

Úvod

1. [Iterační protokol podruhé](#),
2. [while smyčka](#),
3. [nekonečná smyčka](#),
4. [operátor mrože?!](#),
5. [Zkrácené přiřazování](#),
6. [souhrn smyček](#),
7. [velmi použitelné prvky iterací](#),
8. [domácí úkol](#).



✓ Iterační protokol, podruhé

Pro periodické opakování ohlášení existují tzv. *iterační protokoly* (příp. označovány jako *smyčky*, *cykly*, *loopy*).

Pomocí smyčky *for* umíš zapsat takovou *iteraci*, kdy postupně projdeš **všechny hodnoty**.

Co když budeš potřebovat *iterovat* bez zadané hodnoty, ale za jistých podmínek?

Až bude mít `list` 3 hodnoty, dokud uživatel zadává vstupy, atd.

Potom bude potřeba, povědět si ještě o druhém typu smyček:

1. smyčka `for`,
2. smyčka `while`.



✓ While smyčka

Někdy ale není nutné *iterovat* přes **celý objekt**, jak tomu bylo u smyčky `for`.

Naopak, budeš potřebovat provádět proces *iterování* tak dlouho, *dokud* to bude nutné.

Za takovým účelem můžeš využít druhý typ smyček, `while`.

✓ Obecně while loop

```
index = 1
```

```
while index < 6:  
    print("Ještě nemáš 6, ale ", index, ", pokračuji..", sep="")  
    index = index + 1
```

```
print("Hotovo, máš 6!")
```

```
➡ Ještě nemáš 6, ale 1, pokračuji..  
Ještě nemáš 6, ale 2, pokračuji..  
Ještě nemáš 6, ale 3, pokračuji..  
Ještě nemáš 6, ale 4, pokračuji..  
Ještě nemáš 6, ale 5, pokračuji..  
Hotovo, máš 6!
```

1. `while` je **klíčkové slovo** v záhlaví,
2. `index < 6` je **podmínka**. Pokud je vyhodnocená jako `True`, proved' *odsazené ohlášení*,
3. `index < 6 ... False`, ukonči smyčku a pokračuj **s neodsazeným zápisem** pod smyčkou,
4. `:` řádek s předpisem musí být **zakončený dvojtečkou**,
5. `print("Ještě nemáš...")`, následují *odsazené ohlášení*, které se budou opakovat v každém kroku,
6. `print("Hotovo, " ...)`, pokračuje *neodsazený zápis*, pod smyčkou.

✓ While s doplňující podmínkou

Cyklus `while` samotný podmínku obsahuje. Určitě je ale možnost, tento podmínkový strom ještě rozšířit:

```
index = 0
```

```
while index <= 20:
    if len(str(index)) != 2:
        index = index + 1
    else:
        print(index)
        index = index + 1
```

```
→ 10
   11
   12
   13
   14
   15
   16
   17
   18
   19
   20
```

Takové rozšíření může být obzvlášť přínosné, pokud podmínku v předpise nelze jednoduše rozšířit:

```
index = 0
```

```
while index < 20 and len(str(index)) == 2:
    print(index)
    index = index + 1
```

✓ While/else

Cyklus `while` lze rozšířit o podmínkovou větev `else` (podobně jako *for loop*).

K ní se *interpret* dostane, pokud je podmínka v předpisu vyhodnocená jako `False`.

Současně nesmí narazit na ohlášení `break`:

```
index = 0
```

```
while index < 20:
    if len(str(index)) != 2:
        index = index + 1

    else:
        print(index)
        index = index + 1
```

```

else:
    print("-" * 23, "Podmínka -> False".center(23), "-" * 23, sep="\n")

print(">Pokračuji pod smyčkou<")

```

```

➞ 10
   11
   12
   13
   14
   15
   16
   17
   18
   19
   -----
   Podmínka -> False
   -----
   >Pokračuji pod smyčkou<

```

Pokud doplníš ohlášení `break`, *interpret* přeskočí nejenom zbytek smyčky *while* ale také větev `else`:

```

index = 0

while index < 20:
    if len(str(index)) != 2: # pokud není číselná hodnota ze dvou znaků
        index = index + 1

    else:
        print(index)
        index = 1
        break

else:
    print("-" * 23, "Podmínka -> False".center(23), "-" * 23, sep="\n")

print(">Pokračuji pod smyčkou<")

```

```

➞ 10
   >Pokračuji pod smyčkou<

```



✓ Nekonečný while loop

Jednou z aplikací smyčky `while` je zápis tzv. *nekonečného cyklu*.

Obecně řečeno, že v případě **nekonečných smyček** můžeš potkat dva typy:

1. **řízené** nekonečné smyčky,
2. **neřízené** nekonečné smyčky.

✓ Neřízené nekonečné smyčky

Ty mohou nastat v důsledku **špatného zápisu** `while` cyklu:

```
index = 1

while index < 20:
    print(index)
    # neinkrementuji hodnotu v proměnné 'index'
    # .. hodnota je v každém kroce 1 a smyčka nekončí.
    # .. Ctrl + C ->
```

poznámka. výše ukázaná varianta představuje tzv. *nežádoucí nekonečnou smyčku*, kde vznikla chyba **v odsazené části zápisu**.

Chyba ovšem může nastat i při **špatném ohlášení** v zadání smyčky *while*:

```
index = 1

while index > 0: # vyhodnocené ohlášení má stále hodnotu `True`
    print(index)
    index = index + 1
```

poznámka. výše ukázaná varianta představuje tzv. *nežádoucí nekonečnou smyčku*, kde vznikla chyba **ve špatně zapsané podmínce**.

✓ Řízené nekonečné smyčky

Nekonečný cyklus s `while` je možné formulovat jako *řádnou/žádoucí nekonečnou smyčku*.

```
while True:
    uziv_vstup = input("Zapiš libovolný text [nebo 'q' pro ukončení]: ")

    if uziv_vstup == "q":
        break
    print(uziv_vstup.capitalize())

print("Ukončuji ukázkou!")
```

```
➞ Zapiš libovolný text [nebo 'q' pro ukončení]: ahoj
Ahoj
Zapiš libovolný text [nebo 'q' pro ukončení]: matouš
Matouš
Zapiš libovolný text [nebo 'q' pro ukončení]: 1
1
Zapiš libovolný text [nebo 'q' pro ukončení]: @
@
Zapiš libovolný text [nebo 'q' pro ukončení]: q
Ukončuji ukázkou!
```

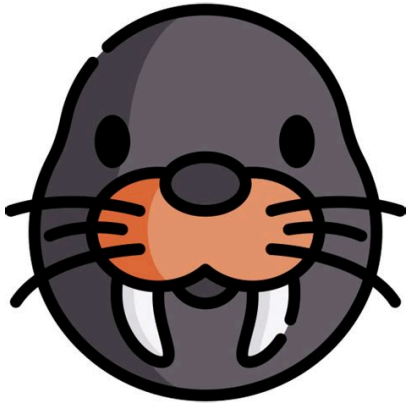
```
switch = True

while switch:
    uziv_vstup = input("Zapiš libovolný text [nebo 'q' pro ukončení]: ")

    if uziv_vstup == "q":
        switch = False
    print(uziv_vstup.capitalize())

print("Ukončuji ukázkou!")
```

```
➞ Zapiš libovolný text [nebo 'q' pro ukončení]: ahoj
Ahoj
Zapiš libovolný text [nebo 'q' pro ukončení]: p
P
Zapiš libovolný text [nebo 'q' pro ukončení]: x
X
Zapiš libovolný text [nebo 'q' pro ukončení]: q
Q
Ukončuji ukázkou!
```



✓ Walrus operátor

Přiřazovací operátor nebo jinak *walrus operátor* je formulace, která je v Pythonu poměrně nová (3.8+).




Jde o zápis, který ti umožní **dva procesy**, při použití **jednoho operátoru**:

1. nejprve **hodnotu přiřadí** proměnné,
2. přímo ji použije.


✓ Vytvoření hodnoty a uložení

```
jmeno = "Matous"
```

```
print(jmeno)
```

 Matous

```
print(jmeno := "Matous")
```

 Matous

V předchozí ukázce jde čistě o **vysvětlivku**.


Proměnné jinak nadále a přehledně zapis po jedné a pod sebe. :)

Praktické ukázky skutečného využití najdeš níže.

✓ Kombinace s podmínkou

```
jmeno = input("Zapiš jméno: ".upper())
```

```
if jmeno == "Matouš":
    print("Toto je ", jmeno, sep="")
else:
    print("Tak ", jmeno, ", toho neznám.", sep="")
```


 ZAPIŠ JMÉNO: Marek
Tak Marek, toho neznám.

Obzvlášť v kombinaci se **zabudovanými funkcemi** a *uživatelskými funkcemi* je nápomocný.

Zásadní je **doplnění kulatých závorek**, kterými *interpretu* zdůrazníš pořadí:

```
TEXT = "Zapiš jméno: ".upper()


if (jmeno := input(TEXT)) == "Matouš":
    print("Toto je ", jmeno, sep="")
else:
    print("Tak ", jmeno, ", toho neznám.", sep="")
```

 ZAPIŠ JMÉNO: Matouš
Toto je Matouš

Pokud zapomeneš kulaté závorky, ohlášení **nemusí logicky pracovat**:

```
TEXT = "Zapiš jméno: ".upper()

if jmeno := input(TEXT) == "Matouš":
    print("Toto je ", jmeno, sep="")
else:
    print("Tak ", jmeno, ", toho neznám.", sep="")
```

 ZAPIŠ JMÉNO: Marek
Tak Marek, toho neznám.

Copak se stane:

1. *Interpret* nejprve uloží vstup do funkce `input()`,
2. následně vloženou hodnotu porovná,
3. výsledek (`True / False`) uloží do proměnné `jmeno`.

✓ Kombinace s cyklem `while`

```
TEXT = "Zapiš libovolný text [nebo 'q' pro ukončení]: "
```

```
while (vstup := input(TEXT)) != "q":
    print(vstup)
else:
    print(vstup, "Konec smyčky!", sep="\n")
```

```
➞ Zapiš libovolný text [nebo 'q' pro ukončení]: Ahoj
Ahoj
Zapiš libovolný text [nebo 'q' pro ukončení]: tak
tak
Zapiš libovolný text [nebo 'q' pro ukončení]: mame
mame
Zapiš libovolný text [nebo 'q' pro ukončení]: druhou
druhou
Zapiš libovolný text [nebo 'q' pro ukončení]: hodinu
hodinu
Zapiš libovolný text [nebo 'q' pro ukončení]: q
q
Konec smyčky!
```

Analogicky můžeš opsat celý postup také **bez přiřazovacího operátoru**:

```
vstup = ""
TEXT = "Zapiš libovolný text [nebo 'q' pro ukončení]: "

while vstup != "q":
    vstup = input(TEXT)
    print(vstup)

else:
    print("Konec smyčky!")
```

```
➞ Zapiš libovolný text [nebo 'q' pro ukončení]: a
a
Zapiš libovolný text [nebo 'q' pro ukončení]: ahoj
ahoj
Zapiš libovolný text [nebo 'q' pro ukončení]: b
b
Zapiš libovolný text [nebo 'q' pro ukončení]: @
@
Zapiš libovolný text [nebo 'q' pro ukončení]: q
```

q
Konec smyčky!

Pokud ti tedy dovede operátor `:=` šikovně pomoci, **určitě jej využij**.

Není nutné jej **zneužívat** v situacích, kdy je zápis málo čitelný, nebo špatně pochopitelný.

Opatrně na **verze**.

Pokud vyvíjíš na jiném prostředí, než produkčním, můžeš zjistit, že **má starší verzi Pythonu** a *walrus* nemusí podporovat.



✓ Zkrácené přiřazování

Jde o **kratší způsob** pro úpravu hodnoty v existující proměnné.

Efektivnější není jen způsob zápisu, ale také způsob zpracování.

Doposud znáš tento zápis:

1. **Vytvoříš hodnotu** v proměnné `x`,
2. **upravíš hodnoty** v proměnné `x`,
3. **použiješ** novou hodnotu `x`.

```
x = 2
```

```
x = x + 3
```

```
print(x)
```

```
⇒ 5
```

✓ Augmented assignment

Zkrácená varianta vypadá tak, že původní proměnnou `x` nepoužiješ a aritmetický operátor přesuneš přes rovnítko:

```
x = 2
```

```
x += 2
```

```
print(x)
```

```
↩ 4
```

✓ Rozdíl

Pro tebe, jako uživatele, je tento zápis pouze o něco kratší.

Vypadá odlišně, jinak je stejně zapsaný pomocí 3 řádků.

V čem je tedy lepší?

Lepší je z hlediska využití paměti.

✓ Klasický zápis rozdělený na jednotlivé kroky:

1. Vytvoříš **novou hodnotu** a **uložíš ji** do proměnné,
2. **načteš** původní hodnotu,
3. **zvětšíš ji** o hodnotu 4,
4. **uložíš** novou hodnotu,
5. **vypíšeš ji**.

```
x = 2
```

```
x = x + 3
```

```
print(x)
```

```
↩ 5
```

✓ Ve zkráceném zápise:

1. Vytvoříš **novou hodnotu** a **uložím ji** do proměnné,
2. **zvětšíš existující hodnotu**,
3. **vypíšeš ji**.

```
x = 2
```

```
x += 2
```

```
print(x)
```

```
↔ 4
```

Některé zkrácené operátory

Původní operátor	Zkrácená varianta
+	+=
-	-=
*	*=
/	/=
**	**=



✓ Souhrn úvodu smyček

Nyní máš za sebou stručný úvod do problematiky **iterátorů**. Jaké pojmy jsou tedy zásadní.

Iterable

Anglické ozn., které představuje **takový objekt**, který umí vytvořit *iterátor* (pomůcka zab. funkce `iter()`).

Iterator

Anglické ozn., které představuje tzv. **iterátor**. Tedy objekt, který dovede podávat jednotlivé hodnoty (pomocí funkce `next()`).

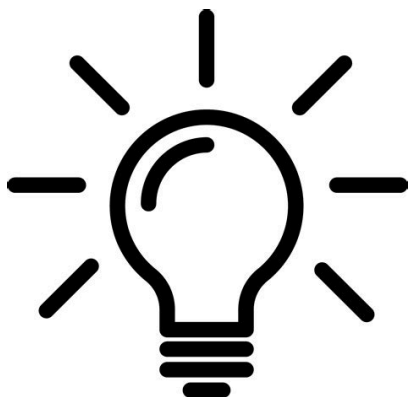
Iteration

Anglické ozn., které představuje proces **iterace**. Což je proces, který postupně prochází hodnoty. Krok za krokem.

For vs. while smyčka

Kdy máš vybrat co, lze popsat jako:

1. Pokud potřebuješ *iterovat* (~procházet) hodnotu od začátku do konce (tedy přes všechny hodnoty), použij `for`,
2. pokud potřebuješ *iterovat*, dokud platí nějaké kritérium, použij `while`.



✓ Comprehensions

Jde o proces, kdy můžeš **kratším a kompaktnějším** zápisem zkombinovat:


1. `for` cyklus,
2. jednodušší podmínky (!),
3. okamžitě plnit nové hodnoty daty.

Jde prakticky o **nejpoužívanější prvek** v Pythonu vůbec, který používá naprostá většina solidních programátorů.

✓ List comprehensions

```
dvojnásobek = [cislo * 2 for cislo in range(30)]
```

```
print(dvojnásobek)
```


 [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 4

Jde tedy o ekvivalent k zápisu, který zatím znáš jako:

```
dvojnásobek = list()
```

```
for cislo in range(30):  
    dvojnásobek.append(cislo * 2)
```

```
print(dvojnásobek)
```


 [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 4

V kombinaci s větví if:

```
data = [1, 2, 3, "a", 5, 6, "@", "7", 7]
```

```
dvojnásobek = [cislo * 2 for cislo in data if isinstance(cislo, int)]
```

```
print(dvojnásobek)
```

 [2, 4, 6, 10, 12, 14]

Pokud je zápis v závorce delší a **málo přehledný**:

```
data = [1, 2, 3, "a", 5, 6, "@", "7", 7]
```

```
dvojnásobek = [  
    cislo * 2  
    for cislo in data  
    if isinstance(cislo, int)  
]
```

```
dvojnásobek = list()  
data = [1, 2, 3, "a", 5, 6, "@", "7", 7]
```

```
for cislo in data:  
    if isinstance(cislo, int):  
        dvojnásobek.append(cislo * 2)
```

Obecný vzorec tedy vypadá jako:

```
# vysledna_promenna = [
#     <hodnota>
#     <for_smycka>
#     <podminka>
# ]
```

▼ Dict comprehensions

Comprehensions můžeš skládat také **ze slovníků**:

```
obyvatele = {
    "Praha": 1_335_084,
    "Brno": 382_405,
    "Ostrava": 284_982,
    "Plzen": 175_219,
    "Liberec": 104_261
}
```

```
velka_mesta = {
    klic.upper(): hodnota
    for klic, hodnota in obyvatele.items()
    if hodnota > 200_000
}
```

```
print(velka_mesta)
```

```
↔ {'PRAHA': 1335084, 'BRNO': 382405, 'OSTRAVA': 284982}
```

```
velka_mesta = dict()
```

```
for klic, hodnota in obyvatele.items():
    if hodnota > 200_000:
        velka_mesta[klic.upper()] = hodnota
```

Podmínka o dvou větvích:

```
lego_ceny = {
    "7104: Desert Skiff": {"cena_$": 6, "rok_vydani": "2000"},
    "7190: Millennium Falcon": {"cena_$": 100, "rok_vydani": "2000"},
    "75044: Droid Tri-Fighter": {"cena_$": 30, "rok_vydani": "2015"}
}
```

```
cenys_inflaci = {
    jmeno: (
        hodnoty["cena_$"] * 2.27
        if hodnoty["rok_vydani"] == "2000"
        else hodnoty["cena_$"] * 2.51
    )
}
```

```
)
    for jmeno, hodnoty in lego_ceny.items()
}
```

```
print(ceny_s_inflaci)
```

```
➞ {'7104: Desert Skiff': 13.620000000000001, '7190: Millennium Falcon': 227.0, '75044:
```

```
ceny_s_infl = dict()
```

```
for jmeno, hodnoty in lego_ceny.items():
    if hodnoty["rok_vydani"] == "2000":
        ceny_s_infl[jmeno] = hodnoty["cena_$"] * 2.27
    else:
        ceny_s_infl[jmeno] = hodnoty["cena_$"] * 2.51
```

```
print(ceny_s_infl)
```

```
➞ {'7104: Desert Skiff': 13.620000000000001, '7190: Millennium Falcon': 227.0, '75044:
```

✓ Set comprehensions

```
ziskane_adresy = {"me@matousholinka.com", "petr@svetr.com", "lukas@gmail.com", "nan_email
```

```
domeny = {
    adresa.split("@")[1]
    for adresa in ziskane_adresy
    if "@" in adresa
}
```

```
print(domeny)
```

```
➞ {'matousholinka.com', 'gmail.com', 'svetr.com'}
```

```
domeny = set()
```

```
for adresa in ziskane_adresy:
    if "@" in adresa:
        domeny.add(adresa.split("@")[1])
```

```
print(domeny)
```

```
➞ {'matousholinka.com', 'svetr.com', 'gmail.com'}
```


✓ Nestovaná komprehence

Nakonec ještě ukázkat nestované *comprehensions*.

Tady je potřeba si zápis prohlédnout a zamyslet se, jestli je dostatečně **pochopitelný a čitelný** (obecně).

Je totiž k ničemu, pokud zapíšeš **nadupanou smyčku**, kterou budeš někomu *koktavě vysvětlovat*, nebo si o nějaký týden později neuvědomíš, co má sebou napsaná smyčka, vůbec dělat:

```
data = [  
    ["jméno", "příjmení", "email", "projekt"],  
    ["Lucie", "Nováková", "lucie.novakova@seznam.cz", "projekt_a"],  
    ["Petr", "Svetr", "petr.svetr@gmail.com", "projekt_b"]  
]
```

```
hledana_jmena = [  
    bunka  
    for radek in data  
    for bunka in radek  
    if "_" in bunka  
]
```

```
print(hledana_jmena)
```

```
➞ ['projekt_a', 'projekt_b']
```

```
hledana_jmena = list()
```

```
for radek in data:  
    for bunka in radek:  
        if "_" in bunka:  
            hledana_jmena.append(bunka)
```

```
print(hledana_jmena)
```

```
➞ ['projekt_a', 'projekt_b']
```



✓ Domácí úkol

✓ Zadání

Tvým úkolem bude odstraňovat písmena ze zadaného seznamu pomocí funkce `input` :

```
pismena = ["a", "a", "b", "c", "d", "a", "e", "g", "m"]
```

Jakmile budou všechna písmena odstraněná, vypíše tvůj program:

Seznam je prázdný!

Pokud zapíšeš písmeno, které v zadaném seznamu není, dostaneš upozornění:

x není součástí písmen!

Průběh může vypadat následovně:

```
Začátek: ['a', 'a', 'b', 'c', 'd', 'a', 'e', 'g', 'm']  
ktere písmeno chceš vyhodit? a  
Zbývají písmena ['a', 'b', 'c', 'd', 'a', 'e', 'g', 'm']  
ktere písmeno chceš vyhodit? a  
Zbývají písmena ['b', 'c', 'd', 'a', 'e', 'g', 'm']  
ktere písmeno chceš vyhodit? a  
Zbývají písmena ['b', 'c', 'd', 'e', 'g', 'm']  
ktere písmeno chceš vyhodit? b  
Zbývají písmena ['c', 'd', 'e', 'g', 'm']  
ktere písmeno chceš vyhodit? c  
Zbývají písmena ['d', 'e', 'g', 'm']  
ktere písmeno chceš vyhodit? d  
Zbývají písmena ['e', 'g', 'm']
```

```
ktere písmeno chceš vyhodit? e
Zbývají písmena ['g', 'm']
ktere písmeno chceš vyhodit? x
x není součástí písmen!
ktere písmeno chceš vyhodit? g
Zbývají písmena ['m']
ktere písmeno chceš vyhodit? m
Seznam je prázdný!
```

```
help(list.remove)
```

 Help on method_descriptor: