```
# Jednořádkový komentář začíná křížkem

""" Víceřádkové komentáře používají tři uvozovky nebo apostrofy
    a jsou často využívány jako dokumentační komentáře k metodám
"""

####################################################
## 1. Primitivní datové typy a operátory
####################################################

# Čísla
3  # => 3

# Aritmetické operace se chovají běžným způsobem
1 + 1   # =>  2
8 - 1   # =>  7
10 * 2  # => 20

# Až na dělení, které vrací desetinné číslo
35 / 5  # => 7.0

# Při celočíselném dělení je desetinná část oříznuta (pro kladná i
záporná čísla)
5 // 3       # => 1
5.0 // 3.0   # => 1.0  #  celočíselně dělit lze i desetinným číslem
-5 // 3      # => -2
-5.0 // 3.0  # => -2.0

# Pokud použijete desetinné číslo, výsledek je jím také
3 * 2.0  # => 6.0

# Modulo
7 % 3  # => 1

# Mocnění (x na y-tou)
2**4  # => 16

# Pro vynucení priority použijte závorky
(1 + 3) * 2  # => 8

# Logické hodnoty
True
False

# Negace se provádí pomocí not
not True   # => False
not False  # => True
```

```
# Logické operátory
# U operátorů záleží na velikosti písmen
True and False  # => False
False or True   # => True

# Používání logických operátorů s čísly
0 and 2     # => 0
-5 or 0     # => -5
0 == False  # => True
2 == True   # => False
1 == True   # => True

# Rovnost je ==
1 == 1  # => True
2 == 1  # => False

# Nerovnost je !=
1 != 1  # => False
2 != 1  # => True

# Další porovnání
1 < 10  # => True
1 > 10  # => False
2 <= 2  # => True
2 >= 2  # => True

# Porovnání se dají řetězit!
1 < 2 < 3  # => True
2 < 3 < 2  # => False


# Řetězce používají " nebo ' a mohou obsahovat UTF8 znaky
"Toto je řetězec."
'Toto je také řetězec.'

# Řetězce se také dají sčítat, ale nepoužívejte to
"Hello " + "world!"  # => "Hello world!"
# Dají se spojovat i bez '+'
"Hello " "world!"  # => "Hello world!"

# Řetězec lze považovat za seznam znaků
"Toto je řetězec"[0]  # => 'T'

# .format lze použít ke skládání řetězců
"{} mohou být {}".format("řetězce", "skládány")

# Formátovací argumenty můžete opakovat
"{0} {1} stříkaček stříkalo přes {0} {1} střech".format("tři sta třicet
tři", "stříbrných")
```

```
# => "tři sta třicet tři stříbrných stříkaček stříkalo přes tři sta
třicet tři stříbrných střech"

# Pokud nechcete počítat, můžete použít pojmenované argumenty
"{jmeno} si dal {jidlo}".format(jmeno="Franta", jidlo="guláš")  # =>
"Franta si dal guláš"

# Pokud zároveň potřebujete podporovat Python 2.5 a nižší, můžete použít
starší způsob formátování
"%s se dají %s jako v %s" % ("řetězce", "skládat", "jazyce C")


# None je objekt (jinde NULL, nil, ...)
None  # => None

# Pokud porovnáváte něco s None, nepoužívejte operátor rovnosti "==",
# použijte raději operátor "is", který testuje identitu.
"něco" is None  # => False
None is None    # => True

# None, 0, a prázdný řetězec/seznam/slovník se vyhodnotí jako False
# Vše ostatní se vyhodnotí jako True
bool(0)   # => False
bool("")  # => False
bool([])  # => False
bool({})  # => False


####################################################
## 2. Proměnné a kolekce
####################################################

# Python má funkci print
print("Jsem 3. Python 3.")

# Proměnné není třeba deklarovat před přiřazením
# Konvence je používat male_pismo_s_podtrzitky
nazev_promenne = 5
nazev_promenne  # => 5
# Názvy proměnných mohou obsahovat i UTF8 znaky
název_proměnné = 5

# Přístup k předtím nepoužité proměnné vyvolá výjimku
# Odchytávání vyjímek - viz další kapitola
neznama_promenna  # Vyhodí NameError

# Seznam se používá pro ukládání sekvencí
sez = []
# Lze ho rovnou naplnit
```

```
jiny_seznam = [4, 5, 6]

# Na konec seznamu se přidává pomocí append
sez.append(1)    # sez je nyní [1]
sez.append(2)    # sez je nyní [1, 2]
sez.append(4)    # sez je nyní [1, 2, 4]
sez.append(3)    # sez je nyní [1, 2, 4, 3]
# Z konce se odebírá se pomocí pop
sez.pop()        # => 3 a sez je nyní [1, 2, 4]
# Vložme trojku zpátky
sez.append(3)    # sez je nyní znovu [1, 2, 4, 3]

# Přístup k prvkům funguje jako v poli
sez[0]  # => 1
# Mínus počítá odzadu (-1 je poslední prvek)
sez[-1]  # => 3

# Přístup mimo seznam vyhodí IndexError
sez[4]  # Vyhodí IndexError

# Pomocí řezů lze ze seznamu vybírat různé intervaly
# (pro matematiky: jedná se o uzavřený/otevřený interval)
sez[1:3]  # => [2, 4]
# Odříznutí začátku
sez[2:]  # => [4, 3]
# Odříznutí konce
sez[:3]  # => [1, 2, 4]
# Vybrání každého druhého prvku
sez[::2]  # =>[1, 4]
# Vrácení seznamu v opačném pořadí
sez[::-1]  # => [3, 4, 2, 1]
# Lze použít jakoukoliv kombinaci parametrů pro vytvoření složitějšího
řezu
# sez[zacatek:konec:krok]

# Odebírat prvky ze seznamu lze pomocí del
del sez[2]   # sez je nyní [1, 2, 3]

# Seznamy můžete sčítat
# Hodnoty sez a jiny_seznam přitom nejsou změněny
sez + jiny_seznam   # => [1, 2, 3, 4, 5, 6]

# Spojit seznamy lze pomocí extend
sez.extend(jiny_seznam)   # sez je nyní [1, 2, 3, 4, 5, 6]

# Kontrola, jestli prvek v seznamu existuje, se provádí pomocí in
1 in sez  # => True

# Délku seznamu lze zjistit pomocí len
```

```
len(sez)  # => 6


# N-tice je jako seznam, ale je neměnná
ntice = (1, 2, 3)
ntice[0]      # => 1
ntice[0] = 3  # Vyhodí TypeError

# S n-ticemi lze dělat většinu operací, jako se seznamy
len(ntice)          # => 3
ntice + (4, 5, 6)  # => (1, 2, 3, 4, 5, 6)
ntice[:2]           # => (1, 2)
2 in ntice          # => True

# N-tice (nebo seznamy) lze rozbalit do proměnných jedním přiřazením
a, b, c = (1, 2, 3)  # a je nyní 1, b je nyní 2 a c je nyní 3
# N-tice jsou vytvářeny automaticky, když vynecháte závorky
d, e, f = 4, 5, 6
# Prohození proměnných je tak velmi snadné
e, d = d, e  # d je nyní 5, e je nyní 4


# Slovníky ukládají klíče a hodnoty
prazdny_slovnik = {}
# Lze je také rovnou naplnit
slovnik = {"jedna": 1, "dva": 2, "tři": 3}

# Přistupovat k hodnotám lze pomocí []
slovnik["jedna"]  # => 1

# Všechny klíče dostaneme pomocí keys() jako iterovatelný objekt. Nyní
ještě
# potřebujeme obalit volání v list(), abychom dostali seznam. To
rozebereme
# později. Pozor, že jakékoliv pořadí klíčů není garantováno - může být
různé.
list(slovnik.keys())  # => ["dva", "jedna", "tři"]

# Všechny hodnoty opět jako iterovatelný objekt získáme pomocí values().
Opět
# tedy potřebujeme použít list(), abychom dostali seznam. Stejně jako
# v předchozím případě, pořadí není garantováno a může být různé
list(slovnik.values())  # => [3, 2, 1]

# Operátorem in se lze dotázat na přítomnost klíče
"jedna" in slovnik  # => True
1 in slovnik        # => False

# Přístup k neexistujícímu klíči vyhodí KeyError
```

```
slovnik["čtyři"]  # Vyhodí KeyError

# Metoda get() funguje podobně jako [], ale vrátí None místo vyhození
KeyError
slovnik.get("jedna")   # => 1
slovnik.get("čtyři")   # => None
# Metodě get() lze předat i výchozí hodnotu místo None
slovnik.get("jedna", 4)   # => 1
slovnik.get("čtyři", 4)   # => 4

# metoda setdefault() vloží prvek do slovníku pouze pokud tam takový
klíč není
slovnik.setdefault("pět", 5)  # slovnik["pět"] je nastaven na 5
slovnik.setdefault("pět", 6)  # slovnik["pět"] je pořád 5

# Přidání nové hodnoty do slovníku
slovnik["čtyři"] = 4
# Hromadně aktualizovat nebo přidat data lze pomocí update(), parametrem
je opět slovník
slovnik.update({"čtyři": 4})  # slovnik je nyní {"jedna": 1, "dva": 2,
"tři": 3, "čtyři": 4, "pět": 5}

# Odebírat ze slovníku dle klíče lze pomocí del
del slovnik["jedna"]  # odebere klíč "jedna" ze slovnik


# Množiny ukládají ... překvapivě množiny
prazdna_mnozina = set()
# Také je lze rovnou naplnit. A ano, budou se vám plést se slovníky.
Bohužel.
mnozina = {1, 1, 2, 2, 3, 4}  # mnozina je nyní {1, 2, 3, 4}

# Přidání položky do množiny
mnozina.add(5)  # mnozina je nyní {1, 2, 3, 4, 5}

# Průnik lze udělat pomocí operátoru &
jina_mnozina = {3, 4, 5, 6}
mnozina & jina_mnozina  # => {3, 4, 5}

# Sjednocení pomocí operátoru |
mnozina | jina_mnozina  # => {1, 2, 3, 4, 5, 6}

# Rozdíl pomocí operátoru -
{1, 2, 3, 4} - {2, 3, 5}  # => {1, 4}

# Operátorem in se lze dotázat na přítomnost prvku v množině
2 in mnozina  # => True
9 in mnozina  # => False
```

```
####################################################
## 3. Řízení toku programu, cykly
####################################################

# Vytvořme si proměnnou
promenna = 5

# Takto vypadá podmínka. Na odsazení v Pythonu záleží!
# Vypíše "proměnná je menší než 10".
if promenna > 10:
    print("proměnná je velká jak Rusko")
elif promenna < 10:  # Část elif je nepovinná
    print("proměnná je menší než 10")
else:                # Část else je také nepovinná
    print("proměnná je právě 10")


"""
Smyčka for umí iterovat (nejen) přes seznamy
vypíše:
    pes je savec
    kočka je savec
    myš je savec
"""
for zvire in ["pes", "kočka", "myš"]:
    # Můžete použít formát pro složení řetězce
    print("{} je savec".format(zvire))

"""
range(cislo) vrací iterovatelný objekt čísel od 0 do cislo
vypíše:
    0
    1
    2
    3
"""
for i in range(4):
    print(i)

"""
range(spodni_limit, horni_limit) vrací iterovatelný objekt čísel mezi
limity
vypíše:
    4
    5
    6
    7
"""
```

```
for i in range(4, 8):
    print(i)

"""
Smyčka while se opakuje, dokud je podmínka splněna.
vypíše:
    0
    1
    2
    3
"""
x = 0
while x < 4:
    print(x)
    x += 1  # Zkrácený zápis x = x + 1. Pozor, žádné x++ neexisuje.


# Výjimky lze ošetřit pomocí bloku try/except(/else/finally)
try:
    # Pro vyhození výjimky použijte raise
    raise IndexError("Přistoupil jste k neexistujícímu prvku v
seznamu.")
except IndexError as e:
    print("Nastala chyba: {}".format(e))
    # Vypíše: Nastala chyba: Přistoupil jste k neexistujícímu prvku v
seznamu.
except (TypeError, NameError):  # Více výjimek lze zachytit najednou
    pass  # Pass znamená nedělej nic - nepříliš vhodný způsob ošetření
chyb
else:  # Volitelný blok else musí být až za bloky except
    print("OK!")  # Vypíše OK! v případě, že nenastala žádná výjimka
finally:  # Blok finally se spustí nakonec za všech okolností
    print("Uvolníme zdroje, uzavřeme soubory...")

# Místo try/finally lze použít with pro automatické uvolnění zdrojů
with open("soubor.txt") as soubor:
    for radka in soubor:
        print(radka)

# Python běžně používá iterovatelné objekty, což je prakticky cokoliv,
# co lze považovat za sekvenci. Například to, co vrací metoda range(),
# nebo otevřený soubor, jsou iterovatelné objekty.

slovnik = {"jedna": 1, "dva": 2, "tři": 3}
iterovatelny_objekt = slovnik.keys()
print(iterovatelny_objekt)  # => dict_keys(["jedna", "dva", "tři"]).
Toto je iterovatelný objekt.

r
# Můžeme použít cyklus for na jeho projití
```

```
for klic in iterovatelny_objekt:
    print(klic)    # vypíše postupně: jedna, dva, tři

# Ale nelze přistupovat k prvkům pod jejich indexem
iterovatelny_objekt[1]  # Vyhodí TypeError

# Všechny položky iterovatelného objektu lze získat jako seznam pomocí
list()
list(slovnik.keys())  # => ["jedna", "dva", "tři"]

# Z iterovatelného objektu lze vytvořit iterátor
iterator = iter(iterovatelny_objekt)

# Iterátor je objekt, který si pamatuje stav v rámci svého
iterovatelného objektu
# Další hodnotu dostaneme voláním next()
next(iterator)  # => "jedna"

# Iterátor si udržuje svůj stav v mezi jednotlivými voláními next()
next(iterator)  # => "dva"
next(iterator)  # => "tři"

# Jakmile interátor vrátí všechna svá data, vyhodí výjimku StopIteration
next(iterator)  # Vyhodí StopIteration


####################################################
## 4. Funkce
####################################################

# Pro vytvoření nové funkce použijte klíčové slovo def
def secist(x, y):
    print("x je {} a y je {}".format(x, y))
    return x + y  # Hodnoty se vrací pomocí return

# Volání funkce s parametry
secist(5, 6)  # => Vypíše "x je 5 a y je 6" a vrátí 11

# Jiný způsob, jak volat funkci, je použít pojmenované argumenty
secist(y=6, x=5)  # Pojmenované argumenty můžete předat v libovolném
pořadí

# Lze definovat funkce s proměnným počtem (pozičních) argumentů
def vrat_argumenty(*argumenty):
    return argumenty

vrat_argumenty(1, 2, 3)  # => (1, 2, 3)

# Lze definovat také funkce s proměnným počtem pojmenovaných argumentů
```

```
def vrat_pojmenovane_argumenty(**pojmenovane_argumenty):
    return pojmenovane_argumenty

vrat_pojmenovane_argumenty(kdo="se bojí", nesmi="do lesa")
# => {"kdo": "se bojí", "nesmi": "do lesa"}


# Pokud chcete, lze použít obojí najednou
# Konvence je používat pro tyto účely názvy *args a **kwargs
def vypis_vse(*args, **kwargs):
    print(args, kwargs)  # print() vypíše všechny své parametry oddělené
mezerou

vypis_vse(1, 2, a=3, b=4)  # Vypíše: (1, 2) {"a": 3, "b": 4}

# * nebo ** lze použít k rozbalení N-tic nebo slovníků!
ntice = (1, 2, 3, 4)
slovnik = {"a": 3, "b": 4}
vypis_vse(ntice)  # Vyhodnotí se jako vypis_vse((1, 2, 3, 4)) - jeden
parametr, N-tice
vypis_vse(*ntice)  # Vyhodnotí se jako vypis_vse(1, 2, 3, 4)
vypis_vse(**slovnik)  # Vyhodnotí se jako vypis_vse(a=3, b=4)
vypis_vse(*ntice, **slovnik)  # Vyhodnotí se jako vypis_vse(1, 2, 3, 4,
a=3, b=4)


# Viditelnost proměnných - vytvořme si globální proměnnou x
x = 5

def nastavX(cislo):
    # Lokální proměnná x překryje globální x
    x = cislo  # => 43
    print(x)  # => 43

def nastavGlobalniX(cislo):
    global x
    print(x)  # => 5
    x = cislo  # Nastaví globální proměnnou x na 6
    print(x)  # => 6

nastavX(43)
nastavGlobalniX(6)


# Funkce jsou first-class objekty
def vyrobit_scitacku(pricitane_cislo):
    def scitacka(x):
        return x + pricitane_cislo
    return scitacka
```

```
pricist_10 = vyrobit_scitacku(10)
pricist_10(3)  # => 13

# Klíčové slovo lambda vytvoří anonymní funkci
(lambda parametr: parametr > 2)(3)  # => True

# Lze použít funkce map() a filter() z funkcionálního programování
map(pricist_10, [1, 2, 3])
# => <map object at 0x0123467> - iterovatelný objekt s obsahem: [11, 12,
13]
filter(lambda x: x > 5, [3, 4, 5, 6, 7])
# => <filter object at 0x0123467> - iterovatelný objekt s obsahem: [6,
7]

# S generátorovou notací lze dosáhnout podobných výsledků, ale vrací
seznam
[pricist_10(i) for i in [1, 2, 3]]  # => [11, 12, 13]
[x for x in [3, 4, 5, 6, 7] if x > 5]   # => [6, 7]
# Generátorová notace funguje i pro slovníky
{x: x**2 for x in range(1, 5)}  # => {1: 1, 2: 4, 3: 9, 4: 16}
# A také pro množiny
{pismeno for pismeno in "abeceda"}  # => {"d", "a", "c", "e", "b"}


####################################################
## 5. Třídy
####################################################

# Třída Clovek je potomkem (dědí od) třídy object
class Clovek(object):

    # Atribut třídy - je sdílený všemi instancemi
    druh = "H. sapiens"

    # Toto je kostruktor. Je volán, když vytváříme instanci třídy. Dvě
    # podtržítka na začátku a na konci značí, že se jedná o atribut nebo
    # objekt využívaný Pythonem ke speciálním účelům, ale můžete sami
    # definovat jeho chování. Metody jako __init__, __str__, __repr__
    # a další se nazývají "magické metody". Nikdy nepoužívejte toto
    # speciální pojmenování pro běžné metody.
    def __init__(self, jmeno):
        # Přiřazení parametru do atributu instance jmeno
        self.jmeno = jmeno

    # Metoda instance - všechny metody instance mají "self" jako první
parametr
    def rekni(self, hlaska):
        return "{jmeno}: {hlaska}".format(jmeno=self.jmeno,
```

```
hlaska=hlaska)

    # Metoda třídy - sdílená všemi instancemi
    # Dostává jako první parametr třídu, na které je volána
    @classmethod
    def vrat_druh(cls):
        return cls.druh

    # Statická metoda je volána bez reference na třídu nebo instanci
    @staticmethod
    def odkaslej_si():
        return "*ehm*"


# Vytvoření instance
d = Clovek(jmeno="David")
a = Clovek("Adéla")
print(d.rekni("ahoj"))    # Vypíše: "David: ahoj"
print(a.rekni("nazdar"))  # Vypíše: "Adéla: nazdar"

# Volání třídní metody
d.vrat_druh()  # => "H. sapiens"

# Změna atributu třídy
Clovek.druh = "H. neanderthalensis"
d.vrat_druh()  # => "H. neanderthalensis"
a.vrat_druh()  # => "H. neanderthalensis"

# Volání statické metody
Clovek.odkaslej_si()  # => "*ehm*"


####################################################
## 6. Moduly
####################################################

# Lze importovat moduly
import math
print(math.sqrt(16.0))  # => 4

# Lze také importovat pouze vybrané funkce z modulu
from math import ceil, floor
print(ceil(3.7))   # => 4.0
print(floor(3.7))  # => 3.0

# Můžete také importovat všechny funkce z modulu, ale radši to nedělejte
from math import *

# Můžete si přejmenovat modul při jeho importu
```

```
import math as m
math.sqrt(16) == m.sqrt(16)  # => True

# Modul v Pythonu není nic jiného, než obyčejný soubor .py
# Můžete si napsat vlastní a prostě ho importovat podle jména
from muj_modul import moje_funkce  # Nyní vyhodí ImportError - muj_modul
neexistuje

# Funkcí dir() lze zjistit, co modul obsahuje
import math
dir(math)


####################################################
## 7. Pokročilé
####################################################

# Generátory jsou funkce, které místo return obsahují yield
def nasobicka_2(sekvence):
    for i in sekvence:
        yield 2 * i

# Generátor generuje hodnoty postupně, jak jsou potřeba. Místo toho, aby
vrátil
# celou sekvenci s prvky vynásobenými dvěma, provádí jeden výpočet v
každé iteraci.
# To znamená, že čísla větší než 15 se v metodě nasobicka_2 vůbec
nezpracují.

# Funkce range() je také generátor - vytváření seznamu 900000000 prvků
by zabralo
# hodně času i paměti, proto se místo toho čísla generují postupně.

for i in nasobicka_2(range(900000000)):
    print(i)  # Vypíše čísla 0, 2, 4, 6, 8, ... 30
    if i >= 30:
        break


# Dekorátory jsou funkce, které se používají pro obalení jiné funkce,
čímž mohou
# přidávat nebo měnit její stávající chování. Funkci dostávají jako
parametr
# a typicky místo ní vrací jinou, která uvnitř volá tu původní.

def nekolikrat(puvodni_funkce):
    def opakovaci_funkce(*args, **kwargs):
        for i in range(3):
            puvodni_funkce(*args, **kwargs)
```

```
    return opakovaci_funkce


@nekolikrat
def pozdrav(jmeno):
    print("Měj se {}!".format(jmeno))

pozdrav("Pepo")  # Vypíše 3x: Měj se Pepo!
```