť viacero rôznych grafov a pripraviť k nim zodpovedajúce vstupné súbory. Pri implementácii matice susednosti si žiaci majú uvedomiť, že každý riadok matice musí byť samostatne vytvorený zoznam. Ak zapíšu vytvorenie zoznamu zoznamov nesprávne, prídu nato pri výpise načítaného grafu na obrazovku sami. Graf v tejto úlohe vypisujeme na obrazovku dvomi spôsobmi, aby sme takto upriamili pozornosť na rozdiel medzi abstraktnou štruktúrou (tou je aj matica susednosti) a jej konkrétnou implementáciou v jazyku Python.

Pri riešení **Úlohy 3** sa dá so žiakmi diskutovať o tom, ako by riešenie vyzeralo v prípade, ak by sme namiesto celočíselných hodnôt do matice susednosti ukladali logické hodnoty.

Úloha 4 poskytuje priestor na formulovanie ďalších otázok, na ktoré je možné odpovedať prehľadávaním načítaného grafu.

V poslednej úlohe predstavujeme inú implementáciu tej istej abstraktnej údajovej štruktúry graf. Žiaci môžu niektoré zo zadaní v **Úlohe 4** prepracovať s použitím zoznamu zoznamov susedov. Veľmi užitočné je aj kritické porovnanie oboch prístupov k implementácii grafu. Žiaci si tak lepšie uvedomia, že pri výbere vhodnej údajovej štruktúry a jej implementácii je potrebné zvážiť aj to, aký vplyv bude mať naše rozhodnutie na efektívnosť riešenia.

Vzorové riešenia všetkých úloh uvádzame v prílohe.

1.3 CONWAYOVA HRA ŽIVOT

Bunkové automaty (celulárne automaty) sa používajú na modelovanie procesov v prírodných, technických aj spoločenských vedách. Realizovaním počítačových simulačných experimentov môžeme objaviť zaujímavé zákonitosti správania sa modelu.

Conwayova hra *Život* je príkladom jednoduchého dvojrozmerného bunkového automatu **o vývoji spoločenstva organizmov** žijúcich v bunkách štvorcovej siete. Každá bunka štvorcovej siete môže byť v jednom z 2 stavov:

- **živá** (je tam organizmus) alebo
- **mŕtva** (nie je tam organizmus).

Hra začína 0. generáciou (vygenerovanou náhodne alebo zadanou používateľom). V každej ďalšej generácii sa **stav všetkých buniek zmení naraz** v závislosti od stavov susedných buniek a pravidiel pre zmenu stavu (tzv. prechodovej funkcie automatu).

Každý organizmus má v štvorcovej sieti **najviac 8 susedov. Pri výpočte nového stavu** bunkového automatu (novej generácie) sa riadime týmito pravidlami:

- organizmus prežije, ak má 2 alebo 3 susedov,
- organizmus zahynie na samotu, ak má 0 susedov alebo 1 suseda,
- organizmus zahynie od hladu, ak má viac ako 3 susedov,
- v prázdnej bunke sa narodí nový organizmus vtedy, ak má bunka práve 3 susedov.

Kedy všetky organizmy vyhynú? Vytvárajú sa v priebehu simulácie pre nejaký vstup opakujúce sa, či iné špecifické vzory? Ak áno, aké?

IMPLEMENTÁCIA DVOJROZMERNÉHO BUNKOVÉHO AUTOMATU V PROGRAME

Dvojrozmerný bunkový automat je sieť tvorená riadkami a stĺpcami buniek, ktorých stav sa mení. V programe ho preto implementujeme pomocou zoznamu zoznamov. Stav bunky môžeme kódovať logickými hodnotami (*False*/*True*) alebo celočíselnými hodnotami (0/1).

V pôvodnej Conwayovej hre sa uvažuje nekončená sieť buniek, v programe budeme pracovať s konečnou sieťou s konkrétnymi rozmermi. Na začiatku vytvoríme sieť bez organizmov (prvky zoznamu zoznamov inicializujeme na hodnotu *False*, resp. 0). Následne zabezpečíme načítanie vstupu od používateľa (hodnoty buniek na zvolených pozíciách nastavíme na hodnotu *True*, resp. 1).

ÚLOHA 1

Vytvorte program, v ktorom sa východisková generácia organizmov vygeneruje náhodne. Zadajte počet generácií a zrealizujte simuláciu. Experimentujte so štvorcovou sieťou menšieho rozmeru a malým počtom generácií.

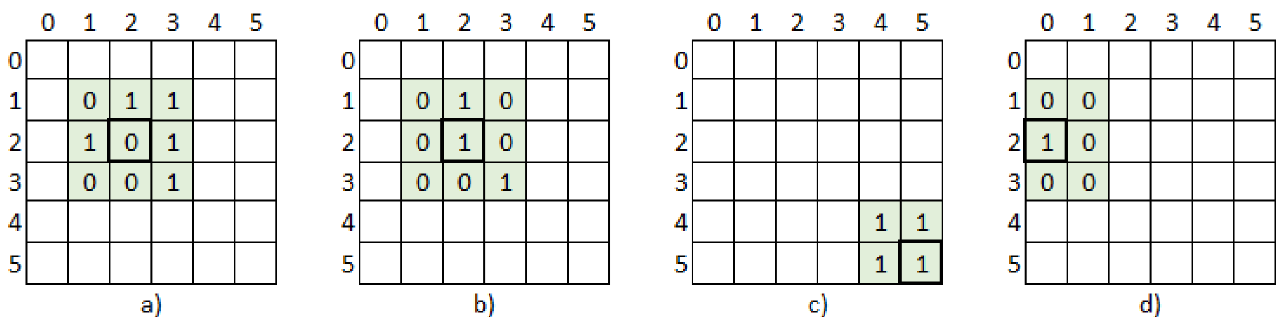
Riešenie

Rozmer štvorcovej siete môžeme v programe definovať ako konštantu.

Najprv naprogramujeme a otestujeme funkcie vytvárajúce prázdnu, resp. náhodne inicializovanú štvorcovú sieť:

```
import random

N = 5        # rozmer siete
ktora = 0    # počitadlo generácií

# vytvoríme novú prázdnu sieť ako zoznam zoznamov
# 0 = mŕtva bunka
def prazdna():
    siet = []
    for i in range(N):
        siet.append([0] * N)
    return siet

# vytvoríme novú sieť inicializovanú náhodne
# 0 = mŕtva bunka, 1 = živá bunka
def nahodna():
    siet = []
    for i in range(N):
        siet.append([random.randrange(2) for i in range(N)])
    return siet
```

Pre výpočet novej generácie je kľúčová pomocná funkcia, ktorá pre konkrétnu bunku v riadku *r* a stĺpci *s* zistí, koľko buniek s ňou susediacich je živých.



Obrázok 1.3 Bunky s rôznym počtom susedných buniek

Každá bunka má **najviac 8 susedných buniek**, na okrajoch siete menej. Na obrázku vidíme zelenou farbou podfarbené okolie zvolenej bunky (vrátane nej samej). V príklade a) má mŕtva bunka v riadku 2 a stĺpci 2 spolu 5 susedov. V príklade b) je bunka na rovnakej pozícii živá a má 2 susedov. V príklade c) je bunka v pravom dolnom rohu živá a má 3 susedov. V príklade d) je bunka v riadku 2 a stĺpci 0 živá a nemá žiadnych susedov.

Vo funkcii *pocet_susedov()* preskúmame celé okolie bunky v zadanom riadku a stĺpci. Ak je táto bunka živá, jej hodnotu je potrebné od zisteného počtu susedov odpočítať:

```
# pre bunku v riadku r a stĺpci s spočítame počet jej susedov
def pocet_susedov(r, s):
    pocet = 0
    for i in r - 1, r, r + 1:
        for j in s - 1, s, s + 1:
            if 0 <= i < N and 0 <= j < N:
                pocet += generacia[i][j]
    return pocet - generacia[r][s]
```

V priebehu simulácie budeme opakovane počítať nasledujúcu generáciu:

```
# výpočet ďalšej generácie
def dalsia_generacia():
    # globálne počítadlo
    global generacia
    global ktora
    # nový stav vypočítame do novej siete
    nova = prazdna()
    # hodnoty buniek vypočítame na základe predchádzajúceho stavu
    for i in range(N):
        for j in range(N):
            # zistíme počet susedov
            p = pocet_susedov(i, j)
            # aplikujeme pravidlá
            if generacia[i][j] == 0:
                if p == 3:
                    nova[i][j] = 1
            else:
                if p == 2 or p == 3:
                    nova[i][j] = 1
    # zapamätáme si novú generáciu ako aktuálnu
    generacia = nova
    # zvýšime počítadlo generácií
    ktora += 1
```

V bunke, ktorá bola predtým prázdna, sa narodí nová, len ak je počet susedov práve 3. bunka, ktorá bola predtým živá (mala hodnotu 1), bude živá v novej generácii iba v prípade, že má dvoch alebo 3 susedov (inak zomiera na samotu alebo od hladu).

Na záver vzorového riešenia uvedieme funkciu pre výpis generácií na obrazovku a hlavný program:

```
# zobrazenie aktuálneho stavu na obrazovke
def vypis():
    print(f'Generacia #{ktora}: ')
    for i in range(N):
        for j in range(N):
            print(generacia[i][j], end='')
        print()


# hlavný program
generacia = nahodna()
vypis()

pocet = int(input('zadaj pocet generacii: '))
for i in range(pocet):
    dalsia_generacia()
    vypis()
```

PRÁCA S GRAFICKÝMI OBJEKTAMI PRI KRESLENÍ POMOCOU MODULU TKINTER

Aby sme predchádzajúci program mohli prerobiť na aplikáciu s pekným grafickým používateľským rozhraním, musíme vedieť vykresliť sieť štvorčekov a v priebehu simulácie meniť ich stav (napr. farbu).

Nakreslime najprv na plátno jeden štvorec. Funkcia *create_rectangle()* vracia ako výsledok číselný identifikátor nakresleného grafického objektu. Pomocou tohto čísla (môžeme si ho vypísať na obrazovku) vieme neskôr zmeniť jeho vlastnosti (farbu výplne, farbu obrysu, pozíciu a pod.).

```
import tkinter

platno = tkinter.Canvas(width=600, height=400)
platno.pack()

id = platno.create_rectangle(10, 10, 110, 110, fill='green')
print(id)

platno.mainloop()
```

Ako zabezpečíme, aby sa pri kliknutí na plátno zmenila štvorcu farba výplne (napr. zo zelenej ma bielu)? Vo funkcii obsluhujúcej udalosť „pri kliknutí“ tlačidlom myši zavoláme funkciu *itemconfig()*. V prvom parametri uvedieme číselný identifikátor grafického objektu, následne hodnoty tých parametrov, ktoré mu chceme nastaviť:

```
import tkinter

def klik(event):
    platno.itemconfig(id, fill='white')

platno = tkinter.Canvas(width=600, height=400)
platno.bind('<Button-1>', klik)
platno.pack()

id = platno.create_rectangle(10 ,10, 110, 110, fill='green')
print(id)

platno.mainloop()
```

ÚLOHA 2

Upravte program z textovým výstupom z **Úlohy 1** tak, aby mal grafické používateľské rozhranie. Umožnite editovať vstupnú generáciu interaktívne pomocou myši. Simuláciu krokujte klikaním na tlačidlo.

Riešenie

Okrem zoznamu zoznamov, v ktorom si pamätáme stavy jednotlivých buniek aktuálnej generácie, si budeme potrebovať pamätať aj všetky štvorčeky siete v grafickom výstupe na plátne:

```
def vytvor_siet_stvorcekov():
    s = []
    for i in range(n):
        s.append([0] * n)
        for j in range(n):
            # výpočet súradníc aktuálneho štvorca siete
            x, y = d * j + x0, d * i + y0
            s[i][j] = platno.create_rectangle(x, y, x + d, y + d,
                                              fill='white',
                                              outline='gray')
    return s
```

Namiesto vypisovania obsahu zoznamu *generacia* na obrazovku v textovej podobe budeme teraz aktualizovať grafický výstup:

```
def vykresli_aktualny_stav():
    for i in range(n):
        for j in range(n):
            x, y = d * j + x0, d * i + y0
            # pre bunku s hodnotou 0 bude výplň biela, pre hodnotu 1
zelená
            farba = ['white', 'green'][generacia[i][j]]
            # zmena farby výplne štvorčeka vzhľadom na stav v generácii
            platno.itemconfig(stvorceky[i][j], fill=farba)
```

Interaktívne editovanie generácie zabezpečíme v osobitnej funkcii, ktorá sa vykonáva ako reakcia na udalosť „pri kliknutí“ na plátno:

```
def klik(event):
    s, r = (event.x - x0) // d, (event.y - y0) // d
    if 0 <= r < len(generacia) and 0 <= s < len(generacia[r]):
        if generacia[r][s] == 0:
            generacia[r][s] = 1
            platno.itemconfig(stvorceky[r][s], fill='green')
        else:
            generacia[r][s] = 0
            platno.itemconfig(stvorceky[r][s], fill='white')
```

Hodnoty *event.x* a *event.y* sú súradnice miesta, na ktoré sme na plátne klikli. Ak *d* je šírka štvorčeka siete, ktorej ľavý horný roh je na pozícii $[x_0, y_0]$, potom hodnoty *r* a *s* predstavujú riadkový a stĺpcový index zasiahnutého štvorčeka. Ak ide o štvorec siete, zmeníme jeho farbu výplne z aktuálnej na opačnú.

ÚLOHA 3

Experimentujte s vytvorenou aplikáciou. Pre rôzne počiatočné generácie organizmov sledujte, ako sa im darí. Prežijú, vyhynú, robia niečo zaujímavé?

Riešenie

Ako počiatočnú generáciu môžeme použiť svoje meno, rôzne pravidelné či nepravidelné obrázky. Príklady zaujímavých vzorov vznikajúcich v priebehu simulácie nájdeme napr. vo Wikipédii: https://cs.wikipedia.org/wiki/Hra_%C5%BEivota.

ÚLOHA 4

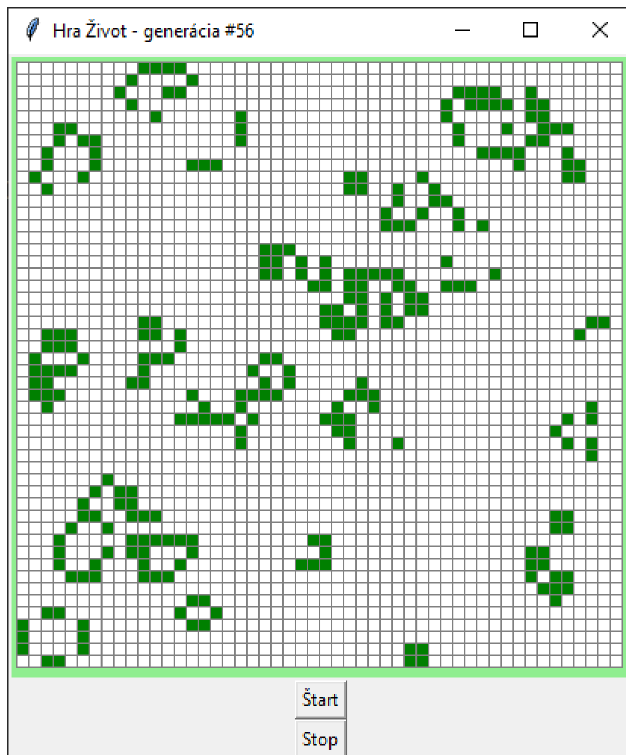
Upravte simuláciu tak, aby bežala automaticky a nemuseli ste ju krokovať pomocou tlačidla.

Riešenie

Namiesto tlačidla „Ďalšia generácia“ pridáme do okna aplikácie tlačidlá „Štart“ a „Stop“. Stláčaním týchto tlačidiel budeme meniť stav globálnej premennej *bezi* z *True* na *False*, resp. naopak. Hodnota tejto premennej bude indikovať, či simulácia ešte beží alebo bola zastavená.

Tlačidlám tiež nastavíme vzhľad (vlastnosť *state*) podľa toho, či majú alebo nemajú byť z pohľadu používateľa aktívne. Zabránime tým nechcenému viacnásobného klikaniu na tlačidlo „Štart“.

Aby sa stav v aplikácii menil automaticky, po výpočte každej ďalšej generácie naplánujeme opätovné zavolanie funkcie *dalsia_generacia()*. Plátno má funkciu *after()*, v ktorej parametroch definujeme časový interval v milisekundách a funkciu, ktorá sa má po uplynutí tohto času zavolať:



Obrázok 1.4 Simulácia Hry Život s grafickým používateľským rozhraním

```
# …
def stop():
    tlacidlo_start.configure(state=tkinter.NORMAL)
    global bezi
    bezi = False

def start():
    tlacidlo_start.configure(state=tkinter.DISABLED)
    global bezi
    bezi = True
    dalsia_generacia()

def dalsia_generacia():
    # ...
    if bezi:
        platno.after(100, dalsia_generacia)

# globálna premenná
bezi = True

# …
tlacidlo_start = tkinter.Button(okno, text='Štart', command=start)
tlacidlo_start.pack()

tlacidlo_stop = tkinter.Button(okno, text='Stop', command=stop)
tlacidlo_stop.pack()
```

ZHRNUTIE

V tejto podkapitole predpokladáme, že žiaci už majú predchádzajúcu skúsenosť s údajovou štruktúrou zoznam zoznamov a dokážu vytvoriť jednoduchú aplikáciu s grafickým používateľským rozhraním. Vyžadujú sa len základy práce s modulom *tkinter*: kreslenie na grafické plátno a programovanie reakcií na udalosti súvisiace s klikaním myšou na plátno a tlačidlami.

Žiaci využijú údajovú štruktúru zoznam zoznamov pri riešení skutočného, zmysluplného problému – tvorbe počítačovej simulácie známeho bunkového automatu. V zozname zoznamov si budú pamätať stav aktuálnej generácie, ale aj grafické objekty, pomocou ktorých simuláciu vizualizujú. Užitočná je tiež skúsenosť s realizovaním simulačných experimentov s cieľom objaviť niečo zaujímavé o správaní modelovaného systému (v našom prípade o spoločenstve organizmov).

Pravidlá pre *Hru Život* sformulované v úvode je vhodné so žiakmi najprv aplikovať na viacerých konkrétnych príkladoch výpočtu novej generácie zo starej pre sieť menšieho rozmeru. Žiakom pripravíme pracovné listy, v papierovej alebo elektronickej podobe. Praktickým pomocníkom môže byť tabuľkový kalkulátor a jeho sieť buniek.

Pred tvorbou aplikácie s grafickým používateľským rozhraním (**Úloha 2**) je vhodné žiakom umožniť, aby získali skúsenosti so zapamätaním si grafického objektu a následným používaním príkazu *itemconfig()*.

V poslednej úlohe sme vysvetlili, ako sa dá v jazyku Python implementovať jednoduchý časovač. Manuálne krokovanie simulácie sme tak nahradili automatickým riadením priebehu s možnosťou jeho pozastavenia. Takáto verzia riešenia bude pre žiakov pravdepodobne najatraktívnejšia.

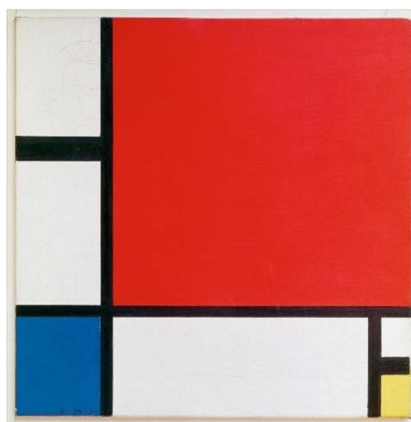
Vytvorená aplikácia má minimalistické rozhranie. Žiakom môžu napadnúť aj ďalšie vylepšenia zjednodušujúce realizovanie experimentov (napr. obnovenie stavu, uloženie priebehu simulácie do súboru, zastavenie simulácie po vymretí populácie a pod.).

Ak nie je k dispozícii dostatok času na naprogramovanie celej aplikácie, môžeme žiakom pripraviť súbory s čiastkovým riešením, v ktorom doplnia niektoré chýbajúce časti (napr. výpočet počtu susedov, vygenerovanie náhodnej počiatočnej generácie, dokončenie alebo prispôsobenie grafického používateľského rozhrania).

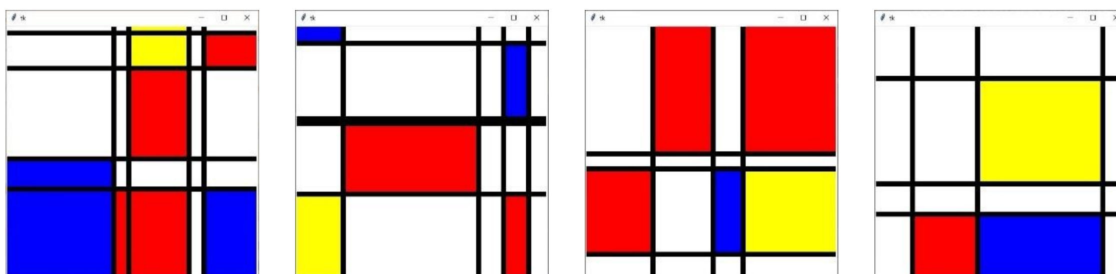
Vzorové riešenia všetkých úloh uvádzame v prílohe.

1.4 PROGRAMUJEME UMENIE

Piet Modrian (1872 - 1944) bol holandský maliar a teoretik umenia. Je považovaný za jedného z najväčších umelcov 20. storočia – priekopníka abstraktného umenia. Vyvinul nereprezentatívnu formu, ktorú nazval neoplasticizmus. Pracoval na vytvorení „univerzálnej krásy" prostredníctvom poriadku a jednoduchosti. Pre jeho obrazy je typické najmä použitie horizontálnych a vertikálnych čiernych čiar rôznych hrúbok a niekoľkých základných farieb (modrá, červená, žltá). Abstraktné obrazy založené na pravidelnosti a geometrických tvaroch sa dajú pomerne ľahko generovať aj počítačovým programom.



Obrázok 1.5 Kompozícia s červenou, modrou a žltou (1930) od Pieta Mondriana (Zdroj: Wikimedia Commons)



Obrázok 1.6 Obrazy v Modrianovom štýle vytvorené programom

GENEROVANIE OBRAZU PODĽA ZADANÝCH KRITÉRIÍ

Generovanie obrazu môže byť založené na náhode.

V jazyku Python sa dajú náhodné čísla získavať s využitím funkcií z modulu *random:*

randint(start, stop) — vracia celé číslo z intervalu <*start, stop*>, teda vrátane dolnej a hornej hranice zadanej v parametroch.

randrange(n) — vracia náhodné celé číslo z hodnôt 0, 1, 2, ..., n-1.

choice(zoznam) — vracia náhodný prvok zo zadaného zoznamu.

Ak chceme vytvoriť grafický výstup v štýle Mondrianovych obrazov, potrebujeme, aby program rozdelil plochu plátna náhodne niekoľkými zvislými a niekoľkými vodorovnými čiernymi čiarami. Spomedzi vzniknutých obdĺžnikov má následne niektoré vyplniť niektorou zo základných farieb.

Práca s rôzne hrubými čiarami a nimi ohraničenými obdĺžnikmi bude jednoduchšia, ak budeme celý obraz reprezentovať mriežkou primerane malých štvorčekov.

ÚLOHA 1

Ktoré parametre obrazu sú meniteľné a budú mať vplyv na výsledný grafický výstup?

Riešenie

Parametre obrazu môžeme definovať ako konštanty na začiatku programu. Zmenou ich hodnôt zabezpečíme generovanie iného výstupu. Určíme minimálny a maximálny počet čiar na obraze a minimálny a maximálny počet pokusov o vyplnenie nejakého náhodne zvoleného obdĺžnika:

```
import tkinter
import random

SIRKA_PLATNA = 500
VYSKA_PLATNA = 500

VELKOST_STVORCA = 10

MAX_CIAR = 10
MIN_CIAR = 6
MAX_OBDLZNIKOV = 10
MIN_OBDLZNIKOV = 3

POCET_RIADKOV = VYSKA_PLATNA // VELKOST_STVORCA
POCET_STLPCOV = SIRKA_PLATNA // VELKOST_STVORCA

FARBY = ['red', 'yellow', 'blue']
```

ÚLOHA 2

Napíšte funkciu, ktorá na plátne zvolených rozmerov vytvorí mriežku malých bielych štvorcov bez obrysu, ktoré budeme neskôr vypĺňať farbami.

Riešenie

Rovnako ako v predošlej podkapitole, využijeme fakt, že funkcia *create_rectangle()* vracia číselný identifikátor nakresleného útvaru. Pre každý štvorček mriežky si toto číslo zapamätáme, aby sme mohli neskôr k nemu pristúpiť a zmeniť jeho vlastnosti (farbu výplne):

```
def mriezka():
    for r in range(POCET_RIADKOV):
        for s in range(POCET_STLPCOV):
            obraz[r][s] = platno.create_rectangle(s * VELKOST_STVORCA,
                                                  r * VELKOST_STVORCA,
                                                 (s+1)*VELKOST_STVORCA,
                                                 (r+1)*(VELKOST_STVORCA),
                                                 fill='white',outline='')

# hlavný program
# príprava grafického plátna
```

```
platno = tkinter.Canvas(width= SIRKA_PLATNA, height= VYSKA_PLATNA,
                        bg='white')

# vytvorenie zoznamu zoznamov pre pamätanie si štvorčekov mriežky
obraz = [[0] * POCET_STLPCOV for i in range(POCET_RIADKOV)]

# vykreslenie štvorčekov na plátne
mriezka()

# zobrazenie aplikácie s plátnom
platno.pack()
platno.mainloop()
```

ÚLOHA 3

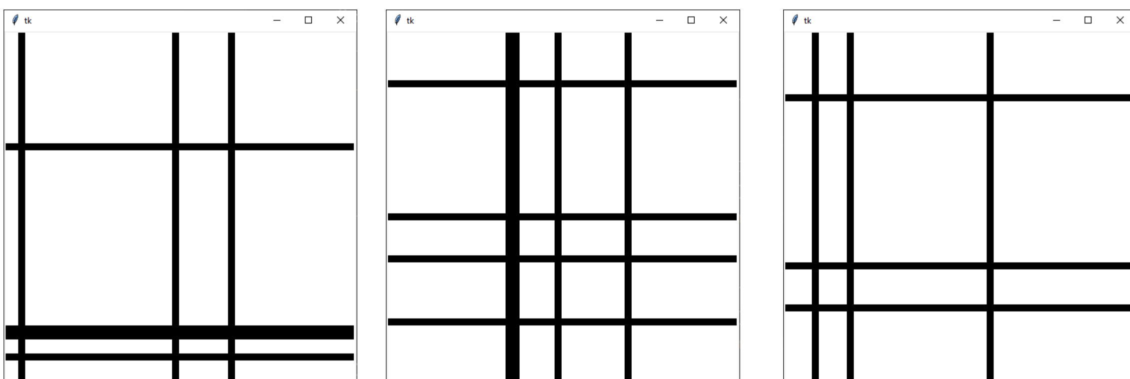
Doplňte do programu vykreslenie náhodného počtu zvislých a vodorovných čiar obrazu.

Riešenie

Túto požiadavku môžeme zrealizovať rôzne. Vo vzorovom riešení najprv náhodne zvolíme celkový počet čiar na obraze. Polovicu z nich nakreslíme na náhodne zvolených riadkoch, druhú polovicu zase v náhodne zvolených stĺpcoch mriežky. Štvorčeky mriežky majú pôvodne bielu farbu výplne. Nakresliť čiaru preto znamená prefarbiť všetky štvorčeky vo zvolenom riadku, resp. vo zvolenom stĺpci načierno. Zmenu farbu výplne štvorčeka *obraz[r][s]* dosiahneme pomocou príkazu *itemconfig()*:

```
def ciary():
    pocet = random.randint(MIN_CIAR, MAX_CIAR)

    for h in range(pocet // 2):
        r = random.randrange(POCET_RIADKOV)
        for s in range(POCET_STLPCOV):
            platno.itemconfig(obraz[r][s], fill='black')

    for v in range(pocet // 2):
        s = random.randrange(POCET_STLPCOV)
        for r in range(POCET_RIADKOV):
            platno.itemconfig(obraz[r][s], fill='black')
```

Keď náhodou vyplníme štvorčeky v susediacich stĺpcoch či riadkoch, dosiahneme, že na obraze bude aj hrubšia čiara:



Obrázok 1.7 Príklady rôznych výstupov po zavolaní funkcie ciary()

IDENTIFIKOVANIE GRAFICKÝCH OBJEKTOV POMOCOU ŠTÍTKOV

Funkcie z modulu *tkinter*, ktorými kreslíme na plátno grafické objekty (napr. obdĺžniky, ovály a pod.) vracajú celé číslo, ktorým je nakreslený objekt jednoznačne určený. Okrem toho môžeme každému grafickému objektu priradiť štítok (*tag*) s reťazcom, ktorý ho popisuje, napr.:

```
# nakreslený štvorček pomenujme vchod (napr. v bludisku)
id = platno.create_rectangle(10, 10, 60, 60, fill='white', tag = 'vchod')

# pomocou štítku môžeme neskôr vlastnosti grafického objektu zmeniť
platno.itemconfig('vchod', fill='green')

# rovnako by sme mohli využiť aj zapamätaný číselný identifikátor
platno.itemconfig(id, fill='green')
```

Ak priradíme viacerým objektom rovnaký štítok, umožní nám to s týmito objektami pracovať jednotne ako so skupinou. N-ticu (*tuple*) všetkých grafických objektov s nejakým štítkom získame pomocou funkcie *find_withtag()*:

```
vchody = platno.find_withtag('vchod')
```

Všetky grafické objekty na plátne môžeme ľahko odstrániť príkazom:

```
platno.delete('all')
```

Analogicky môžeme odstrániť alebo konfigurovať akúkoľvek skupinu objektov s rovnakým štítkom:

```
platno.itemconfig('vchod', fill='red')
```

Pri generovaní Mondrianovho obrazu sa nám štítky zídu. Pomôžu nám rozlíšiť, ktoré štvorčeky mriežky sú prázdne, ktoré tvoria čiary alebo sú už vyplnené nejakou farbou.

ÚLOHA 4

Doplňte do funkcie na vykreslenie základnej mriežky pre každý vykreslený štvorček štítok *'prazdne'* a do funkcie na vykreslenie čiar pre každý čiarový štvorček štítok *'ciary'*.

Riešenie

Po zavolaní funkcií *mriezka()* a *ciary()* môžeme overiť, koľko štvorčekov má priradený rovnaký štítok:

```
mriezka()
prazdne = platno.find_withtag('prazdne')
print(len(prazdne))

ciary()
ciary = platno.find_withtag('ciary')
print(len(ciary))
```

ÚLOHA 5

Na obraze už máme zvislé a vodorovné čierne čiary. Potrebujeme ešte farbami vyplniť niektoré z obdĺžnikov, ktoré tieto čiary ohraničujú. Štvorčeky týchto obdĺžnikov sú označené štítkom *'prazdne'*. Ak niektorý z nich náhodne vyberieme, môžeme zvolenou farbou vyplniť celý obdĺžnik, do ktorého tento štvorček patrí. Naprogramujte opísané riešenie.

Riešenie

Funkciu vypĺňajúcu náhodný počet obdĺžnikov by sme mohli napísať takto:

```
def obdlzniky():
    pocet = random.randint(MIN_OBDLZNIKOV, MAX_OBDLZNIKOV)
    for i in range(pocet):
        farba = random.choice(FARBY)
        stvorcek = random.choice(prazdne)
        vypln(stvorcek, farba)

def vypln(stvorcek, farba):
    pass
```

Najprv zvolíme náhodne počet obdĺžnikov. Následne opakovane vyberáme niektorú z farieb a štvorček niektorého z obdĺžnikov, ktorý touto farbou vyplníme.

Vo funkcii *vypln()* potrebujeme okrem štvorčeka posielaného v parametri vyplniť aj jeho okolie v každom smere až po stĺpec či riadok tvoriaci hraničnú čiaru (alebo po okraj obrazu). Najprv však musíme zistiť, v ktorom riadku a stĺpci sa štvorček nachádza. Funkcia *coords()* vracia súradnice jeho ľavého a pravého dolného rohu ako zoznam. Získané súradnice prepočítame na riadkový a stĺpcový index štvorčeka:

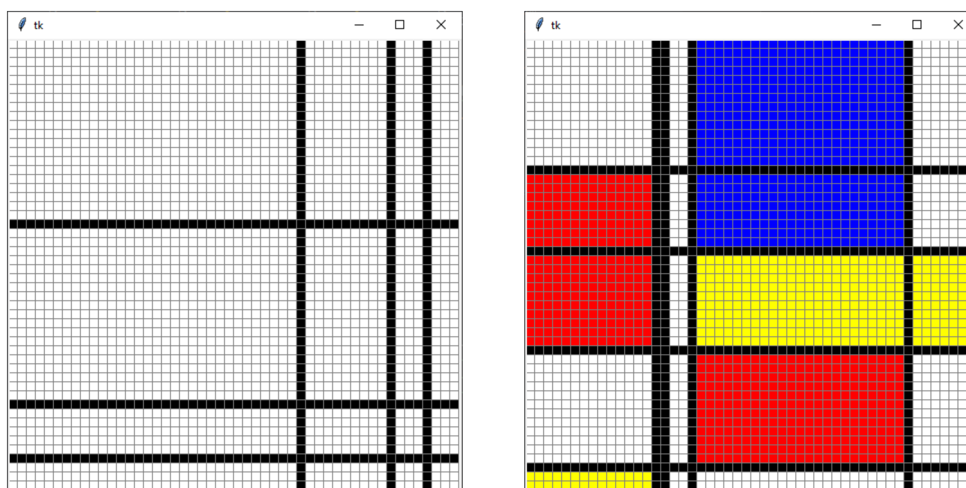
```
def vypln(stvorcek, farba):
    suradnice = platno.coords(stvorcek)
    start_r = int(suradnice[1]) // VELKOST_STVORCA
    start_s = int(suradnice[0]) // VELKOST_STVORCA
    platno.itemconfig(obraz[start_r][start_s], fill='red', tag='vypln')

    # vyhľadanie okrajových riadkových a stĺpcových indexov obdĺžnikovej
    # oblasti, v ktorej tento štvorček leží
    # rh - horný riadok, rd = dolný riadok
    # sl - ľavý stĺpec, sp = pravý stĺpec

    # prefarbenie všetkých švorčekov obdĺžnika
    for r in range(rh, rd + 1):
        for s in range(sl, sp + 1):
            platno.itemconfig(obraz[r][s], fill=farba, tag='vypln')
```

Na tomto mieste uvedieme vyhľadanie indexu prvého riadku obdĺžnikovej oblasti (analogicky postupujeme aj v ostatných smeroch):

```
    r = start_r
    while r >= 0 and obraz[r][start_s] in prazdne: r -= 1
    rh = r + 1
```

Začíname na riadku a v stĺpci východiskového štvorčeka. Riadkový index postupne klesá („v príslušnom stĺpci ideme nahor“). Sledujeme, či index *r* neklesol pod nulu (vybehli by sme z obrazu nad horný okraj) alebo či sa už nenachádzame na štvorčeku tvoriacom čiaru. Ak áno, cyklus skončí a zistený index si zapamätáme. V podmienke cyklu využívame, že po nakreslení čiar sme v hlavnom programe do globálneho zoznamu *prazdne* uložili čísla všetkých tých štvorčekov, ktoré nie sú súčasťou čiar.

Pri uvažovaní o prehľadávaní okolia náhodne zvoleného prázdneho štvorčeka (ale aj kreslení čiar v úvode riešenia) môže pomôcť dočasné zobrazenie obrysov štvorčekov mriežky (parameter *outline* nastavíme napr. na sivú farbu, t. j. *'gray'*).



Obrázok 1.8 Základná mriežka obrazov pred a po vyplnení obdĺžnikových oblastí 3 farbami

ZHRNUTIE

V tejto podkapitole nadväzujeme na skúsenosť s pamätaním si grafických objektov z úloh o tvorbe simulácie bunkového automatu z podkapitoly 1.3. Generovaný obraz tvorený mriežkou štvorčekov ukladáme do zoznamu zoznamov. Prvok *obraz[r][s]* predstavuje číselný kód konkrétneho štvorčeka nachádzajúceho sa na plátne v riadku *r* a stĺpci *s*. Každému zo štvorčekov mriežky sa v programe náhodne vyberá farba výplne (mení sa jeho vlastnosť *fill*). Využívajú sa funkcie na generovanie náhodných čísel a funkcia na výber náhodného prvku zo zoznamu z modulu *random*.

Potreba rozlišovať medzi štvorčekmi vypĺňaných obdĺžnikov a hraničných čiar nás priviedla k vysvetleniu významu štítkov („tagov"), ktoré je možné priradiť grafickým objektom.

Pomocou internetového vyhľadávača rýchlo nájdeme veľa originálnych Mondrianovych diel, ako aj príklady aplikovania Mondrianovych motívov na rôznych typoch produktov (plagáty, kalendáre, mapy, tašky, prívesky na kľúče, tričká, šaty, ponožky, šálky, taniere, karosérie áut, koberce, uteráky a pod.).

Pre žiakov odporúčame pripraviť pracovný list obsahujúci mriežky štvorčekov, ktoré si môžu najskôr ručne vyfarbiť a vykonať tak algoritmus, ktorý budú implementovať v jazyku Python. Pri programovaní kreslenia čiar a vypĺňania obdĺžnikov poslúži tento pracovný list ako pomôcka pri premýšľaní o indexoch štvorčekov a podmienke zotrvania v cykle pri hľadaní hraníc vypĺňaného obdĺžnika.

V **Úlohe 1** uvádzame ako vstupné parametre obrazu rozmery plátna, minimálne a maximálne počty čiar, minimálne a maximálne počty vypĺňaných obdĺžnikov a používané farby. Aby sme sa odlíšili od pôvodných Mondrianovych diel, môžeme k žltej, modrej a červenej farbe pridať nejakú ďalšiu základnú farbu, napr. zelenú.

Pri výbere obdĺžnika na vyplnenie sa môže stať, že opakovane vyberieme ten istý a prefarbíme ho viackrát inou farbou. Celkový počet vyplnených obdĺžnikov tak môže byť menší ako by sme čakali.

Z praktického hľadiska je vhodné doplniť rozhranie aplikácie o tlačidlo, ktorým by žiaci mohli opakovane vyvolávať vygenerovanie nového obrazu.

Vzorové riešenia všetkých úloh uvádzame v prílohe.

2 FUNKCIONÁLNE TECHNIKY PROGRAMOVANIA V JAZYKU PYTHON

Funkcionálny štýl programovania väčšinou nie je prezentovaný ako prvý pri vyučovaní algoritmizácie a základov programovania u začiatočníkov. Je to preto, že intuitívnemu chápaniu pojmu algoritmus väčšinou zodpovedá predstava návodu ako postupnosti príkazov. Na takejto intuitívnej predstave algoritmu je založený **imperatívny** (príkazový) štýl programovania.

Funkcionálne programovanie patrí medzi **deklaratívne** štýly programovania. Ani tento prístup však nemusí byť pre začiatočníkov celkom neznámy a cudzí. Môžu sa s ním stretnúť napríklad v matematike alebo pri práci v tabuľkovom kalkulátore, kde riešia problémy pomocou definovania vzorcov. Návod na výpočet je vyjadrený vzorcom, ktorý opisuje, čo má byť výsledkom. Premenné v matematike ani v tabuľkovom kalkulátore nemenia svoju hodnotu v čase. Iným príkladom deklaratívnych návodov z bežného života môžu byť návody na hranie rôznych spoločenských hier, ktoré sú vyjadrené množinou pravidiel, a nie postupnosťou príkazov, ktoré sa majú vykonávať v určenom poradí.

Ak pojem algoritmus definujeme ako „návod“, ktorým môže byť aj množina pravidiel alebo vzorec, a nezužujeme ho len na „postupnosť príkazov“, otvárame priestor aj pre iné ako imperatívne programovanie.

V tejto kapitole predstavíme základné techniky funkcionálneho štýlu programovania v jazyku Python. V podkapitole 2.1 predstavujeme jazyk Python ako multiparadigmový jazyk, ktorý umožňuje programovať rôznymi štýlmi – imperatívnym, objektovým a udalosťami riadeným, funkcionálnym štýlom.

V podkapitole 2.2 sú uvedené jednoduché príklady riešenia problémov pomocou definovania funkcií vo funcionálnom štýle bez vedľajších efektov v podobe zmien hodnôt premenných. Príklady v tejto podkapitole nepresahujú obsahové štandardy vyučovania informatiky na strednej škole.

V podkapitole 2.3 sa venujeme téme funkcií vyššieho rádu, ktorá je rozširujúcou témou vzhľadom na obsahové štandardy strednej školy. Používame len také funkcie, ktoré sú súčasťou základnej inštalácie interpretera jazyka Python a nevyžadujú si importovanie z rozširujúceho modulu.

V poslednej podkapitole 2.4 zhrnieme podstatné znaky funkcionálneho štýlu programovania v porovnaní s imperatívnym programovaním.

2.1 PYTHON AKO MULTIPARADIGMOVÝ JAZYK

V tejto podkapitole predstavíme rôzne štýly programovania v jazyku Python. Ako príklad sme zvolili jednoduchý problém výpočtu indexu telesnej hmotnosti BMI (Body Mass Index). Vstupmi pre výpočet sú hmotnosť človeka (*m*) v kilogramoch a výška (*v*) v metroch. Index sa vypočíta podľa vzorca:

$$BMI = \frac{m}{v^2}$$

KONZOLOVÁ APLIKÁCIA

Prvé riešenie je tzv. *konzolová aplikácia*. Má textové používateľské rozhranie, ktoré sa zobrazuje v okne Shell príkazového riadka interpretera. Vstupy sa zadávajú pomocou klávesnice, výstup je textový.

```
1   #vstup
2   m = float(input('Zadaj hmotnosť v kilogramoch: '))
3   v = float(input('Zadaj výšku v metroch: '))
4
5   #výpočet
6   bmi = m / v**2
7
8   #výstup
9   print('BMI: ', bmi)
```

Riešenie algoritmického problému výpočtu BMI je jednoduché (riadok 6), zvyšné časti kódu zabezpečujú komunikáciu s používateľom – vstupy a výstup.

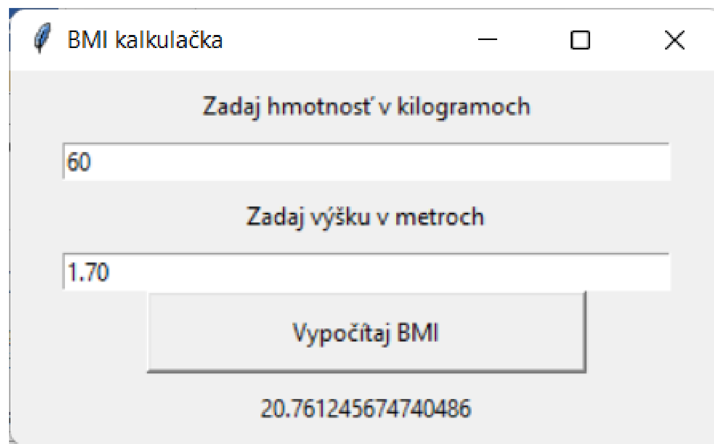
Riešenie predstavuje *imperatívnu paradigmu* programovania, v ktorej program je postupnosť príkazov (imperatívov). V našom programe sú to tri priraďovacie príkazy a príkaz výstupu. Po spustení programu sa reštartuje interpreter a príkazy sa vykonajú. Príklad výpočtu:

```
================ RESTART: C:\Python pre ucitelov\bmi_konzola.py ================
Zadaj hmotnosť v kilogramoch: 75
Zadaj výšku v metroch: 1.85
BMI:  21.913805697589478
>>>
```

Program má jednoduchú sekvenčnú štruktúru: má začiatok (načítanie vstupov), riešenie problému (výpočet BMI) a koniec (výpis riešenia). Aplikácia má priateľské používateľské rozhranie – vypisuje výzvy na zadanie vstupov (chybné vstupy nerieši) a slovný popis výsledku. Jej nevýhodou je, že počíta BMI pre jednu dvojicu vstupov. Pre inú dvojicu vstupov treba program znovu spustiť.

APLIKÁCIA S GRAFICKÝM POUŽÍVATEĽSKÝM ROZHRANÍM

Druhé riešenie problému výpočtu BMI je aplikácia, ktorá beží v okne a poskytuje grafické používateľské rozhranie na zadávanie vstupov, spúšťanie výpočtu a zobrazovanie výsledku. Aplikácia sa ovláda myšou a klávesnicou. V dvoch riadkových editoroch sa dajú editovať vstupy a stlačením tlačidla sa vypočíta a zobrazí BMI. Výpočet sa dá opakovať ľubovoľne veľa krát pre rôzne vstupy. Aplikácia sa ukončí zavretím okna pomocou systémového tlačidla X.



Obrázok 2.1 Aplikácia BMI kalkulačka s grafickým používateľským rozhraním

Na realizáciu grafického používateľského rozhrania aplikácie sme využili knižnicu *tkinter*, ktorá je súčasťou štandardnej inštalácie interpretera jazyka Python. V aplikácii sme použili ovládacie prvky (widgety) na zobrazenie nápisu v okne (*Label*), riadkové editory na zadanie vstupov (*Entry*), tlačidlo na vykonanie príkazu (*Button*).

```
1   import tkinter as tk
2
3   window = tk.Tk()
4   window.title('BMI kalkulačka')
5
6   label1 = tk.Label(
7       text="Zadaj hmotnosť v kilogramoch",
8       height=2,
9       width = 50)
10  entry1 = tk.Entry(width = 50)
11  entry1.insert(0,'60')
12
13  label2 = tk.Label(
14      text="Zadaj výšku v metroch",
15      height=2,
16      width = 50)
17  entry2 = tk.Entry(width = 50)
18  entry2.insert(0,'1.70')
19
20  label3 = tk.Label(
21      height=2,
22      width = 50)
23
24  def klik():
25      label3.config( \
26          text = str(float(entry1.get())/float(entry2.get())**2))
27
28  button = tk.Button(
29      text="Vypočítaj BMI",
30      height=2,
31      width = 30,
32      command = klik)
33
34  label1.pack()
35  entry1.pack()
36  label2.pack()
```

```
37  entry2.pack()
38  button.pack()
39  label3.pack()
```

Väčšina programu rieši vytvorenie grafického používateľského rozhrania aplikácie. Problém výpočtu BMI sa rieši na riadku 26 vo funkcii *klik*, ktorá sa spúšťa v reakcii na udalosť stlačenia tlačidla. Textové hodnoty z riadkových editorov *entry1* a *entry2* sa pretypujú na desatinné čísla a dosadia sa do vzorca na výpočet BMI. Výsledok sa pretypuje na textový reťazec, na ktorý sa nastaví hodnota vlastnosti text objektu *label3*.

Program na vytvorenie aplikácie s grafickým používateľským rozhraním využíva *objektovú paradigmu* programovania. Ovládacie prvky v rozhraní aplikácie sú objekty vytvorené ako inštancie tried z knižnice *tkinter*. Tok výpočtu je riadený udalosťami.

Na rozdiel od predchádzajúceho poskytuje toto riešenie krajšie grafické používateľské rozhranie v okne, v ktorom umožňuje používateľovi viacnásobné editovanie vstupov a spúšťanie výpočtu. Pri pohľade na kód však vidíme, že pre programátora sa triviálny algoritmický problém výpočtu BMI zmenil na netriviálnu úlohu, aj keď sa využije knižnica s hotovými triedami objektov.

FUNKCIA

Tretím riešením, ktoré uvedieme, je definícia funkcie na výpočet BMI. Nejedná sa o aplikáciu, lebo kód nerieši používateľské rozhranie. Výpis na riadkoch 1, 2 je pridaný len ako dokumentácia.

```
1   print('Funkcia bmi počíta pre zadanú hmotnosť v kg a výšku v m \
2          Body Mass Index')
3
4   def bmi(hmotnost,vyska):
5       return hmotnost / vyska**2
```

Na komunikáciu s používateľom využijeme príkazový riadok interpretera Python Shell. Po spustení programu sa vypíše dokumentačná veta k funkcii, ale žiadny výpočet BMI neprebehne. Ten spustíme až v príkazovom riadku volaním funkcie *bmi* s konkrétnymi parametrami hmotnosti a výšky. Výpočet môžeme v príkazovom riadku opakovať ľubovoľne veľa krát s rôznymi vstupnými parametrami:

```
================== RESTART: C:\Python pre ucitelov\bmi_fun.py ==================
Funkcia bmi počíta pre zadanú hmotnosť v kg a výšku v m Body Mass Index
>>> bmi(75,1.85)
21.913805697589478
>>> bmi(60,1.65)
22.03856749311295
>>> bmi(60,1.40)
30.612244897959187
>>> 
```

Riešenie predstavuje *funkcionálnu paradigmu* programovania. Funkcia je matematická abstrakcia algoritmického problému – vstupom priraďuje výstupy. Definícia funkcie (na riadkoch 4 – 5) nie je postupnosť príkazov ako v imperatívnom programovaní, ale je to výraz, ktorý formálnym spôsobom deklaruje, čo je výsledok (podiel hmotnosti a druhej mocniny výšky). Takýto prístup k riešeniu problémov voláme tiež *deklaratívny*.

Pri volaní funkcie sa zadaným vstupom podľa definície funkcie priradí výsledok. Volanie funkcie v príkazovom riadku je výraz, ktorého hodnotu príkazový riadok automaticky zobrazí aj bez príkazu *print*.

Využitie príkazového riadka Shell ako používateľského rozhrania na výpočet BMI má svoje výhody a nevýhody. Výhodou je, že výpočet môžeme opakovať a experimentovať s rôznymi vstupmi. Takáto interaktivita je užitočná pre programátora pri písaní, testovaní a ladení programu. Ako pomôcka pri zadávaní vstupov namiesto výzvy slúži bublina, ktorá pri písaní volania funkcie do príkazového riadka zobrazuje formálne parametre funkcie. Ak si ich vhodne pomenujeme, ich názov môže slúžiť ako zrozumiteľná výzva, čo treba zadať ako vstup:

```
>>> bmi(
        (hmotnost, vyska)
```

Technické rozhranie príkazového riadka je určené pre programátorov, pre bežného používateľa bez znalosti programového kódu je nepoužiteľné. Ak chceme vytvoriť aplikáciu pre laického používateľa, je potrebné naprogramovať okrem funkcie na riešenie problému aj používateľské rozhranie. Na to sa viac hodia imperatívny alebo objektový prístup.

2.2 FUNKCIONÁLNY ŠTÝL

V nasledujúcich podkapitolách sa zameriame na techniky programovania pri riešení algoritmických problémov s využitím funkcií a deklaratívneho štýlu programovania. Nebudeme riešiť tvorbu používateľského rozhrania, ktorá s riešením problému nesúvisí, a budeme využívať technické rozhranie príkazového riadka. Taktiež nebudeme riešiť zadávanie neplatných vstupov a budeme predpokladať, že pri volaní do funkcie vstupujú len platné hodnoty z jej definičného oboru.

PRÍKLAD: DÁTUMOVÁ ARITMETIKA

Dátumová aritmetika je súbor funkcií, ktoré opisujú vlastnosti dátumov a operácie s nimi. Dátum budeme reprezentovať ako usporiadanú trojicu celých čísel deň, mesiac, rok. V Pythone sa na to hodí údajový typ n-tica (*tuple*), napríklad trojica (1, 1, 2000) predstavuje dátum 1. 1. 2000.

Dátum však nie je akákoľvek trojica celých čísel. Mesiace sú celé čísla z množiny 1, 2, ..., 12, dni sú celé čísla z množín závislých od mesiaca a priestupnosti roka. Údaje o počte dní v jednotlivých mesiacoch v nepriestupnom roku si na úvod modulu s dátumovou aritmetikou uložíme do premennej *pocet_dni* typu n-tica dĺžky 12.

Okrem dátumu sa zvykne určovať aj deň v týždni z množiny pondelok, utorok, ..., nedeľa. Názvy dní usporiadané do postupnosti od pondelka do nedele si uložíme v premennej *den* typu n-tica dĺžky 7.

```
1   ## 3. 1. 2000 je pondelok
2
3   pocet_dni = (31,28,31,30,31,30,31,31,30,31,30,31)
4   den = ('pondelok', 'utorok', 'streda', 'štvrtok', 'piatok', \
5            'sobota', 'nedeľa')
```

VÝRAZY A PODMIENENÉ VÝRAZY

Prvá funkcia *priestupny* zisťuje, či je zadaný rok priestupný. Výsledkom volania funkcie je logická hodnota (*True* alebo *False*). Jej výpočet je zapísaný deklaratívne ako logický výraz (boolovská formula) s premennou *rok*, ktorá je vstupným parametrom funkcie *priestupny*.

Pravidlá pre určenie priestupného roka sa dajú nájsť napríklad na Wikipédii[1]. Logický výraz vo výsledku funkcie vyjadruje, že priestupný rok je deliteľný štyrmi a zároveň nedeliteľný stomi alebo deliteľný štyristomi:

```
7   def priestupny(rok):
8       return (rok % 4 == 0) and (rok % 100 != 0) \
9               or (rok % 400 == 0)
```

[1] https://sk.wikipedia.org/wiki/Priestupný_rok

Správnosť výpočtu otestujeme v príkazovom riadku okna Shell volaním funkcie s rôznymi hodnotami parametra *rok*:

```
>>> priestupny(1900)
False
>>> priestupny(2000)
True
>>> priestupny(2012)
True
>>> priestupny(1989)
False
>>> |
```

Druhá funkcia *platny* s parametrom *datum* je tiež logická funkcia, ktorá vracia logickú hodnotu *True*, ak zadaná trojica čísel je platný dátum, inak vráti hodnotu *False*.

Priraďovací príkaz na riadku 12 slúži na vytvorenie pomocných lokálnych premenných den, mesiac, rok, ktorým priradíme jednotlivé prvky usporiadanej trojice vstupného parametra *datum*.

Výsledok sa určí vyhodnotením logického výrazu na riadkoch 13 – 15 za kľúčovým slovom *return*: mesiac musí byť celé číslo medzi 1 a 12 a zároveň deň je celé číslo medzi 1 a počtom dní v mesiaci alebo je to 29. 2. v prípade priestupného roka.

```
11  def platny(datum):
12      den, mesiac, rok = datum
13      return 1 <= mesiac <= 12 \
14              and 1 <= den <= pocet_dni[mesiac - 1] \
15              or (den, mesiac) == (29, 2) and priestupny(rok)
```

Príklady výpočtu:

```
>>> platny((31,2,2000))
False
>>> platny((29,2,2000))
True
>>> platny((50,1,1950))
False
>>> platny((5,13,2020))
False
>>> platny((1,9,2036))
True
>>>
```

Vonkajšie zátvorky vo volaní funkcie ohraničujú parametre funkcie *platny*, vnútorné zátvorky ohraničujú hodnotu typu n-tica, ktorou je reprezentovaný vstupný parameter dátum.

Deklaratívny štýl programovania, akým je aj funkcionálne programovanie, nepoužíva príkazy, ale definuje výsledok ako výraz. V jazyku Python existuje jazyková konštrukcia, ktorá umožňuje definovať aj podmienené výrazy.

Jednoduchý príklad výrazu s podmienkou je v definícii nasledujúcej funkcie *dni_v_roku*, ktorá určí počet dní v danom roku. Výsledkom je buď 366, ak je rok priestupný, inak je počet dní v roku 365. Podmienený výraz je na riadku 18 za kľúčovým slovom *return* vracajúcim výsledok funkcie.

```
17  def dni_v_roku(rok):
18      return 366 if priestupny(rok) else 365
```

Príklady výpočtu funkcie *dni_v_roku* v rozhraní príkazového riadka:

```
>>> dni_v_roku(1900)
365
>>> dni_v_roku(2000)
366
>>> dni_v_roku(2020)
366
>>> dni_v_roku(2022)
365
>>> |
```

Podmienený výraz možno použiť rovnako ako iné výrazy aj vnorene ako súčasť iného výrazu. Príklad zložitejšieho aritmetického výrazu s podmieneným výrazom je v nasledujúcej funkcii *poradove_cislo*, ktorá počíta poradové číslo dňa v danom roku. Vstupom je dátum.

```
20  def poradove_cislo(datum):
21      den, mesiac, rok = datum
22      return sum(pocet_dni[:mesiac-1]) \
23             + den \
24             + (1 if priestupny(rok) and (mesiac > 2) else 0)
```

Na riadku 21 sú podobne ako v predchádzajúcom príklade pomenované lokálnymi premennými *den*, *mesiac*, *rok* prvky trojice *datum*. Poradové číslo dňa v roku sa vypočíta ako súčet:

- súčtu dní kompletných mesiacov pred zadaným dátumom – urobí sa rez z postupnosti *pocet_dni* od začiatku po mesiac pred vstupným dátumom *pocet_dni[:mesiac-1]* a sčíta sa funkciou *sum* (riadok 22),
- počtu dní v zadanom mesiaci v premennej *den* (riadok 23),
- 1 v prípade, že je vstupný dátum v priestupnom roku po 29. 2., lebo premenná *pocet_dni* obsahuje počty dní v mesiacoch nepriestupného roka, alebo 0 v opačnom prípade – tu je podmienený výraz použitý v súčte s inými výrazmi (riadok 24).

Príklady výpočtov v príkazovom riadku v okne Shell:

```
>>> poradove_cislo((1,1,2000))
1
>>> poradove_cislo((29,2,2000))
60
>>> poradove_cislo((31,12,2000))
366
>>>
```

V ďalších dvoch príkladoch použijeme viacnásobne vnorený podmienený výraz. Vytvorme funkcie *zajtra* a *vcera* s parametrom *datum*, ktoré vrátia dátum nasledujúci resp. predchádzajúci zadanému dátumu.

```
26  def zajtra(datum):
27      (den, mesiac, rok) = datum
28      return (den + 1, mesiac, rok) if den < pocet_dni[mesiac - 1] \
29                      or den == 28 and priestupny(rok) else \
30             (1,1,rok + 1) if (den, mesiac) == (31, 12) else \
31             (1, mesiac + 1, rok)
32
33  def vcera(datum):
34      (den, mesiac, rok) = datum
35      return (den - 1, mesiac, rok) if den > 1 else \
36             (29, 2, rok) if mesiac == 3 and priestupny(rok) else \
37             (31, 12, rok - 1) if (den, mesiac) == (1, 1) else \
38             (pocet_dni[mesiac - 2], mesiac - 1, rok)
```

Zajtrajší dátum vo funkcii *zajtra* je určený ako vnorený podmienený výraz s tromi možnými vetvami:

- zajtrajší deň je o 1 vyšší ako vstupný deň, ak je deň menší ako maximálny počet dní v mesiaci alebo ak je deň 28 v prípade priestupného roka (riadky 28 – 29),
- inak zajtrajší deň je 1. 1. nasledujúceho roka, ak vstupný dátum je 31. 12. (riadok 30),
- inak zajtrajší deň je 1. deň v nasledujúcom mesiaci (riadok 31).

Včerajší deň vo funkcii *vcera* je určený ako podmienený výraz so štyrmi možnými vetvami:

- včerajší deň je o 1 nižší ako vstupný deň, ak je deň väčší ako 1 (riadok 35),
- inak včerajší deň je 29. 2., ak vstupný mesiac je marec a rok je priestupný (riadok 36),
- inak včerajší deň je 31. 12. predchádzajúceho roka, ak vstupný dátum je 1. 1. (riadok 37),
- inak včerajší deň je posledný deň predchádzajúceho mesiaca (riadok 38).

Príklady výpočtov v príkazovom riadku:

```
>>> zajtra((28,2,2000))
(29, 2, 2000)
>>> zajtra((28,2,2001))
(1, 3, 2001)
>>> vcera((1,1,2000))
(31, 12, 1999)
>>> vcera((1,9,2022))
(31, 8, 2022)
>>> zajtra((31,12,2050))
(1, 1, 2051)
>>> vcera((4,9,1990))
(3, 9, 1990)
>>>
```

Výsledkom funkcií *zajtra* a *vcera* je štruktúrovaná hodnota dátum (trojica čísel).

REKURZIA VERZUS CYKLUS

Funkcie *zajtra* a *vcera* môžeme použiť pri definovaní ďalších funkcií v dátumovej aritmetike: na pripočítanie nejakého počtu dní k dátumu a na odpočítanie nejakého počtu dní od dátumu. Obe funkcie majú dva vstupné parametre: východzí dátum a počet dní, výsledkom je dátum.

V imperatívnom štýle vyriešime pripočítanie počtu dní k dátumu pomocou cyklu *for* tak, že príslušný počet krát zvýšime dátum na zajtrajší pomocou funkcie *zajtra*. Výsledok sa počíta vykonávaním postupnosti príkazov, ktoré iteratívne menia hodnotu premennej *vysledok*, až kým nenadobudne tú správnu hodnotu:

```
40  def plus_cyk(datum, pocet_dni):
41      vysledok = datum
42      for i in range(pocet_dni):
43          vysledok = zajtra(vysledok)
44      return vysledok
```

Príklady výpočtu:

```
>>> plus_cyk((1,1,2000), 30)
(31, 1, 2000)
>>> plus_cyk((1,1,2000), 100)
(10, 4, 2000)
>>> plus_cyk((1,1,2000), 366)
(1, 1, 2001)
>>> plus_cyk((1,1,2000), 1500)
(9, 2, 2004)
>>> |
```

Vo funkcionálnom programovaní sa opakovanie výpočtu dosahuje rekurzívnou definíciou funkcie:

```
46  def plus_rek(datum, pocet_dni):
47      return plus_rek(zajtra(datum),pocet_dni - 1) if pocet_dni > 0 \
48              else datum
```

Rekurzívna definícia funkcie je podmienený výraz, ktorý rozlišuje rekurzívny a triviálny prípad. Podmienka závisí od počtu dní, ktoré treba k danému dátumu pripočítať:

- ak je počet dní väčší ako nula, výsledkom je zajtrajší dátum, ku ktorému sa pripočíta rekurzívne tou istou funkciou o jeden menší počet dní (riadok 47),
- inak dátum netreba zvyšovať o žiadny počet dní (riadok 48).

Príklady výpočtu:

```
>>> plus_rek((1,1,2000), 30)
(31, 1, 2000)
>>> plus_rek((1,1,2000), 100)
(10, 4, 2000)
>>> plus_rek((1,1,2000), 366)
(1, 1, 2001)
>>> plus_rek((1,1,2000), 1500)
Traceback (most recent call last):
  File "<pyshell#151>", line 1, in <module>
    plus_rek((1,1,2000), 1500)
  File "C:\Python pre ucitelov\datumy.py", line 47, in plus_rek
    return plus_rek(zajtra(datum), pocet_dni - 1) if pocet_dni > 0 \
  File "C:\Python pre ucitelov\datumy.py", line 47, in plus_rek
    return plus_rek(zajtra(datum), pocet_dni - 1) if pocet_dni > 0 \
  File "C:\Python pre ucitelov\datumy.py", line 47, in plus_rek
    return plus_rek(zajtra(datum), pocet_dni - 1) if pocet_dni > 0 \
  [Previous line repeated 1021 more times]
  File "C:\Python pre ucitelov\datumy.py", line 28, in zajtra
    return (den + 1, mesiac, rok) if den < pocet_dni[mesiac - 1] \
RecursionError: maximum recursion depth exceeded in comparison
>>>
```

Rekurzívna funkcia *plus_rek* dáva rovnaké výsledky ako funkcia *plus_cyk* využívajúca cyklus, avšak pri väčších hodnotách parametra *pocet_dni* skončí výpočet chybou „prekročená maximálna hĺbka rekurzie“. Je to preto, že Python nie je funkcionálny jazyk a jeho interpreter neoptimalizuje rekurzívne výpočty tak, ako interpretery čistých funkcionálnych jazykov, v ktorých je rekurzia jediná technika na opakovanie výpočtu. V jazyku Python je preto rozumnejšie kombinovať štýly programovania podľa toho, ktorý je výhodnejší. V tomto prípade je to imperatívne programovanie s využitím cyklu.

Podobne ako pripočítavanie dní k dátumu môžeme naprogramovať aj odpočítavanie dní od dátumu opakovanou aplikáciou funkcie *vcera*. Na riadkoch 50 – 54 je riešenie s využitím cyklu (funkcia *minus_cyk*), na riadkoch 56 – 58 je rekurzívne riešenie (funkcia *minus_rek*).

```
50  def minus_cyk(datum, pocet_dni):
51      vysledok = datum
52      for i in range(pocet_dni):
53          vysledok = vcera(vysledok)
54      return vysledok
55
56  def minus_rek(datum, pocet_dni):
57      return minus_rek(vcera(datum),pocet_dni - 1) if pocet_dni > 0 \
58              else datum
```

GENERÁTOROVÁ NOTÁCIA – LIST COMPREHENSION

V doterajších príkladoch boli hodnoty výrazov, ktoré sme počítali pomocou funkcií, jednoduchého typu (logická hodnota alebo číslo) alebo štruktúrovaná hodnota konštantnej dĺžky (dátum ako trojica čísel). Na vygenerovanie štruktúrovanej hodnoty typu zoznam alebo n-tica ľubovoľnej dĺžky sa používa technika list comprehension – generátorová notácia. Generátorová notácia vytvorí zoznam alebo n-ticu hodnôt daných výrazom pre všetky hodnoty generátora za splnenia podmienky (filtra). Napríklad:

```
>>> [rok for rok in range(2000,2022) if priestupny(rok)]
    [2000, 2004, 2008, 2012, 2016, 2020]
```

Vypočíta sa zoznam priestupných rokov medzi rokmi 2000 a 2021. Analyzujme zápis list comprehension bližšie. Výsledkom je:

- zoznam – určujú to vonkajšie hranaté zátvorky,
- prvky zoznamu sú hodnoty výrazu *rok*,
- prvky zoznamu sa generujú pre všetky hodnoty v generátorovej časti *for* (roky z rozsahu *range(2000, 2022)*)
- prvky zoznamu sa filtrujú podľa podmienky v časti *if* (priestupné roky).

Ak žiadny filter nie je uvedený, jeho hodnota je *True* a generujú sa hodnoty výrazov pre všetky prvky z generátora *for*. Napríklad:

```
>>> [dni_v_roku(r) for r in (1900,2000,2020,2022)]
    [365, 366, 366, 365]
```

Výsledkom je zoznam počtov dní (výraz *dni_v_roku(r)*) vo všetkých vymenovaných rokoch (generátor *for r in (1900, 2000, 2020, 2022)*) bez ďalšieho filtrovania podmienkou.

Iný príklad:

```
>>> [dni_v_roku(r) for r in range(2000,2010)]
    [366, 365, 365, 365, 366, 365, 365, 365, 366, 365]
```

Výsledkom je zoznam počtov dní v rozsahu rokov 2000 – 2009.

Generátorová notácia sa podobá na cyklus v imperatívnom štýle, ale formálne je to výraz, nie príkaz, a teda predstavuje deklaratívny štýl programovania.

Ďalšia funkcia z našej dátumovej aritmetiky, v ktorej využijeme generátorovú notáciu list comprehension, počíta rozdiel rokov v dňoch. Keďže roky môžu mať 365 alebo 366 dní, na výpočet môžeme použiť zoznam počtu dní v rokoch medzi danými dvomi rokmi z predchádzajúceho príkladu.

```
60  def rozdiel_rokov(rok1,rok2):
61      if rok1 > rok2: rok1, rok2 = rok2, rok1
62      return sum([dni_v_roku(r) for r in range(rok1, rok2)])
```

Na riadku 61 sa v prípade potreby robí výmena hodnôt parametrov *rok1* a *rok2* tak, aby boli usporiadané v čase, teda prvý rok bol menší alebo rovný ako druhý rok. Na riadku 62 sa definuje výsledok funkcie ako súčet (funkcia *sum*) zoznamu počtu dní v roku pre všetky roky z rozsahu *range(rok1,rok2)*. Zoznam je definovaný pomocou generátorovej notácie list comprehension.

Príklad výpočtu:

```
>>> rozdiel_rokov(2000,2022)
    8036
>>> 
```

ĎALŠIE FUNKCIE DÁTUMOVEJ ARITMETIKY

V ďalšej funkcii porovnajme dva dátumy. Dátum bude menší ako iný dátum vtedy, keď sa vyskytuje skôr v čase. Pri porovnávaní dvoch dátumov daných trojicou deň, mesiac, rok najprv porovnávame roky. V prípade, že roky sú rovnaké, porovnávame mesiace. V prípade, že aj mesiace sú rovnaké, závisí poradie od dní.

Na hodnotách údajového typu n-tica je definovaná relácia usporiadania, teda dajú sa porovnávať pomocou operátorov <, >, ==, <=, >=. Usporiadanie je lexikografické, pričom väčšiu váhu pri porovnávaní majú prvky n-tice zľava. Napríklad:

```
>>> (1,2,3) < (2,2,3)
True
>>> (1,2,3) < (1,2,4)
True
>>> (3,2,1) < (3,1,2)
False
>>> (5,5,5) > (4,6,8)
True
>>> (2,3,4,5) > (2,3,4,0)
True
>>>
```

Na porovnávanie dátumov nemôžeme použiť toto štandardné porovnávanie n-tíc, lebo v našom zápise dátumov majú pri porovnávaní vyššiu váhu položky sprava. Napríklad platí:

(31,12,2000) < (1,1,2001)

Avšak štandardné porovnanie n-tíc dá výsledok *False*:

```
>>> (31,12,2000) < (1,1,2001)
False
```

Zadefinujme vlastnú funkciu *mensi* s dvomi parametrami dátum, ktorá zistí, či prvý dátum je menší (skôr na časovej osi) ako druhý dátum.

V riešení využijeme štandardné porovnanie n-tíc na dátumy v obrátenom poradí položiek rok, mesiac deň.

```
64  def mensi(datum1, datum2):
65      return datum1[::-1] < datum2[::-1]
```

Na obrátenie poradia prvkov trojice sú použité rezy. Rez je časť iterovateľnej hodnoty (reťazca, zoznamu alebo n-tice) daný indexom začiatku rezu, indexom konca rezu (nie je súčasťou rezu) a nepovinným krokom, ktoré sú v hranatých zátvorkách oddelené dvojbodkou

[start : stop : krok]

Ak sa vynechá začiatočný alebo koncový index, rez sa robí od začiatku alebo do konca štruktúry. Ak sa vynechá krok, berú sa všetky prvky medzi danými indexami. Krok môže byť aj záporný.

V zápise na riadku 65 *datum1[::-1]* sú vynechané začiatok a koniec rezu, krok je -1, výsledkom bude celá trojica v obrátenom poradí.

Funkciu *mensi* použijeme v ďalšej funkcii *rozdiel_datumov*, ktorá pre dva dané dátumy vypočíta ich rozdiel v dňoch:

```
67  def rozdiel_datumov(datum1, datum2):
68      if mensi(datum2, datum1): datum1, datum2 = datum2, datum1
69      return rozdiel_rokov(datum1[2], datum2[2]) \
70              - poradove_cislo(datum1) \
71              + poradove_cislo(datum2)
```

Na riadku 68 sa v prípade potreby vymenia hodnoty vstupných parametrov tak, aby prvý dátum predchádzal v čase druhý dátum. Použitá je funkcia *mensi*.

Na riadkoch 69 – 71 sa počíta rozdiel dátumov (na obrázku 2.2 červený úsek) takto:

Obrázok 2.2 Výpočet rozdielu dvoch dátumov

- vypočíta sa rozdiel celých rokov od 1. 1. prvého dátumu do 1. 1. druhého dátumu pomocou funkcie *rozdiel_rokov* (riadok 69, na obrázku modrý úsek),
- odpočíta sa počet dní od začiatku prvého roka po prvý dátum ako poradové číslo dňa v roku pomocou funkcie *poradove_cislo* (riadok 70, na obrázku žltý úsek),
- pripočíta sa počet dní od začiatku druhého roka po druhý dátum ako poradové číslo dňa v roku pomocou funkcie *poradove_cislo* (riadok 71, na obrázku zelený úsek).

Príklady výpočtu:

```
>>> rozdiel_datumov((28,10,1918),(1,1,1993))
27094
>>> rozdiel_datumov((1,9,1939),(8,5,1945))
2076
>>>
```

Poslednou funkciou v našej dátumovej aritmetike bude určenie dňa v týždni pre daný dátum. Ak poznáme deň v týždni pre jeden konkrétny dátum, z rozdielu počtu dní vieme vypočítať deň v týždni pre akýkoľvek iný dátum.

```
4   den = ('pondelok', 'utorok', 'streda', 'štvrtok', 'piatok', \
5          'sobota', 'nedeľa')
```

```
73  den_v_tyzdni(datum):
74      return den[rozdiel_datumov((3,1,2000),datum) \
75                 * ((-1) if mensi(datum,(3,1,2000)) else 1) \
76                 % 7]
```

Na riadkoch 4 – 5 sme už definovali n-ticu *den* dĺžky 7 s názvami dní v týždni začínajúc pondelkom. Preto bude výhodné za pevný známy deň v týždni zvoliť nejaký pondelok, napríklad prvý pondelok v našom storočí, čo je 3. 1. 2000. Na riadkoch 73 – 76 sa nachádza definícia funkcie *den_v_tyzdni*. Poradové číslo dňa v týždni v n-tici *den* sa počíta:

- na riadku 74 sa vypočíta počet dní medzi daným dátumom a pondelkom 3. 1. 2000 pomocou funkcie *rozdiel_datumov*,

- na riadku 75 sa tento počet upraví na záporný, ak daný dátum predchádza pondelok 3. 1. 2000, alebo na kladný v opačnom prípade (násobenie číslom -1 alebo 1),
- na riadku 76 sa vypočíta zvyšok po delení tohto čísla siedmimi, ktorý určí poradové číslo dňa v týždni začínajúcim pondelkom.

Príklady výpočtu:

```
>>> den_v_tyzdni((17,11,1989))
'piatok'
>>> den_v_tyzdni((1,5,2004))
'sobota'
>>>
```

Ako príklady výpočtu sme zvolili významné dni v histórii Slovenska (začiatok Nežnej revolúcie a deň vstupu Slovenska do Európskej únie) v minulom a v tomto storočí.

2.3 FUNKCIE VYŠŠIEHO RÁDU

Funkcie vyššieho rádu sú také funkcie, ktoré ako vstupný parameter očakávajú inú funkciu alebo vracajú funkciu ako výsledok. Funkcie s funkcionálnym parametrom umožňujú zovšeobecniť podobné výpočty nielen na úrovni údajov, teda rovnaký algoritmus aplikovať na rôzne vstupné údaje, ale aj na úrovni algoritmu, teda zovšeobecniť podobné, ale nie rovnaké algoritmy. Konkrétna podoba všeobecného algoritmu, ktorý funkcia vyššieho rádu opisuje, sa špecifikuje v jej parametri odkazom na funkciu upresňujúcu jej všeobecný tvar.

V jazyku Python je implementovaných niekoľko štandardných funkcií vyššieho rádu, ktoré zovšeobecňujú niektoré výpočtové vzory s iterovateľnými štruktúrami údajov ako napríklad n-tice alebo zoznamy.

MAPOVANIE

Mapovanie je výpočtový vzor, ktorý s každým prvkom iterovateľnej postupnosti údajov vykoná nejaký výpočet – aplikuje nejakú funkciu. Mapovanie je implementované v Pythone ako funkcia vyššieho rádu *map* s dvomi parametrami: mapovacou funkciou a postupnosťou údajov. Výsledkom je iterovateľný objekt obsahujúci rovnaký počet prvkov ako pôvodná postupnosť.

Napríklad na všetky prvky zoznamu čísel aplikujme vo funkcii *map* funkciu *abs*:

```
>>> map(abs,[1,2,-3,0,-4])
    <map object at 0x000001F9922202E0>
>>>
```

Takýto objekt sa dá prechádzať napríklad vo *for* cykle alebo sa z neho dá vytvoriť zoznam alebo n-tica pomocou funkcií *list*, *tuple*.

```
>>> for i in map(abs,[1,2,-3,0,-4]):
...     print(i)
...
...
    1
    2
    3
    0
    4
>>> list(map(abs,[1,2,-3,0,-4]))
    [1, 2, 3, 0, 4]
>>> tuple(map(abs,[1,2,-3,0,-4]))
    (1, 2, 3, 0, 4)
```

Použime mapovanie pri načítaní postupnosti čísel. Úlohou je načítať zoznam čísel, ktoré zadá používateľ do riadka a oddelí ich medzerami. Vstup sa načíta ako reťazec, ktorý vieme pomocou reťazcovej metódy *split* rozdeliť podľa oddeľovača medzera (parameter *sep*) na zoznam reťazcov.

```
>>> vstup = input('Zadaj zoznam čísel oddelených medzerou: ')
    Zadaj zoznam čísel oddelených medzerou: 123 57 698 42 6 19
>>> vstup
    '123 57 698 42 6 19'
>>> vstup.split(sep=' ')
    ['123', '57', '698', '42', '6', '19']
```

Pomocou funkcie *map* vieme z takéhoto zoznamu vytvoriť postupnosť celých čísel aplikovaním funkcie *int* na každý prvok zoznamu alebo postupnosť desatinných čísel aplikovaním funkcie *float*. Výsledok mapovania ešte upravíme na zoznam pomocou funkcie *list*.

```
>>> list(map(int,vstup.split(sep=' ')))
[123, 57, 698, 42, 6, 19]
>>> list(map(float,vstup.split(sep=' ')))
[123.0, 57.0, 698.0, 42.0, 6.0, 19.0]
>>>
```

V ďalšom príklade použijeme mapovanie na zamaskovanie všetkých výskytov písmen i, í, y, ý v zadanom texte znakom podtržník (_). Funkcia *maska* s parametrom typu *char* vráti pre vstupné znaky i, í, y, ý výstup _, iné vstupy vráti nezmenené.

```
def maska(x):
    return '_' if x in "iíyý" else x
```

Zamaskovanie písmen i, í, y, ý v reťazci *text* urobíme pomocou funkcie *map* aplikovaním funkcie *maska* na reťazec v premennej *text*. Výsledkom mapovania bude objekt obsahujúci postupnosť znakov, ktoré pomocou metódy *join* spojíme prázdnym reťazcom a vytvoríme nový zamaskovaný reťazec:

```
>>> text = "Náš tím sa skladá z ôsmich ľudí, z nich štyri sú ženy."
>>> ''.join(map(maska,text))
'Náš t_m sa skladá z ôsm_ch ľud_, z n_ch št_r_ sú žen_.'
>>>
```

FILTROVANIE

Filtrovanie je výpočtový vzor, ktorý z nejakej postupnosti vyberie podpostupnosť len takých prvkov, ktoré zodpovedajú nejakej podmienke. Filtrovanie je v Pythone implementované pomocou štandardnej funkcie vyššieho rádu *filter*, ktorá má dva parametre: filtrovaciu funkciu s jedným parametrom, ktorá vracia hodnotu True alebo False, a filtrovanú postupnosť prvkov (napríklad zoznam, n-ticu, reťazec a pod.). Výsledkom je iterovateľný objekt, ktorý obsahuje podpostupnosť prvkov pôvodnej filtrovanej postupnosti.

Príklad: Definujme si logickú funkciu *parne*, ktorá zisťuje, či zadané celé číslo je párne.

```
def parne(x):
    return x % 2 == 0
```

Funkciu *parne* môžeme použiť na prefiltrovanie zoznamu čísel na párne čísla, alebo vytvorenie zoznamu párnych čísel z určitého rozsahu.

```
>>> list(filter(parne,[21,254,326,25,120]))
[254, 326, 120]
>>> list(filter(parne,range(1,20)))
[2, 4, 6, 8, 10, 12, 14, 16, 18]
>>>
```

Vytvoriť filtrovaný zoznam vieme aj technikou generátorová notácia (list comprehension), ktorá bola predstavená v podkapitole 2.2:

```
>>> [x for x in range(1,20) if parne(x)]
[2, 4, 6, 8, 10, 12, 14, 16, 18]
>>>
```

ANONYMNÉ FUNKCIE

Do funkcií vyššieho rádu, ako sú *map* a *filter*, vstupuje funkcia ako parameter prostredníctvom svojho mena. Môže ísť o štandardnú funkciu, ako napríklad, *int*, *float*, *abs*, alebo programátorom definovanú funkciu, ako napríklad *maska*, *parne*. Niekedy je táto funkcia veľmi jednoduchá alebo ju programátor nevyužije inde. Vtedy je možné nedefinovať ju ako vlastnú funkciu s menom, ale využiť tzv. anonymnú funkciu, teda funkciu bez mena. Anonymná funkcia sa definuje pomocou kľúčového slova *lambda* takto:

```
lambda <parametre oddelené čiarkami> : <výsledok ako výraz>
```

Napríklad funkcie *maska* a *parne* ako anonymné funkcie:

```
maska = lambda x: '_' if x in "iíyý" else x
parne = lambda x: x % 2 == 0
```

Keďže anonymná funkcia nemá meno, jej definícia sa používa všade tam, kde by sa použilo meno funkcie, napríklad pri volaní funkcie:

```
>>> (lambda x: '_' if x in "iíyý" else x)('i')
'_'
>>> (lambda x: x % 2 == 0)(4)
True
>>> (lambda x: not x % 2 == 0)(4)
False
>>> (lambda x: x + 1)(2022)
2023
>>> (lambda x: x[-1])("pondelok")
'k'
>>> (lambda x: x[-1])("streda")
'a'
>>> |
```

Anonymnú funkciu môžeme použiť ako skutočný parameter pri volaní funkcie vyššieho rádu namiesto mena funkcie, napríklad na vyfiltrovanie nepárnych čísel v zozname, zväčšenie všetkých čísel v zozname o 1, nahradenie samohlások v reťazci hláskou 'a':

```
>>> list(filter(lambda x: not x % 2 == 0, range(1,20)))
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>> list(map(lambda x: x + 1,[5,6,7,8,9]))
[6, 7, 8, 9, 10]
>>> ''.join(map(lambda x: 'a' if x in "aeiouy" else x,"Sedi mucha na stene"))
'Sada macha na stana'
>>> |
```

Vyriešme niekoľko jednoduchých úloh s využitím funkcií vyššieho rádu, v ktorých ako funkcionálny parameter použijeme anonymnú alebo pomenovanú funkciu.

Úloha: Napísať funkciu, ktorá pre dané prirodzené číslo nájde jeho všetky vlastné delitele.

V riešení pomocou funkcie *filter* z postupnosti čísel z rozsahu *range(2, n)* vyfiltrujeme tie, ktorými je zadané číslo *n* deliteľné. Výsledný iterovateľný objekt je funkciou *list* konvertovaný na zoznam.

```
def vlastne_delitele(n):
    return list(filter(lambda k: n%k == 0, range(2,n)))
```

Príklady výpočtu:

```
>>> vlastne_delitele(10)
[2, 5]
>>> vlastne_delitele(100)
[2, 4, 5, 10, 20, 25, 50]
>>> 
```

Úloha: Napísať funkciu, ktorá zistí, či dané prirodzené číslo je prvočíslo.

Na riešenie nepoužijeme priamo funkciu vyššieho rádu, ale funkciu *vlastne_delitele*, ktorou vygenerujeme zoznam vlastných deliteľov zadaného prirodzeného čísla. Keďže prvočíslo nemá žiadne vlastné delitele, riešením je výsledok porovnania počtu prvkov tohto zoznamu s nulou.

```
def je_prvocislo(n):
    return len(vlastne_delitele(n))==0
```

Príklady výpočtov:

```
>>> je_prvocislo(41)
True
>>> je_prvocislo(20)
False
>>> 
```

Úloha: Napísať funkciu, ktorá nájde všetky prvočísla menšie ako zadané prirodzené číslo.

V riešení pomocou funkcie *filter* z postupnosti čísel z rozsahu *range(2, n)* vyfiltrujeme prvočísla. Funkcionálnym parametrom bude pomenovaná funkcia *je_prvocislo*. Výsledný iterovateľný objekt je funkciou *list* konvertovaný na zoznam.

```
def prvocisla(n):
    return list(filter(je_prvocislo, range(2,n)))
```

Príklad výpočtu – všetky prvočísla menšie ako 200:

```
>>> prvocisla(200)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67,
71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149,
151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199]
>>> 
```

REDUKOVANIE

Redukovanie je výpočtový vzor, v ktorom je vstupná postupnosť postupne prechádzaná a redukovaná na jednu hodnotu. Univerzálna funkcia vyššieho rádu *reduce* zovšeobecňujúca redukciu postupnosti na jednu hodnotu sa nachádza v špeciálnom module *functools*. Bez jej importovania môžeme využiť štandardné funkcie ako *min*, *max*, *sum*, *len*, *any*, *all*, ktoré sú konkrétnymi príkladmi tohto výpočtového vzoru.

Funkcie *min*, *max* určia najmenšiu, resp. najväčšiu hodnotu z postupnosti hodnôt. Ak sú tieto hodnoty typu, na ktorom sú definované operácie <, >, použijú sa na porovnanie tieto operácie. Ak nie sú, alebo chceme porovnávať podľa iného kritéria, využijeme vo funkcii nepovinný funkcionálny parameter *key*. Tu uvedieme funkciu, ktorou porovnávané hodnoty zmeníme na také, podľa ktorých ich budeme porovnávať.

Napríklad: Ak sú hodnoty, z ktorých určujeme *min*, *max*, typu *string*, operácie <, > ich budú porovnávať lexikograficky (abecedne). Minimum teda bude reťazec prvý podľa abecedy, maximum

bude reťazec posledný podľa abecedy. Ak by sme chceli určiť najkratší, resp. najdlhší reťazec, operácie <, > majú porovnávať nie reťazce, ale ich dĺžky. Ako funkcionálny parameter *key* uvedieme funkciu *len*.

```
>>> max(['pondelok','utorok','streda','stvrtok','piatok','sobota','nedela'])
'utorok'
>>> max(['pondelok','utorok','streda','stvrtok','piatok','sobota','nedela'],
key = len)
'pondelok'
>>> min(['pondelok','utorok','streda','stvrtok','piatok','sobota','nedela'])
'nedela'
>>> min(['pondelok','utorok','streda','stvrtok','piatok','sobota','nedela'],
key = len)
'utorok'
>>>
```

V podkapitole 2.2 sme pracovali s dátumami ako trojicami celých čísel. Na určenie, ktorý dátum zo zoznamu dátumov je na časovej osi najskôr, resp. najneskôr, môžeme použiť funkcie *min*, *max*, ale pri porovnávaní musíme obrátiť poradie deň, mesiac, rok na opačné – rok, mesiac, deň. V riešení je ako parameter *key* vo funkciách *min*, *max* použitá anonymná funkcia, ktorá obracia poradie prvkov trojice reprezentujúcej dátum.

```
def najskor(zoznam_datumov):
    return min(zoznam_datumov, key = (lambda x: (x[2],x[1],x[0])))

def najneskor(zoznam_datumov):
    return max(zoznam_datumov, key = (lambda x: (x[2],x[1],x[0])))
```

Príklady výpočtov:

```
>>> najskor([(10,3,2000),(1,12,2000),(31,12,1999)])
(31, 12, 1999)
>>> najneskor([(10,3,2000),(1,12,2000),(31,12,1999)])
(1, 12, 2000)
>>>
```

CELEBRITY EŠTE RAZ

V podkapitole 1.1 sme riešili úlohy o celebritách, ktoré sú prítomné na festivale v určitých časových intervaloch. Spracovávali sme zoznam *intervaly*, ktorý obsahoval trojice údajov: meno, hodina príchodu, hodina odchodu.

```
intervaly = [('Adele', 6, 8),
             ('Meryl Streep', 7, 8),
             ('Leonardo DiCaprio', 5, 9),
             ('Justin Timberlake', 7, 10),
             ('Peter Sagan', 6, 9),
             ('Ed Sheeran', 9, 10)]
```

Na porovnanie s imperatívnym štýlom programovania vyriešme niektoré úlohy ešte raz s využitím techník funkcionálneho programovania.

Úloha: Koľko celebrít bude prítomných o konkrétnej hodine? Ktoré sú to?

Funkcia *pocet_pritomnych* má dva parametre – zoznam údajov o prítomnosti celebrít (*intervaly*) a hodinu. V riešení použijeme funkciu *filter*, ktorou zoznam *intervaly* prefiltrujeme na tie prvky, pre ktoré sa vstupná hodina nachádza medzi hodinou príchodu (údaj na indexe 1 v trojici) a odchodu celebrity (údaj na indexe 2 v trojici). Ako podmienka filtrovania je použitá anonymná funkcia.

Výsledok filtrovania konvertujeme funkciou *list* na zoznam a funkciou *len* určíme počet jeho prvkov – počet prítomných celebrít v danej hodine.

```
def pocet_pritomnych(intervaly, hodina):
    return len(list(filter(lambda x: x[1] <= hodina < x[2], intervaly)))
```

Príklad výpočtu – počet prítomných o 8. hodine:

```
>>> pocet_pritomnych(intervaly,8)
3
>>>
```

Ak chceme zistiť nielen počet, ale aj mená celebrít, ktoré budú prítomné o konkrétnej hodine, namiesto funkcie *len* použijeme na prefiltrovaný zoznam mapovanie funkciou *map*, ktorou zo zoznamu trojíc (meno, čas príchodu, čas odchodu) vyberieme len meno – údaj na indexe 0.

```
def pritomni(intervaly, hodina):
     return list(map((lambda x: x[0]), \
                     filter(lambda x: x[1] <= hodina < x[2], intervaly)))
```

Príklad výpočtu:

```
>>> pritomni(intervaly,8)
['Leonardo DiCaprio', 'Justin Timberlake', 'Peter Sagan']
>>>
```

Úloha: Kedy príde prvá celebrita, odíde posledná celebrita?

Úlohu vyriešime dvomi spôsobmi. Prvým spôsobom vyriešime zistenie prvého príchodu celebrity – začiatok akcie, druhým zistenie posledného odchodu celebrity – koniec akcie. Oba problémy sa analogicky dajú riešiť oboma spôsobmi.

Vo funkcii *zaciatok* zistíme príchod prvej celebrity ako minimum z príchodov všetkých celebrít. Zoznam príchodov celebrít získame mapovaním zoznamu trojíc *intervaly* anonymnou funkciou, ktorá z trojice vráti údaj o príchode na indexe 1. Analogicky môžeme zistiť odchod poslednej celebrity, keď namiesto funkcie *min* použijeme funkciu *max* a v mapovacej funkcii vyberáme údaj o odchode na indexe 2.

```
def zaciatok(intervaly):
    return min(map(lambda x: x[1],intervaly))
```

Vo funkcii *koniec* ukážeme iné riešenie s využitím funkcionálneho parametra *key* funkcie *max*. Rozdiel oproti predchádzajúcemu riešeniu je ten, že sa neurčuje maximum zo zoznamu odchodov, ale maximum z trojíc podľa času odchodu (údaj na indexe 2) – kritérium sa uvádza v nepovinnom parametri *key* funkcie *max*. Výsledkom teda bude celá trojica meno, čas príchodu, čas odchodu. Z nej treba ešte vybrať údaj o odchode na indexe 2. Analogicky môžeme takto zistiť aj príchod prvej celebrity.

```
def koniec(intervaly):
    return max(intervaly, key = lambda x: x[2])[2]
```

Úloha: V akom poradí budú celebrity prichádzať, v akom odchádzať?

Riešením úlohy je usporiadanie údajov podľa času príchodu, resp. času odchodu. Použijeme funkciu *sorted*, ktorá daný zoznam vracia usporiadaný. Anonymná funkcia v parametri *key* vracia hodnotu, podľa ktorej sa má zoznam usporiadať.

Zoznam *intervaly* usporiadaný podľa času príchodu – anonymná funkcia v parametri *key* vracia čas príchodu na indexe 1 v trojiciach údajov v zozname intervaly:

```
>>> sorted(intervaly, key=(lambda x: x[1]))
[('Leonardo DiCaprio', 5, 9), ('Adele', 6, 8), ('Peter Sagan', 6, 9),
('Meryl Streep', 7, 8), ('Justin Timberlake', 7, 10), ('Ed Sheeran', 9
, 10)]
>>>
```

Zoznam *intervaly* usporiadaný podľa času odchodu – anonymná funkcia v parametri *key* vracia čas odchodu na indexe 2 v trojiciach údajov v zozname intervaly:

```
>>> sorted(intervaly, key=(lambda x: x[2]))
[('Adele', 6, 8), ('Meryl Streep', 7, 8), ('Leonardo DiCaprio', 5, 9),
 ('Peter Sagan', 6, 9), ('Justin Timberlake', 7, 10), ('Ed Sheeran', 9
, 10)]
>>>
```

Vo funkcii *poradie_prichadzajucich* z usporiadaného zoznamu intervaly podľa časov príchodu vytvoríme zoznam obsahujúci len mená celebrít v tomto poradí. Funkcia *poradie_odchadzajucich* vytvorí zoznam mien celebrít v poradí podľa časov odchodu.

Na riešenie môžeme použiť mapovanie funkciou *map* a výsledok konvertovať na zoznam funkciou *list*, alebo vygenerovať zoznam pomocou generátorovej notácie list comprehension. Vo funkcii *poradie_prichadzajucich* je prezentované použitie mapovania, vo funkcii *poradie_odchadzajucich* použitie list comprehension:

```
def poradie_prichadzajucich(intervaly):
    return  list(map(lambda x: x[0],\
                     sorted(intervaly, key=(lambda x: x[1]))))

def poradie_odchadzajucich(intervaly):
    return [x[0] for x in sorted(intervaly, key=(lambda x: x[2]))]
```

Výstupmi budú zoznamy mien celebrít v príslušných poradiach:

```
>>> poradie_prichadzajucich(intervaly)
['Leonardo DiCaprio', 'Adele', 'Peter Sagan', 'Meryl Streep', 'Justin
Timberlake', 'Ed Sheeran']
>>> poradie_odchadzajucich(intervaly)
['Adele', 'Meryl Streep', 'Leonardo DiCaprio', 'Peter Sagan', 'Justin
Timberlake', 'Ed Sheeran']
>>>
```

Výsledok ešte môžeme spracovať do slovnej odpovede spojením mien v zozname do jedného reťazca pomocou reťazcovej metódy *join* a oddeliť čiarkami:

```
print(f'Celebrity budú prichádzať v poradí {", ".join(poradie_prichadzaju
cich(intervaly))}.')

print(f'Celebrity budú odchádzať v poradí {", ".join(poradie_odchadzajuci
ch(intervaly))}.')
```

```
Celebrity budú prichádzať v poradí Leonardo DiCaprio, Adele, Peter Sagan,
Meryl Streep, Justin Timberlake, Ed Sheeran.
Celebrity budú odchádzať v poradí Adele, Meryl Streep, Leonardo DiCaprio,
Peter Sagan, Justin Timberlake, Ed Sheeran.
>>>
```

Úloha: O koľkej hodine je prítomných najviac celebrít? Ktoré sú to?

V riešení použijeme funkcie *zaciatok* a *koniec* na zistenie časového intervalu od príchodu prvej do odchodu poslednej celebrity (premenné *od*, *do*). Pomocou generátorovej notácie list comprehension si vytvoríme zoznam *pritomnost*, ktorý obsahuje dvojice údajov hodina, zoznam prítomných v danej hodine pre každú hodinu v intervale *od* – *do*. Z tohto zoznamu vyberieme maximum podľa dĺžky (funkcia *len*) zoznamu prítomných v danej hodine (údaj na indexe 1 v dvojici hodina, zoznam prítomných).

```
def najviac_pritomnych(intervaly):
    od = zaciatok(intervaly)
    do = koniec(intervaly)
    pritomnost = [(h, pritomni(intervaly, h)) \
                          for h in range(od, do)]
    return max(pritomnost, key= lambda x: len(x[1]))
```

Príklad výpočtu:

```
>>> najviac_pritomnych(intervaly)
    (7, ['Adele', 'Meryl Streep', 'Leonardo Dicaprio', 'Justin Timberlake',
    'Peter Sagan'])
>>>
```

Výsledkom je dvojica (hodina, zoznam celebrít). Výsledok môžeme spracovať do slovnej odpovede napríklad takto:

```
vysledok = najviac_pritomnych(intervaly)

print(f'Najviac osobnosti bude prítomných o {vysledok[0]}. hodine.')
print(f'Prítomní budú {", ".join(vysledok[1])}.')
```

Výstup:

```
   Najviac osobnosti bude prítomných o 7. hodine.
   Prítomní budú Adele, Meryl Streep, Leonardo Dicaprio, Justin Timberlake,
   Peter Sagan.
>>>
```

2.4 ZHRNUTIE

Riešením úloh z podkapitoly 1.1 sme mali možnosť porovnať imperatívny a funkcionálny (deklaratívny) štýl programovania. Zopakujme si rozdiely:

- V imperatívnom programovaní je algoritmus vyjadrený ako postupnosť **príkazov** (čo sa má robiť), vo funkcionálnom programovaní ako **výraz** (čo má byť výsledkom).
- V imperatívnom programe sa **vykonávajú** príkazy, ktoré menia stav výpočtu, hodnoty premenných v čase. Vo funkcionálnom programe sa **vyhodnocujú** výrazy, hodnoty premenných sa len zisťujú, ale nemenia sa v čase.

Rozdiely dobre demonštruje dvojica funkcií *sort* a *sorted*. Použitie metódy *sort* je príkaz, ktorý usporiada daný zoznam – zmení jeho hodnotu. Jej názov *sort* vyjadruje príkaz, čo sa má urobiť: usporiadať. Použitie funkcie *sorted* je výraz, ktorý vracia usporiadaný zoznam. Jej názov *sorted* vyjadruje, čo je výsledok: usporiadaný. Pri vyhodnotení výrazu sa hodnoty premenných len vyhodnocujú, nemenia sa.

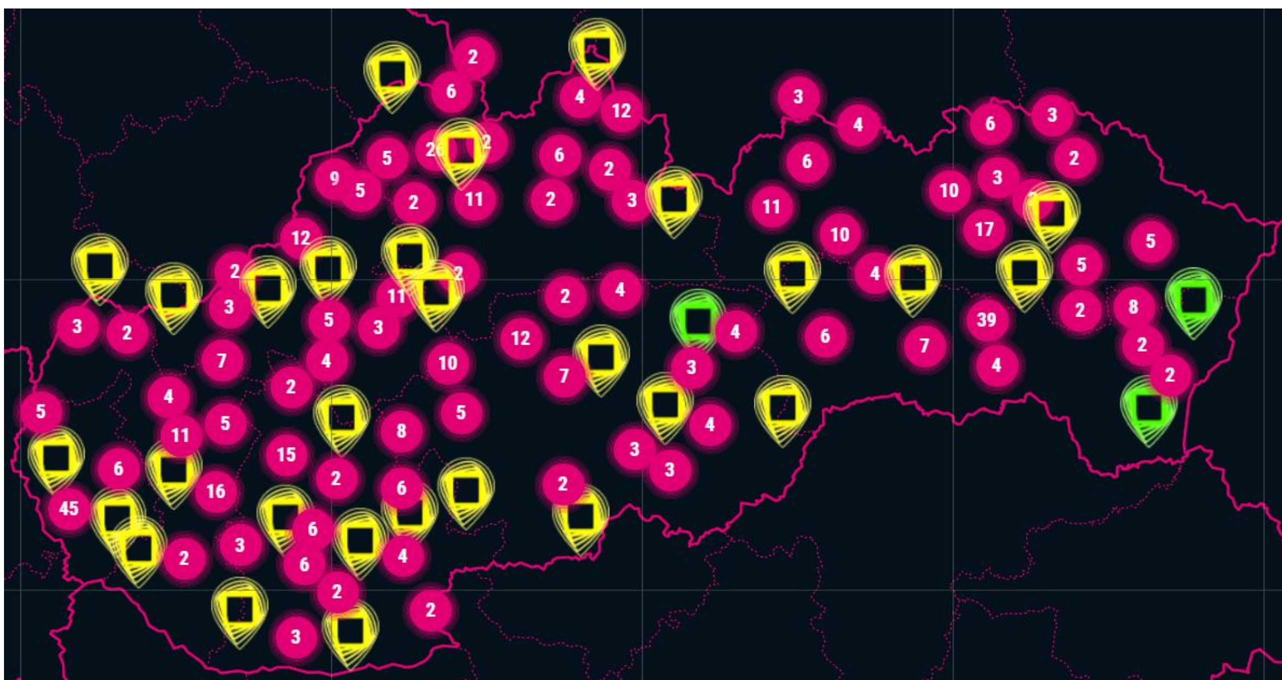
Príklad výpočtov:

```
>>> z = [25,456,3,14,9874,58,16,1]
>>> print(z)
    [25, 456, 3, 14, 9874, 58, 16, 1]
>>> print(sorted(z))
    [1, 3, 14, 16, 25, 58, 456, 9874]
>>> print(z)
    [25, 456, 3, 14, 9874, 58, 16, 1]
>>> print(z.sort())
    None
>>> print(z)
    [1, 3, 14, 16, 25, 58, 456, 9874]
>>>
```

V premennej *z* je na začiatku neusporiadaný zoznam čísel. Výsledkom výrazu *sorted(z)* je usporiadaný zoznam. Pri vyhodnocovaní výrazu sa hodnota premennej *z* nezmenila, len sa použila pri výpočte. Volanie *z.sort()* je príkaz, ktorý nevracia žiadnu hodnotu, ale usporiada daný zoznam, čím zmení hodnotu premennej *z*.

3 PROGRAMOVANIE HARDVÉRU BBC MICRO:BIT V JAZYKU MICROPYTHON

Programovanie v školách môže mať rôzne podoby, jednou z nich je aj programovanie reálnych zariadení – hardvéru. Na Slovensku sa často používa BBC micro:bit, ktorému sa budeme venovať v tejto kapitole. Vychádzame z údajov, ktoré môžete nájsť aj v tejto mape (Obrázok 3.1). Zahŕňa 556 základných alebo stredných škôl (žlté/zelené/číselné označenie), ktoré majú základné alebo pokročilé hardvérové sady BBC micro:bit. V mape je reálne zakreslený len jeden dvojročný projekt (Slovak Telekom, 2022), preto počet škôl môže byť aktuálne ešte vyšší. Možno však povedať, že v každom kraji sa nachádzajú desiatky škôl vyučujúcich s touto hardvérovou pomôckou.



Obrázok 3.1 BBC micro:bit v školách na Slovensku (Slovak Telekom, 2022)

BBC micro:bit bol prvýkrát predstavený vo Veľkej Británii v roku 2015 ako zariadenie pre edukačné účely. Výskum, bezplatný hardvér pre mnoho žiakov a enormný dosah tohto projektu vyústil do konceptov vyučovacích hodín prepojených s britským kurikulom (KS2, KS3, KS4) v predmete Computing (u nás preložené ako informatika).

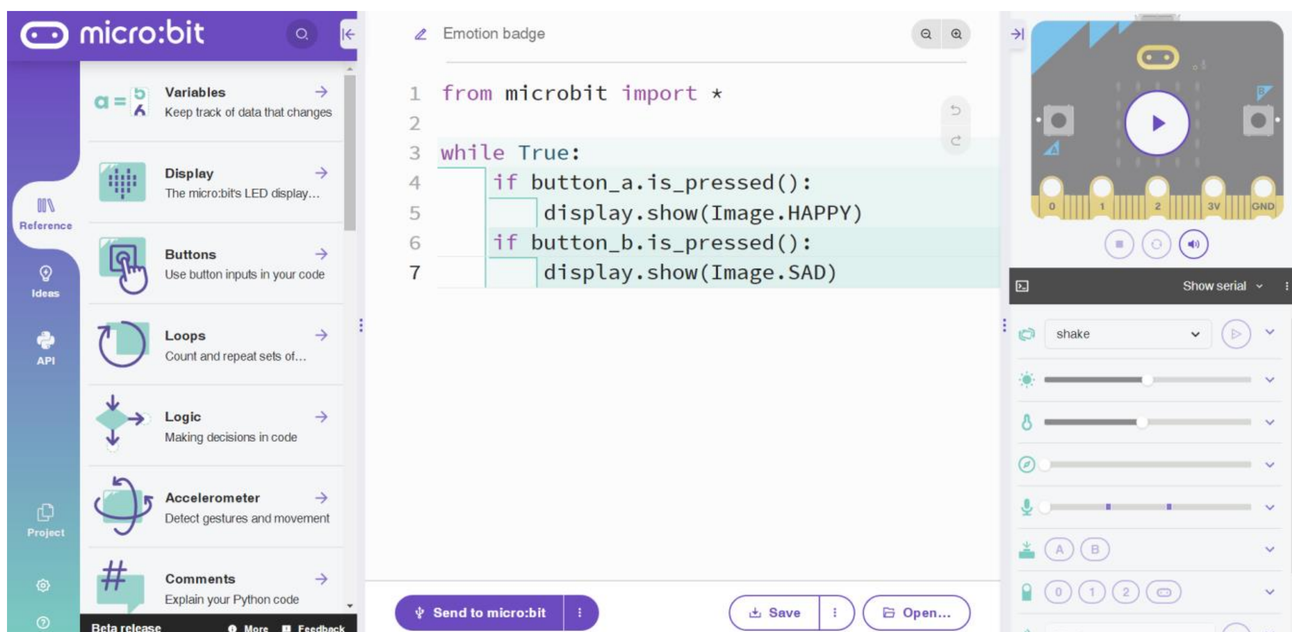
BBC micro:bit je postavený na prístupe, kedy poznanie si buduje žiak. Papert z Piagetovho konštruktivizmu odvodil konštrukcionizmus, kedy žiak je v roli konštruktéra/tvorcu a má k dispozícii hmatateľnú, reálnu, viditeľnú vec (Situating Constructionism, 1991) (Stiller, 2009). Okrem toho zariadenia, ktoré sa dajú chytiť, podporujú brikolážny prístup, ktorý podľa Levi-Straussa, Turkle, Paperta a Stillera predstavuje kontrast k analytickému prístupu. Ide o spôsob, kedy myšlienkový postup nezačína na axiómach, ale poznatky (konštrukty) sa budujú na základe usporiadania, preskupenia a skúmania už známych „vecí".

BBC micro:bit verzia 2 (pred rokom 2020 verzia 1.5) sa skladá z 5x5 červených LED diód na zobrazovanie obrázkov, textu a čísel a dá sa s nimi merať intenzita svetla dopadajúca na LED diódy. Okrem toho má 2 programovateľné tlačidlá, RESET tlačidlo na zadnej strane, anténu, procesor, žltú

a červenú indikačnú LED diódu, mikrofón, integrovaný reproduktor, kapacitné dotykové tlačidlo, micro USB konektor, konektor na batérie, kompas a akcelerometer. Následne sa tam nachádzajú vstupno/výstupné piny, ktoré sú programovateľné, príp. majú skratku GND (ZEM) a 3V (na napájanie externých senzorov).

Obrázok 3.2 BBC micro:bit (Micro:bit Educational Foundation, 2022)

Najčastejšie sa vyskytujúcou sadou je základná sada Učíme s Hardvérom, s ktorou budeme pracovať aj v tejto kapitole. Sada obsahuje 1x BBC micro:bit V2, 1x USB kábel, 1x držiak na batérie k micro:bitu, 2x AAA batéria, 10x krokosvorkový kábel, 1x LED pásik, 1x reproduktor, 5x LED diódy s rezistormi.



Obrázok 3.3 Online Python editor

V školách sa bežne využíva programovacie prostredie Microsoft MakeCode, ktoré je typické svojím blokovým programovaním. Ponúka však aj možnosť programovať textovo v jazyku MicroPython, ktorý je založený na programovacom jazyku Python 3 a určený špeciálne na programovanie mikrokontrolérov. Neodporúčame však používať MicroPython priamo v prostredí MakeCode, nakoľko v ňom nebudú fungovať MicroPython programy správne. Pre MicroPython odporúčame

nainštalovať si editor MU alebo online Python editor. Poďme sa pozrieť najskôr na online Python editor (Obrázok 3.3).

V ľavej časti okna sa nachádza bočné menu s niekoľkými kartami. Na karte Reference môžeme nájsť konkrétne príkazy aj s vysvetlením, čo vykonávajú. Na karte Ideas nájdeme nápady s kompletnými projektmi, v ktorých sa nachádza celý program, vysvetlivky k nemu, či námety na rozšírenia. Karta API (Application Programming Interface) poskytuje detailný prehľad príkazov jazyka MicroPython na prácu s micro:bitom. V strede okna sa nachádza editor kódu. Dole sa nachádzajú tlačidlá na odoslanie programu do micro:bita, uloženie do počítača či otvorenie zo súboru. Na pravej strane sa nachádza simulátor, kde sa dajú meniť parametre micro:bita a simulovať výpočet bez fyzického zariadenia.

3.1 WEARABLES – NOSITEĽNÁ ELEKTRONIKA

Wearables (zo slovesa to wear = nosiť) alebo nositeľná elektronika je technológia, ktorá sa nosí na tele alebo oblečení. Možno ju nosíte aj vy - môže ísť o smart hodinky, smart náramky a iné zariadenia. V tejto podkapitole si predstavíme aj rôzne špecifické komponenty určené pre nositeľnú elektroniku, v mnohých prípadoch si však viete pomôcť aj so základnou sadou Učíme s Hardvérom.

„Šaty s vlečkou, stříbrem vyšívané, ale princezna to není, jasný pane.“

Zamysleli ste sa niekedy nad tým, čo by sa stalo, keby bola Popoluška programátorkou? Ak mala šaty prešívané striebornou niťou, znamená to, že bola už len kúsok od nositeľnej elektroniky. Stačila by jej k tomu len baterka a nejaká LED dióda.

No a čo by sa stalo? Jej šaty mohli napríklad svietiť. Práve strieborná niť je často používaná v nositeľnej elektronike, pretože je elektricky vodivá. Keď sa zamyslíme nad nositeľnými technológiami vo všeobecnosti, zistíme, že sú tu s nami už dlho.

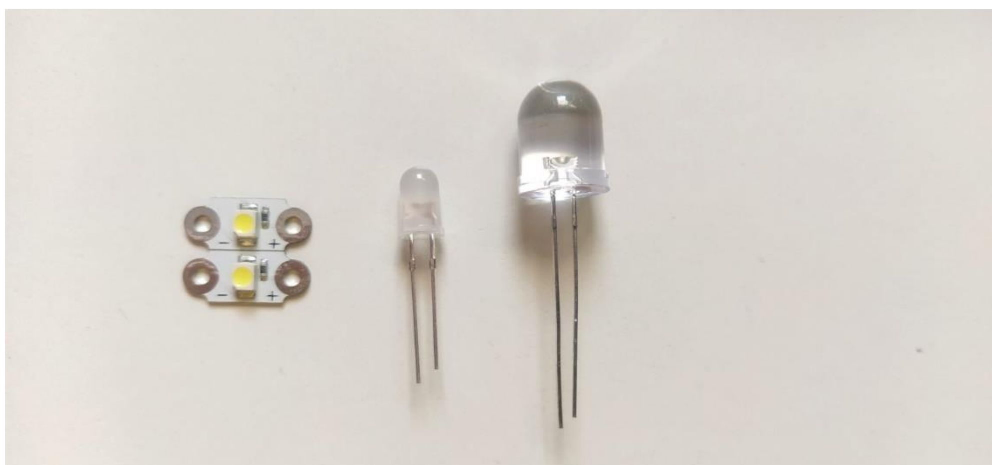
Prvými nositeľnými technológiami boli napríklad okuliare vynájdené v 13. storočí alebo abakus vo forme prsteňa nájdený v Číne. Nositeľné technológie sa využívajú aj u zvierat. Niekedy sa fotoaparáty upevňovali na bruška holubov, aby mohli fotiť terén z veľkej výšky. Dnes je najčastejšie využívanou nositeľnou elektronikou alebo technológiou napríklad mobilný telefón. Možno ale máte aj smart hodinky, fitness náramky alebo iné malé elektronické zariadenia, ktoré nosíte na sebe. Nositeľná elektronika má základný parameter, a to elektrický obvod. To znamená, že napríklad môžeme mať tričko, ktoré vďaka elektrickému obvodu môže svietiť.

Skúsme sa zamyslieť, či by sme mohli vytvoriť svietiace tričko iba so základnou sadou Učíme s Hadvérom. Budeme potrebovať tričko, krokosvorkové káble, LED diódy , rezistory, napájanie a ak chceme LED diódu naprogramovať, budeme potrebovať aj BBC microbit (iba pre svietenie postačí batéria bez micro:bita). Bolo by však takéto káblové tričko nositeľné? Asi áno, ale samotné nosenie by bolo veľmi náročné a nepríjemné. Preto existujú takzvané nositeľné komponenty, ktoré nahrádzajú bežne dostupnú elektroniku.

Najdôležitejšou je elektrovodivá niť. Je to alternatíva ku krokosvorkovým káblom. Takáto vodivá niť vyzerá ako bežná niť na šitie, avšak je možné, aby ňou prechádzal elektrický prúd. Môže ísť o striebornú niť alebo niť z nerezovej ocele. Takúto niť môžeme ohnúť alebo ju našiť na oblečenie, kde sa bude správať ako bežná niť s vlastnosťou navyše - bude ňou pretekať elektrický prúd. Inou alternatívou je medená samolepiaca páska či elektrovodivá farba. Všetky tieto veci fungujú na princípe, že obsahujú kovy ako striebro, oceľ, či meď, ktoré sú elektricky vodivé.

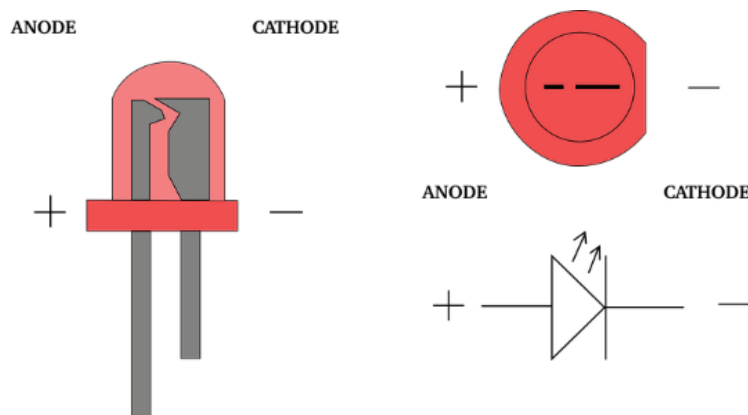
Obrázok 3.4 Elektrovodivá niť (SPy o.z., 2022)

Ďalšou častou súčiastkou je LED dióda. Môžeme používať rôzne LED diódy v závislosti od ich veľkosti. Existujú nositeľné LED diódy, ktoré sú určené na šitie a sú menšie, ich puzdro nie je také vypuklé, ako to býva pri bežných LED diódach. Pri týchto nositeľných LED diódach existuje aj ďalší rozdiel. Všimnite si, že tieto LEDky nemajú tzv. nožičky. Namiesto toho majú na oboch stranách elektrovodivé kruhovité útvary, ktoré sú pomenované plus a mínus. Plus je pre nás „dlhšia nožička“, čiže anóda a „kratšia nožička“, teda katóda, je označená mínus. Okrem týchto vlastností existuje ešte jedna, ktorá je veľmi dôležitá. Pri pozornom pohľade na LED diódu si môžete všimnúť, že sa tam nachádza ešte jedna súčiastka. Je ňou rezistor. Pri zapojení bežnej LED diódy zvyčajne potrebujeme rezistor, ktorý pridávame do elektrického obvodu. Pri nositeľnej LED dióde nepotrebujeme žiadne rezistory, pretože sú už zabudované, čiže integrované. Takýmto riešením dostávame veľkú výhodu – nemusíme prišívať zvlášť rezistor a LED diódu, prípadne ich spájkovať.



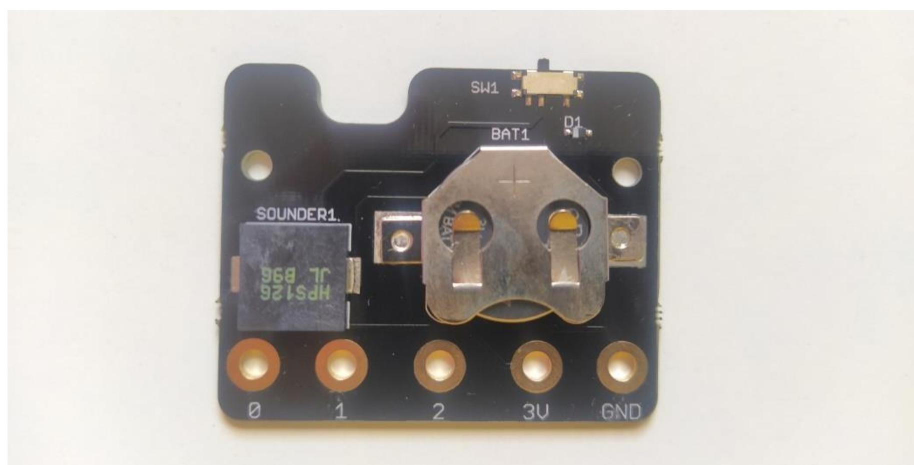
Obrázok 3.5 LED diódy (nositeľná LED dióda vľavo) (SPy o.z., 2022)

Ako sa zapájajú bežné LED diódy? Na nositeľných LED diódach máme označenie plus a mínus, plus pripájame ku kolíku 3V alebo k programovateľným kolíkom, napríklad PIN 0, 1 alebo 2. Naopak, mínus pripájame vždy ku kolíku s označením GND, čiže zem alebo ground. Takisto vieme, že rezistor alebo odpor je elektronická súčiastka, ktorá nám „ochraňuje“ LED diódu od toho, aby sa „vypálila“. Keďže sú v nositeľných LED diódach rezistory integrované, nie je potrebné pridávať ďalší rezistor, ani sa zaoberať hodnotou rezistoru, koľko ohmov má mať.



Obrázok 3.6 Bežná LED dióda (SPy o.z., 2022)

Pre elektrický obvod potrebujeme ešte zdroj napätia, napríklad batériu. Bežne dostupné AAA batérie sú pomerne veľké k nositeľnej elektronike a zároveň sú aj ťažké, a preto môžeme využívať napríklad gombíkovú teda okrúhlu plochu 3 voltovú batériu. Takáto batéria sa dá sa zasadiť do MI: power dosky/board. Doska sa pomocou elektrovodivých skrutiek a matiek pripojí ku kolíkom na micro:bite, čím sa kolíky prepoja . MI:power doska obsahuje reproduktor, vypínač, kolíky a držiak na okrúhlu batériu.



Obrázok 3.7 MI:power board (SPy o.z., 2022)

Existujú dve verzie MI:power dosky, jedna pre micro:bit verzie 1 a druhá pre micro:bit verzie 2. No jediným rozdielom medzi dvoma verziami MI: power doskami je, že verzia 2 má dlhšie skrutky a plastové oddeľovače. Doska sama o sebe nemá žiadne zmeny. Čiže ak si kúpite dlhšie skrutky v nejakom obchode s elektronickými súčiastkami, aj doska MI: power verzie 1 bude kompatibilná s micro:bit verziou 2.

Poďme si MI:power dosku pripojiť k microbitu. Ako prvé je potrebné vložiť 3V batériu, keďže vkladať a vyberať batériu je možné, iba keď nie je doska priskrutkovaná k micro:bitu. V balíku nájdeme tri skrutky, tri matky a tri plastové oddeľovače. Skrutky najskôr zasunieme do kolíkov zem, 3 volty, a kolík nula. Prečo kolík nula? Pretože reproduktor na MI:power doske pripájame ku koliku 0 na micro:bite. Teraz na skrutky nasadíme plastové oddeľovače a pridáme dosku. Pripojenie zakončíme matičkami, ktoré musia dobre doliehať, aby boli všetky kolíky prepojené.

V tejto podkapitole ste sa dozvedeli o základných komponentoch pre nositeľnú elektroniku. Takýchto súčiastok existuje viacero, tie špecifické si budeme ukazovať pri jednotlivých projektoch.

3.2 NOSITEĽNÉ TRIČKO

Na jednoduchší nositeľný projekt nám stačí skutočne len pár súčiastok:

- zdroj napätia, teda batéria,
- elektrický obvod
- elektronické súčiastky, napríklad LED diódy.

Pokiaľ chceme nositeľný projekt programovať, budeme potrebovať pripojiť BBC micro:bit. Poďme si to v tejto lekcii vyskúšať. Najčastejší projekt, ktorý sa spája s nositeľnou elektronikou, je nepochybne svietiace tričko. Tento projekt by sme však mohli prepojiť s biológiou, aby sme zahrnuli medzipredmetové vzťahy. Poznáte zviera, ktoré má krídla, lieta v noci a so sebou si nosí „svietiaci lampášik“? Je ňou svetluška svätojánska, ľudovo nazývaná aj svätojánska muška.

Skúsme si objasniť pár zaujímavostí. Svetluška svätojánska sa nazýva preto, lebo ju zvyčajne vidieť na sviatok sv. Jána, 24. júna. Svetlušky na našom území svietia žltou farbou, v iných častiach Zeme ich však možno nájsť svietiť aj modrou či červenou farbou. Obsahujú v sebe obsahujú luciferín, čiže chemickú látku, ktorá pri kontakte s kyslíkom vo vzduchu vyžaruje svetlo, a preto svetlušky svietia.

Spojme svetlušku a nositeľnú elektroniku, pričom úloha by mohla byť koncipovaná nasledovane: Vytvorte tričko s nositeľnou elektronikou, kde sa bude nachádzať svetluška, ktorej „lampášik“ bude svietiť po tom, čo zatlieskame.

Na projekt potrebujeme:

- BBC micro.bit,
- MI:powerboard,
- elektrovodivú niť,
- ihlu,
- LED diódy,
- tričko,
- krokosvorky.

Krokosvorky, ktoré budeme používať, nie sú prepojené vodivým drôtom. Budeme potrebovať aj chlpatý drôtik, ktorý nájdeme v papiernictve, ale aj textíliu, z ktorej bude vytvorená svätojánska muška. My budeme používať plošne netkanú textíliu alebo plstené vlákenné rúno, ktoré isto poznáte pod názvom plsť. Ako dekoráciu si môžeme pripraviť aj trblietavé lepidlo, či pohyblivé očká.

Najskôr si poďme vytvoriť elektrický obvod, ktorý našijeme na tričko. Zoberieme si MI:powerboard, vložíme batériu a spojíme s micro:bitom pomocou skrutiek. Micro:bit musí byť pevne zaskrutkovaný, pretože skrutky a matice predstavujú uzatvorený elektrický obvod medzi batériou a micro:bitom. Následne priložíme svorky ku kolíku GND a ľubovoľnému programovateľnému pinu, napríklad 1. Používame to na to , aby sme micro:bit mohli kedykoľvek odpojiť z elektrického obvodu. Micro:bit teda nebude nikdy prišitý. Tričko prevrátime naruby, teda vnútornou stranou, kde priložíme micro:bit spolu so svorkami. Pozor, svorky zapájame k micro:bitu, ktorý umiestňujeme smerom dopredu.

Následne začíname šiť. Odstrihneme si asi 15 cm vodivej nite, prevlečieme cez ihlu a na konci spravíme uzlík. Začíname od svorky, ktorá je upevnená ku GND. Keďže nechceme, aby nám svorka vypadla z trička, prišívame ju viacerými ťahmi. Pri svorke sa nachádza krúžok, ktorým niť aspoň dvakrát obídeme. Následne šijeme predným stehom. Poznáte predný steh? Je to jeden z tých

základných stehov (Obrázok 3.8). Vytvoríme ho tak, že šijeme akúsi rovnú priamku, pričom ihlu zapichujeme raz smerom hore, potom smerom dole. Snažíme sa o to, aby elektrický obvod nebol príliš veľký, pretože ho nemá byť vidno. Teraz ešte prišijeme LED diódu. Prišívame ju na líc, to znamená časť trička, ktorú bude vidieť. V našom prípade to znamená, že svorku na micro:bit sme prišili na vnútornú stranu trička a LED diódu prišívame na vonkajšiu. Keď šijeme od pinu GND, tak prišívame niť k mínusu a následne šitie zakončíme uzlíkom a zvyšnú niť odstrihneme.

Obrázok 3.8 Predný steh elektrovodivou niťou (SPy o.z., 2022)

Novú niť začíname s uzlíkom pri svorke micro:bitu, ktorá je pripojená na pin 1 a predným stehom pokračujeme k LED dióde k plusu. Zakončíme uzlíkom. Dávame si pozor na to, aby sa nám nite neprekrývali. Rozdiel medzi klasickým krokosvorkovým káblom a niťou je ten, že niť nie je izolovaná, a môže dôjsť ku skratu.

Keď máme elektrické obvody našité, pripojíme micro:bit k počítaču cez USB kábel. Môžeme naprogramovať, aby LED dióda svietila vtedy, keď zatlieskame. No túto úlohu môžeme robiť len s micro:bitom verzie 2, pretože obsahuje mikrofón.

Poďme teraz programovať v programovacom jazyku MicroPython. Môžete na to využiť MU editor, alebo online Python editor. Najskôr si do micro:bita naimportujeme potrebné príkazy z knižnice microbit, teda pin1, sleep, microphone aj SoundEvent. Pokiaľ platí, že mikrofón deteguje udalosť hlasný zvuk, do kolíka 1 sa zapíše digitálna jednotka, čo znamená, že sa LED dióda rozsvieti. Následne sa 5000 milisekúnd čaká. A potom sa opäť do kolíka 1 zapíše digitálna nula, a teda LED dióda zhasne. Táto udalosť sa vykonáva v cykle while.

```
from microbit import pin1, sleep, microphone, SoundEvent

while True:
    if microphone.current_event() == SoundEvent.LOUD:
        pin1.write_digital(1)
        sleep(5000)
        pin1.write_digital(0)
```

Existuje alternatíva, že do knižnice naimportujeme príkazy cez *from microbit import **. Hviezdička v tomto prípade znamená všetky príkazy, avšak takáto možnosť môže pri zložitejších programoch spôsobiť problémy, ak sa vo viacerých knižniciach nachádzajú rovnaké identifikátory (napr. príkazy).

Teraz potrebujeme vystrihnúť z plste telo svetlušky. Vyberieme si čiernu plsť a vytvoríme si prvý pár krídel, tzv. krovky. Krídla majú byť symetrické, preto si ich vystrihneme spolu. Pred strihaním si môžeme krídla naznačiť. Môžeme ísť pripevniť alebo prišiť na tričko. V našom prípade použijeme obojstrannú lepiacu pásku. Ďalšou časťou je lampášik. Keďže chceme svetlo zvýrazniť, použijeme na

to žltú plsť. Opäť si ju môžeme naznačiť a neskôr vystrihnúť. Lampášik sa čiastočne skryje pod čiernu hruď, preto je dôležité, aby časti, ktoré budeme spájať, boli rovnako široké. Okrem toho si musíme pamätať, že pod lampášikom je LED dióda, ktorú nesmie byť vidno vrátane elektrického obvodu z nití. Svetluška má dva páry krídel, a preto si vytvoríme z chlpatého drôtu ďalšie dve krídla, ktoré taktiež nalepíme. Na to pridávame telo svetlušky, na ktorú sme nalepili tykadlá z chlpatého drôtu. Posledným krokom je nalepenie pohyblivých očí a k tomu môžeme na tričko nalepiť žlté trblietky okolo lampášika. Svietiaca svetluška je na svete!

Alternatívou k tomuto projektu môžeme byť aj „pískacie“ tričko, kedy sa po stlačení tlačidla prehrá nejaký zvuk z integrovaného reproduktora v micro:bite.

Obrázok 3.9 Finálny produkt (SPy o.z., 2022)

3.3 ODZNAK

Ďalší projektom je odznak. Ten nosí mnoho ľudí. Deti ich nosia na taškách či tričkách najmä kvôli radosti. Vojaci, policajti a hasiči ich nosia kvôli tomu, aby navzájom videli, akú hodnosť majú. Odznak o nás môže povedať veľa. Možno sa z neho dozvedieť, akú hudbu máme radi, kto je náš obľúbený idol, či dokonca, aké je naše hobby! Poďme si v tejto podkapitole jeden odznak vytvoriť.

Na tento projekt potrebujeme:

- plste: biela, modrá, fialová (tmavomodrá),
- BBC micro:bit,
- USB kábel,
- elektrovodivá niť,
- ihla,
- LED dióda,
- MI:power board,
- MI:pro ochranný obal,
- zatvárací špendlík (ľudovo zicherka),
- obyčajná niť,
- kombinované kliešte.

Najskôr sa pozrime na logo, ktoré chceme vytvoriť – pokojne buďte tvoriví a vymyslite si vlastné logo. Ako príklad budeme pracovať s logom Učíme s Hardvérom. Čo je v ňom? Vidíme tam elektrický obvod, ktorý má v strede značku diódy. Čo keby sme diódu nahradili LED diódou, ktorú už dobre poznáme? Skúsme náš projekt opäť prepojiť s inou témou, napr. Morseovou abecedou. Potom môžeme vytvoriť napríklad takýto projekt: Bude to odznak Učíme s Hardvérom s LED diódou, ktorý bude Morseovou abecedou vysielať nejakú informáciu. Takýto svietiaci odznak bude neprehliadnuteľný.

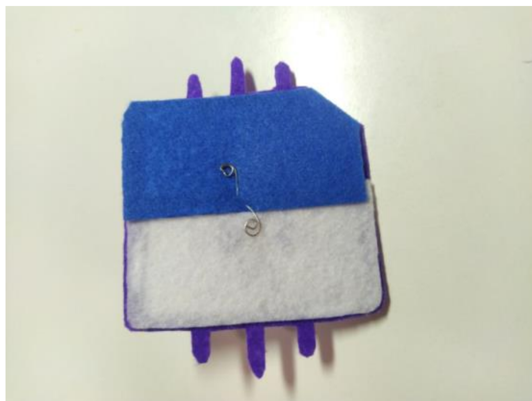
Obrázok 3.10 Práca s logom

Najskôr si vytvoríme nižšiu vrstvu loga, ktorá zahŕňa bielu a modrú farbu. Zoberieme si bielu a modrú plsť, z ktorej vystrihneme základné tvary podobajúce sa obdĺžnikom. Musíme však dbať na to, že pri bielej plsti je biela výplň aj v značke LED diódy. Snažme sa plste vystrihnúť tak, aby sme jeden okraj mali dlhší a obe plste sme mohli jednoducho zlepiť. Tie zlepíme v tomto poradí: Modrá plsť je spodok, na ňu sa čiastočne priliepa biela plsť a všetko ukončíme fialovou plsťou vo forme elektrického obvodu.

Keďže chceme LED diódou komunikovať Morseovou abecedou, je potrebné, aby bola dostatočne viditeľná. Preto nepoužijeme LED diódu určenú na nositeľnú elektroniku, ale zapojíme si bežnú LED diódu s nožičkami. LED diódu v strede odznaku prepichneme nožičkami, avšak nesmú sa dotýkať!

Kombinovanými kliešťami potrebujeme z nožičiek vytvoriť kruhy, pretože ich budeme našívať a niť z nich nesmie vypadnúť.

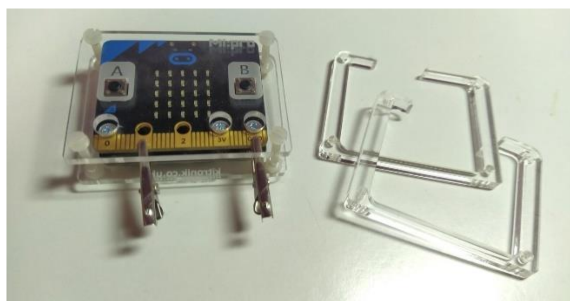
Poďme si to vyskúšať. Ubezpečíme sa, ktorá nožička je dlhšia a zapamätáme si ju, prípadne si ju môžeme nejako označiť. Spravíme si väčší kruh pre plus a menší kruh pre mínus. Zistíme si, či LED dióda potrebuje rezistor a v prípade potreby ju pripojíme k LED dióde pomocou kombinovaných kliešťov tak, aby koniec jednej nožičky bol kruhovitý. Takáto LED dióda potrebuje samostatný rezistor, ktorý si môžeme pridať akýmkoľvek spôsobom. Môžeme si rezistorové nožičky zahnúť a pripojiť k dlhšej nožičke LED diódy.



Obrázok 3.11 Práca s LED nožičkami (SPy o.z., 2022)

Ochranný obal MI:power má dva účely: chráni micro:bit, ktorý už má zapojený MI:power board a predná časť micro:bita sa stáva plochou, pričom tlačidlá sú stále stlačiteľné. To znamená, že micro:bit sa stane základom pre textilný odznak, ktorý nebude skrčený a vyčnievajúce tlačidlá nebudú tvoriť škaredý dojem, pretože odznak bude na rovnom podklade. Tento ochranný obal sa skladá zo štyroch častí. My potrebujeme len prvú a poslednú, ktoré priskrutkujeme.

Teraz potrebujeme pripájať krokosvorky ku kolíkom, a to by bolo pomerne náročné. Preto z powerboard vyberieme dve vnútorné plastové časti, ktoré sa nachádzajú na kolíkoch GND, teda zem a kolík 0. Krokosvorky budeme pripájať k skrutkám, pretože sú vodivé.



Obrázok 3.12 Práca s krokosvorkami (SPy o.z., 2022)

Následne použijeme elektrovodivú niť, ktorú uchytíme na koncoch kratšej nožičky a pripojíme k zemi (GND). Dlhšiu nožičku, anódu, pripojíme k pinu 0. Pokiaľ to nie je nutné, stačí, aby na oboch koncoch boli uzlíky a nemusíme šiť. Dbajme na to, aby bola niť dostatočne krátka na to, aby ju pri nosení nebolo vidno. Odznak z plste prilepíme na predný obal micro:bitu.

Odznak potrebuje mať niečo, čím si ho na seba prichytíme. Môžeme na to využiť zatvárací špendlík a obyčajnú niť. Cez obal niekoľkokrát prevlečieme niť na oboch stranách a následne ich zopneme zatváracím špendlíkom.

Teraz môžeme začať programovať. Využiť na to môžeme už spomínanú Morseovu abecedu ako šifrovací jazyk. Vyskúšajme Morseovu abecedu, kde sa na LED dióde bude zobrazovať napr. slovo SPy. Pre začiatok je dobré využiť nejaké krátke slovo.

Určite si vie každý nájsť na internete Morseovu abecedu, v našom prípade to znamená, že písmeno s má tri bodky, písmeno p má bodku čiarku bodku, a písmeno ypsilon má čiarku bodku čiarku. Abeceda nerozlišuje veľké a malé písmená. Dohodnime sa, že si ešte pridáme dlhšiu pauzu medzi jednotlivými písmenami, aby sme videli, kde písmeno začína a kde končí. Pozrime sa na zápis:

S …

P .-.

Y -.-

Poďme si to naprogramovať. Používať budeme len tri základné príkazy:

```
    pin0.write_digital(1)
    sleep(300)
    pin0.write_digital(0)
```

Pri stlačení tlačidla A sa stlačí, pozastaví a digitálne zapíše kolík. Nezabudnime ešte použiť dlhšiu pauzu, aby sme vedeli, že skončil jeden znak (písmeno) a nasleduje ďalší.

Ako naprogramovať bodku a čiarku? Bodka znamená, že zvuk alebo svetlo svieti kratšie. Pre nás to môže byť napríklad 300 ms, ktoré dopíšeme (nevyberáme z možností). Pri čiarke musí svetlo svietiť podstatne dlhšie, napríklad 1000 ms (1 sekunda). Každú pauzu si môžeme nastaviť napr. na 500 ms. Skúste si to sami a zistite, koľkokrát sa v programe opakujú vzorce (patterns). Je ich viac ako 40? Skúsme sa teda zamyslieť, ako by sme mohli v programe zjednodušiť zápis.

Porozmýšľajme nad programom najskôr všeobecne. Ak si chceme uľahčiť písanie programu, je možné vytvárať tzv. funkcie. Skúsme si vytvoriť funkciu *bodka*. Následne si vytvorme ďalšiu funkciu *ciarka* a vložme 4 príkazy pre čiarku. Treťou funkciou bude *koniec_pismena*, kde vložíme dlhšiu pauzu. Posledná funkcia nám počtovo nezlepší program, ale v konečnom výsledku bude program lepšie čitateľný.

Hlavný program bude fungovať tak, že pri stlačení tlačidla A budeme volať *bodka* v hlavnom programe, čím sa vykonajú všetky príkazy pre bodku. To isté platí pre *ciarka* a *koniec_pismena*. Hlavný program je takto čitateľnejší a keď chceme program zmeniť, stačí zmeniť príkaz vo funkcii, čím sa akoby prepíše celý hlavný program.

Ďalším zefektívnením je pozrieť sa na program a zistiť, čo sa opakuje. Pre písmeno S sú to tri za sebou idúce bodky. Je zbytočné volať bodka, bodka, bodka, keď môžeme zavolať bodku trikrát cez opakovať 3-krát vykonať. Takýto program je stále ľahko čitateľný a oveľa kratší ako náš pôvodný.

```
from microbit import pin0, sleep, button_a

def bodka():
    pin0.write_digital(1)
    sleep(300)
    pin0.write_digital(0)
    sleep(300)
```

```
def ciarka():
    pin0.write_digital(1)
    sleep(1000)
    pin0.write_digital(0)
    sleep(300)

def koniec_pismena():
    sleep(500)

while True:
    if button_a.is_pressed():
        for i in range(3):
           bodka()
        koniec_pismena()
        ciarka()
        ciarka()
        bodka()
        koniec_pismena()
        ciarka()
        bodka()
        ciarka()
        ciarka()
```

Všimnime si, že pri volaniach funkcií sú prázdne zátvorky. Je to z toho dôvodu, že nemajú parametre. V prípade opakujúceho sa vzoru (tri bodky) môžeme použiť cyklus *for*.

V tejto podkapitole sme sa zaoberali najmä cyklom *while*, *for*, udalosťou stlačenie tlačidla a funkciami.

3.4 VALENTÍNSKY ODKAZ

Už ste niekedy písali alebo dostali pohľadnicu? Určite áno. Vraj keď niečo darujeme, tešíme sa z toho ešte viac, akoby sme niečo dostali. V tejto podkapitole si vytvoríme valentínsky odkaz, ktorý môžeme darovať mame, otcovi, starým rodičom či našim blízkym, ktorých máme radi. Každý má rád príjemné prekvapenia, preto ich skúsme prekvapiť nielen odkazom, ale aj tým, že odkaz bude v micro:bite, ktorý sa bude zapínať LEN vtedy, keď otvoríme pohľadnicu.

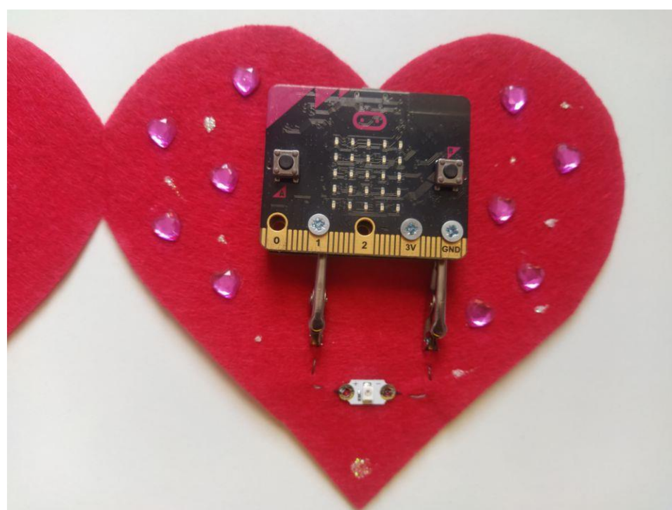
Isto ste už podobnú pohľadnicu alebo oznámenie videli. Keď je zatvorená, nič sa nedeje, keď sa otvorí, začne hrať napríklad narodeninová hudba. V tomto prípade ide o 2 plôšky, ktoré sa pri otvorení prepoja a vznikne uzatvorený elektrický obvod. Hudba sa začne prehrávať. V našom prípade nebudeme spájať plôšky, ale merať úroveň osvetlenia.

Budeme potrebovať:

- BBC micro:bit,
- USB kábel,
- MI:power board,
- nositeľné LED diódy,
- ihla,
- elektrovodivá niť,
- plsť,
- trblietavé lepidlo,
- nožnice.

Odporúčame si pohľadnicu najskôr vystrihnúť. Podstatné je, aby sa dokázala otvoriť, a teda plsť preložíme napoly a vystrihneme požadovaný tvar. Pohľadnica by mala spĺňať takéto pravidlá:

- dá sa zatvoriť,
- oba geometrické útvary sú v jednej časti spojené,
- oba geometrické útvary sú rovnako veľké.



Obrázok 3.13 Finálny produkt (SPy o.z., 2022)

Nemusí ísť vždy len o srdce. Môže to byť obdĺžnik, štvorec, štvorlístok, či iné tvary, ktoré sú vhodné napríklad na iné sviatky a poskytujú príležitosť využiť medzipredmetové vzťahy. Možno sa na anglickom jazyku učia deti o sviatku svätého Patrika, ktorý je 17. marca a základným symbolom je práve štvorlístok, ktorý predstavuje šťastie.

K micro:bitu pripojíme MI:power board s batériou. Pozor! Krokosvorky pripájame k MI:power board a pripojené piny musia obsahovať skrutky a matice. To znamená, že na doske bude pretekať elektrický prúd len v tých pinoch, ktoré sú prepojené s micro:bitom. Krokosvorky slúžia k tomu, aby sme si v prípade potreby mohli micro:bit vybrať a opätovne ho použiť. Keby sme si našili priamo piny, v prípade odobratia micro:bita by sme museli porušiť celý elektrický obvod. LED diódu zapájame tak, že šijeme z krokosvorky GND do mínusovej časti LED-ky a ukončíme šitie uzlíkom. Nové šitie začíname v krokosvorke pinu 1, pokračujeme predným stehom a ukončíme ho uzlíkom v plusovej častí LED diódy.

Keď to máme, môžeme začať programovať. Našou úlohou je, aby micro:bit vykonával niečo len vtedy, keď sa pohľadnica otvorí. Musíme teda merať intenzitu osvetlenia. Keďže nemáme takýto senzor, použijeme LED displej zabudovaný do micro:bita, ktorý meria dopadajúce svetlo na LED displej. Hodnota osvetlenia sa pohybuje od 0 do 255, pričom 0 je úplná tma a 255 najjasnejšie svetlo. Kedy budeme merať úroveň osvetlenia? Stále, pretože nevieme, kedy pohľadnicu otvoríme. Použijeme na to cyklus *while*.

Tento projekt môžeme využiť aj na to, aby sme si naprogramovali búšenie srdca. Viete nájsť nejaký opakujúci sa vzor pri búšení srdca? Je to bum bum pauza bum bum pauza a tak ďalej. To, čo sa opakuje, je: bum bum pauza.

Pri programe budeme vychádzať z cyklu *while*, v ktorom budeme merať hodnotu osvetlenia. Ak je väčšia alebo rovná 20 (môžete si číslo upraviť), bude niečo vykonávať. Využijeme na to *display.read_light_level()*. Následne programujeme búšenie srdca. Samotné bum bude znamenať, že sa nositeľná LED dióda rozsvieti (digitálne zapíšeme 1 do pinu 1), následne dáme pauzu (napr. 150 ms), vypnutie LED diódy naprogramujeme cez digitálne zapísanie 0 do pinu 1 a opäť dáme pauzu. Máme teda 4 príkazy. Lenže my potrebujeme dvakrát bum-bum, čiže 4 príkazy sa musia opakovať dvakrát. Na to využijeme cyklus *for*. Čo sa stane potom? Aj bum-bum, bum-bum, bum-bum sa opakuje, ale medzi bum-bum, bum-bum prebieha dlhšia pauza. Na to využijeme cyklus v cykle, pričom každý cyklus je *for*. Do tohto napíšeme iba *sleep* s dlhšou pauzou a rozhodneme sa, koľkokrát sa má bum-bum zobraziť. Ak je cyklus *for* s opakovaním 3 a v ňom cyklus *for* s opakovaním 2, spolu ide o 6-krát bum.

Ak chceme program ešte vylepšiť, môžeme si na displej zobraziť nejaký pekný text. Nezabudnime potom na import, v tomto prípade *display*, *sleep* a *pin1*.

```
from microbit import display, sleep, pin1

while True:
   if  display.read_light_level() >= 20:
       for i in range(3):
           for i in range(2):
               pin1.write_digital(1)
               sleep(150)
               pin1.write_digital(0)
               sleep(150)
           sleep(200)
   display.scroll('Si naj!')
```

Skúsme si to zhrnúť. Program v pohľadnici sa vykonáva stále. Podstatné je, že LED dióda začne blikať na spôsob búšenia srdca a zobrazí odkaz len vtedy, keď otvoríme pohľadnicu. Inak sa nevykonáva žiadny príkaz.

V tejto podkapitole sme sa naučili vytvoriť si pohľadnicu s micro:bitom. Použili sme pri tom nositeľnú LED diódu, v ktorej je zabudovaný rezistor a naprogramovali program tak, aby pohľadnica po otvorení vykonávala dané príkazy. Pracovali sme aj s úrovňou osvetlenia, digitálnym zapisovaním, podmienkou, ale aj cyklom v cykle. Na ktorých iných predmetoch dokážeme tento projekt využiť?

3.5 NOČNÁ OBLOHA

Isto ste sa niekedy pozerali na nočnú oblohu, možno dokonca viete pomenovať rôzne súhvezdia. Poďme si vytvoriť nočnú oblohu, ktorá bude svietiť aj v izbe, napríklad s pomocou micro:bitu a nositeľných LED diód. Je len na vás, aké súhvezdie si vyberiete. V tejto podkapitole si môžeme vybrať súhvezdie Váhy.

Potrebujeme:

- čierna (alebo tmavomodrá) plsť,
- BBC micro:bit,
- USB kábel,
- elektrovodivá niť,
- ihla,
- 6 kusov LED diód (počet kusov závisí od počtu hviezd),
- trblietavé lepidlo,
- kombinované kliešte,
- biela niť.

A4 plsť prestrihneme na polovicu, pretože budeme potrebovať vytvoriť vonkajšiu a vnútornú stranu projektu. Najskôr pracujeme s vnútornou stranou. Po tom, ako sme si vybrali súhvezdie, si označíme jednotlivé "hviezdy", teda umiestnenia LED diód. Môžeme si pri tom pomôcť vytlačením alebo nakreslením súhvezdia, vytlačený papier položíme na plsť a ihlou urobíme diery tam, kde chceme mať LED diódy.

Potrebujeme zapojiť viacero LED diód, ktoré budú pripojené k jednému micro:bitu na spoločný programovateľný pin. To dosiahneme prostredníctvom paralelného pripojenia, t.j. pod sebou. Dajme pozor na to, aby sme rozpoznali katódy od anód. LED diódy naukladáme na vnútornú stranu tak, aby sme dokázali jedným predným ťahom zapojiť elektrovodivou niťou katódy, teda mínusy a ďalším jedným ťahom anódy, čiže plusy, pričom sa jednotlivé ťahy nebudú prekrývať, aby nedošlo k skratu.

Obrázok 3.14 Práca s kombinovanými kliešťami (SPy o.z., 2022)

Na ihlu si navlečieme niť a rozdelíme na polovicu. Micro:bit si tentoraz nebudeme pripájať pomocou krokosvoriek, ale uzlíkov. Na konci budeme mať oba kraje nite a uzlík spravíme až o 5 - 6 cm. Predným stehom si prišijeme všetky LED diódy, najskôr mínusové kruhy, ukončíme steh a potom urobíme to isté s plusovými časťami. Keď chceme pridať micro:bit, stačí urobiť uzlík z vyčnievajúcich koncov nití. Uzlík prichytíme o kolíky micro:bita.

Obrázok 3.15 Kostra projektu (SPy o.z., 2022)

Zamysleli ste sa nad tým, prečo hviezdy blikajú? Skúste vyjsť večer na vyššie miesto, napríklad kopec alebo rozhľadňu a všimnite si mesto. Nielen hviezdy, ale aj osvetlenie v meste bude blikať. Je to spôsobené pohybom vzduchu a putovaním svetla. Keďže sa vzduch pohybuje a má rôznu teplotu, putovanie svetla sa mihoce. Takýto jav sa nazýva scintilácia a my si ju môžeme simulovať aj prostredníctvom programu.

Blikajú hviezdy pravidelne? Určite nie. Môžeme povedať, že blikajú náhodne, teda určitý čas svietia a zrazu na malú chvíľu zhasnú, opäť sa rozsvietia a znova svietia nejaký čas. Musíme si teda naprogramovať náhodný čas svietenia a následne rýchle bliknutie, čo môžeme nastaviť napr. na 200 ms.

Najskôr si potrebujeme inicializovať premennú. Pomenujme si ju *cas* a priradíme jej 0, aby vždy po zapnutí/reštarte bola počítaná od 0. Poďme nastaviť náhodný počet sekúnd od 1 do 10, musíme ich však previesť do milisekúnd.

1 sekunda = 1000 milisekúnd

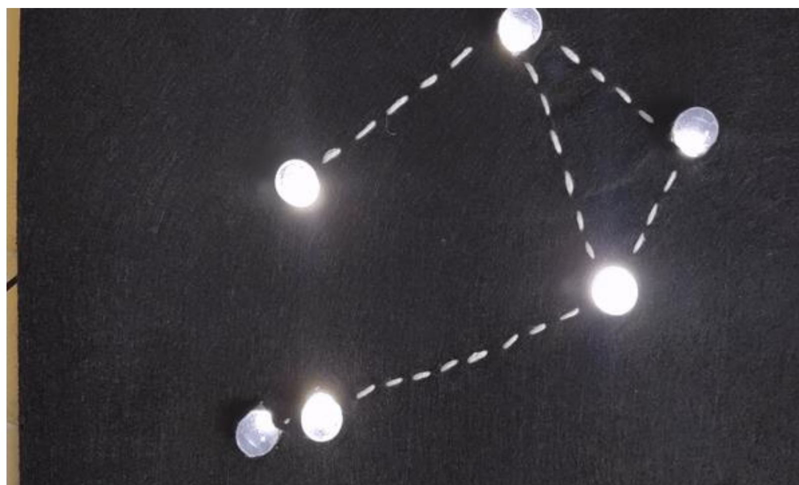
10 sekúnd = 10 000 milisekúnd

Chceme, aby sa LED diódy rozsvecovali viackrát, preto použijeme cyklus while. V ňom budeme ukladať do premennej cas náhodne generované hodnoty medzi 1000 až 10 000 milisekúnd. LED diódu zapneme na pine 0 prostredníctvom príkazu pin0.write_digital(1). Následne bude dióda svietiť (čakať) náhodný čas, ktorý už je vygenerovaný: sleep(cas). Potom sa LED dióda vypne prostredníctvom pin0.write_digital(0) a aby sme vypnutie uvideli, dáme tam opäť krátku pauzu, napr. sleep(200).

```
from microbit import sleep, pin0
import random

cas = 0

while True:
    cas = random.randint(1000, 10000)
    pin0.write_digital(1)
    sleep(cas)
    pin0.write_digital(0)
    sleep(200)
```

Ešte je potrebné dokončiť projekt po estetickej stránke. Zvyšná strana plste bude slúžiť ako vonkajšia. Vzájomne ich prekryjeme a tam, kde sa nachádzajú LED diódy, vystrihneme otvory, aby sme mohli obe plste spojiť, čím budú LED diódy viditeľné. Keď vidíme umiestnenia LED-iek, obyčajnou bielou niťou spojíme na vonkajšej strane jednotlivé LED-ky, aby vytvárali želané súhvezdie.

Potom spojíme obe plste a je na nás, či si nočnú oblohu vytvoríme do tvaru obrazu a zavesíme na stenu, alebo si ju prišijeme na oblečenie.



Obrázok 3.16 Finálny produkt

3.6 ČELENKA

Pri čelenke budeme využívať neopixel LED pásik, ktorý sme v predchádzajúcich projektoch nevyužívali.

Budeme na to potrebovať:

- čelenka,
- BBC micro:bit,
- USB kábel,
- LED pásik,
- batéria,
- dekorácie:
  - kvietky, trblietavé lepidlá, chlpatý drôt, ligotavé kruhy.

Mnoho žiakov zaujme práve neopixel LED pásik. Keď sa naň pozrieme bližšie, zistíme, že sa skladá z LED diód. Tento pásik má však jednu veľkú výhodu - v každej LED dióde sa nachádza malý mikroprocesor, ktorý samostatne nastavuje farbu na RGB LED dióde. Nazývame to, že LED diódy sú na pásiku individuálne adresovateľné, teda každá môže svietiť v rovnakom čase inou farbou, príp. nesvietiť - každá je úplne nezávislá. Samozrejme, každá má aj vlastný rezistor, ktorý limituje množstvo prúdu a tým ochráni LED diódu od vypálenia.

Najskôr potrebujeme spojiť neopixel pásik s čelenkou, máme dve možnosti:

- Lepiacou páskou uchytíme pásik k čelenke.
- Zoberieme si chlpatý drôt, pásik priložíme k čelenke a omotáme ho okolo bočnej strany čelenky. Na druhej strane urobíme to isté, ale snažíme sa zamaskovať kábliky, ktoré budeme neskôr pripájať. Čelenka by mala byť podobná na oboch stranách, aby pôsobila symetrickým dojmom.

Následne si môžeme čelenku skrášliť dekoráciami. Aby sme plastový obal nepoškodili, je možné nalepiť lepiacu pásku na pásik. Na ňu sa ukladajú ligotavé nálepky a iné dekorácie.

Keď chceme programovať, potrebujeme si najskôr pripojiť neopixel pásik k micro:bitu:

- čierny alebo biely káblik pripojíme ku kolíku uzemnenia, teda k pinu GND,
- červený káblik pripojíme k stálemu napätiu 3V,
- žltý alebo zelený káblik slúžia na prenos dát, ktoré určujú jednotlivým LED diódam, ako sa majú správať. Pripájame ho k programovateľným pinom.

Ako bude pásik svietiť? LED pásik chceme rozsvietiť tak, aby sme začali od prvej a poslednej LED diódy, následne sa budú rozsvecovať po susednej línii a posledné LED diódy sa rozsvietia v strede. Znamená to, že si LED pásik v mysli rozdelíme na polovicu a spojíme pozície, ktoré chceme spolu rozsvietiť. Pozor, prvá LED dióda má pozíciu 0, posledná má pozíciu 7:

- najprv sa rozsvieti 0. a 7. LEDka,
- potom 1. a 6. LEDka.
- potom 2. a 5. LEDka.
- nakoniec 3. a 4. LEDka.

```
from microbit import sleep, pin0, microphone, SoundEvent
import neopixel

```

```
led_pasik = neopixel.NeoPixel(pin0, 8)

while True:
    if microphone.current_event() == SoundEvent.LOUD:
        for i in range(2):
            led_pasik[0] = (255, 20, 147)
            led_pasik[7] = (255, 20, 147)
            led_pasik.show()
            sleep(500)
            led_pasik[1] = (255, 20, 147)
            led_pasik[6] = (255, 20, 147)
            led_pasik.show()
            sleep(500)
            led_pasik[2] = (255, 20, 147)
            led_pasik[5] = (255, 20, 147)
            led_pasik.show()
            sleep(500)
            led_pasik[3] = (255, 20, 147)
            led_pasik[4] = (255, 20, 147)
            led_pasik.show()
            sleep(500)
            led_pasik.clear()
            led_pasik.show()
```

Najskôr je potrebné importovať neopixel a ďalšie triedy. Následne inicializujeme neopixel LED pásik. Vytvoríme premennú *led_pasik*, do ktorej uložíme *neopixel.Neopixel(pin0, 8)*, pretože ho mám pripevnený na pin0 a má 8 LED diód. Ak je počuť väčší hluk, 2-krát sa zopakuje rozsvietenie pásika, pričom v hranatých zátvorkách sa nachádza ich pozícia, ktorá sa počíta vždy od 0. Následne sa tam nachádzajú farby podľa RGB modelu. Samotný príkaz ešte nerozsvieti pásik, je potrebné využiť aj *led_pasik.show()*. Ak chceme farby vymazať, použijeme *led_pasik.clear()*.

Obrázok 3.17 Čelenka (SPy o.z., 2022)

3.7 DISKO NEOPIXEL LED TOPÁNKY

Pri neopixel LED pásiku ešte ostaneme a môžeme si vytvoriť podobný projekt, ale neopixel LED pásik nepôjde na čelenku, ale na topánky. Vytvoríme si teda disko topánky.

Pripravíme si:

- BBC micro:bit,
- baterky,
- topánky ideálne s hrubšou gumenou podrážkou,
- neopixel LED pásik,
- špendlík alebo obojstrannú lepiacu pásku.

Projekt je veľmi jednoduchý na realizáciu, pričom by sme mohli využiť aj praktickosť – ak je tma, svetlo na topánke sa rozsvieti. Rovnako by malo fungovať napr. aj na párty, kde je prítmie.

Najskôr je potrebné si inicializovať LED pásik s pripojením na pin 0 a 8 LED diódami. Následne v cykle while ak je osvetlenie menej alebo rovné 40, celý LED pásik sa rozsvieti podľa RGB parametrov. Posledný príkaz zobrazí led_pasik.

```
from microbit import display, pin0
import neopixel

led_pasik = neopixel.NeoPixel(pin0, 8)

while True:
    if display.read_light_level() <= 40:
        led_pasik.fill((0,100,120))
        led_pasik.show()
```

Realizácia spočíva iba v nalepení LED pásika na topánku, ideálne po okrajoch gumenej podrážky (Obrázok 3.18). Ak nalepenie nedrží, môžu sa použiť aj 2 – 3 špendlíky.

Obrázok 3.18 Disko neopixel LED topánka

3.8 MERANIE TICHA

Možno ste sa v triede stretli s hlukom, ktorý bol neprimeraný. V tejto podkapitole sa zameriame na merač ticha, ktorý bude zelený v čase, keď bude hlasitosť v triede primeraná a červený, keď bude limit presiahnutý.

Vytvorme si projekt pre každú osobu, ktorá bude mať pri sebe nositeľný micro:bit verzie 2 s mikrofónom, ktorý bude sledovať hlasitosť žiaka. Ak presiahne maximum, pošle to bezdrôtovou komunikáciou do micro:bita na učiteľský stôl (katedru) a ten zobrazí LED pásik na červeno, čím upozorní žiaka, že treba byť potichšie.

Micro:bit si chceme upevniť na naše telo, a to spravíme pomocou zatváracieho špendlíka alebo obojstrannej pásky. Keď túto časť realizácie máme, môžeme ísť programovať. Samozrejme, budeme pri bezdrôtovej komunikácii používať 2 rôzne programy.

Prvý program je nasledovný. Po naimportovaní si zapneme rádio a v cykle sa vykonáva to, že akonáhle je hladina zvuku vyššia ako 100, pošle sa 1 do ďalšieho micro:bita.

```
from microbit import microphone, sleepimport radio

radio.on()

while True:
    if microphone.sound_level() > 100:
    radio.send("1")
    sleep(1000)
```

V druhom programe si po importovaní zapneme rádio a RGB parametre uložíme do jednotlivých farieb. Inicializujeme LED pásik a keď rádio prijme 1, pixely budú zobrazovať červenú farbu, inak budú zobrazovať iba zelenú.

```
from microbit import microphone, sleep, pin0
import radio
import neopixel

RED = (255, 0, 0)
GREEN = (0, 255, 0)

radio.on()

led_strip = neopixel.NeoPixel(pin0, 8)

while True:
    incoming = radio.receive()
    if incoming == '1':
        for pixel_id in range(0, len(led_strip)):
                    led_strip[pixel_id] = RED
        led_strip.show()
        sleep(2000)
        incoming = '0'
    else:
        for pixel_id in range(0, len(led_strip)):
                    led_strip[pixel_id] = GREEN
            led_strip.show()
```

3.9 PLYŠOVÁ ROZPRÁVAJÚCA HRAČKA

Dnes existuje veľké množstvo hračiek. Jedna kategória sa nazýva Do It Yourself = urob si sám. Ide o hračky, ktoré si sami vyrobíme a to, čo kúpime v obchode, sú len nespojené látky, časti, súčiastky a iné. Je na nás, čo si vytvoríme, no ak dokážeme dať do hračky micro:bit s napájaním a reproduktor (je kvalitnejší ako integrovaný), môžeme hračke umožniť rozprávať.

V jazyku MicroPython máme možnosť, aby micro:bit rozprával. Poznáte ikonické kľúčenky, ktoré hovoria „Hello, I love you“? My si toto môžeme vytvoriť v MicroPythone.

Aby sme mali lepší zvuk, vypneme si zabudovaný reproduktor. Keď potrasieme hračkou, vykoná sa príkaz *speech.say*, ktorý povie „Hello, I love you“. Importujeme si *speech*, vypneme integrovaný reproduktor a pri potrasení povie „Hello. I love you“. Môžeme vyskúšať aj ďalšie slová alebo vety v angličtine.

```
import speech
from microbit import speaker, accelerometer

speaker.off()

while True:
    if accelerometer.was_gesture('shake'):
        speech.say("Hello. I love you")
```

NA ZÁVER

V tejto kapitole sme predstavili niekoľko námetov na projekty, v ktorých sme použili programovací jazyk MicroPython na programovanie mikrokontrolérov micro:bit. Na rozdiel od predchádzajúcich kapitol nie sú v programoch použité zložité štruktúrované údaje, ani netriviálne programovacie techniky. Práca na projektoch je zameraná komplexne na vytvorenie hmatateľného produktu, pričom programovanie je len jedna časť realizácie projektu. Nemenej dôležitá je práca s hardvérom a s ďalším materiálom na výrobu produktu, čo rozvíja manuálnu zručnosť a tvorivosť. Práca na takýchto projektoch môže osloviť aj žiakov, ktorí nemajú pokročilé vedomosti z programovania, ale sú tvoriví alebo majú vzťah k technike. Námety projektov z oblasti nositeľnej elektroniky, ktoré sme v tejto kapitole prezentovali, majú potenciál osloviť a pritiahnuť k programovaniu a technike aj dievčatá.

LITERATÚRA

Amos, David. 2022. Python GUI Programming with Tkinter. [Online] 30. 03 2022. [Dátum: 29. 08 2022.] https://realpython.com/python-gui-tkinter/#building-your-first-python-gui-application-with-tkinter.

Blaho, Andrej. 2018. Programovanie v Pythone. [Online] 2018. [Dátum: 13. 10 2022.] https://input.sk/pdf/python.pdf.

Devadas, Srini. 2017. *Programming for the Puzzles: Learn to Program While Solving Puzzles.* s.l. : The MIT Press, 2017. ISBN 978-0262534307.

Gombrich, Ernst H. 2017. *Príbeh umenia.* Bratislava : Ikar, 2017. ISBN 978-80-55153-81-0.

Guniš, Ján a kol. 2020. *Riešenie problémov a programovanie.* Bratislava : Centrum vedecko-technických informácií SR, 2020. ISBN 978-80-89965-62-5.

Micro:bit Educational Foundation. 2022. BBC micro:bit. [Online] 2022. [Dátum: 15. 09 2022.] https://microbit.org/.

Situating Constructionism. **Papert, S. a Harel, I. 1991.** New York : Ablex Publishing, 1991, Zv. 36.

Slovak Telekom. 2022. ENTER. [Online] 2022. [Dátum: 15. 09 2022.] https://enter.study/kde-vsade-najdete-enter/.

SPy o.z. 2022. Učíme s Hardvérom. [Online] 09. 11 2022. https://www.ucimeshardverom.sk/.

Stiller, E. 2009. Teaching Programming Using Bricolage. *Journal of Computing Sciences in Colleges.* 2009, Zv. 24.

TIOBE. 2022. TIOBE Index. [Online] 2022. [Dátum: 1. 11 2022.] https://www.tiobe.com/tiobe-index/.

PRÍLOHY

Príloha A. Zdrojové súbory riešení úloh z kapitoly 1

Príloha B. Zdrojové súbory riešení úloh z kapitoly 2

Názov:	**Programovanie v jazyku Python**
Podnázov:	Vybrané kapitoly pre učiteľov informatiky
Autori:	Gabriela Lovászová
	Viera Michaličková
	Nika Kvaššayová

Vydavateľ:	Univerzita Konštantína Filozofa v Nitre
Edícia:	Prírodovedec č. 797
Formát:	A4
Rok vydania:	2022
Miesto vydania:	Nitra
Počet strán:	80

ISBN 978-80-558-1951-8

ISBN 978-80-558-1951-8