Funkce sorted
dostane jako parametr libovolnou iterovatelnou posloupnost (například seznam, řetězec, …) a vrátí uspořádaný seznam. Původní posloupnost se přitom nemění (je to immutable).

```
def vzestupně(seznam):
return sorted(seznam) == seznam
```

Metoda split() - *řetězec.split()*
Metoda rozbije daný řetězec na samostatné řetězce a uloží je do seznamu (tedy vrací seznam řetězců). Metoda split() se často využívá při rozdělení přečteného řetězce ze vstupu (input() nebo subor.readline()) na více částí, například:

```
>>> ret = input('zadej 2 čísla: ')
zadej 2 čísla: 15 999
>>> sez = ret.split()
>>> sez
['15', '999']
```

Někdy můžeme vidět i takový zápis:

```
>>> jméno, příjmení = input('zadej jméno a příjmení: ').split()
zadej jméno a příjmení: Janko Hraško
>>> jméno
'Janko'
>>> příjmení
'Hraško'
```

Metoda ***split()*** může obdržet jako parametr oddělovač, například pokud jsme přečetli čísla oddělená čárkami:

```
sucet = 0
for prvek in input('zadej čísla: ').split(','):
sucet += int(prvek)
print('jejich součet je', sucet)
zadej čísla: 10,20,30,40
jejich součet je 100
```

Metoda join() - *oddělovač.join(seznam)*
Metoda slepí všechny řetězce z daného seznamu řetězců do jednoho, přičemž je navzájem oddělí uvedeným oddělovačem, tj. nějakým zadaným řetězcem. Jako seznam můžeme uvést libovolnou posloupnost (iterovatelný objekt) řetězců.

Ukažme to na příkladu:

```
>>> sez = ['první', 'druhý', 'třetí']
>>> sez
['první', 'druhý', 'třetí']
>>> ''.join(sez)
'první druhý třetí'
>>> '...'.join(sez)
'první...druhý...třetí'
>>> dopis(str(2021))
['2', '0', '2', '1']
>>> '.'.join(list(str(2021)))
'2.0.2.1'
>>> '.'.join('Python')
'P.y.t.h.o.n'
```

Seznamy n-tice (tuple)

Jsou to vlastně jen neměnitelné (immutable) seznamy. Pythonovský typ ***tuple()*** dokáže dělat skoro všechno to jisté jako ***list()*** kromě mutable operací. Takže to nejprve shrňme a pak si ukážeme, jak to pracuje:

• operace ***+*** na zřetězování (spojování dvou n-tic)
• operace ******* pro vícenásobné zřetězování (vícenásobné spojování jedné n-tice)
• operace ***in*** pro zjišťování, zda se nějaká hodnota nachází v n-tici
• operace indexování ***[i]*** ke zjištění hodnoty prvku na zadaném indexu
• operace řezu ***[i:j:k]*** ke zjištění hodnoty nějaké hlavě n-tice
• relační operace ***==, !=, <, <=, >, >=*** pro porovnávání obsahu dvou n-tic
• metody ***count()*** a ***index()*** pro zjišťování počtu výskytů, resp. indexu prvního výskytu
• procházení n-tice pomocí ***for-cyklu*** (iterování)
• standardní funkce ***len(), sum(), min(), max()*** které zjišťují něco o prvcích n-tice

Takže n-tice, stejně jako seznamy jsou strukturované typy, t.j. jsou to typy, které obsahují hodnoty nějakých (možná různých) typů (jsou to tzv. kolekce):

• konstanty typu n-tica uzavíráme do kulatých závorek a navzájem oddělujeme čárkami
• funkce ***len()*** vrátí počet prvků n-tice,

n-tice s jednou hodnotou

Zapíšeme-li libovolnou hodnotu do závorek, není to n-tice (v našem případě je to jen jedno celé číslo).
Pro jednoprvkovou n-tici musíme do závorek zapsat i čárku:

```
>>> p = (12,)
```

Pro Python jsou důležitější čárky než závorky. V mnoha případech si Python závorky domyslí (čárky si nedomyslí nikdy).

Operace s n-ticemi
Operace fungují přesně stejně, jako fungovaly s řetězci a seznamy:

• operace ***+*** zřetězí dvě n-tice, to znamená, že vyrobí novou n-tici, která obsahuje nejprve všechny prvky první n-tice a za tím všechny prvky druhé n-tice; zřejmě oba operandy musí být n-tice: nemůžeme zřetězovat n-tici s hodnotou jiného typu
• operace ******* zadané číslo-krát zřetězí jednu n-ticu, to znamená, že vyrobí novou n-ticu, která požadovaný-krát obsahuje všechny prvky zadané n-tice; operace vícenásobného zřetězování má jeden operand typu n-tica a druhý musí být celé číslo
• operace ***in*** umí zjistit, zda se nějaký prvek nachází v n-tici
• operace ***is*** zjišřuje zada dva objekty mají identickou referenci

n-tice může obsahovat jako své prvky i jiné n-tice:

```
>>> střed = (150, 100)
>>> p = ('střed', střed)
>>> p
('střed', (150, 100))
>>> len(p)
2
```

Funkce tuple()

Vytváří n-ticu z libovolné posloupnosti iterovatelného objektu, například se znakového řetězce, ze seznamu, vygenerované posloupnosti celých čísel pomocí *range(), a i z otevřeného textového souboru, v tomto případě, se řádky postupně stanou prvkami n-tice.*

For-cyklus s n-ticemi

for-cyklus je programová konstrukce, která postupně prochází všechny prvky
nějakého iterovatelného objektu. Doposud jsme se setkali s iterováním pomocí funkce *range(),*
procházením prvků řetězce *str* a seznamu *list(),* i celých řádků textového souboru.

Ale už od 2. přednášky jsme používali i takový zápis:

```
for i in 2, 3, 5, 7, 11, 13:
    print('prvocislo', i)
```

V tomto zápisu už nyní vidíme n-ticu (tuple): 2, 3, 5, 7, 11, 13. Jen jsme tomu nemuseli dávat závorky.
Jelikož i n-tica je iterovatelný objekt, Můžeme ji používat ve for-cyklu stejně jako jiné iterovatelné typy.
Totéž jako předchozí příklad bychom zapsali například i takto:

```
čísla = (2, 3, 5, 7, 11, 13)
for i in cisla:
    print('prvocislo', i)
```

Prvky n-tice mohou být nejrůznější objekty, a i funkce:

```
>>> rozne = ('retazec', (100, 200), 3.14, len)
>>> for prvok in rozne:
      print(prvok, type(prvok))
```

Pomocí for-cyklu můžeme n-tice také vytvářet a skládat podobně, jako se seznamy

Funkce enumerate()

Funkce *enumerate()* ve skutečnosti z jedné libovolné posloupnosti vygeneruje posloupnost dvojic, které jsou již typu *tuple().* Tuto posloupnost dále procházíme *for-cyklem.*
Pro zadání:

```
sez = (2, 3, 5, 7, 9, 11, 13, 17, 19)
for dvojice in enumerate(sez):
    ix, pr = dvojice
    print(f'{ix}. prvocislo je {pr} ... dvojica = {dvojica}')
```

Dostaneme tento výsledek:

```
0. prvocislo je 2 ... dvojica = (0, 2)
1. prvocislo je 3 ... dvojica = (1, 3)
2. prvocislo je 5 ... dvojica = (2, 5)
3. prvocislo je 7 ... dvojica = (3, 7)
4. prvocislo je 9 ... dvojica = (4, 9)
5. prvocislo je 11 ... dvojica = (5, 11)
6. prvocislo je 13 ... dvojica = (6, 13)
7. prvocislo je 17 ... dvojica = (7, 17)
8. prvocislo je 19 ... dvojica = (8, 19)
```

Takže funkce *enumerate()* vytváří *n-ticu,* posloupnosti *n-ticových* dvojic:

```
>>> tuple(enumerate(sez))
  ((0, 2), (1, 3), (2, 5), (3, 7), (4, 9), (5, 11), (6, 13), (7, 17), (8, 19))
```

Indexování

stejně jako jsme indexovali seznamy:
• prvky posloupnosti můžeme indexovat v ***[]*** závorkách, přičemž index musí být od 0 až po počet prvků-1
• pomocí řezu (slice) umíme indexovat část n-tice (něco jako podřetězec) tak, že [] závorek zapíšeme i dvojtečku:

ntice[od:do] n-tice z prvků s indexy od až po do-1
ntice[:do] n-tice z prvků od začátku až po prvek s indexem do-1
ntice[od:] n-tice z prvků s indexy od až po konec n-tice
ntica[od:do:krok] n-tice z prvků s indexy od až po do-1, přičemž bereme každý krok prvek

Porovnávání n-tic

Porovnávání n-tic je velmi podobné jako porovnávání řetězců a seznamů.
Připomeňme si, jak je to při seznamech:

• postupně porovnává i-te prvky obou seznamů, dokud jsou stejné; při první nerovnosti je výsledkem srovnání těchto dvou hodnot
• je-li při první neshodě v prvním seznamu menší hodnota než ve druhém, tak první seznam je menší jako druhý
• je-li první seznam kratší než druhý a odpovídající prvky se shodují, tak první seznam je menší jako druhý

Říkáme tomu lexikografické srovnávání.

Tedy i při porovnávání n-tic se budou postupně porovnávat odpovídající si prvky a při první nerovnosti se zkontroluje, který z těchto prvků je menší. Třeba zde ale dodržovat jedno velmi důležité pravidlo: porovnávat hodnoty například na menší můžeme jen tehdy, když jsou shodných typu.

Nejlepší je porovnávat takové n-tice, které mají prvky stejného typu. U n-ticí, které mají smíšené typy si musíme dávat větší pozor.

Vícenásobné přiřazení

Nejprve připomeňme, jak funguje vícenásobné přiřazení: je-li před znakem přiřazení = více proměnných, které jsou odděleny čárkami, tak za znakem přiřazení musí být iterovatelný objekt, který má přesně tolik hodnot, jako počet proměnných.
Iterovatelným objektem může být seznam **(list),** n-tica **(tuple),** znakový řetězec **(str),** generovaná posloupnost čísel **(range())** ale také otevřený textový soubor **(open()),** který má přesně tolik řádků, kolik je proměnných v přiřazení.

Pokud do jedné proměnné přiřazujeme více hodnot oddělených čárkou, Python to chápe jako přiřazení ntice:
Uvědomte si rozdíl, mezi těmito dvěma přiřazení:

```
>>> a = 3.14
>>> b = 3,14
>>> print(a, type(a))
   3.14 <class 'float'>
>>> print(b, type(b))
   (3, 14) <class 'tuple'>
```

Vícenásobné přiřazení používáme například i pro výměnu obsahu dvou (i více) proměnných:

```
>>> x, y, z = y, z, x
```

I v tomto příkladu je na pravé straně přiřazení (za =) n-tica: (y, z, x).

N-tica jako návratová hodnota funkce

V Pythonu se dost využívá toho, že návratovou hodnotou funkce může být n-tica, tzn. výsledkem funkce je najednou několik návratových hodnot.

Například následující příklad počítá celočíselné dělení a současně
zbytek po dělení:

```
def zjistit(a, b):
    return a // b, a % b
```

Funkci můžeme použít například takto:

```
>>> podíl, zbytek = zjisti(153, 33)
>>> print('podíl =', podíl, 'zvysek =', zbytek)
podíl = 4 zbytek = 21
```

Pokud z výsledku takové funkce potřebujeme použít jen jednu z hodnot, můžeme zapsat:

```
>>> print('zvysek =', zjisti(153, 33)[1])
```

Další funkce vrátí posloupnost všech dělitelů nějakého čísla:

```
def dělitele(cislo):
    vysl = ()
    for i in range(1, cislo+1):
        if cislo % i == 0:
            vysl = vysl + (i,)
    return vysl
```

Další funkce a metody

S n-ticami umí pracovat následující standardní funkce:
• ***len(ntica)*** - vrátí počet prvků n-tice
• ***sum(ntica)*** - vypočítá součet prvků n-tice (všechny musí být čísla)
• ***min(ntica)*** - zjistí nejmenší prvek n-tice (prvky se musí dát navzájem porovnat, nemůžeme zde míchat různé typy)
• ***max(ntica)*** - zjistí největší prvek n-tice (jako při min() ani zde se nemohou typy prvků míchat)

Na rozdíl od seznamů a znakových řetězců, které mají velké množství metod, n-tice mají jen dvě:
• ***ntica.count(hodnota)*** - zjistí počet výskytů nějaké hodnoty v n-tici
• ***ntica.index(hodnota)*** - vrátí index (pořadí) v n-tici prvního (nejlevnějšího) výskytu dané hodnoty, pokud se hodnota v n-tici nenachází, metoda způsobí spadnutí na chybě:
(ValueError: tuple.index(x): x not in tuple)

n-tice a grafika

canvas.update()
canvas.after(300)

Tyto dva příkazy označují, že se grafická plocha překreslí (aktualizuje) pomocí metody ***update()*** a poté se na 300 milisekund (0.3 sekundy) pozdrží výpočet pomocí metody ***after().***

Metoda ***extend(),*** přilepí k stávajícímu seznamu na konec nový seznam.

```
>>> a.extend(b)
```

Je totéž jako:

```
>>> a += b
```

Následující příklad předvede použití n-tic v grafickém režimu.
Definujeme několik bodů v rovině a poté pomocí nich kreslíme nějaké barevné polygony:

```
a = (70, 150)
b = (200, 200)
c = (150, 250)
d = (120, 70)
e = (50, 220)

canvas = tkinter.Canvas()
canvas.pack()

utvar1 = e, a, c
utvar2 = e, d, b
utvar3 = a, b, c, d
utvar4 = a, c, d, b

cyklus = (utvar1, 'green'), (utvar2, 'yellow'), (utvar3, 'red'), (utvar4, 'blue')

for utvar, barva in cyklus:
        canvas.create_polygon(utvar, fill=barva)
```

Pokud se do proměnné cyklu postupně přiřazují nějaké dvojice hodnot a tyto by se na začátku těla rozdělily do dvou proměnných, můžeme přímo tyto dvě proměnné použít jako proměnné cyklu (jako kdyby vícenásobné přiřazení). Podobnou ideu jsme mohli vidět při použití enumerate().

Podívejme se na n-ticu, která se prochází tímto for-cyklem:

```
>>> cyklus
((((50, 220), (70, 150), (150, 250)), 'green') (((50, 220), (120, 70),
(200, 200)), 'yellow') (((70, 150), (200, 200), (150, 250), (120, 70)),
'red') (((70, 150), (150, 250), (120, 70), (200, 200)), 'blue'))
```

Vidíme, že n-tice v takovém tvaru je dost obtížně čitelná, ale for-cyklus jí normálně rozumí.

Seznamy a grafika

Již víme, že většina grafických příkazů, například *create_line(), create_polygon(), …* akceptují jako parametry nejen čísla, ale také n-tice nebo i seznamy čísel, resp. seznamy/dvojice čísel.

Pokud bychom chtěli využít grafickou funkci ***coords(),*** která modifikuje souřadnice nakreslené křivky, nemůžeme jí poslat seznam souřadnic (dvojic čísel x a y), ale tato vyžaduje plochý seznam (nebo n-ticu) čísel.

```
import tkinter
import random
canvas = tkinter.Canvas(bg='white')
canvas.pack()
poly = canvas.create_polygon(0, 0, 0, 0, fill='yellow', outline='blue')
křivka = []
for i in range(100):
        bod = [random.randrange(350), random.randrange(250)]
        křivka.extend(bod)                          # totéž jako křivka += bod
        canvas.coords(poly, křivka)
        canvas.update()
        canvas.after(300)
tkinter.mainloop()
```