Python pro SPŠ a VOŠ Kladno

Jaroslav Holeček
holecek@spskladno.cz

Poslední úprava: 3. května 2020

Obsah

1 Vývojový diagram

Vývojový diagram nesouvisí konkrétně s Pythonem, ale je obecnou součástí programování.
Je to (grafický) diagram, který znázorňuje v jakém pořadí se provádí jaké výpočty a kroky programu. Znázorňuje ve kterých místech se program dělí do více větví, podle určitého kritéria (podmínky) a také v jakou chvíli program končí.
Vývojový diagram nekreslíme pro konkrétní programovací jazyk - je to obecný zápis (znázornění) průběhu programu či algoritmu. Jeho smyslem je přehledně znázornit „co se děje“. Na základě vývojového diagramu jde obvykle snadno zapsat kód v konkrétním programovacím jazyku.

1.1 Odkazy

Ve vývojovém diagramu existuje spoustu norem a zvyklostí, existuje jich také několik typů.
`https://cs.wikipedia.org/wiki/V%C3%BDvojov%C3%BD_diagram`

1.2 Značky

Pro naše potřeby postačí zjednodušený vývojový diagam s pouze několika základními značkami.
Pro začátek a konec algoritmu použijeme ovál (elipsu). Pro krok algoritmu (výpočet, akci) použijeme obdélník. A pro větvení algoritmu (na základě nějaké podmínky - otázky) použijeme kosodélník.

(a) Začátek a konec algoritmu (programu)

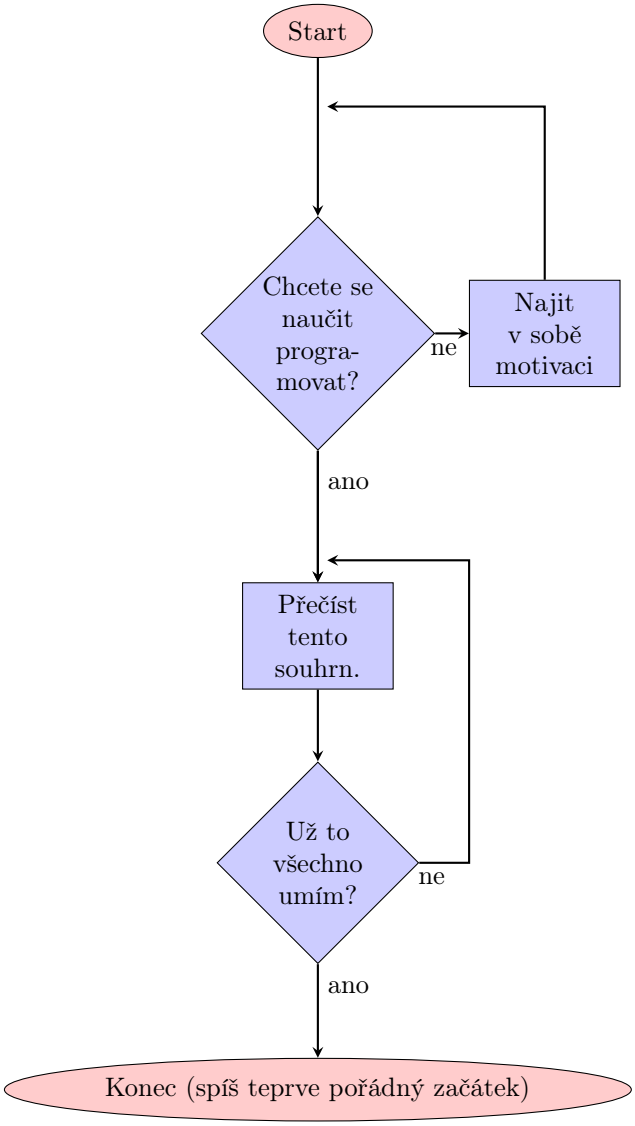
(b) Jeden krok algoritmu (programu)

(c) Větvení programu - Podmínka, otázka

Obrázek 1: Základní značky ve vývojovém diagramu

1.3 Celý diagram

Ke spojení jednotlivých kroků, resp. míst ve vývojovém diagramu používáme šipky. To můžeme vidět např. na obrázku **??**. Někdy se šipka vrací zpět do určitého místa diagramu (spojuje se s jinou šipkou - spojovací čárou). V takovém případě šipky (čáry) spojíme speciální spojovací značkou (obvykle malým kolečkem), nebo necháme šipku mířit na danou čáru. Nespojujeme čáry v blocích algoritmu. Pokud vedou do bloku algoritmu dvě šipky znamená to, že pro tento krok potřebujeme oba vstupy zároveň (což obvykle není to, co chceme naznačit).



Obrázek 2: Kompletní ukázka vývojového diagramu

2 Datové typy

V hardwaru počítače neexistují zvláštní místa pro ukládání čísel, nápisů, obrázků, písniček apd..
Vše je uloženo ve stejné mechanické součástce - paměti (operační, cash, pevný disk).
Z pohledu programu a programátora si můžeme představit, že je vše uloženo ve formě 0 a 1 - např. v ASCII znamená 1000001: "A", 1110111: "w", 0111000: "8".
Zda jsou v počítači uloženy znaky ("Aw8"), čísla (420), nebo třeba písnička či obrázek pozná program podle datového typu.
Datový typ je tedy označení, **co znamenají** uložená data (**jedničky a nuly**) - zda je to číslo, znak, obrázek, ...
;

2.1 Odkazy

`https://www.w3schools.com/python/python_datatypes.asp`

2.2 Datové typy v Pythonu

V Pythonu nemusíme (ale můžeme) zapisovat, jaký datový typ má být v proměnné uložený - dokonce se může v průběhu programu datový typ proměnné měnit.
Python přiřadí datový typ automaticky - podle toho, co uložíme do proměnné.
Zjistit, jakého datového typu je nějaká proměnná můžeme pomocí funkce „type()“.

```
1 x = 8
2 typ_x = type(x)
3 print(typ_x)
4
5 x = "Ahoj"
6 print(type(x))
```

Zdrojový kód 1: Type

ř. 1: Uloží do proměnné „x“ hodnotu 8. Automaticky se k „x“ přiřadí datový typ „Celé číslo“ („Integer - int“)
ř. 2: Zjistíme datový typ proměnné „x“ a uložíme ho do proměnné „typ_x“
ř. 3: Vytiskneme datový typ do konzole
ř. 5: Uloží do proměnné „x“ hodnotu "Ahoj". Automaticky se u „x“ změní datový typ na „Nápis - Řetězec“ („String - str“)
ř. 6: Datový typ si nemusíme ukládat, můžeme ho pouze vytisknout do konzole

2.2.1 Integer - int

Integer označuje **celé číslo**.
Jeho zkratka je „int“.

```
1 pocet_mesicu = 12
2 penez_v_penezence = 0
3 prumerne_iq_tridy = -23
4
5 print(type(pocet_mesicu))
6 print(type(penez_v_penezence))
7 print(type(prumerne_iq_tridy))
```

Zdrojový kód 2: Integer

ř. 1-3: Celé číslo (Integer) může být kladné, nula, nebo záporné.
ř. 5-7: Vytiskneme do konzole datový typ proměnných

Operace s int S celými čísly můžeme provádět:

+ Sčítání
- Odčítání

* Násobení

/ Dělení - výsledek je **float - desetinné číslo**

// Celočíselné dělení - výsledek je celé číslo - znáte ze 3. třídy ZŠ

% Zbytek po celočíselném dělení - znáte ze 3. třídy ZŠ

2.2.2 Float - float

Float označuje **desetinné číslo**, číslo s desetinnou čárkou (plovoucí desetinnou čárkou - odtud slovo „float“).
Takové číslo můžeme do proměnné uložit přímo („pozor - s desetinnou tečkou“). Také ho můžeme získat jako výsledek dělení.
Jeho zkratka je „float“.

```
1  pocet_odpracovanych_hodin = 14.5
2  print(type(pocet_odpracovanych_hodin))
3
4  pocet_mesicu = 12
5  rocni_plat = 1200
6  mesicni_plat = rocni_plat / pocet_mesicu
7
8  print(type(pocet_mesicu))
9  print(type(rocni_plat))
10 print(type(mesicni_plat))
```

ř. 1-3: Desetinné číslo (Float) zapisujeme s **desetinnou tečkou**.
ř. 6: Pokud vydělíme dvě celá čísla, získáme desetinné číslo - i když jdou vydělit beze zbytku.

Zdrojový kód 3: Float

Operace s float S desetinnými čísly můžeme provádět:

\+ Sčítání

\- Odčítání

* Násobení

/ Dělení

2.2.3 Boolean - bool

Boolean označuje **logickou hodnotu - Pravda, Nepravda (True, False)**.
V proměnné tohoto datového typu je tedy uloženo buď „True“ (Pravda, Platí), nebo „False“ (Nepravda, Neplatí).
Výsledek s tímto datovým typem vrací např. operátory porovnání: <, >, =<, =>, ==, != .
Boolean se využije mimo jiné všude, kde se pracuje s podmínkami - if 10, while 15.
Jeho zkratka je „bool“.

```
1  mam_brigadu = True
2  print(type(mam_brigadu))
3
4  moje_vyska = 175
5  muzu_byt_letuska = moje_vyska > 160
6  print(muzu_byt_letuska)
7  print(type(muzu_byt_letuska))
```

Zdrojový kód 4: Boolean

řádek 1: Uložení „True“ přímo.
řádek 5: Uložení Boolean jako výsledek porovnání dvou čísel.

Operace s bool S boolean hodnotami můžeme provádět:

or Logické sčítání - aby byl výsledek True, stačí aby byla jedna z hodnot True

and Logické násobení - aby byl výsledek True, všechny hodnoty musí být True

not Logická inverze - převrací hodnotu

2.2.4 String - str

String označuje **Řetězec znaků - nápis**. Zapisujeme je do uvozovek - jednoduchých, nebo dvojitých - výběr mezi jednoduchými a dvojitými nám umožní zapsat uvozovku i jako součást řetězce. **Pozor** při práci s čísly - je rozdíl mezi 15 (číslo - int) a "15"(řetězec - str). Číslo označuje počet - dá se sčitat, dělit, násobit s ostatními čísly tak , jak jste zvyklí z normálního počítání. Oproti tomu Řetězec je „obrázek“ (jak vypadá písmenko, číslo atp.) a tedy operace, které znáte (+,*) nebudou dělat to, co znáte z počítání (co to znamená sečíst dva obrázky?). Jeho zkratka je „str“.

```
1  jmeno = "Jarda"
2  print(type(jmeno))
3
4  vek_int = 16
5  vek_str = "16"
6  print(type(vek_int))
7  print(type(vek_str))
8
9  novy_vek = vek_int + 1
10 print(novy_vek)
11 novy_vek = vek_str + 1
12 print(novy_vek)
```

Zdrojový kód 5: String

řádek 1: Vytvoření proměnné „jmeno“ datového typu String.
řádek 4-5: „vek_int“ je celé číslo, „vek_str“ je String.
řádek 9.: Toto projde - čisla můžeme sčítat
řádek 11.: Toto „neprojde“ - co to znamená sečíst „obrázek“ s číslem?

Do proměnné typu string můžeme také zapisovat speciální znaky - zapisujeme je jako normální součást textu. Nejčastěji využijeme:

"$\backslash t$" Tabulátor

"$\backslash n$" Konec řádku

Operace se str S řetězcem lze provádět nepřeberné množství operací: `https://www.w3schools.com/python/python_strings.asp`
Jistě musíte znát:

\+ Zřetězení - spojí dva nápisy za sebe.

.split(odelovac) Rozdělení - roztrhá nápis na části v místech daných odelovac-em.

.isdecimal() Kontrola, zda jsou všechny znaky číslovky.

len(napis) Zjistí délku napis-u.

Pokud budete chtít s řetězcem cokoliv udělat - vždy se nejprve podívejte, zda již taková funkce neexistuje - téměř jistě ano a vy si ušetříte spoustu práce.

2.3 Výčtové datové typy

Velmi často potřebujeme v programech pracovat s více hodnotami stejným způsobem. Např. ke všem číslům přičíst 1, všechny nápisy převést na velká písmena apd.
Proto existují datové typy, které umožňují pod jedním názvem uložit více hodnot. Máme tedy jednu proměnnou, ve které je uloženo např. 10 různých čísel. Tyto jednotlivá čísla je samozřejmě potřeba nějak rozlišit, abychom mohli pracovat i s každým z nich zvlášť.

2.3.1 List - pole

List, někdy také označovaný jako pole (array) je jednoduše seznam prvků.
V těchto jednotlivých prvcích může být uloženo cokoliv a klidně může být v jednom poli více prvků různých typů. Můžeme tedy mít jeden List, ve kterém bude číslo, nápis a klidně i další pole.
Prvky jsou seřazené (mají pevně dané pořadí), mohou se opakovat a můžeme je měnit (přepisovat, mazat, přidávat).

Vytvoření pole Nejednodušeji vytvoříme pole pomocí hranatých závorek, ve kterých jsou jednotlivé prvky pole oddělené čárkami „[prvek0, prvek1, prvek2, . . .]“.

```
1  moje_pole = ["Emanuel", 17, 184.5]
```

Zdrojový kód 6: List - pole

řádek 1: Vytvoření pole se třemi prvky - každý z nich může (ale nemusí) být jiného datového typu.

Přístup k prvkům K prvkům pole můžeme přistupovat:

1. „Roztrháme“ pole na různé proměnné - do každé proměnné se uloží jeden prvek pole
2. Přistupujeme k průvků podle jejich pořadí. Tomuto pořadí říkáme **index** a **začíná od 0**

```
1  moje_pole = ["Emanuel", 17, 184.5]
2
3  jmeno, vek, vyska = moje_pole
4  print("Jmeno:", jmeno)
5  print("Vek:", vek)
6  print("Vyska:", vyska)
7
8  print("Jmeno:", moje_pole[0])
9  print("Vek:", moje_pole[1])
10 print("Vyska:", moje_pole[2])
11
12 moje_pole[0] = "Tonda"
13 print("Jmeno:", moje_pole[0])
```

Zdrojový kód 7: List - prvky

řádek 3: Rozdělení pole na tři různé proměnné

řádek 8: Přístup k prvkům pole pomocí indexů

řádek 12: Pomocí indexu můžeme měnit hodnoty v poli

Operace s List S polem můžeme provádět nepřeberné množství operací, proto se vždy nejprve pokuste najít v dokumentaci to, co chcete provést.
Jistě se budou hodit operace:

+	Spojení dvou Listů za sebe
.append()	Přidání prvku na konec Listu
.pop(index)	Odebrání prvku na zadaném indexu. Pokud index nezadáme, odebere se poslední prvek.

2.3.2 Tuple

Tuple je velmi podobná Listu 2.3.1.
Hlavní rozdíl spočívá v tom, že **prvky** Tuple se **nedají měnit**. Uvadíme ho zde proto, že datový typ Tuple je návratový typ některých funkcí. Abychom nebyli překvapeni.

Vytvoření tuple Nejednodušeji vytvoříme pole pomocí kulatých závorek, ve kterých jsou jednotlivé prvky pole oddělené čárkami „(prvek0, prvek1, prvek2, . . .)“.

```
1  moje_typle = ("Emanuel", 17, 184.5)
```

řádek 1: Vytvoření Tuple se třemi prvky - každý z nich může (ale nemusí) být jiného datového typu.

Zdrojový kód 8: Tuple

Přístup k prvkům K prvkům Tuple můžeme přistupovat:

1. „Roztrháme“ Tuple na různé proměnné - do každé proměnné se uloží jeden prvek
2. Přistupujeme k prvkům podle jejich pořadí. Tomuto pořadí říkáme **index** a **začíná od 0**

```
1  moje_tuple = ("Emanuel", 17, 184.5)
2
3  jmeno, vek, vyska = moje_tuple
4  print("Jmeno:", jmeno)
5  print("Vek:", vek)
6  print("Vyska:", vyska)
7
8  print("Jmeno:", moje_tuple[0])
9  print("Vek:", moje_tuple[1])
10 print("Vyska:", moje_tuple[2])
11
12 # Tento řádek skončí chybou
13 moje_tuple[0] = "Tonda"
```

řádek 3: Rozdělení tuple na tři různé proměnné

řádek 8: Přístup k prvkům tuple pomocí indexů

řádek 13: Tuple je neměnný datový typ - tento příkaz tedy **nemůžeme** provést

Zdrojový kód 9: List - prvky

Operace s List Tuple můžeme pouze číst, případně vytvořit nové:

+	Spojení dvou Tuple za sebe

3 Řídící struktury

Každý program běží postupně po řádcích od 1. řádku dále a vykonává příkazy přesně v tom pořadí, jak jdou za sebou.
Někdy (velmi často) ale chceme, aby se některé řádky (přikazy) přeskočily - neprovedly se. Někdy také chceme, aby se některé přikazy provedly opakovaně - vícekrát.
K tomuto slouží takzvané řídící struktury - speciální příkazy, které řídí, který řádek (příkaz) se provede jako další. Říkáme, že řídí běh programu.
To, že je nějaký řádek uvnitř řídící struktury (např. je to řádek, který chceme přeskočit) se v Pythonu pozná podle **odsazení**. Odsazení může být jakékoliv (mezera, tři mezery, ...) ale doporučuji tabulátor. Všechny řádky, které jsou ve stejné „skupince“ musí být odsazené stejně (např. všechny o jeden tabulátor;).

3.1 Odkazy

Operátory
`https://www.w3schools.com/python/python_operators.asp`
`https://naucse.python.cz/course/pyladies/beginners/comparisons/`
Podmínka - Větvení
`https://www.w3schools.com/python/python_conditions.asp`
`https://naucse.python.cz/course/pyladies/beginners/comparisons/`
Cykly
`https://www.w3schools.com/python/python_while_loops.asp`
`https://www.w3schools.com/python/python_for_loops.asp`
`https://naucse.python.cz/course/pyladies/beginners/while/`

3.2 Podmínka - Větvení

Můžeme si představit, že program běží po jednotlivých větvých - proto tomu říkáme větvení.
Stejně jako u větví na stromě, se v nějakém místě programu můžeme vydat dvěma směry (po dvou různých větvích). Na rozdíl od větví na stromě se ale jednotlivé větve v programu mohou opět spojit do jedné.

3.2.1 if

Nejjednodušší podmínka „if“ nám poslouží v případě, že chceme některé příkazy provést jen někdy - jen pokud je splněna daná podmínka. Pokud podmínka splněna není, přikazy se přeskočí.

```
1 trabant.jed()
2
3 if trabant.je_v_domecku():
4         trabant.dopln_palivo()
5
6 trabant.jed()
```

řádek 1: Provede se vždy
řádek 3: Zjišťuje se, zda je trabant v domečku
řádek 4: Provede se jen v případě, že je trabant v domečku
Pokud v domečku není, řádek se přeskočí
řádek 6: Provede se vždy

Zdrojový kód 10: if

Za klíčové slovo „if“ píšeme podmínku - cokoliv, o čem umí Python rozhodnout, zda je to splněné, nebo ne - zda je to *True*, nebo *False*.
Můžeme zde přímo zapsat *True*, nebo *False*, často zde píšeme porovnání: $<$, $>$, $==$, $!=$, $<=$, $>=$.

```
1 trabant.jed()
2
3 if trabant.get_rychlost() > 50:
4         policista.dej_pokutu(trabant)
5
6 trabant.jed()
```

Zdrojový kód 11: if porovnání

řádek 1: Provede se vždy
řádek 3: Zjišťuje se rychlost trabantu a porovná se s hodnotou 50
řádek 4: Provede se jen v případě, že trabant jede rychleji, než 50
Pokud jede 50 a méně, řádek se přeskočí
řádek 6: Provede se vždy

3.2.2 if-else

Často chceme aby se při splnění podmínky vykonaly některé příkazy a při nesplnění podmínky se vykonaly jiné. Chceme tedy pomocí podmínky vybrat jednu ze dvou skupin příkazů (řádků).
K tomu slouží konstrukce „if-else“. Při splnění podmínky (True) se (stejně jako v obyčejném „if“) provedou příkazy v části po „if“. Při nesplnění podmínky (False) se provedou příkazy v části po „else“. Jedna z těchto dvou skupin příkazů se tedy provede vždy.

```
1 if trabant.get_body() > ferrari.get_body():
2         hra.set_vitez(trabant)
3 else:
4         hra.set_vitez(ferrari)
```

Zdrojový kód 12: if-else

řádek 1: Porovnání počtu bodů trabantu a ferrari
řádek 2: Pokud je podmínka splněna == True, je vítěz trabant
řádek 4: Pokud není podmínka splněna == False, je vítěz ferrari
Kontrolní otázka... Kdo je vítěz, pokud mají stejný počet bodů?

3.2.3 elif

Opravíme kód 12. Někdy potřebujeme vybrat ne mezi dvěma příkazy, ale mezi více. K tomu slouží konstrukce „elif“ - dovolí nám vložit další „větev“. Takových větví může být libovolné množství a **program se vydá** vždy jen tou z nich, u které je **podmínka splněna jako první** (kontrolováno od **shora dolů**). Pokud není splněna žádná z podmínek, vydá se program větví „else“.

```
1 if trabant.get_body() > ferrari.get_body():
2         hra.set_vitez(trabant)
3 elif ferrari.get_body() > trabant.get_body():
4         hra.set_vitez(ferrari)
5 else:
6         hra.set_vitez("remíza")
```

Zdrojový kód 13: if-elif-else

řádek 2,4,6: Provede se jen jeden z těchto příkazů - poté, co jedna z podmínek projde (je splněna), se ostatní ani nekontrolují

3.2.4 Vnoření

Podmínky (stejně jako jiné řídící struktury) můžeme takzvaně **vnořovat** - tedy **vkládat jednu do druhé**.

```
1 if trabant.je_ve_meste():
2         if trabant.get_rychlost() > 50:
3                 policista.dej_pokutu(trabant)
4 else:
5         if trabant.get_rychlost() > 90:
6                 policista.dej_pokutu(trabant)
```

Zdrojový kód 14: if-else vnoření

ř. 1: Zjištění zda je trabant ve městě, nebo mimo město
ř. 2-3: Pokud je ve městě porovnává se rychlost s 50
ř. 4: Není ve městě
ř. 5-6: Pokud není ve městě porovnává se rychlost s 90

3.3 Cykly

Velmi často chceme, aby program provedl nějaký výpočet vícekrát. Buď proto, že tímto opakováním získáme požadovaný výsledek, nebo chceme stejnou operaci provést s více „objekty“ (např. poslat zprávu všem kamarádům ze seznamu).
K tomuto opakování slouží takzvané cykly.

3.3.1 While

Cyklus „while“ opakuje příkazy, **dokud je splněna** zadaná **podmínka**. Nejprve zkontroluje, zda je podmínka splněna - pokud ano, **vykoná všechny zadané příkazy** a poté zkontroluje podmínku znovu - a tak stále dokola, dokud při kontrole podmínky nezjistí, že podmínka již splněna není. Ve chvíli, kdy podmínka splněná není (ať už hned napoprvé, nebo kdykoliv později), přeskočí všechny zadané příkazy a program pokračuje dále za tímto „while“ cyklem.

```
1 trabant.jed()
2
3 while trabant.chci_palivo():
4         benzinka.vyber_korunky(trabant)
5         benzinka.dopln_trochu_paliva(trabant)
6
7 trabant.jed()
```

Zdrojový kód 15: While

ř. 1: Provede se jednou
ř. 3: Kontrola, zda trabant chce palivo
ř. 4-5: Pokud je podmínka splněna (chce palivo (True)) vykonají se všechny příkazy uvnitř while
Poté se znovu zkontroluje podmínka
ř. 7: Ve chvíli, kdy podmínka splněna není (nechce palivo (False)) pokračuje program dále za celým cyklem (za všemi řádky uvnitř cyklu)

3.3.2 For

Cyklus „for“ používáme v Pythonu k procházení libovolného seznamu (resp. čehokoliv, co se skládá z více prvků). Takovým seznamem může být mimo jiné:

1. Pole (v Pythonu List) - seznam libovolných prvků

2. Řetězec (String) - seznam znaků (písmeno, číslice, tečka, mezera, atd.)
3. Range - seznam čísel
4. Textový soubor - seznam řádků

Při procházení seznamu pomocí „for“ máme uložen **jeden** prvek ze seznamu při každém průchodu (procházíme příkazy uvnitř „for“). Při prvním průchodu máme uložen první prvek, při druhém průchodu druhý prvek atd.
Můžeme tak nějakou operaci (jeden nebo více příkazů) provést s každým prvkem seznamu zvlášť - a provést ji postupně se všemi prvky.

```
1 hra.zmen_rocni_obdobi(zima)
2
3 for auticko in hra.get_vsechna_auta():
4         auticko.prezuj_pneu(zima)
5         auticko.dej_do_kufru_retezy()
6
7 trabant.jed()
```

Zdrojový kód 16: For

řádek 1: Provede se jednou
řádek 3: Procházíme seznam všech aut (za „in“) Všechna auta jsou postupně po jednom uložena do proměnné „auticko“
řádek 4-5: S autem, které je zrovna na řadě se provedou zadané příkazy - postupně se provedou se všemi auty ze seznamu
řádek 7: Po zpracování celého seznamu pokračuje program dále

3.3.3 Break

Jakýkoliv cyklus můžeme také ukončit kdykoliv v jeho těle (mezi příkazy, které jsou v něm napsané). Slouží k tomu příkaz **„break“**. Ve chvíli kdy **program** dorazí na řádek s tímto příkazem, **okamžitě skočí za cyklus**, ve kterém je tento příkaz zapsaný.

```
1 trabant.jed()
2
3 while trabant.chci_palivo():
4         benzinka.vyber_korunky(trabant)
5         if not benzinka.vyber_penez_v_poradku():
6                 break
7         benzinka.dopln_trochu_paliva(trabant)
8
9 trabant.jed()
```

Zdrojový kód 17: Break

řádek 1: Provede se jednou
řádek 3: Kontrola, zda trabant chce palivo
řádek 5: Kontrola, zda trabant zaplatil
řádek 6: Pokud trabant nezaplatil, provede se příkaz „break“ - program skočí **za** cyklus, tedy na řádek 9
řádek 9: Ve chvíli, kdy podmínka splněna není (nechce palivo (False)) pokračuje program dále za celým cyklem (za všemi řádky uvnitř cyklu)

3.3.4 Continue

Pomocí předchozího „break“ se ukončil celý průběh cyklu. Můžeme také ukončit průběh cyklu jen v jednom „kolečku“ (pro aktuální průchod). Cyklus pak bude pokračovat v dalším kolečku - od jeho prvního řádku.

Na příkladu níže budou přezuty a budou mít řetězy všechna auta, kromě trabantu.

```
1 hra.zmen_rocni_obdobi(zima)
2
3 for auticko in hra.get_vsechna_auta():
4         if auticko is trabant:
5                 continue
6         auticko.prezuj_pneu(zima)
7         auticko.dej_do_kufru_retezy()
8
9 trabant.jed()
```

Zdrojový kód 18: Continue

řádek 1: Provede se jednou
řádek 3: Procházíme seznam všech aut (za „in“)
řádek 4-7: S autem, které je zrovna na řadě se provedou zadané příkazy - postupně se provedou se všemi auty ze seznamu
řádek 4: Kontrola, zda je auticko trabant
řádek 5: Pokud auticko je trabant provede se „continue“ a cyklus skočí do dalšího kolečka - k dalšímu autu.
řádek 9: Po zpracování celého seznamu pokračuje program dále

4 Funkce

Funkce (někdy také nazývané metody) slouží k významnému zjednodušení a zpřehlednění kódu. Funkci si můžete představit jako „krabičku“, která umí dělat něco užitečného a kterou mám uloženou v paměti. Tuto „krabičku“ můžu v programu použít kolikrát chci.
Často chceme stejný proces (několik příkazů) spustit na několika místech v programu. Funkce nám umožní tento proces (několik příkazů) zapsat pouze jednou - pojmenovat ho - a poté ho spustit jen zadáním jeho jména. Nemusíme tak stále dokola psát stejný kód na všech místech, kde ho chceme spustit.
Pokud někde v programu píšete **podruhé** stejnou část kódu, již je to chvíle, kdy je čas na **použití funkce**.

Pro ukázku použití funkcí vytvoříme funkce, které budou zdravit naše kamarády a učitele

4.1 Vytvoření funkce

Předtím, než můžeme funkci používat, ji musíme samozřejmě vytvořit - definovat. Definicí funkce ji pouze uložíme do paměti. V paměti funkce čeká do té doby, dokud ji nezavoláme (nespustíme). Funkci vytvoříme zapsáním klíčového slova „**def**“. Za ním následuje **název funkce** - jak chceme funkci volat. Dále jsou **kulaté závorky** - později do závorek zapíšeme takzvané argumenty, ale i když žádné argumenty psát nechceme, kulaté závorky zde být musí. Na konec řádku zapíšeme **dvojtečku**. Tomuto prvnímu řádku se říká **hlavička funkce**. A nyní již na další řádky píšeme příkazy, které chceme uvnitř funkce - tedy ty, které se provedou, až funkci zavoláme (spustíme). Těmto řádkům říkáme „tělo funkce“.

```
1  def pozdrav():
2          print("Nazdar")
3          print("Nazdar")
4          print("Nazdar")
5
6  print("Franta")
7  pozdrav()
8
9  print("Lojza")
10 pozdrav()
11
12 print("Marenka")
13 pozdrav()
```

Zdrojový kód 19: Definice funkce

řádek 1: Definice (vytvoření, připravení) funkce: Klíčové slovo „def“, název funkce, kulaté závorky a dvojtečka.

řádek 2-4: Tyto řádky (příkazy) se provedou po zavolání funkce - jsou uvnitř funkce - jsou odsazené. Říkáme jim „tělo funkce“

řádek 6: Program pokračuje na prvním neodsazeném řádku - již nepatří do funkce.

řádek 7, 10, 13: Voláme funkci. Na všech těchto řádcích skočí program do volané funkce (tedy na řádek 1) a provede všechny příkazy uvnitř funkce.

4.1.1 Prázdná funkce

Někdy si chceme jen připravit hlavičku funkce, ale zatím do ní nenapsat žádné příkazy. Například víme, že funkci budeme později potřebovat a nechceme na ní zapomenout, nebo chceme, aby se nám zobrazovala v našeptávání při psaní dalšího kódu.

```
1  def pozdrav_zdvorily():
2          pass
3
4  def pozdrav():
5          print("Nazdar")
6          print("Nazdar")
7          print("Nazdar")
```

řádek 1: Hlavička funkce je zcela stejná, jako u neprázdné funkce.

řádek 2: Dovnitř (do těla) funkce zapíšeme klíčové slovo „pass“.

Zdrojový kód 20: Prázdná funkce

4.2 Argumenty

U funkce sice chceme, aby prováděla stále stejné příkazy, ale byly bychom rádi, aby uměla tyto stejné příkazy provést na různých datech (vstupech). Například pozdravit společně se jménem - a toto jméno bude pokaždé jiné (podle toho, koho zrovna zadravíme).
Abychom mohli dostat nějakou informaci (třeba jméno) dovnitř do funkce, použijeme argumenty funkce.

```
1  def pozdrav(jmeno):
2          print("Nazdar", jmeno)
3          print("Nazdar", jmeno)
4          print("Nazdar", jmeno)
5
6
7  pozdrav("Franta")
8
9  pozdrav("Lojza")
10
11 pozdrav("Marenka")
```

řádek 1: Definice funkce s tím, že jí při volání předáme jeden argument - jméno, které má pozdravit: Argumenty (zde je pouze jeden, ale může jich být více) píšeme do kulatých závorek.

řádek 7, 9, 11: Protože jsem vytvořili funkci s argumentem - musíme jí nějakou hodnotu tohoto argumentu předat.

Zdrojový kód 21: Funkce s argumentem

Argumentů můžeme funkci předat více, mohou být jakýchkoliv datových typů a mohou se ve funkci použít libovolně - tedy může být jeden argument string, druhý integer, atd. Více argumentů píšeme do kulatých závorek a **oddělujeme je čárkou**.

Upravíme funkci tak, abychom jí mohli říct (předat argument), kolikrát má daného člověka pozdravit

```
1  def pozdrav(jmeno, pocet):
2          for i in range(1,pocet+1):
3                  print("Po ", i, ".: Nazdar ", jmeno)
4
5
6  pozdrav("Franta", 1)
7  pozdrav("Lojza", 3)
8  pozdrav("Marenka", 10)
```

řádek 1: Definice funkce s dvěma argumenty.

řádek 6, 7, 8: Protože jsme vytvořili funkci s dvěma argumenty - musíme jí při volání předat dvě hodnoty.

Zdrojový kód 22: Funkce s dvěma argumenty

4.3 Návratová hodnota - return

Funkce mohou také spočítat výsledek (tak, jak znáte z matematiky - např. funkce 2x+1 spočítá pro vstup 1 výsledek 3, pro vstup 4 výsledek 9, pro vstup 7 výsledek 15 atd.).
Často u funkce chceme, aby nám výsledek, který spočítá, takzvaně „vrátila“ na místo (řádek) programu, odkud jsme funkci zavolali. V tomto místě (kde jsme funkci zavolali) typicky výsledek použijeme k dalším výpočtům, ale můžem s ním dělat cokoli chceme (nic, uložit ho, dále s ním počítat).
Že je již výpočet u konce (došli jsme k výsledku, který chceme vrátit) a chceme tedy funkci ukončit a vrátit výsledek, zepíšeme ve funkci pomocí klíčového slova „return“. Po tomto klíčovém slově se již ve funkci neprovedou žádné příkazy. Pokud za slovo „return“ zapíšeme co má funkce vrátit, přenese se tato hodnota do místa, odkud jsme funkci zavolali.

```
1  def pozdrav_zdvorile(jmeno, pocet):
2          for i in range(1,pocet+1):
3                  print("Po ", i, ".: Dobry den ", jmeno)
4
5          return len(jmeno)
6          print("Toto se jiz nevypise :-( ")
7
8
9  delka_jmena_1 = pozdrav("Jan", 1)
10 print(delka_jmena_1)
11
12 print(pozdrav("Vladislav", 3))
13
14 delka_jmena_2 = pozdrav("Cecilie",10)
15
16 soucet_delky_jmen = delka_jmena_1 + delka_jmena_2
17 print("Pocet znaku na pozvance: ", soucet_delky_jmen)
```

Zdrojový kód 23: Funkce s return

řádek 5: Klíčové slovo „return“ a za ním hodnota, která se má vrátit - zde např. délka jména, které jsme funkci předali jako argument (např. u "Jan"je vráceno 3, u "Cecilie"je vráceno 7). Žádný další řádek už se neprovede.

řádek 9: Vrácenou hodnotu si můžeme uložit do proměnné (zde „delka_jmena_1“).

řádek 12: Vrácenou hodnotu můžeme také přímo použít (zde vytisknout do konzole).

řádek 16: Uložené hodnoty můžeme samozřejmě kdykoliv použít.

5 Zpracování chyb

Při spuštění programu může vzniknout velké množství chyb.
V této kapitole se nebudeme zabývat chybami, které nedovolí program ani spustit - chybně zapsané příkazy, špatné odsazení bloku, použití proměnné nebo funkce, která neexistuje, ...
Budeme se zde věnovat tak zvaným **RunTimeError - chybám za běhu programu**.

Na těchto chybách je zákeřné, že nastanou pouze někdy. Může se stát, že někdy náš program proběhne zcela v pořádku a někdy skončí chybou.
Typickým příkladem je:

Necháme uživatele zadat dvě čísla, která následně program vydělí a ukáže výsledek.
Pokud je uživatel hodný a zadá čísla, která se dají vydělit, tak náš program proběhne v pořádku. ***Uživatelé*** *ale* ***nejsou hodní*** *a tak hned při druhéme výpočtu zadá uživatel dělení nulou. Dělit nulou Python neumí a tak program skončí chybou - „spadne“.*

V Pythonu se s takovými chybami můžeme elegantně vypořádat pomocí bloku „try-except“.

```
1  delenec = float(input("Zadej delence: "))
2  delitel = float(input("Zadej delitele: "))
3
4  try:
5          podil = delenec / delitel
6          print("Vysledek je:" podil)
7  except ZeroDivisionError as chyba:
8          print("Ale ty jeden... Nulou preci delit nejde.")
9          print("Python by ti rekl: ", chyba)
10
```

řádek 4: Začátek bloku, ve kterém očekáváme možnou chybu za běhu.

řádek 5: Pokud na tomto řádku nastane chyba, kterou jsme očekávali, přejde program na řádek 8

řádek 7: Zde zapíšeme, jaký typ chyby chceme ošetřit a případně si chybu můžeme uložit do proměnné (kdybychom z ní třeba chtěli získat další informace)

Zdrojový kód 24: Try - except

Bloků except můžeme za sebe zapsat kolik chceme - chyba která nastane, se porovnává se zadanými typy chyb odshora dolů a program provede ten blok, u kterého první sedí typ chyby.

6 Práce se souborem

Často chceme, aby si náš program uchoval informaci i po svém ukončení (např. dosažený výsledek ve hře, uživatelské nastavení). Pokud těchto informací není mnoho, můžeme je uložit do souboru. Do souboru je můžeme uložit i když jich mnoho je, ale v takovém případě je lepší využít databázi, která bude s uloženými daty pracovat rychleji. Někdy také přímo chceme, aby náš program vytvořil soubor - například seznam uiživatelů či výsledkovou listinu.

6.1 Typy souborů

Pokud pomineme specializované typy souborů, jako jsou obrázky (jpg, png, ...) či dokumenty (pfd, xml, xlsx, doc, ...) umí samotný Python pracovat především s dvěma typy souborů:

1. **Textový**, ve kterém je uložen text - tedy znaky ASCII, UTF8, cp1250, případně dalších kódování.
   Tento typ souboru je defaultní, tedy pokud neurčíme jinak, použije se právě textový soubor.
2. **Binární**, ve kterém jsou uložená „surová" data - tedy 1 a 0. Například obrázky, videa, zvukové soubory.

Samozřejmě i v textovém soubou jsou uloženy jen 1 a 0 (stejně jako všude v počítači). Zde jsou seřazeny v přesně daném uspořádání (určeném kodováním) a tedy je takový soubor čitelný pro jakýkoliv jiný program, který umí text s daným kódováním číst - např. SublimeText, Poznámkový blok, Gedit apd.

6.2 Textový soubor

Pokud chceme pracovat se souborem, musíme tento **soubor** nejprve **otevřít**. To provedeme funkcí „**open()**". Soubor můžeme otevřít v různých režimech - pro čtení, pro zápis, pro obojí. Podle zvoleného režimu můžeme ze souboru načítat data (funkce **read()**, **readline()**), zapisovat data (funkce **write()**), nebo obojí.

6.2.1 Otevření souboru - open()

Před začátkem práce se souborem jej musíme nejprve otevřít. Při otevírání souboru může nastat chyba (viz. níže, nebo nemáme práva pro přístup do složky apd.). Takovou chybu je vhodné v programu ošetřit (viz. kód **??**).
K otevření souboru využijeme funkci **open()**, které musíme předat argumenty:

název souboru , kterým určíme s jakým souborem chceme pracovat.
Pokud zadáme celou cestu k souboru ($C:\\muj\adresar\se\soubory\muj_prvni_soubor.txt$), můžeme otevřít soubor kdekoliv. Pokud zadáme pouze název souboru ($muj_prvni_soubor.txt$), otevře se tento soubor ve stejné složce, ve které máme náš program (skript).

režim (mód) , kterým určime, zda chceme ze souboru číst, chceme do něj zapisovat, nebo obojí. Defaultní režim (tedy takový, který se použije, pokud žádný nezadáme je: čtení textového souboru). Režim změníme pomocí následujících hodnot:

Režim
- r čtení
  pokud soubor neeexistuje, nastane chyba
- a přidání na konec - ponechá současný obsah souboru a nově zapsaný připíše na konec
  pokud soubor neexistuje, vytvoří se
- w přepsání souboru - smaže současný obsah souboru a začne psát od začátku
  pokud soubor neexistuje, vytvoří se

	x	vytvoření souboru pokud soubor existuje, nastane chyba
	+	doplnění druhé možnosti ("r+"je čtení a zápis, "w+"je zápis a čtení)
Typ	t	textový
	b	binární

Funkce „open()“ ovetře soubor, ale také nám **vrátí objekt**, ve kterém je **uloženo spoustu užitečných informací** - zda se soubor podařilo otevřít, v jakém režimu je soubor otevřený, v jakém místě v souboru právě čteme (případně do jakého místa právě zapisujeme). Tento objekt použijeme vždy, když budeme chtít se souborem něco provést.

Po ukončení práce se souborem je vhodné jej **zavřít** funkcí „**close()**“.

```
1 soubor = open("muj_prvni_soubor.txt", "r")
2 # zde muzeme cist ze souboru soubor
3 soubor.close()
4
5 with open("muj_prvni_soubor.txt", "r") as soubor:
6         # zde muzeme cist ze souboru soubor
7         # POZOR! Prikazy jsou odsazene - jsme uvnitr bloku with
```

Zdrojový kód 25: Otevření souboru

řádek 1: Otevření souboru pro čtení a vytvoření objektu „soubor“.

řádek 3: Uzavření souboru

řádek 5: Můžeme také využít blok „with“, který za nás zavře soubor automaticky

6.2.2 Čtení ze souboru

Při čtení ze souboru se nám obsah souboru (nebo jeho část) načte do string proměnné v kódu. S tímto načteným obsahem poté pracujeme jako s jakýmkoliv jiným řetězcem (stringem).
Při čtení se v souboru posouvá ukazatel, který ukazuje na místo, kde jsme se čtením skončili (tento ukazatel je uložený v objektu, který nám vrátí funkce open()). Pokud tedy přečteme 7 znaků a dále v programu ve čtení pokračujeme, budou se znaky načítat od místa, kde jsme se čtením skončili (tedy od 8. znaku). Při čtení souboru máme několik možností:

Přečtení celého souboru Pomocí funkce „read()“ přečteme celý soubor najednou.

```
1 with open("muj_prvni_soubor.txt", "r") as soubor:
2         obsah_souboru = soubor.read()
3         print(obsah_souboru)
```

Zdrojový kód 26: Přečtení celého souboru

řádek 2: Uložení celého obsahu souboru do proměnné „obsah_souboru“

řádek 3: Vypsání do konzole

Přečtení daného počtu znaků Pomocí funkce „read()“ můžeme také přečíst pouze zadaný počet znaků.

```
1 with open("muj_prvni_soubor.txt", "r") as soubor:
2         sedm_znaku = soubor.read(7)
3         print(sedm_znaku)
```

Zdrojový kód 27: Přečtení zadaného počtu znaků

řádek 2: Načtení pouze 7 znaků

Přečtení daného počtu znaků Pomocí funkce „readline()“ přečteme jeden řádek v souboru.

```
1 with open("muj_prvni_soubor.txt", "r") as soubor:
2         radek = soubor.readline()
3         print(radek)
```

řádek 2: Načtení jednoho řádku

Zdrojový kód 28: Přečtení jednoho řádku

Přečtení celého souboru po řádcích Velmi často chceme přečíst celý soubor po jednotlivých řádcích. V Pythonu k tomu můžeme využít jednoduchou konstrukci za použití „for“. For-cyklus slouží k procházení libovolného seznamu prvek po prvku - a v Pythonu je soubor také seznam - seznam řádků.

```
1 with open("muj_prvni_soubor.txt", "r") as soubor:
2
3         for radek in soubor:
4                 print(radek)
```

řádek 3: Tato konstrukce projde postupně celý soubor - do proměnné „radek“ bude v každém průchodu cyklem uložen jeden řádek souboru

Zdrojový kód 29: Přečtení celého souboru po řádcích

6.2.3 Zápis do souboru

Zápis do souboru je velmi přímočarý. Po otevření souboru ve správném režimu, použijeme funkci „write()“, která umí zapsat string do souboru.

```
1 with open("muj_prvni_soubor.txt", "w") as soubor:
2         soubor.write("Toto je můj první zápis do souboru.")
3         soubor.write(str(5))
```

řádek 1: Při otevírání souboru zvolíme správný režim.

řádek 2: Zápis textu do souboru.

řádek 3: Pokud chceme zapsat číslo, musíme ho nejprve převést na string.

Zdrojový kód 30: Zápis do souboru

Speciální znaky Obzvláště při práci se souborem využijeme speciální znaky:

"$\backslash t$" Tabulátor

"$\backslash n$" Konec řádku

7 Objektově orientované programování - OOP

7.1 Odkazy

https://naucse.python.cz/lessons/beginners/class/
https://www.w3schools.com/python/python_classes.asp

7.2 Třída a objekt

Třídu si můžeme představit jako „šablonu“ nebo „výrobní plán“, podle kterého se vytváří jednotlivé objekty - říkame jim objekty dané třídy. Ve chvíli, kdy v programu zavoláme příkaz pro vytvoření objektu nějaké třídy, vytvoří se objekt, který má vlastnosti a dovednosti (správně řečeno atributy a metody), které jsme mu předepsali ve třídě.
Objektů můžeme od stejné třídy vytvořit libovolné množství. Každý z nich pak „žije vlastním životem“ - hodnoty jeho vlastností jsou nezávislé na ostatních objektech. Všechny objekty stejné třídy ale mají stejné vlastnosti (hodnoty těchto vlastností stejné být nemusí) a dovednosti.

Máme třídu Auto. Každé auto má: pozici na mapě (x,y), rychlost a barvu. Vytvoříme tři auta trabant, skoda a ferrari. Každé z nich má v průběhu hry různou pozici, rychlost i barvu (např. je jedno modré, druhé červené a třetí zelené) - každé auto má různé hodnoty vlastností (např. "modra", "cervena", "zelena"), ale vlastnosti mají všechna auta stejné (např. všechny mají barvu).

```
1  class Auto:
2          x = 0
3          y = 0
4          barva = "cerna"
5          rychlost = 5
6
7          def jed(self):
8                  self.x = self.x + 1
9
10         def vypis_se(self):
11                 print(self.barva , " - x: ", self.x, " y: ", self.y)
12
13 trabant = Auto()
14 skoda = Auto()
15 ferrari = Auto()
```

Zdrojový kód 31: První ukázka třídy

Tato ukázka je značně nešikovná, protože všechna vyrobená auta (trabant, skoda, ferrari) maji ze začátku (od jejich „výroby“) stejnou barvu, rychlost i pozici.
Situace je dokonce ještě horší - takto zapsané, mají všechny objekty atributy společné (i hodnoty). Je to pouze ilustrativní příklad - opravíme ho o kousek níže.

ř. 1: Definice (vytvoření) třídy
ř. 2-5: Předepsání 4 vlastností (atributů)
ř. 7-11: Předepsání 2 dovedností (metod)
ř. 13-15: Vytvoření 3 objektů Auto

7.3 Přístup k vlastnostem (čtení, změna)

U každého objektu můžeme měnit hodnotu jeho vlastností. Tato změna ovlivní pouze jeden objekt - na žádný jiný nemá vliv a tak hodnoty u všech ostatních objektů zůstanou stejné.
K vlastnostem a dovednostem (metodám) objektu přistupujeme pomocí tečky.

```
1  class Auto:
2          x = 0
3          y = 0
4          barva = "cerna"
5          rychlost = 5
6
7          def jed(self):
8                  self.x = self.x + 1
9
10         def vypis_se(self):
11                 print(self.barva , " - x: ", self.x, " y: ", self.y)
12
13 trabant = Auto()
14 skoda = Auto()
15 ferrari = Auto()
16
17 trabant.barva = "cervena"
18 skoda.barva = "modra"
19 ferrari.barva = "zelena"
20
21 trabant.jed()
22 trabant.vypis_se()
23 skoda.vypis_se()
```

řádky 17-19: Změna hodnoty vlastnosti barva u všech objektů
řádek 21: Zavolání metody jed() na objekt trabant
řádky 22-23: Zavolání metody vypis_se() na objekty trabant a skoda

Zdrojový kód 32: Přístup k vlastnostem

7.4 Self

Ve zdrojových kódech 31, 32 a 33 si můžeme všimnout slova „self“ v metodách (dovednostnech) třídy.
Toto slovo (mimochodem, nemusí to být přímo „self“, může to být jakékoliv slovo), které je **první** mezi argumenty metody (první v kulatých závorkách) označuje, že pracujeme právě s jedním daným objektem - v metodě označeným právě „self“.
Nepřehlédneme také, že „self“ je mezi argumenty pouze při vytváření metody. Ve chvíli, kdy metodu (dovednost) voláme na nějakém objektu (pomocí tečky), žádné „self“ již do kulatých závorek nepíšeme. „Self“ totiž označuje (je v něm uložen) právě ten objekt, na kterém jsme metodu (dovednost) zavolali - ten, který je před tečkou.

```
1  class Auto:
2          x = 0
3          y = 0
4          barva = "cerna"
5          rychlost = 5
6
7          def jed(self):
8                  self.x = self.x + 1
9
10         def vypis_se(self):
11                 print(self.barva , " - x: ", self.x, " y: ", self.y)
12
13 trabant = Auto()
14 skoda = Auto()
15 ferrari = Auto()
16
17 trabant.barva = "cervena"
18 skoda.barva = "modra"
19 ferrari.barva = "zelena"
20
21 trabant.jed()
22 trabant.vypis_se()
23 skoda.vypis_se()
```

řádek 21: Při vykonávání tohoto příkazu (metody jed() - tedy řádků 7-8) bude v „self“ uložen objekt **trabant**
řádek 22: Při vykonávání tohoto příkazu (metody vypis_se() - tedy řádků 10-11) bude v „self“ uložen objekt **trabant**
řádek 23: Při vykonávání tohoto příkazu (metody vypis_se() - tedy řádků 10-11) bude v „self“ uložen objekt **skoda**

Zdrojový kód 33: Self

7.5 Metoda __init__()

V ukázkách kódu 31, 32 a 33 je třída vytvořena značně nešikovně. Všechny objekty Auto mají při svém vytvoření určené hodnoty vlastností (x: 0, y: 0, barva: "cerna", rychlost: 5). To ale není to, co chceme - chceme, aby auta byla různá (např. měla různou barvu). Trochu tento stav vylepšuje kód 32 a 33, kde změníme hodnotu barvy tak, aby ji mělo každé auto jinou. Stále ale máme pevně dané hodnoty vlastností při vytváření objektů - pro jejich změnu musíme psát řádky navíc - a to není pohodlné, ani bezpečné. Situace je dokonce ještě horší (pro úvod se tím nebudeme zabývat, ale takto vytvořené atributy jsou nejprve pro všechny objekty stejné (i hodnoty))

Tento problém řeší metoda __init__() - **Pozor: Před i za slovem „init“ jsou 2 podržítka**
Metoda **__init__()** je speciální metoda (speciální metody jsou označny právě těmi podtržítky), která se zavolá ve chvíli, kdy **vytváříme objekt**.
Při vytváření objektu se vykonají všechny příkazy v této metodě __init__() - a stejně jako každá jiná metoda umí pracovat i s **argumenty** a to je přesně to, co využijeme.

```
1  class Auto:
2          def __init__(self, zadane_x, zadane_y, zadana_barva, zadana_rychlost):
3                  self.x = zadane_x
4                  self.y = zadane_y
5                  self.barva = zadana_barva
6                  self.rychlost = zadana_rychlost
7
8          def jed(self):
9                  self.x = self.x + 1
10
11         def vypis_se(self):
12                 print(self.barva , " - x: ", self.x, " y: ", self.y)
13
14 trabant = Auto(1, 2, "cervena", 5)
15 skoda = Auto(13, 14, "modra", 5)
16 ferrari = Auto(25, 26, "zelena", 5)
17
18 trabant.jed()
19 trabant.vypis_se()
20 skoda.vypis_se()
```

ř. 2-6: Tyto řádky se vykonají při vytváření objektu. Do vlastností objektu se uloží ty hodnoty, které jsou zadané jako argumenty.
ř. 14-16: Vytváříme 3 objekty a rovnou každému z nich přiřazujeme jeho správné hodnoty - každé auto má jinou pozici na mapě, jinou barvu a všechny mají stejnou rychlost

Zdrojový kód 34: Metoda __init__()

7.6 Zapouzdření

Velmi často chceme, aby atributy (vlastnosti) měly jen některé hodnoty.
Python jako takový nezná význam našeho programu - a nemá tedy šanci poznat, zda se nám někde neobjevil nesmyslný výsledek, nebo hodnota. Nesmyslný je totiž pouze pro nás (v naší hře). Pro Python je to prostě jen nějaké číslo.
Ještě doplním, že v Pythonu nemá „zapouzdření“ zcela stejný význam jako např, v Javě nebo C++. Přistupujte prosím k této kapitole s rezervou - jde nám hlavně o princip.

V případě aut chceme, aby v atributu „barva“ byly jen názvy barev a v souřadnicích (atributech) „x“ a „y“ nebyly záporná čísla - byly bychom totiž mimo mapu (obrazovku)). Samozřejmě si můžete vymyslet i další omezení.
Tomu, aby auto nevyjelo mimo obrazovku na druhou stranu (vpravo a dolů) se zde zatím věnovat nebudeme. Situace je stejná, jako v případě záporných čísel (vlevo a nahoru), jen musíme znát velikost mapy. Zatím zde žádnou mapu nemáme a tedy ani nevíme její velikost...

Bohužel, není možné někam přímo zapsat, jaké hodnoty chceme v jakém atributu. Musíme zapsat do kódu příkazy, které pohlídají, zda je hodnota taková, jakou chceme.

```
1  class Auto:
2          def __init__(self, zadane_x, zadane_y, zadana_barva, zadana_rychlost):
3                  # self.__x = zadane_x
4                  self.setX(zadane_x)
5
6                  self.y = zadane_y
7                  self.barva = zadana_barva
8                  self.rychlost = zadana_rychlost
9
10         def setX(self, hodnota):
11                 if hodnota < 0:
12                         self.__x = 0
13                 else:
14                         self.__x = hodnota
15
16         def getX(self):
17                 return self.__x
18
19         def jed(self):
20                 # self.__x = self.__x + 1
21                 self.setX(self.__x + 1)
22
23         def vypis_se(self):
24                 print(self.barva , " - x: ", self.__x, " y: ", self.y)
25
26 trabant = Auto(-4, 2, "cervena", 5)
27 skoda = Auto(13, 14, "modra", 5)
28 ferrari = Auto(25, 26, "zelena", 5)
29
30 trabant.jed()
31 trabant.vypis_se()
32 skoda.setX(-78)
33 skoda.vypis_se()
34 x_of_skoda = skoda.getX()
```

Zdrojový kód 35: Zapouzdření

řádek 3: Atribut „schováme“ tak, že začíná na dvě podtržítka - říkáme, že je „private“.

řádek 10: Vytvoříme metodu „setX“ a zavoláme ji pokaždé, když budeme měnit hodnotu „x“.
Tedy se při každé změně spustí příkazy 11-14, které hlídají, správnou hodnotu „x“.

řádek 4, 21, 32: Metodu „setX“ používáme všude, kde měníme hodnotu „x“

34: Kdykoliv potřebujeme přečíst hodnotu „x“ mimo třídu, použijeme metodu „getX, která nám ji vrátí (pošle do místa, kde jsme metodu zavolali)“

7.6.1 Private - skrytí atributu

Aby nebylo možné zapsat do atributu jakoukoliv hodnotu, je potřeba tento atribut schovat. Náš kód může používat i někdo jiný než my - a ten nemusí nic vědět o významu jednotlivých proměnných (atributů) a mohl by do něj napsat nějaký nesmysl. Také se při složitějších operacích může stát, že prostě neuhlídáme hodnotu, kterou do atributu zapisujeme.
Takovému **schování** říkáme, že je atribut **private**. Není dostupný „zvnějšu“ - tedy se k němu nedostaneme mimo kód třídy, ke které atribut patří.
Abychom z atributu veřejného udělali atribut „private“, připíšeme před jeho název dvě podtržítka (např. $x \rightarrow \_\_x$).s

7.6.2 Setter

Hodnotu atributů potřebujeme měnit na spoustě míst v programu - když objekt vytváříme, ve chvíli, kdy se ve hře něco děje (např. má auto popojet). Je proto výhodné zapsat jeden kousek kódu, který umí pohlídat hodnotu atributu a tento kousek kódu použít na všech místech, kde hodnotu měníme. Takovému „kousku kódu“, který lze použít vícekrát, se říká funkce (nebo metoda). Tato funkce má za úkol **nastavit** hodnotu atributu (a pohlídat ji). Proto jí říkáme **setter**. Obvyklé je, že se funkce jmenuje „setNazevAtributu“ - může se samozřejmě jmenovat jakkoliv, ale je lepší dodržet tuto „normu“.

7.6.3 Getter

Protože jsme z atributu udělali atribut „private“ - tedy není dostupný mimo třídu, nemohli bychom se ani podívat, co je v něm uloženo (přečíst ho).
My však potřebujeme znát hodnotu atributů i mimo třídu (abychom mohli auto nakreslit, musíme vědět, kde je).
K **přečtení** hodnoty „private“ atributů slouží metoda, které se říká **getter**. Nedělá nic jiného, než že se podívá na hodnotu atributu a vrátí ho - pošle do místa, kde jsme getter zavolali. Stejně jako u setterů se metoda obvykle nazývá „getNazevAtributu“ - tento název není nutný, ale je nanejvýš vhodné ho dodržet.

7.6.4 Opravdu Pythonovský zápis

Ukázkový kód v úvodu sekce 35 je pouze zjednodušený. Nebylo by totiž příliš pohodlné psát všude „setX“ a „getX“ a navíc má takový kód i další nepěkné vlastnosti. Pro naše použití ve škole postačí, ale pro ty, kteří by chtěli mít kód opravdu Pythonosvky správný přikládám ukázku, jak to udělat doopravdy.

```
1  class Auto:
2          def __init__(self, zadane_x, zadane_y, zadana_barva, zadana_rychlost):
3                  self.x = zadane_x
4
5                  self.y = zadane_y
6                  self.barva = zadana_barva
7                  self.rychlost = zadana_rychlost
8
9          @property
10         def x(self):
11                 return self.__x
12
13         @x.setter
14         def x(self, hodnota):
15                 if hodnota < 0:
16                         self.__x = 0
17                 else:
18                         self.__x = hodnota
19
20         def jed(self):
21                 self.x = self.x + 1
22
23         def vypis_se(self):
24                 print(self.barva , " - x: ", self.x, " y: ", self.y)
25
26 trabant = Auto(-4, 2, "cervena", 5)
27 skoda = Auto(13, 14, "modra", 5)
28 ferrari = Auto(25, 26, "zelena", 5)
29
30 trabant.jed()
31 trabant.vypis_se()
32 skoda.x = -78
33 skoda.vypis_se()
34 x_of_skoda = skoda.x
```

Zdrojový kód 36: Zapouzdření - Pythonovsky

řádek 14: Setter vytvoříme tak, že na řádek před něj zapíšeme „@NazevAtributu.setter“ a můžeme tuto metodu pojmenovat stejně jako atribut.
Tím, se tato metoda spustí pokaždé, když budeme zapisovat do atributu.

řádek 10: Getter vytvoříme tak, že před tuto metody zapíšeme „@property“ a můžeme tuto metodu pojmenovat stejně jako atribut.
Pozor - můsíme ho vytvořit před vytvořením setteru.

řádek 3, 21, 32: Setter voláme jednoduše tak, tako by byl atribut veřejný. Setter se ale spustí.

34: Getter voláme také tak, jako by atribut byl veřejný.

7.7 Dědičnost - inheritance

V běžném životě dělíme objekty okolo nás do kategorii (např. ryby, ptáci, savci jsou zvířata; kočka, člověk, pes jsou savci; muž, žena jsou lidé)
Děláme to proto, že pak můžeme snadněji popsat, jaké vlastnosti a dovednosti má nějaký objekt. Z příkladu o dvě věty zpět je například okamžitě jasné, že muž je savec - a tedy se na něj vztahují všechny vlastnosti savců (např. po narození pijí mateřské mléko).
Pokud rozdělíme objekty do kategorií, nemusíme potom u každého zvlášť vypisovat všechny jeho vlastnosti - stačí, když např. o muži řekneme, že je savec - a hned víme spoustu jeho vlastností

(které si přečteme v knížce o savcích).

Obdobným způsobem můžeme vytvořit třídy (a od těchto tříd objekty) v našem programu. Pokud řekneme, že jedna třída **je potomkem (dědí, je podtřídou)** nějaké jiné třídy, říkáme tím, že všechny vlastnosti (atributy) a dovednosti (metody) **rodiče (nadtřídy)** umí i tato podtřída. Pokud bychom nechali dědit jednu třídu od jiné jedné třídy, žádnou výhodu bychom tím nezískali. Výhodné je ale to, že můžeme nechat od jedné třídy dědit (tedy převzít její vlastnosti) spoustu jiných tříd a naopak, můžeme nechat jednu třídu přejmout vlastnosti od několika jiných.

7.7.1 super()

Často se stane, že chceme využít metodu, kterou umí nadtřída a ještě k ní přidat něco navíc - trochu metodu upravit pro podtřídu - a protože dělá vlastně to samé (např. vypisuje všechny vlastnosti) bylo by fajn, kdyby se jmenovala stejně (např. vypis_se()). Zavolat metodu nadtřídy (metodu, která má stejně pojmenovaného „kamaráda“ v podtřídě) můžeme pomocí metody „super()“.
super() je v podstatě odkaz (ukazatel) na nadtřídu.

```
1  class Vozidlo:
2          def __init__(self, zadane_x, zadane_y, zadana_barva, zadana_rychlost):
3                  self.setX(zadane_x)
4                  self.y = zadane_y
5                  self.barva = zadana_barva
6                  self.rychlost = zadana_rychlost
7
8          def setX(self, hodnota):
9                  if hodnota < 0:
10                         self.__x = 0
11                 else:
12                         self.__x = hodnota
13
14         def getX(self):
15                 return self.__x
16
17         def jed(self):
18                 self.setX(self.__x + 1)
19
20         def vypis_se(self):
21                 print(self.barva , " - x: ", self.__x, " y: ", self.y)
22
23
24 class Auto(Vozidlo):
25         def __init__(self, zadane_x, zadane_y,
26                             zadana_barva, zadana_rychlost, zadany_pocet_dveri):
27                 super().__init__(zadane_x, zadane_y, zadana_barva, zadana_rychlost)
28                 self.pocet_dveri = zadany_pocet_dveri
29
30         def vypis_se(self):
31                 super().vypis_se()
32                 print("Pocet dveri:", self.pocet_dveri)
33
34
35 class Motorka(Vozidlo):
36         def __init__(self,   zadane_x, zadane_y,
37                             zadana_barva, zadana_rychlost, zadany_tlumic):
38                 super().__init__(zadane_x, zadane_y, zadana_barva, zadana_rychlost)
39                 self.tlumic_riditek = zadany_tlumic
40
41         def vypis_se(self):
42                 super().vypis_se()
43                 print("Tlumic riditek:", self.tlumic_riditek)
44
45
46
47 trabant = Auto(4, 2, "cervena", 5, 5)
48 trabant.jed()
49 trabant.vypis_se()
50
51 suzuki = Motorka(10, 50, "modra", 10, True)
52 suzuki.jed()
53 suzuki.vypis_se()
```

Zdrojový kód 37: Dědičnost

řádek 1: Vytvoríme nadtřídu - do ní zapíšeme vše společné pro podtřídy (zde vše, co bylo v předchozích ukázkách).

řádek 24, 35: To, že je Auto Vozidlo (a tedy má převzít všechny vlastnosti Vozidla) řekneme tím, že zapíšeme Vozidlo do kulatých závorek za název podtřídy.

řádek 27, 38, 31, 42: Pomocí metody „super()“ můžeme zavolat metodu z nadtřídy (která se jmenuje stejně, jako metoda v podtřídě). Zde např. při výrobě (__init__()) nejprve vyrobíme Vozidlo a poté přidáme vlastnosti navíc.
Obdobně při výpisu (vypis_se()) - nejprve použijeme metody z nadtřídy a poté vypíšeme informaci navíc.

8 Databáze

Zdrojové kódy převzaty z `https://www.w3schools.com/python/python_mysql_getstarted.asp`, kde najdete i další ukázky.
Pokud chceme, aby náš program zpracovával **velké množství informací** a tyto informace **zůstaly uloženy** i po vypnutí programu (např. více uživatelů hry, skóre) je vhodné tyto informace uložit do souboru nebo v databázi. Výhoda databáze je hlavně ve chvíli, kdy potřebujeme v uložených datech **rychle vyhledávat**, hledat různé **podskupiny** a podobně - přesně k tomu jsou totiž databáze vytvořené a jistě se budou takové operace provádět rychleji, než v našem programu (např. prohledáváním souboru).
Navíc se lze k databázi připojit i pomocí jiných programovacích jazyků a tak půjde naše databáze snadno zobrazit např. v prohlížeči. Nebudeme muset vytvářet speciální kód, který čte náš vlastní soubor.
Zde ve škole můžeme využít školní databázi na serveru `dbs.spskladno.cz` (MySQL)- pro přístup přes prohlížeč použijte `http://dbs.spskladno.cz/myadmin/` . Přístupové údaje získáte od svých vyučujících.
Způsob jakým se přistupuje k databázi závisí na použité knihovně - zde si ukážeme práci s **knihovnou mysql-connector**

8.1 Přístupové údaje

Pro připojení k databázi (přesněji k databázovému serveru) je potřeba zadat údaje:

adresa serveru ip, nebo url adresa - dbs.spskladno.cz

jméno získáte od vyučujících

heslo získáte od vyučujících

pro připojení ke konkrétní databázi navíc zadáme název databáze:

název databáze zde na školním databázovém serveru závisí na účtu, pod kterým se k serveru přihlásíme

Pomocí výše uvedených údajů vytvoříme spojení našeho php skriptu s databází. Přes toto spojení můžeme zasílat do databáze požadavky pomocí jazyku SQL.

```
1  import mysql.connector
2
3  mydb = mysql.connector.connect(
4    host="dbs.spskladno.cz",
5    user="dotaz_na_vyucujiciho",
6    passwd="dotaz_na_vyucujiciho",
7    database="vyuka__"
8  )
9  mycursor = mydb.cursor()
10
11 mycursor.close()
12 mydb.close()
```

Zdrojový kód 38: Připojení k databázi

řádek 1: Import knihovny mysql-connecor

řádek 3: Vytvoření spojení pomocí adresy serveru, přístupových údajů, případně i konkrétní databáze

řádek 9: Vytvoření kurzoru. Kurzor je „ukazatel“ do databáze, pomocí kterého v ní budeme spouště dotazy a získávat výsledky.

řádek 11: Uzavření kurzoru a odpojení spojení s databází.

8.2 Dotazy, které mění data v databázi

U některých dotazů do databáze nezískáváme žádný výsledek, ale **měníme** samotnou **databázi** (např. vytvoření tabulky, vložení dat do tabulky).

Takové dotazy se neprovedou přímo, ale pouze se „připraví“ a **databáze čeká na potvrzení**, že se mají data skutečně změnit. Tomuto potvrzení říkáme „**commit**“.

```
1  import mysql.connector
2
3  mydb = mysql.connector.connect(
4    host="dbs.spskladno.cz",
5    user="dotaz_na_vyucujiciho",
6    passwd="dotaz_na_vyucujiciho",
7    database="vyuka__"
8  )
9  mycursor = mydb.cursor()
10
11 sql_create = 'CREATE TABLE Uzivatel (jmeno TEXT, prijemni TEXT)'
12 sql_insert = 'INSERT INTO Uzivatel (jmeno, prijmeni)
13                            VALUES ("Jarda", "Vomáčka"), ("Tonda", "Kobliha")'
14
15 mycursor.execute(sql_create)
16 mycursor.execute(sql_insert)
17 mydb.commit()
18
19 mycursor.close()
20 mydb.close()
```

řádek 11: Příprava dotazů pro vytvoření tabulky a vložení dvou uživatelů.

řádek 15: Spuštění dotazů. Dotazy se pouze „připraví“, protože mění data v databázi.

řádek 17: Potvrzení spuštěných dotazů. Teprve v tuto cvhíli se dotazy projeví v databázi - ve stejném pořadí, v jakém jsme je spustili.

Zdrojový kód 39: Tvorba tabulky a vložení dat

8.3 Dotazy s výsledkem

U některých dotazů získáváme z databáze informaci (proto databázi máme . . .).
Výsledkem dotazu může být velmi mnoho řádků (např. máme mnoho uživatelů) a bylo by značně náročné, přenášet celý výsledek najednou.
V následující ukázce si můžeme představit, že po odeslání dotazu se v databázi **dočasně vytvoří** náš **výsledek** v našem programu se v kurzoru uloží **ukazatel** na tyto data (nepřenáší se tedy data samotná, ale pouze informace, kde v databázi jsou). **Část dat** (tolik, kolik i v programu vyžádáme) **se přenese** až **po zavolání funkce**:

fetchone() přenese jeden řádek výsledku
do datového typu **tuple**

fetchmany(kolik) přenese tolik řádků, kolik zapíšeme do argumentu
do datového typu **list** (pole). Každý prvek pole je poté jeden řádek výsledku (datový typ tuple).

fetchall() přenese všechny řádky výsledku
do datového typu **list** (pole). Každý prvek pole je poté jeden řádek výsledku (datový typ tuple).

Po stažení části výsledku se kurzor posune, takže pokud zavoláme funkci pro stažení výsledku znovu - dostaneme následující část (řádky).

```
1  import mysql.connector
2
3  mydb = mysql.connector.connect(
4    host="dbs.spskladno.cz",
5    user="dotaz_na_vyucujiciho",
6    passwd="dotaz_na_vyucujiciho",
7    database="vyuka__"
8  )
9  mycursor = mydb.cursor()
10
11 sql_select = 'SELECT jmeno, prijmeni FROM Uzivatel'
12
13 mycursor.execute(sql_select)
14
15 ziskano = mycursor.fetchone()
16 krestni, prijmeni = ziskano
17 print(krestni, "se jmenuje ", prijmeni)
18
19 ziskano = mycursor.fetchmany(2)
20
21 for radek in ziskano:
22         krestni, prijmeni = radek
23         print(krestni, "se jmenuje ", prijmeni)
24
25 mycursor.close()
26 mydb.close()
```

řádek 15: Stáhneme si požadovanou část výsledku. Zde např. 1 řádek přímo do tuple.

řádek 19: Stáhneme si požadovanou část výsledku. Zde např. 2 řádky do pole (prvky pole jsou jednotlivé řádky - tuple).

řádek 16, 22: Rozdělení tuple do jednotlivých dat (sloupců).

řádek 21: Pokud máme výsledek v poli, můžeme postupně projít všechny prvky pomocí for-cyklu.

Zdrojový kód 40: Select - dotaz s výsledkem

8.4 Zpracování chyb

Při přístupu k databázi může vzniknout spoustu chyb - odpojený přístup k internetu, špatně zadaný dotaz, nemáme práva k požadované operaci v databázi, ... Takové chyby zpracováváme stejně, jako jakékoliv jiné „chyby za běhu - RunTimeError“ v Pythonu.
Kritickou část kódu (tu, kde tušíme možný problém) **uzavřeme do bloku „try: ... except:“** a takzvaně **ošetříme** možnou **chybu**. etření znamená, že zapíšeme, co se má provést ve chvíli, kdy chyba nastane.

```
1  import mysql.connector
2
3  try:
4    mydb = mysql.connector.connect(
5      host="dbs.spskladno.cz",
6            user="dotaz_na_vyucujiciho",
7            passwd="dotaz_na_vyucujiciho",
8            database="vyuka__"
9    )
10 except mysql.connector.Error as chyba:
11   print(chyba)
12   print("Error Code:", chyba.errno)
13   print("SQLSTATE:", chyba.sqlstate)
14   print("Message:", chyba.msg)
15
16 mycursor = mydb.cursor()
17
18 mycursor.close()
19 mydb.close()
```

Zdrojový kód 41: Zachytávání chyb - try-except

řádek 3: Začátek bloku příkazu, ve kterých tušíme možou chybu
zde připojení k databázi.

řádek 10: zapíšeme kterou chybu chceme zachytit
zde všechny chyby „Error“, které jsou připraveny v knihovně „mysql.connector“
Pro další příkazy si chybu pojmenujeme - zde „chyba“

řádek 11: Do bloku „except“ zapíšeme, co se má stát ve chvíli, kdy daná chyba nastane. Těchto bloků (pro různé typy chyb) může být kolik chceme.

9 Grafika

„Nakreslit“ obrázek na monitor počítače není tak jednoduchý úkol, jak se na první pohled může zdát. Proto již spoustu hodných programátorů vytvořilo knihovny (soubory, ve kterých jsou předpřipravené funkce), které můžeme jednoduše používat a které udělají spoustu těžké práce za nás. I tak, ale musíme těmto připraveným funkcím říct, co od nich vlastně chceme.
Grafických knihoven je poměrně velké množství - některé se specializují na 2D, některé na 3D, na grafy, ...
Ve škole budeme používat knihovny PyQt, PyGame a PyQtGraph.

V testech se nebude očekávat, že znáte konkrétní funkce - jejich názvy a použití
Jsou ale **obecné principy**, které se pro zobrazování grafiky používají - a to je to, co **máte za úkol se naučit**. V níže uvedených příkladech se proto zaměřte na to, **jak to funguje** a ne na to, co přesně je na kterém řádku napsané.

9.1 PyQt

Pomocí knihovny PyQt můžeme vytvářet grafické (klikací) aplikace. Vytvářet budeme především, okna, tlačítka, nápisy a vstupní políčka.

9.1.1 Odkazy

Ukázky jsou převzaty především z `https://naucse.python.cz/lessons/intro/pyqt/`.

9.1.2 Aplikace

Před vytvořením hlavního okna musíme nejprve „vytvořit“ samotnou aplikaci - bude to objekt, který si pamatuje sposutu informací (s jakými argumenty byla aplikace spuštěna, s jakým nastavením, které okno je zrovna aktivní, ...). Provedem to pomocí „QtWidgets.QApplication([])“.
Celá aplikace se spustí, po zavolání funkce „exec()“. Do té doby budou všechny úpravy, které provedeme skryty.

9.1.3 Hlavní okno

Dále si vytvoříme samotné hlavní okno, do kterého budeme vkládat další grafické prvky (tlačítka, nápisy, ...). To uděláme pomocí „QtWidgets.QWidget()“.
Toto hlavní okno (stejně jako jakýkoliv jiný prvek do kterého chceme něco vložit) musí vědět, jakým způsobem do něj chceme další prvky vkládat (vertikálně (svisle), horizontálně (vodorovně), do mřížky, ...).
Způsob vkládání nastavíme pomocí tak zvaného „Layout managera“ - např. „QtWidgets.QHBoxLayout()“ pro horizontální uspořádání.
Toto okno (a všechny jeho potomci - prvky, které jsme do něj vložili) se zobrazí až ve chvíli, kdy na něj zavoláme funkci „show()“

9.1.4 Typy prvků

Prvků, které můžeme vládat je velké množství a v případně, že máte představu, co byste chtěli, podívejte se do dokumentace.
Zde si ukážeme některé z nich:

QtWidgets.QLabel('Napis'**)** je pole, do kterého můžeme zapisovat nápisy. Tyto nápisy nemůže uživatel přímo měnit.

QtWidgets.QPushButton('Klik'**)** je tlačítko, na které může uživatel kliknout (např. myší).

QtWidgets.QHBoxLayout() je prvek (přesněji zvláštní typ prvku - layout), který umožňuje uspořádat ostatní prvky - tento konkrétní Horizontálně (proto má v názvu Q**H**BoxLayout).

QtWidgets.QLineEdit() je pole, do kterého může uživatel cokoliv zapsat. To, co je momentálně zapsané v tomto poli přečteme pomocí **vstupni_text = nazev_vstupniho_pole.text()**

```
1  from PyQt5 import QtWidgets
2
3  app = QtWidgets.QApplication([])
4
5  hlavni_okno = QtWidgets.QWidget()
6  hlavni_okno.setWindowTitle('Můj supr program')
7
8  usporadani = QtWidgets.QHBoxLayout()
9  hlavni_okno.setLayout(usporadani)
10
11 napis = QtWidgets.QLabel('Tlačítko nic nedělá...')
12 usporadani.addWidget(napis)
13
14 tlacitko = QtWidgets.QPushButton('Klikni na mě')
15 usporadani.addWidget(tlacitko)
16
17 hlavni_okno.show()
18 app.exec()
```

Zdrojový kód 42: PyQt - první okno

řádek 1: Importujeme používané knihovny

řádek 3: Vytvoříme hlavní objekt naší aplikace

řádek 5: Vytvoříme první Widget - hlavní okno.

řádek 6: Pomocí funkcí můžeme měnit jednotlivé vlastnosti prvků (Widetů)

řádek 8: Vytvoříme prvek pro uspořádání ostatních prvků

řádek 9: Nastavíme, které uspořádání chceme použít (každé okno může mít jiné uspořádání)

řádek 11: Vytvoříme prvek - nápis

řádek 14: Vytvoříme prvek - tlačítko

řádek 12, 15: Vložíme prvky do připraveného prvku pro uspořádání (aplikace by jinak nevěděla, v jakém okně, v jaké části) chceme prvky mít

řádek 17: Zobrazíme hlavní okno - a s ním všechny jeho potomky (prvky, které jsou v něm)

řádek 18: Spustíme aplikaci

9.1.5 Funcionalita

Samořejmě chceme, aby se po kliknutí na tlačítko (nebo jiné akci) provedla nějaká operace a její výsledek se zobrazil uživateli.
Abychom toho docílili, vytvoříme funkci, do které zapíšeme požadovanou operaci a tuto funkci napojíme na tlačítko - konkrétně na zmáčknutí tlačítka.

```
1  from PyQt5 import QtWidgets
2
3  app = QtWidgets.QApplication([])
4
5  hlavni_okno = QtWidgets.QWidget()
6  hlavni_okno.setWindowTitle('Můj supr program')
7
8  usporadani = QtWidgets.QHBoxLayout()
9  hlavni_okno.setLayout(usporadani)
10
11 napis = QtWidgets.QLabel('Teď už tlačítko funguje...')
12 usporadani.addWidget(napis)
13
14 tlacitko = QtWidgets.QPushButton('Klikni na mě')
15 usporadani.addWidget(tlacitko)
16
17 vstup = QtWidgets.QLineEdit()
18 usporadani.addWidget(vstup)
19
20 def zmen_napis():
21         vstupni_text = vstup.text()
22         napis.setText( "Toto jsi napal: " + vstupni_text )
23
24 tlacitko.clicked.connect(zmen_napis)
25
26 hlavni_okno.show()
27 app.exec()
```

řádek 20: Připravíme si funkci, která se má spustit po kliknutí na tlačítko

řádek 24: Připravenou funkci (pozor! bez kulatých závorek) připojíme na naše tlačítko při kliknutí.

Zdrojový kód 43: PyQt - Zkopíruj nápis

9.2 PyGame

9.2.1 Havní smyčka

Jednou z nejpodstatnějších částí grafické aplikace je takzvaná hlavní smyčka. Celý běh aplikace je zpracováván ve smyčce (while cyklu) ve které se stále dokola provádí ty stejné operace - po provedení všech operací (jednoho kroku) se vše opakuje znovu.
Samozřejmě, že chceme, aby se v průběhu programu jeho výstup měnil (jednou jede auto doleva, jednou doprava) podle toho, co uživatel udělá (např. zmáčkne klávesu). To je zajištěno takzvanými **událostmi**. Události jsou téměř vše, co vás napadne - stisk klávesy, klik myši, pohyb myši, můžeme ve hře vytvořit i naše vlastní libovolné události. Podle toho, jaká událost nastala vykonáme v kódu různé příkazy - a tím změníme výstup programu.

9.2.2 Animace - pohyb

Při zobrazování na monitor se může zobrazit vždy jen jeden konkrétní (pevný) obrázek. Jak tedy zajistit, aby to vypadalo, že se naše autíčko pohybuje?
V každém kroku smyčky smažeme předchozí obrázek auta a nakreslíme ho znovu o malou vzdálenost vedle.
Vzdálenost, o kterou obrázek posouváme, nastavíme dostečně malou a rychlost, jakou obrázek překreslujeme nastavíme dostatečně vysokou. Tím vytvoříme „optický klam“ plynulého pohybu.

```
1  import pygame
2  from pygame.locals import *
3
4  display_width = 800
5  display_height = 600
6
7  screen = pygame.display.set_mode((display_width, display_height))
8  pygame.display.set_caption('Super hra')
9
10 black = (0, 0, 0)
11 white = (255, 255, 255)
12
13 CarPos = [50, 100]
14 CarRadius = 10
15
16 hodiny = pygame.time.Clock()
17 cekej_ms = 60
18
19 def draw_car(S, r):
20     pygame.draw.circle(screen, black, S, r , 0)
21     pygame.draw.line(screen, white, S, (S[0], S[1]+r), 3 )
22
23 pygame.key.set_repeat(int(round(cekej_ms/2)))
24 running = True
25 while running:
26     for event in pygame.event.get():
27         if event.type == pygame.QUIT:
28                     running = False
29
30         if event.type == KEYDOWN:
31             if event.key == K_a:
32                 CarPos[0] = CarPos[0] - 5
33             elif event.key == K_d:
34                 CarPos[0] = CarPos[0] + 5
35
36     screen.fill(white)
37     draw_car(CarPos, CarRadius)
38
39     hodiny.tick(cekej_ms)
40     pygame.display.update()
41
42 pygame.quit()
```

Zdrojový kód 44: První ukázka „hýbací“ grafiky

řádky 1, 2: Importujeme knihovny, které chceme používat.

řádky 4, 5: Uložíme si, jak velké chceme okno naší aplikace.

řádek 7: Vytvoříme hlavní okno („plátno“) naší aplikace. Vše, co nakreslíme na toto „plátno“ se zobrazí na monitoru.

řádky 10, 11, 13, 14: Uložíme si informace, které budeme používat - bílou a černou barvu, pozici a velikost kolečka (v této ukázce je kolečko naše auto.)

řádek 16: V naší hře budeme také chtít měřit čas - proto si vytvoříme hodiny.

řádek 19: Funkce, která nám na zadaném místě (S) nakreslí kolečko o zadaném poloměru (r)

řádek 23: Nastavení, aby se po zmáčknutí a držení klávesy, stále dokola odesílala událost (informace) o tom, že je klávesa zmáčknutá.

řádek 25: Hlavní smyčka hry.

řádek 26: Smyčka, ve které se zpracují všechny události - např. kliknutí na zavírací křížek ř: 27, nebo zmáčknutí klávesy ř: 30.

řádky 31, 33: Zjšitění, která klávesa je zmáčknutá.

řádek 36: Vybarvení celého plátna bílou barvou (tedy smažeme vše, co je na plátně nakreslené)

řádek 37: Zakreslíme „auto“ na aktuální pozici.

řádek 39: Každý krok hry bude trvat stanovený čas - zde 60ms (včetně zpracování všech předchozích příkazů)

řádek 40: Všechny provedené změny zobrazíme na monitor.

9.2.3 Blit - chytřejší vykreslování

Zobrazení nějaké změny na monitor (například změna barvy pozadí, smazání a zobrazení kolečka apd.) je jedna z „nejdražších“ operací v grafickém programu. Její provedení trvá velmi dlouho - je do ní totiž zapojeno mnoho součástek počítače a i samotný výpočet, kde se má co zobrazit je poměrně náročný. Při takovéto „drahé“ operaci se „každý pixel počítá“. Chceme zkrátka kreslit na obrazovku co nejméně.
Při pohledu na řádek ř. 36 v kódu 44 si jistě domyslíme, že **celou plochu** přebarvíme na bílo. Měníme tedy **všechny pixely** naší hry (program nezajímá, co za barvu bylo na nějakém pixelu doposud - provede příkaz, který jsme zadali a to je „nakresli sem bílou“). To je extrémně neefektivní, protože drtivá většina pixelů bude mít stejnou barvu jako předtím.
K výrazně efektivnějsímu kódu slouží v knihovně PyGame funkce „blit()“

Funkci **blit()** si můžeme představit jako Ctrl+C a Ctrl+V. Abychom ji ale mohli používat, musíme nejprve mít odkud a kam kopírovat. K tomu nám poslouží Surface.

Surface - plátno V podstatě vše grafické v PyGame je plátno - Surface. Surface je načtený obrázek, surface může být libovolný geometrický tvar a těchto **pláten můžeme mít** v našem programu **kolik chceme**. Jedno surface je **speciální** - to, co na něj nakreslíme **se zobrazí na monitoru** - toto plátno, vytvoříme pomocí příkazu:

```
1 screen = pygame.display.set_mode((display_width, display_height))
```

Všechna ostatní sice máme v programu a můžeme s nimi libovolně pracovat, ale to, co s nimi provedeme se na monitoru neukáže.

Důležité na surface je, že každé má svůj **obdélník (rectangle)**, který ho **ohraničuje**. Tento obdélník není nijak pootočený - jinými slovy je to obdélník od bodu nejvíce vlevo k bodu nejvíce vpravo a od bodu nejvíce nahoře k bodu nejvíce dole. Tento obdélník je v PyGame uložený jako **pozice x-y levého horního rohu, šířky a výšky** obdélníku.

blit() Blit si můžeme přestavit jako Ctrl+C a Ctrl+V. Z jednoho plátna (surface) zkopírujeme pixely ze zadaného místa (zadaného pozicí levého horního rohu, šířkou a výškou - tedy obdélníkem) „area“ a vložíme je na zadané místo (zadaného pozicí levého horního rohu) do druhého plátna „dest“.

```
1 platno_kam.blit(plano_odkud, dest=pozice_kam, area=pozice_a_velikost_odkud)
```

```
1  import pygame
2  from pygame.locals import *
3
4  pygame.init()
5  display_width = 800
6  display_height = 600
7
8  screen = pygame.display.set_mode((display_width,display_height))
9
10 pygame.display.set_caption('Super hra')
11
12 black = (0, 0, 0)
13 white = (255, 255, 255)
14 red = (255, 0, 0)
15
16 CarPos = [50, 100]
17 CarRadius = 10
18 CarColor = black
19
20 hodiny = pygame.time.Clock()
21 cekej_ms = 60
22
23 def draw_car(S, r):
24     changed_rect = pygame.draw.circle(screen, CarColor, S, r, 0)
25     pygame.draw.line(screen, red, S, (S[0], S[1] + r - 1), 3)
26
27     return changed_rect
28
29
30 pozadi = pygame.Surface((display_width, display_height))
31 pozadi.fill(white)
32 pozadi_rect = pozadi.get_rect()
33
34 screen.blit(pozadi, dest=pozadi_rect, area=pozadi_rect)
35
36 car_rect = draw_car(CarPos, CarRadius)
37
38 pygame.key.set_repeat(int(round(cekej_ms / 2)))
39 running = True
40 while running:
41     for event in pygame.event.get():
42         if event.type == pygame.QUIT:
43             running = False
44
45         if event.type == KEYDOWN:
46             if event.key == K_a:
47                 CarPos[0] = CarPos[0] - 5
48             elif event.key == K_d:
49                 CarPos[0] = CarPos[0] + 5
50
51     screen.blit(pozadi, dest=car_rect, area=car_rect)
52     car_rect = draw_car(CarPos, CarRadius)
53
54     pygame.display.flip()
55
56     hodiny.tick(cekej_ms)
57
58 pygame.quit()
```

řádek 8, 30: Vytvoříme dvě plátna. Jedno hlavní, jedno s pozadím.

řádek 24, 27: Při kreslení auta si uložíme kam (do jakého obdélníku) jsme ho nakreslili. Tento obdélník nám bude kreslící funkce vracet.

řádek 31, 32: Pozadí vybarvíme na bílo. A uložíme si jeho obdélník.

řádek 34: Pozadí zkopírujeme na hlavní plátno.

řádek 36: Nakreslíme auto - a uložíme si obdélník, kam jsme ho nakreslili.

řádek 51: Překreslíme na hlavní plátno tu část pozadí, která odpovídá nakreslenému autu. Tedy v podstatě smažeme auto.

řádek 52: Nakreslíme auto na aktuální pozici - a uložíme si jeho obdélník.

řádek 54: Zobrazíme změny, které jsme provedli v tomto kroku.

Zdrojový kód 45: Grafika s blit()