NOFY080 poznámky k přednášce

Emil Varga

22. září 2025

Obsah

Python Jazykové Intermezza

Code Snippets

1 Nastavení prostředí

Běžné programování v Pythonu silně spoléhá na externí balíčky, které mohou mít různé verze a závislosti. Aby se předešlo konfliktům mezi různými projekty, doporučuje se pro každý projekt používat virtuální prostředí (*virtual environment*, alebo v skratke venv). V tomto kurzu budeme takové virtuální prostředí používat, abychom zajistili konzistentní vývojové prostředí pro všechny. Virtuální prostředí jsou doporučeným způsobem správy závislostí v projektech v Pythonu.

Pro správu těchto virtuálních prostředí budeme používat open-source nástroj `uv`, dostupný na `https://github.com/astral-sh/uv`. Pokud již máte nainstalovanou jakoukoli verzi Pythonu, můžete `uv` nainstalovat jednoduchým spuštěním

```
pip install uv
```

Pokud Python nainstalovaný nemáte, můžete si jej stáhnout z `https://www.python.org/downloads/` a poté použít výše uvedený příkaz nebo použít samostatné instalátory na `https://github.com/astral-sh/uv`.

Pro otestování instalace spusťte v příkazovém řádku

```
uv --version
```

a mělo by se objevit něco podobného jako `uv 0.5.30`.

Nyní vytvoříme projekt s názvem `NOFY080_2025`, který bude obsahovat všechny soubory související s tímto kurzem, spuštěním

```
uv init NOFY080_2025
```

který vytvoří nový adresář se stejným názvem, s jednoduchým souborem "Hello World"v Pythonu, několika dalšími soubory, které `uv` používá ke sledování závislostí vašeho projektu, a nastaví v něm git repozitář (v tomto kurzu nemusíte používat git, ale zájemci se mohou naučit základy v dodatku **??**).

Nyní se přesuňte do adresáře projektu

```
cd NOFY080_2025
```

a přidejte balíčky, které budeme v průběhu tohoto kurzu potřebovat

```
uv add numpy scipy matplotlib
```

což vytvoří virtuální prostředí a nainstaluje zadané balíčky (v adresáři projektu by se měl objevit adresář `.venv`).

Pokud byste chtěli vytvořit virtuální prostředí bez instalace jakýchkoli balíčků, mohli byste použít příkaz

```
uv venv <name of the virtual environment>
```

Pro aktivaci virtuálního prostředí spusťte následující příkaz v Linuxu nebo Mac OS

```
source .venv/bin/activate
```

a ve Windows

```
.venv\Scripts\activate
```

Nyní, když spustíte `python`, spustí se verze z virtuálního prostředí, nikoli vaše systémová verze. Pro deaktivaci virtuálního prostředí jednoduše spusťte `deactivate`.

1.1 Spouštění kódu v Pythonu

Python je interpretovaný jazyk, který není třeba před spuštěním kompilovat. Kód v Pythonu ukládáme do textových souborů s příponou `.py` nebo do takzvaných Jupyter notebooků (přípona souboru `.ipynb`).

Pro otestování naší instalace vytvořte v našem projektovém adresáři nový soubor Pythonu (textový soubor s příponou `.py`) `test.py` s následujícím obsahem

```
import matplotlib.pyplot as plt

plt.plot([1, 2, 3], [4, 5, 6], '-o')
plt.show()
```

Pro jeho spuštění můžete buď manuálně aktivovat virtuální prostředí a spustit jej pomocí `python test.py` nebo použít `uv run test.py`. Měl by se objevit jednoduchý graf. Ve zbytku těchto skript, kdykoli vyvoláme příkaz `python` z příkazového řádku, myslíme tím ten ve virtuálním prostředí.

Pro spuštění kódu v Pythonu přímo bez uložení do souboru otevřete vhodný terminál a zadejte `python` a objeví se výzva (*prompt*) jako

```
    >>>
```

Toto je Read-Evaluate-Print-Loop (REPL); jakýkoli zadaný kód bude spuštěn a výsledek vytištěn na obrazovku. Pro ukončení zadejte `quit()`. Pro spuštění souboru `myfile.py` se nejprve přesuňte do jeho adresáře a poté jej jednoduše spusťte pomocí `python myfile.py` (všimněte si, že pokud máte aktivované virtuální prostředí, skutečný skript Pythonu se nemusí nacházet ve vašem projektovém adresáři).

Cvičení 1. Vytvořte textový soubor `hello.py`, který obsahuje jeden řádek

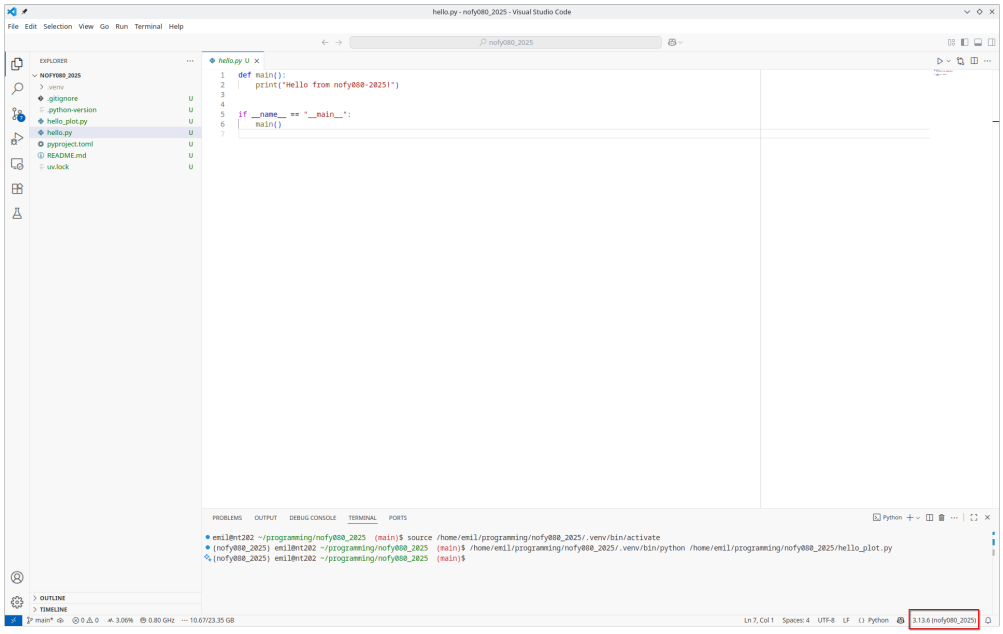
```
        print("Hello world!")

```

a poté tento soubor spusťte.

Někdy nechceme, aby se interpreter Pythonu ukončil ihned po dokončení běhu našeho programu (např. chceme zkontrolovat proměnné vytvořené během běhu programu); to lze provést pomocí `python -i file.py`, kde `-i` znamená *interaktivní*. Alternativně lze uživatelsky přívětivější (barevně odlišená syntaxe, automatické doplňování atd.) verzi interaktivního Python REPL vyvolat pomocí `ipython` (nebo `ipython3`, v závislosti na vaší instalaci), který lze také použít k interaktivnímu spouštění souborů pomocí `ipython3 -i myfile.py`.

1.2 Krátká poznámka k textovým editorům

K psaní kódu v Pythonu lze použít jakýkoli textový editor, včetně výchozího Poznámkového bloku ve Windows. Pro vaše duševní zdraví je však přínosné používat alespoň něco se zvýrazňováním syntaxe, jako je Notepad++, nebo editor s více funkcemi, jako je VS Code nebo Spyder, kde můžete soubor spustit bez přepínání do terminálu. V tomto kurzu doporučuji používat VS Code s nainstalovaným rozšířením pro Python, protože usnadňuje práci s virtuálními prostředími. Otevření adresáře ve VS Code automaticky detekuje virtuální prostředí a použije ho při spuštění souboru. Název aktuálně používaného virtuálního prostředí je uveden v pravém dolním rohu (zvýrazněno červeně):



Práce s prostředími ve Spyderu je poněkud složitější; viz průvodce zde.

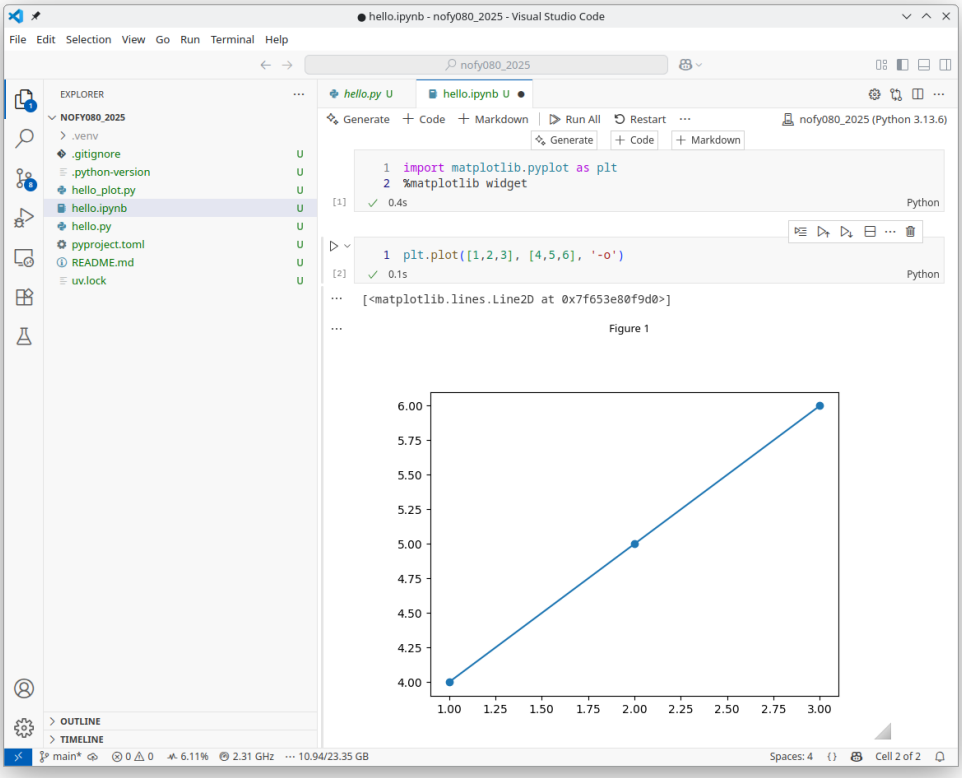
1.2.1 Jupyter Notebooky

Jupyter notebooky (spouštěné pomocí `jupyter-notebook` v příkazovém řádku) poskytují snadno použitelné interaktivní prostředí s rozhraním běžícím ve webovém prohlížeči. Jsou dobré pro zkoušení věcí[1]. Jupyter notebooky také dobře fungují s virtuálními prostředími a VS Code. Přejděte do adresáře svého projektu a spusťte

```
uv pip install jupyter ipympl
```

Balíček `ipympl` umožňuje použití interaktivních grafů Matplotlib v Jupyter noteboocích. Dále si také nainstalujte rozšíření `jupyter` ve VS Code. Nyní jednoduše vytvořte soubor s příponou `.ipynb` a můžete jej používat ve VS Code stejně jako v prohlížeči. Při prvním spuštění budete požádáni o výběr virtuálního prostředí, ve kterém by měl notebook běžet; to, které jsme vytvořili, by mělo být jednou z možností, pokud je notebook uložen v projektovém adresáři (stejný adresář jako adresář `.venv`).

[1] V tomto kurzu budeme (většinou) používat skripty kvůli snazšímu rozdělení do modulů a menšímu počtu problémů s paralelním programováním, se kterými se setkáme později.


● hello.ipynb - nofy080_2025 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
nofy080_2025
EXPLORER
NOFY080_2025
.venv
.gitignore U
.python-version U
hello_plot.py U
hello.ipynb U
hello.py U
pyproject.toml U
README.md U
uv.lock U
hello.py U
hello.ipynb U
Generate Code Markdown Run All Restart
nofy080_2025 (Python 3.13.6)
Generate Code Markdown
1 import matplotlib.pyplot as plt
2 %matplotlib widget
[1] 0.4s Python
1 plt.plot([1,2,3], [4,5,6], '-o')
[2] 0.1s Python
[<matplotlib.lines.Line2D at 0x7f653e80f9d0>]
Figure 1
6.00
5.75
5.50
5.25
5.00
4.75
4.50
4.25
4.00
1.00 1.25 1.50 1.75 2.00 2.25 2.50 2.75 3.00
OUTLINE
TIMELINE
main* 0 0 6.11% 2.31 GHz 10.94/23.35 GB
Spaces: 4 Cell 2 of 2

2 Základy Pythonu

2.1 Proměnné a typy

Proměnné se vytvářejí přiřazením hodnoty k dosud nepoužitému jménu proměnné, deklarace, jako v C/C++ nebo Pascalu, nejsou nutné.

```
my_integer = 5
my_float = 3.14
my_string = "double quote string"
my_string2 = 'single quote string'
my_string3 = "you can use 'single' quote in double quote string"
my_bool = False
```

Listing 1: Definování proměnných

Všimněte si, že proměnné obecně nemají pevný typ (tj. celé číslo nebo řetězec) a provedení něčeho jako `my_integer = "a string"` nezpůsobí typovou chybu. Samotný Python se nestará o to, jakého typu proměnná je, pokud jsou operace, které s ní provádíme, podporovány. Pokud však potřebujeme znát typ proměnné `v`, můžeme jej zjistit pomocí `type(v)`.

Základní aritmetické operace `+,-,*,/` fungují na číslech podle očekávání. Umocňování je `**`. Všimněte si však, že `/` automaticky *povýší (promotion)* celá čísla na desetinná (float). Pro celočíselné dělení můžeme použít `//` a pro zbytek po dělení `%`. Aritmetické operace jsou také definovány pro některé nečíselné typy, kde "dávají smysl", tj. řetězce lze spojovat pomocí `+`. Porovnávací operátory jsou `==` pro rovnost (**nezaměňovat s =, což je přiřazení hodnoty proměnné**), `!=` pro nerovnost a `<,<=,>,>=` pro uspořádání. Booleovské operátory jsou `and, or, not`.

Existují také **bitové** operátory `&,|,~,^` (bitové AND, OR, NOT a XOR). Tyto operují na jednotlivých bitech čísla. Dejte si pozor na záměnu mezi umocňováním `**` a bitovým exkluzivním *or* (XOR) `^`.

Komplexní čísla jsou v Pythonu podporována. Syntaxe pro zadání komplexního čísla je např. `3 + 5j`, což se překládá do matematického zápisu $3 + 5i$. Konkrétně imaginární jednotka je `1j`. Všimněte si, že písmeno `j` následuje bezprostředně za číslem.

Cvičení 2. Vyzkoušejte si základní aritmetiku v REPL. Zkuste vynásobit řetězec celým číslem. Co se stane, když se ho pokusíte vydělit?

Shrnutí základních vestavěných operátorů

`+,-,*,/, **`	základní aritmetika, umocňování
`and, or, not`	booleovské operace
`<, <=, ==, >, >=`	porovnávání
`&,|, ~, ^`	bitové operace, bitové AND, OR, NOT a XOR (exkluzivní nebo)

2.2 Uživatelský vstup

Většinou budeme psát neinteraktivní skripty, nicméně pro získání základního textového vstupu od uživatele můžeme použít funkci `input()`, **která vždy vrací řetězec**. Musíme jej sami převést na číslo (pomocí `int()` nebo `float()`), pokud očekáváme, že uživatel zadá číslo, např.:

```
age = int(input("How old are you? "))
print("You will be 100 in", 100-age, "years.")
```

2.3 Výpis a řetězce

Již jsme se setkali s funkcí `print()`, která na standardní výstup vypíše textovou reprezentaci objektů, které jí předáme. Přijímá jeden nebo více argumentů.

Pro převod jakékoli hodnoty na řetězec můžeme použít funkci `str()`. Speciální znaky, jako je nový řádek nebo tabulátor, je třeba *escapovat*, např. `\n` a `\t`. Zpětná lomítka je také třeba escapovat `\\`. Abychom zabránili Pythonu v interpretaci zpětných lomítek jako speciálních znaků (užitečné např. pro zadávání cest k souborům ve Windows), můžeme použít tzv. raw řetězce, např.:

```
print("s\tri\ng") #\t and \n are interpreted as tab and new line
print(r"s\tri\ng") # raw string, note the r immediately before "
```

Pro vytváření složitějších výpisů našich proměnných můžeme použít *formátovací* řetězce (nebo f-stringy) tak, že proměnnou, kterou chceme vložit do řetězce, uzavřeme do `{}`, např.:

```
vari = 5
varf = 3.14
fstr = f"An interger value {vari:05d}, a float value {varf:010.6f}, exponential
form {varf:g}"
```

Nahrazení textu hodnotou proměnné (tzv. interpolace) se specifikuje jako `{výraz:specifikátor_formátu}`, přičemž specifikátory formátu jsou volitelné. Zde specifikátor formátu `05d` znamená celé číslo s celkovou šířkou alespoň 5 znaků a `010.6f` znamená číslo s plovoucí desetinnou čárkou se 6 číslicemi za desetinnou čárkou a celkovou šířkou alespoň 10 znaků.

Pro úplnou specifikaci možných formátovacích specifikátorů viz https://docs.python.org/3/library/string.html#formats

2.4 Funkce

Funkce se definují pomocí klíčového slova `def`. Funkce mohou přijímat argumenty a vracet hodnotu.

```
def my_function(my_string):
    print("Someone wants to know how long \"" + my_string + "\" is!")
    return len(my_string) #returns the length of a my_string
```

Všimněte si odsazení, kterým se v Pythonu identifikují bloky kódu, tj. neexistuje zde `begin,end` jako v Pascalu nebo `{}` jako v C/C++.

Více argumentů se odděluje čárkou a argumenty mohou mít výchozí hodnotu, např. argument c bude 1, pokud není uvedeno jinak.

```
def calculate(a, b, c=1):
    """
    A function that calculates c*(a + b)

    Parameters:
    -----------
    a, b, c: numbers
        arguments of the calculation

    Returns:
    --------
    result: number
        The result of calculating c*(a + b)
    """
    return c*(a + b)

>>> calculate(2, 3)
5
>>> calculate(2, 3, 4)
20
```

Listing 2: Function definition

Zde je dlouhý řetězec ohraničený trojitými uvozovkami `"""` takzvaný dokumentační řetězec funkce (nebo zkráceně **docstring**). Trojité uvozovky lze použít kdekoli a znamenají pouze to, že řetězec se rozkládá na více řádků. K docstringu lze poté přistupovat pomocí funkce `help()` nebo pomocí textových editorů. Formátování použité ve výše uvedeném příkladu je běžně srozumitelné pro IDE, např. Spyder jej zobrazí pěkně naformátovaný v panelu Nápověda (standardně v pravém horním rohu okna).

Argumenty s výchozími hodnotami musí následovat za povinnými argumenty. S více volitelnými argumenty můžeme funkci volat pomocí *pojmenovaných argumentů* (keyword arguments):

```
    def calculate2(a, b, c=1, d=1):
        return c*(a + b)**d

    >>> calculate(2, 3, d=2) # necha vychozi hodnotu c, ale nastavi d
    25
```

Listing 3: Volitelné a pojmenované argumenty.

Funkce lze předávat v proměnných a argumentech, konkrétně funkce mohou přijímat jiné funkce jako argumenty, např.:

```
    def add1(x):
        return x + 1

    def change_number(x, func):
        return func(x)

    >>> change_number(1, add1)
    2
```

Listing 4: Funkce vyššího řádu.

Krátké, jednoduché funkce (které se vejdou na jeden řádek) lze definovat přímo jako takzvané anonymní neboli *lambda* funkce, např.

```
    >>> change_number(1, lambda x: 2 + x)
    3
```

Listing 5: Lambda funkce.

Lambda funkce jsou obzvláště užitečné pro částečné vyplnění seznamů argumentů existujících funkcí, např.

```
    >>> change_number(1, lambda x: calculate2(x, 4, c=4, d=0.5))
    8.94427190999916
```

2.5 Složené typy

2.5.1 Seznami a n-tice (Lists and tuples)

Seznam (**list**) je kolekce jakýchkoli pythonovských objektů definovaná pomocí hranatých závorek `[]`,

```
    my_list = [1, 2.17, "a string", ['another', 'list']]
```

K hodnotám v seznamu lze přistupovat pomocí indexování seznamu **začínajícího od 0**, např. `my_list[1] == 2.17`. Záporné indexy se počítají od konce, tj. `my_list[-1]` je poslední prvek, `my_list[-2]` předposlední atd.

Seznamy jsou příkladem *objektu*, objekty mají přidružené funkce zvané metody, které se volají pomocí tečkové notace. Užitečné metody pro seznamy jsou:

- `append` and `pop`

```
>>> my_list = [1, 2.0, 'three']
>>> my_list.append('one more')
>>> my_list.pop() # odstrani posledni prvek a vrati ho
"one more"
```

- `my_list.sort()`, seřadí seznam na místě, pokud lze všechny prvky seznamu porovnat; pro vytvoření nového seznamu s seřazenými prvky můžeme použít `sorted(my_list)`
- `my_list.index('a')` najde index prvního výskytu 'a' nebo vrátí chybu

Kromě jednoduchého indexování lze k podmnožinám seznamů přistupovat pomocí *řezů* (slicing), např.:

```
l = [1,2,3,4,5,6,7,8,9]
l[2:5] # [3,4,5]
l[-5::2] # [5,7,9]
```

Obecně je syntaxe `my_list[start_idx:stop_idx:step]`, `start_idx` je včetně, `stop_idx` je vyjma. Všechny tři jsou volitelné, výchozí hodnota start je 0, výchozí hodnota stop je -1 a výchozí hodnota step je 1.

N-tice (**Tuples**) se v mnoha ohledech chovají podobně jako seznamy, s výjimkou, že jsou **neměnné** (immutable), tj. samotnou n-tici ani hodnoty v ní nelze měnit. Indexování a řezy však fungují stejně. N-tice se definují pomocí kulatých závorek `()`. Pokud n-tice obsahuje pouze jednu hodnotu, je třeba přidat čárku, aby se odlišila od výrazu v závorkách, tj. `single_tuple = (1,)`.

Běžným použitím n-tic je vracení více hodnot z funkce, tj.:

```
def add_and_subtract(x, y):
    return x+y, x-y # Vsimnete si, ze zavorky () nejsou potreba v nekterych pripadech

# tu rozbalime vracenou dvojici do dvou promennych
a, s = add_and_subtract(3, 2) # a==5, s==1
```

Všimněte si, že jsme neuložili návratovou hodnotu jako n-tici, ale okamžitě jsme ji rozdělili do dvou samostatných proměnných. Tomu se říká **rozbalování** (unpacking) a lze jej použít pro jakýkoli složený typ.

2.5.2 Slovníky (Dictionaries)

Slovníky (Dictionaries) jsou kolekce přiřazení klíče k hodnotě libovolného typu, často jsou klíče řetězce, např.:

```
my_dict = {
    'key1': value1,
    'key2': value2,
}
```

which can then be accessed as `my_dict['key2']` etc.

2.6 Kontrola toku programu

2.6.1 if-elif-else

Příkaz `if` přijímá booleovský výraz a spustí kód ve svém bloku, pokud se podmínka vyhodnotí jako `True`.

```
if condition:
    do_if_true()
elif condition2: # nepovinne
    do_if_2true() # spusti se jenom kdyz condition je False a condition2 je True
else: # nepovinne
    do_if_false() # spusti se jenom kdyz oboje condition and condition2 jsou False
```

2.6.2 match

Dostupné od Pythonu 3.10 TODO

```
command = 'yell'
match command:
    case 'talk':
        print("hello")
    case 'yell':
```

```
            print("HELLO!!")
        case _: #default behavior
            print("unknown command")
```

2.7 Smyčky (Loops)

Pro cyklení s daným počtem iterací můžeme použít cykly `for` s funkcí `range()`. Můžeme také cyklit přes cokoli iterovatelného, např. seznam nebo slovník.

```
    for k in range(10):
        print(k)

    for value in my_list:
        print(value)
```

Obecný tvar `range()` je `range(start, stop, step)`, start je včetně, stop je vyjma a krok je standardně 1. `range()` nevrací seznam, ale něco, co se nazývá *iterátor*. Můžeme jej převést na seznam pomocí `list(range(start, stop, step))`.

Jestliže potřebujeme iterovat přes nějaké hodnoty a zároveň pracovat s indexem můžeme použít `enumerate`.

```
    for index, value in enumerate(my_list):
        print(f"my_list[{index}] = {value}")
```

Pro cyklení, dokud je podmínka pravdivá, můžeme použít cyklus `while`,

```
    while condition:
        do_stuff()
```

Běh cyklů lze upravit pomocí `continue`, které přeskočí zbytek kódu v iteraci a přejde na další, nebo `break`, které cyklus okamžitě ukončí. Běžné použití je s nekonečnými cykly, např.:

```
    while True: # bude bezet do nekonecna
        do_stuff()
        if should_we_stop: # pokud to neukoncime
            break
```

Cykly se často používají k vytváření seznamů, což lze zjednodušit pomocí **list comprehension** (generování seznamu), jako je

```
    [expression(k) for k in iterable if condition(k)] # if podminka je nepovinna
```

For example,

```
    >>> [k**2 for k in range(5)] # seznam druhych mocnin pro
    [0, 1, 4, 9, 16]

    >>> [k**2 for k in range(5) if k % 2 == 0] # seznam druhych mocnin pro sude k
    [0, 4, 16]
```

Listing 6: List comprehension.

Cvičení 3. Napište program, který se zeptá uživatele na jméno a odpoví `"Hello <jméno>!"`.

1. Změňte program tak, aby pokud je jméno uživatele "Andrej", pozdrav se změnil na "Ciao Andrej!".
2. Oddělte logiku vytváření pozdravu do samostatné funkce.

Cvičení 4. Napište program, který přijme libovolný počet jmen z příkazového řádku a na konci je vypíše v abecedním pořadí.

1. Uživatel předem zadá počet jmen, která chce zadat. *Nápověda*: `range()` a cyklus `for`
2. Program přestane žádat o nová jména, jakmile uživatel zadá prázdný řetězec. *Nápověda:* `break`
3. Pokud uživatel zadal jméno "Emil", bude ve výsledném výpisu přeskočeno. *Nápověda:* `continue`.

2.8 Používaní externích modulů

Python je známý tím, že má velké množství dostupných knihoven pro téměř cokoli pod sluncem.[2] Abychom je mohli použít, musíme nejprve importovat *modul*, který potřebujeme, např. pro použití funkcí, které pracují s časem, můžeme udělat

```
import time as t #importuj modul time a dej mu skraceny nazev t
print(t.time()) #nazev modulu pripajime z leva ke jmenu funkce
```

Pokud chceme z daného modulu použít jen malý počet funkcí, můžeme je importovat konkrétně a nemusíme pro jejich volání používat název modulu, např.

```
from module import func1, func2
func1()
func2()
```

Cvičení 5. Napište program pro výpočet Fibonacciho čísla F_n pomocí cyklů i rekurze. Změřte čas potřebný k výpočtu prvních 20 F_n oběma metodami pomocí `time.time()`. *Nápověda:* $F_k = F_{k-1} + F_{k-2},\ F_1 = F_2 = 1$

[2] A brzy se naučíme, jak si vytvořit vlastní.