



Přehled lekce

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / PŘEHLED LEKCE

Úplné základy tedy známe. Umíme pracovat se základními datovými typy, známe smyčky a umíme pracovat s proměnnými. Než se ale posuneme dál, na úvod této lekce si ukážeme, jak můžeme pracovat s proměnnými efektivněji.

Poté můžeme překročit pomyslný milník, čímž jsou funkce. Funkce jsou jedním ze základních stavebních kamenů Pythonu, ale mnoha dalších programovacích jazyků.

Proč jsou funkce tak důležité?

- Funkce ti umožní používat svůj kód opakovaně. Díky tomu jej nemusíš psát pořád znova a znova.
- Používáním funkcí tedy ušetříme hromady času.
- Znalosti funkcí posunou tvé programátorské schopnosti o úroveň výše. Také je to základ pro následující lekci - **Funkční rámce a Vstupy!**

Na závěr také nakousneme práci s moduly.

Pojďme na to!

PROMĚNNÉ 2.0

Opakování proměnných

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / PROMĚNNÉ 2.0 / OPAKOVÁNÍ PROMĚNNÝCH

Do této chvíle jsme vždy přiřadili k jedné proměnné jeden výraz/hodnotu.

proměnná = výraz

Když Python narazí na takový zápis, udělá následující operace:

1. přiřadí hodnotu k proměnné,
2. vytvoří referenci mezi hodnotou a proměnnou.

Příklady klasického přiřazování výrazů a hodnot k proměnným

```
1 # Příklad
2 rok_narozeni = 1988
3 aktualni_rok = 2018
4 vek = aktualni_rok - rok_narozeni
5
6 #Tisk
7 print("Věk: " + str(vek))
```

30

Základy máme dobré. Nyní si ukážeme, jak je možné vytvořit **více proměnných najednou** nebo jinými slovy, jak do více proměnných přiřadit více hodnot **na jednom řádku**.

Více proměnných najednou

Jak co nejjednodušejí **vytvořit více proměnných najednou**? Stačí zapsat více jmen proměnných oddělených čárkou za sebe, poté napsat rovná se a nakonec **hodnoty**, které chceme proměnným přiřadit. Počet hodnot a proměnných musí být stejný.

Příklad:

```
1 # Vytvoreni vice promennych na jednom radku
2 a, b, c = 1, 2, 3
3
4 # Tisk promennych
5 print(a, b, c)
```

1 2 3

Pojďme si nyní rozebrat, jak pracovat s více proměnnými u konkrétních datových typů.

String, list

Vytvářet více proměnných najednou můžeme také pomocí **sekvenčních datových typů** (list, tuple, string). Začneš stejně jako v předchozím případě - zapíšeš jména proměnných oddělená čárkou. Za rovná se pak ale místo jednotlivých položek zapíšeš nějakou sekvinci.

Pozor! Délka této sekvence nesmí přesáhnout počet proměnných. Do každé proměnné se pak uloží jeden prvek ze sekvence. Pokud tedy použiješ string, v každé proměnné bude jeden znak. Pokud použiješ list nebo string, v každé proměnné bude jeden prvek.

List

```
1 # Prirazeni prvku Lst do promennych
2 11, 12, 13, 14 = [1, 'pes', 2.3, ['vnoreny list']]
3
4 # Tisk promennych z Lst
5 print(11, 12, 13, 14)
```

1 pes 2.3 ['vnoreny list']

U listu můžeme využít slicing

```
1 # Nas list
2 lst = [1, 'pes', 2.3, ['vnoreny list']]
3
4 # Prirazeni prvku Lst do promennych
5 predposledni, posledni = lst[-2:]
6
7 # Tisk
8 print(predposledni)
9 print(posledni)
```

2.3
['vnoreny list']

String

```
1 # Prirazeni znaku str do promennych
2 p1, p2, p3, p4 = 'ahoj'
3
4 # Tisk promennych ze str
5 print(p1)
6 print(p2)
7 print(p3)
8 print(p4)
```

a
h
o
j

Tvoje úloha

- Na řádku 2 vytvoř příkazem na jeden řádek dvě proměnné: **a**, **b**. Přiřaď jim libovolné hodnoty.
- Na řádku 5 vytiskni tyto proměnné.
- Na řádku 11 si znova vypiš **a** a **b**.
- Na řádku 17 si vytiskni promennou **a03**.
- Na řádku 20 rozlož poměnnou **a03** na 4 proměnné **b01**, **b02**, **b03** a **b04**.
- Na řádku 23 vypiš proměnnou **b04** tak, aby vypsala jenom obsah listu s **x** a **y**.

```
1 # Prirazení promenných
2 a, b =
3
4 # Tisk a,b
5 print()
6
7 # Prepsani promennych
8 a, b = b, a + b
9
10 # Opakovany tisk a,b
11 print()
12
13 # Prirazení novych promennych
14 a01, a02, a03 = [a, b, [1, 2, 3, ['x', 'y']]]
15
16 # Tisk a03
17 print(a03)
18
19 # Rozlozeni promenne a03
20 b01...
21
22 # Tisk "x" a "y" z b04
23 print(b04[0], b04[1])
```

SPUSTIT ZNOVU

```
Traceback (most recent call last):
  File <string>, line 2
    a, b =
SyntaxError: invalid syntax,<string>,2,7,a, b =
```

Řešení

V rozbalovacím boxu najdeš řešení úlohy s vysvětlením.

Zobrazit řešení

```
1 # Prirazení promenných
2 a, b = 1, 2
3
4 # Tisk a,b
5 print(a, b)
6
7 # Prepsani promennych
8 a, b = b, a + b
9
10 # Opakovany tisk a,b
11 print(a,b)
12
13 # Prirazení novych promennych
14 a01, a02, a03 = [a, b, [1, 2, 3, ['x', 'y']]]
15
16 # Tisk a03
17 print(a03)
18
19 # Rozlozeni promenne a03
20 b01, b02, b03, b04 = a03
21
22 # Tisk "x" a "y" z b04
23 print(b04[0], b04[1])
```

Range, dictionary, set

PYTHON #2: FUNKCE A SMÝČKY / ÚVOD DO FUNKCÍ / PROMĚNNÉ 2.0 / RANGE, DICTIONARY, SET

Pojďme se podívat na další datové typy:

Range

```
1 # Prirazení pomocí range
2 a, b = range(2)
3
4 # Tisk promenných
5 print(a, b)
```

0 1

Dictionary

Rozbalování pomocí slovníku má defaultně nastaveno, že nahraje klíče do proměnných. Klíče nebudou seřazeny!

Dictionary klíče

```
1 # Slovník
2 d = {'name': 'Bob', 'surname': 'Francis'}
3
4 # Prirazení klícu
5 kategorie1, kategorie2 = d
6
7 # Tisk promenných
8 print(kategorie1)
9 print(kategorie2)
```

'name'
'surname'

Dictionary hodnoty

Kdybychom chtěli místo klíčů přiřadit hodnoty, museli bychom k nim přistoupit pomocí metody `dict.values()`:

```
1 # Prirazení hodnot
2 name, surname = d.values()
3
4 # Tisk promenných
5 print(name)
6 print(surname)
```

'Bob'
'Francis'

Dictionary prvky

A kdybychom chtěli uložit do proměnných tuple obsahující páry `'klíč': 'hodnota'`, museli bychom použít metodu `dict.items()`:

```
1 # Prirazení hodnot
2 item1, item2 = d.items()
3
4 # Tisk promenných
5 print(item1)
6 print(item2)
```

('name', 'Bob')
(surname, 'Francis')

Set

Syntaxe bude obdobná jako u sekvencí nebo klíčů slovníku. Neměli bychom se spoléhat na to, že se nám hodnoty sítě uloží do proměnných v pořadí, ve kterém byly nahrány do setu!:

```
1 # Set
2 names = {'Hans', 'Helga', 'Hilda'}
3
4 # Prirazení promenných
```

```
5 n1, n2, n3 = names
6
7 # Tisk promennych
8 print(n1)
9 print(n2)
10 print(n3)
```

```
'Hilda'
'Hans'
'Helga'
```

Tuple

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / PROMĚNNÉ 2.0 / TUPLE

V kurzu Python online začátečník jsme si říkali, že **tuple** můžeš vytvořit až **třemi** různými způsoby:

- pomocí kulatých závorek: `()`,
- pomocí funkce `tuple()`,
- a třetí způsob je bez závorek:

```
1 tpl = 'prvni vec', 'druha vec', 3
```

Před chvílí jsme si ukazovali, jak vytvořit více proměnných najednou. V podstatě jsme tedy vytvořili tuple, aniž bychom o tom věděli. Rovnou jsme jej ho rozbalili do více proměnných:

```
1 # Hodnoty 'prvni vec', 'druha vec', 3 - tvori tuple
2 prom1, prom2, prom3 = 'prvni vec', 'druha vec', 3
```

V předchozí kapitole jsme se naučili, že **počet přiřazovaných hodnot se musí rovnat počtu proměnných**. Tuto překážku můžeš obejít **použitím závorek**. Hodnoty v závorkách se chovají jako jedna samostatná hodnota:

```
1 # Pouziti kulatych zavorek
2 a, b, c = 1, 2, (3, 4)
3
4 # Tisk a, b, a c
5 print(a, b)
6 print(c)
```

```
1 2
(3, 4)
```

Příklad výše je vlastně **tuple vnořený v jiném tuple**. Dal by se zapsat i takto:

```
1 # Pouziti kulatych zavorek pro oba tuply
2 a, b, c = ( 1, 2, (3, 4) )
3
4 # Tisk a, b, a c
5 print(a, b)
6 print(c)
```

```
1 2
(3, 4)
```

Rozbalování

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / PROMĚNNÉ 2.0 / ROZBALOVÁNÍ

Existuje ještě jeden způsob, jak vyřešit situaci, kdy máš méně proměnných než přiřazovaných hodnot. Místo hodnot budeš manipulovat s **proměnnými**. Pomocí `*` určíš, do které proměnné se má uložit **přebytek** hodnot:

```
1 # Priazeni prebytecnych hodnot
2 prvni, *zbytek = 'AHOJ'
3
4 # Tisk promennych
5 print('Nejdrive', prvni, 'a potom', zbytek)
```

```
Nejdrive A a potom ['H', 'O', 'J']
```

Do proměnné `prvni` se uložil první znak ze slova `'AHOJ'` - tady není nic nového. Na proměnnou `zbytek` ale zbývá víc než jeden znak. Za normálních okolností by ti Python vyspal chybu, ale protože před proměnnou `zbytek` je hvězdička, Python s tím nemá problém a do proměnné uloží zbývající znaky.

Symbol **rozbalení nenesí být jen na poslední proměnné**:

```
1 # Rozbaleni
2 prvni,*stred, posledni = range(10)
3
4 # Tisk
5 print(prvni)
6 print(posledni)
7 print(stred)
```

```
0
9
[1, 2, 3, 4, 5, 6, 7, 8]
```

Chybné přiřazování

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / PROMĚNNÉ 2.0 / CHYBNÉ PŘIŘAZOVÁNÍ

Pokud chceme přiřazovat více hodnot více proměnným, musíme mít na paměti, že **počet prvků musí být na obou stranách stejný!** Pokud tomu tak není, mohou nastat dvě situace, jejichž výsledkem je chyba:

1. Větší počet **hodnot** než proměnných

```
>>> a, b = 1, 2, 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
```

2. Větší počet **proměnných** než hodnot

```
>>> a, b, c = 1, 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not enough values to unpack (expected 3, got 2)
```

V obou případech dostaneme chybové hlášení - `ValueError`. Této chybě se ale dá zabránit rozbalováním, které už známe.

Tvoje úloha

- Na řádku 2 oprav kód pomocí rozbalování tak, abych nevracel chybové hlášení.
- Na řádku 5 vytiskni proměnné **a** a **b**.

```
1 # Oprav kod
2 a, *b = 'Kolo'
3
4 # Tisk promennych
5 print(a,b)
```

SPUSTIT ZNOVU

```
K ['o', 'l', 'o']
```

Řešení

V rozbalovacím boxu najdeš řešení úlohy s vysvětlením.

Zobrazit řešení

```
1 # Oprav kod
2 a, *b = 'Kolo'
3
4 # Tisk promennych
5 print(a,b)
```

Chybné rozbalování

PYTHON #2: FUNKCE A SMÝČKY / ÚVOD DO FUNKCÍ / PROMĚNNÉ 2.0 / CHYBNÉ ROZBALOVÁNÍ

Existuje několik chyb, kterým je dobré se při rozbalování vyvarovat:

1. CHYBA

Co se stane, pokud je proměnná se symbolem pro rozbalení `*` přebytečná? Tedy jinými slovy, máme více proměnných než objektů, které chceme přiřadit?

Přebytečné proměnné je přiřazen prázdný list:

```
1 # Použití rozbalovacího operátoru
2 a,*b,c,d = 1,2,3
3
4 # Tisk proměnných
5 print(a,b,c,d)
```

(1, [], 2, 3)

Pokud ale není rozbalovací proměnná jedinou přebytečnou proměnnou, dostaneme chybu:

```
1 # Použití rozbalovacího operátoru
2 a, *b, c, d, e= range(3)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not enough values to unpack (expected at least 4, got 3)
```

2. CHYBA

Není možné používat 2 rozbalovací proměnné v jedné operaci:

```
1 # Použití rozbalovacího operátoru
2 a,*b,*c = 1,2,3,4,5,6
3
4 # Tisk proměnných
5 print(a,b,c)
```

```
File "<stdin>", line 1
SyntaxError: two starred expressions in assignment
```

3. CHYBA

Rozbalovací proměnná musí být umístěna v listu nebo tuplu. **Všimni si** čárky, která následuje za proměnnou `*a`.

```
1 >>> *a='abcd'
2 >>> a
... . . . . .
```

```
3 [ 'a', 'b', 'c', 'd' ]
```

Čárka nám simuluje tuple. Kdybychom ji nezahrnuli, nevložili bychom ji do tuple (který obsahuje jednu proměnnou) a dostali bychom chybu:

```
1 # Použití rozbalovacího operátoru
2 *a = 'abcd'
```

```
File "<stdin>", line 1
SyntaxError: starred assignment target must be in a list or tuple
```

Rozšířené přiřazení

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / PROMĚNNÉ 2.0 / ROZŠÍŘENÉ PŘIŘAZENÍ

Rozšířená přiřazení jsou zkratkou pro použití aritmetických operací na proměnnou, které přidělujeme výsledek daného výrazu.

Například kdybychom chtěli zvýšit hodnotu proměnné `pocitadlo` ve while loop, mohli bychom změnit zápis `pocitadlo = pocitadlo + 1` na rozšířené přiřazení: `pocitadlo += 1`.

Klasické přiřazení	Rozšířené přiřazení
<code>pocitadlo = pocitadlo + 1</code>	<code>pocitadlo += 1</code>

Podívejme se tedy, jak by rozšířené přiřazení vypadalo pro jednotlivé aritmetické operace:

Operace	Klasické přiřazení	Rozšířené přiřazení
součet	<code>a = a + b</code>	<code>a += b</code>
rozdíl	<code>a = a - b</code>	<code>a -= b</code>
násobení	<code>a = a * b</code>	<code>a *= b</code>
dělení	<code>a = a / b</code>	<code>a /= b</code>
dělení beze zbytku	<code>a = a // b</code>	<code>a //= b</code>
modulo	<code>a = a % b</code>	<code>a %= b</code>
mocnění	<code>a = a ** b</code>	<code>a **= b</code>

Syntaxe rozšířeného přiřazení šetří místo a umožňuje nám vyhnout se únavnému opakování stejné proměnné v daném výrazu. V podstatě se snažíme o aplikování operace (v tomto případě aritmetické) na určitý objekt, který je uložený v proměnné a **výsledek potom uložit do stejné proměnné**.

Rozšířené přiřazení se často používá ve **while loop**:

```
1 # Priřazení promenne
2 muj_string = 'Hello'
3
4 # Pocitadlo a index
5 i = 0
6
7 # While Loop
8 while i < len(muj_string):
9     print('Index ' + str(i) + ': ' + muj_string[i])
10
11     # ROZSIRENE PRIRAZENI <-
12     i += 1
```

```
Index 0: H
Index 1: e
Index 2: l
Index 3: l
Index 4: o
```

Úvod do funkcí

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / FUNKCE / ÚVOD DO FUNKCÍ

V minulosti jsem už používal různé funkce: `print()`, `input()`, `str()`, apod. Všechny fungují na principu vstupu a výstupu. V této sekci si rozebereme funkce podrobněji a ukážeme si, co se unitř funkcí vlastně děje a hlavně, jak si můžeme vytvářet svoje vlastní funkce.

Co je to funkce?

Funkce je nástroj, který umožňuje programátorům **seskupit blok kódu**, který dohromady **slouží společnému účelu**, tedy vykoná zamýšlenou akci. To znamená, že účelem funkcí je vykonat specifický úkol (například převést všechna písmena ve slovu z malých na velká písmena).

Špatná funkce byla například ta, která:

1. získá text ze souboru,
2. naformátuje text,
3. vytiskne text na obrazovku.

Dobrá funkce má pouze jediný účel! Funkce by tedy měla provést pouze jednu z výše uvedených operací.

Jak vypadá funkce v kódu?

Funkci poznáme tak, že hned za jejím názvem jsou kulaté závorky. Například funkce, která počítá sumu numerických hodnot, vypadá takto: `sum()`

Funkce přijímají vstupy

Čísla, jejichž sumu se snažíme získat, musíme vložit mezi závorky. Každá funkce má své specifické požadavky - jakou formu by měly vstupy mít, kolik vstupů bychom měli vložit. V případě `sum()` musíme vložit vstup jako tuple numerických hodnot:

```
1 sum((1,2,4))
```

Tuple bychom mohli vložit i **bez závorek tuplu**:

```
1 sum(1,2,4)
```

Funkce DĚLAJÍ věci

Funkce `sum()` seče a vráti výsledek výpočtu jako výstup:

```
1 # Vstup
2 soucet = sum( 1,2,4 )
3
4 # Tisk vystupu
5 print(soucet)
```

```
7
```

Vždy, když vložíme něco mezi závorky, například `str(25)` - vstup, uvnitř funkce se uskuteční nějaký proces a funkce nám něco vrátí - `'25'` - výstup.

Built-in vs definované funkce

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / FUNKCE / BUILT-IN VS DEFINOVANÉ FUNKCE

Některé funkce dostaneme k dispozici v Pythonu, abychom museli cokoliv dalšího instalovat. Těmto funkci říkáme **built-in funkce**. Python dokumentace uvádí seznam všech [built-in funkci](#). Už známe některé z nich, například `print()`, `input()`, apod.

Většina našeho programování se samozřejmě bude soustředit na **definování** vlastních funkcí. Způsob, jakým definujeme funkci, nám ukáže vše, co má funkce dělat, kolik vstupů očekává mezi závorkami a také jaký výstup má vrátit.

Definice naší vlastní funkce

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / FUNKCE / DEFINICE NAŠÍ VLASTNÍ FUNKCE

Funkce je vytvořena (**definována**) použitím klíčového slova **def**. Obecný zápis takto definované funkce vidíme níže:

```
1 def jmeno_funkce(argument1, arg2,...):  
2     odsazený blok kódu  
3     odsazený blok kódu  
4     odsazený blok kódu  
5     nepovinný výraz return
```

Konkrétním příkladem by mohla být funkce, která sčítá dvě čísla:

```
1 def secti_dve_cisla(a,b):  
2     return a + b
```

Každá část zápisu funkce je důležitá a bude rozebrána podrobněji. Zde jsou jednotlivé komponenty funkce:

- klíčové slovo **def**, které říká Pythonu, aby vytvořil nový objekt funkce,
- **jmeno_funkce**, což je jméno proměnné, které má být objekt přidelen,
- **()** kulaté závorky, která následují hned za jménem funkce - zde přijdou vstupy,
- **argumenty**, vstupy funkce, které jsou vždy uvnitř kulatých závorek,
- **:** dvojtečka, která nám říká, že vytváříme složený výraz (hlavička + tělo),
- odsazení - všechn kód, který patří k funkci, by měl být odsazen doprava,
- klíčové slovo **return**, které dává Pythonu vědět, že následující výraz/vypočtená hodnota by měla být vrácena.

Funkce neexistuje, dokud program nedojde ke klíčovému slovu **def**. Toto klíčové slovo vytvoří objekt funkce (ano, funkce jsou také objekty - typu funkce) a přidělí jej do jména funkce (proměnná).

Když náš program dojde k definici funkce **secti_dve_cisla**, vytvoří proměnnou **secti_dve_cisla** a přidělí jí objekt funkce. Níže můžeme vidět, jak je objekt funkce reprezentován v paměti:

```
1 print(secti_dve_cisla)
```

```
<function secti_dve_cisla at 0x7f655a7b0ae8>
```

Jméno funkce

Jména funkcí by měla jasné vystihovat operace, které provádí. Funkci, která počítá odmocninu, bychom mohli nazvat **odmocnina**. Jméno funkce je jen další název proměnné, a proto zde platí stejná pravidla jako u pojmenování klasických proměnných.

Vstupy funkcí

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / FUNKCE / VSTUPY FUNKCÍ

Některé funkce vyžadují vstupy, které jsou potom dálé upraveny uvnitř funkce. Jak jsem si řekl, tyto vstupy musí být umístěny mezi **kulaté závorky - ()**, které pišeme za názvem funkce.

Definice funkce

Pokud chceme, aby naše funkce vyžadovaly argumenty, musíme je tak definovat. Funkce níže je definována tak, aby vyžadovala 2 argumenty:

```
1 def opakuj_znak(znak, pocet_opakovani):  
2     return znak * pocet_opakovani
```

Vstupy v definici funkce se nazývají **parametry funkce**.

Volání funkce

Nyní můžeme **zavolat funkci** **opakuj_znak** a poskytnout jí vstupy, které očekává:

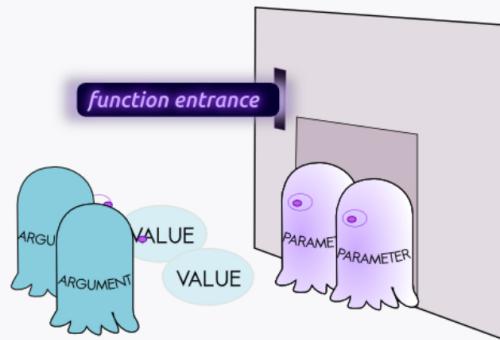
```
1 opakuj_znak('=',20)
```

```
'====='
```

Vstupy funkce, které vkládáme při volání funkce, se nazývají **argumenty funkce**.

Proč rozlišujeme mezi parametry a argumenty funkcí?

Když voláme funkci, argumenty předají své hodnoty parametrům. Parametry jsou vlastně proměnné, které odkazují na hodnoty, které jsme funkci poskytli.



Výstupy funkcí

PYTHON #2: FUNKCE A SMÝČKY / ÚVOD DO FUNKCÍ / FUNKCE / VÝSTUPY FUNKCÍ

Funkce jsou většinou definovány tak, aby mohly přijímat vstupy a vracet (**return**) výstupy. Vrácení výstupů se odehrává pomocí klíčového slova **return**. Když Python narazí na **return**, funkce se **okamžitě ukončí** a vrátí hodnotu, která následuje za klíčovým slovem **return**.

```
1 def max(sekvence):
2     max_prvek = sekvence[0]
3
4     for prvek in sekvence[1:]:
5         if max_prvek < prvek:
6             max_prvek = prvek
7
8     return max_prvek
```

Hodnota je vrácena na místo, na kterém jsme funkci zavolali. Čili **výsledek funkce může být uchován v proměnné**:

```
>>> max_cislo = max([32,2,4,1,4,54,23,12])
>>> max_cislo
54
```

Return vícekrát

Uvnitř jedné funkce můžeme mít více klíčových slov **return**. Princip je ale pořád stejný. Jakmile Python narazí na **return**, funkce se **okamžitě ukončí** a vrátí hodnotu, která následuje za klíčovým slovem. Pokud tedy následující výraz z funkce níže **if not s.istitle():** je vyhodnocen jako pravdivý (True), provede se výraz **return False** a **print('Bye')** není nikdy provedeno.

Definice funkce:

```
1 def vsechna_nazev(slova):
2
3     for s in slova:
4
5         if not s.istitle():
6             return False
7
8     print('Bye')
9     return True
```

Volání funkce:

```
>>> vsechna_nazev(['Bob', 'Frank', 'mike', 'John'])
```

```
False
```

Znovu voláme funkci:

```
>>> vsechna_nazev(['Bob','Frank','Mike', 'John'])
Bye
True
```

Return ani jednou

Funkce nemusí být vždy napsána tak, aby vracela určitou hodnotu. Například funkce níže pouze tiskne šachovnici.

Definice funkce:

```
1 def ukaz_sachovnici(rozmery):
2     for radek in range(rozmery):
3         print()
4
5         for sloupec in range(rozmery):
6             znak = '#' if (radek + sloupec) % 2 == 1 else ' '
7             print(znak, end='')
8     print()
```

Všimni si, že ve vnitřním loopu používáme druhý parametr v `print()` funkci - `end=' '`. Parametr je defaultně nastaven na `\n`. To říká Pythonu, že má vytisknout hodnotu uvnitř funkce a skočit na nový řádek - `\n`.

Proto jsme druhý parametr explicitně uvedli a změnili ho z nového řádku (`\n`) na prázdný string (`' '`). Pro Python to znamená, že při tisku v tomto loopu zůstane na stejném řádku.

Shrnutí příkladu:

- vytiskneme řádek znaků '#' v `range(rozmery)` - vnitřní loop
- po vytisknutí první řady, posuneme se na řádek další - vnější loop `for radek in range(rozmery)`
- použijeme `print()` a skočíme na další řádek
- jdeme znova do vnitřního loopu `for sloupec in range(rozmery)` čímž vytiskneme další řadu.

Volání funkce:

```
>>> ukaz_sachovnici(5)
# #
# #
# # #
# #
# # #
```

Pokud jsme do funkce nenapsali `return` a Python dojde na konec funkce, vrátíme value `None`. Tohle můžeme vidět, když vložíme výsledek funkce `ukaz_sachovnici` do proměnné a proměnnou vytiskneme:

```
>>> result = ukaz_sachovnici(5)
>>> result
# #
# #
# # #
# #
# # #
```

Jiné příklady funkce, které vráčí `None`, jsou třeba `print()` funkce nebo `.sort()` metoda listu a mnoho dalších. Pokud přidělíme hodnotu vrácenou témito funkcemi do proměnné, zjistíme, že proměnná obsahuje hodnotu `None`.

Vytvoř funkci - cvičení

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / FUNKCE / VYTVOŘ FUNKCI - CVIČENÍ

Než budeme pokračovat, udělejme si malé cvičení. Naším úkolem je vytvořit funkci, která:

Vezme vstupy:

- Počáteční číslo rozmezí celých čísel (typ range),
- poslední číslo rozmezí celých čísel (typ range)

Zpracuje vstupy:

- Projde range od počátečního do posledního čísla a seče všechna čísla.

Vrátí výstupy:

- Vrátí sumu všech čísel v range od počátečního do posledního čísla.

Tvoje úloha

- Na řádku 2 zapiš správně hlavičku funkce s parametry.
- Na řádku 5 vlož vhodnou hodnotu do pomocné proměnné **vysledek**
- Na řádku 8-9 projdi pomocí for loop a range jednotlivá čísla daného rozmezí.
- Na řádku 12 vrať výsledek.

```
1 # Hlavicka funkce
2 def :
3
4     # Pomocna promenna
5     vysledek =
6
7     # For loop
8     for :
9
10
11     # Vraceni vysledku
12     return
```

SPUSTIT KÓD

Zde se zobrazí výstup po spuštění kódu.

Řešení

V rozbalovacím boxu najdeš řešení úlohy s vysvětlením.

Zobrazit řešení

```
1 # Hlavicka funkce
2 def suma_range(od_cisla, do_cisla):
3
4     # Pomocna promenna
5     vysledek = 0
6
7     # For Loop
8     for cislo in range(od_cisla, do_cisla):
9         vysledek += cislo
10
11     # Vraceni vysledku
12     return vysledek
```

Když zavoláme funkce po tom, co jsme ji definovali, takto by měl vypadat výsledek:

```
1 # Zavolani funkce a ulozeni vysledku
2 vysledek = suma_range(1,5)
3
4 # Tisk vysledku
5 print(vysledek)
6
```

```
7 # Vysledek  
8 10
```

Volání funkce

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / FUNKCE / VOLÁNÍ FUNKCE

Abychom mohli spustit kód uvnitř funkce, **musíme funkci zavolat**. Funkci zavoláme tím, že napišeme její jméno a hned poté kulaté závorky. Závorky mohou, ale nemusí, obsahovat vstupy. V příkladu níže vidíme built-in funkci `all()`, která vyžaduje pouze jeden vstupní objekt v podobě sbírky. Funkce `all()` vrací hodnotu `True`, pokud jsou všechny prvky/výrazy/hodnoty sbírky pravdivé:

```
1 # Funkce all  
2 result = all([True, False, True, True])  
3  
4 # Tisk vystupu  
5 print(result)
```

```
False
```

To znamená, že:

1. funkce musí být nejdříve definovaná,
2. po jejím definování ji můžeme opakován volat kdekoliv v našem kódu,
3. výsledek funkce, může být uchován v proměnné.

Pokud zkusíme zavolat funkci, která ještě nebyla definována, dostaneme chybové hlášení v podobě `NameError`:

```
1 # Zavolani neexistujici funkce  
2 zadna_fce()
```

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'no_func' is not defined
```

Funkce `all()` je built-in uvnitř Pythonu. Můžeme ji bez problémů používat.

Jméno funkce bez závorek

Kdž napišeme jméno funkce bez závorek, dostaneme zpět objekt funkce jako takový, ne výstup funkce.

```
1 # Funkce bez zavorek  
2 print(all)
```

```
<built-in function all>
```

Pokud tedy chceme funkce **aktivovat** nebo zavolat, **musíme přidat kulaté závorky** za jméno funkce a vložit očekávané vstupy:

```
1 # Funkce se zavorkami a vstupy  
2 print( sum((1,2,3) ) )
```

```
6
```

Funkce je objekt

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / FUNKCE / FUNKCE JE OBJEKT

Jak jsem si řekl, v Pythonu je funkce také objekt. To znamená, že níže uvedený objekt funkce vytvořený klíčovým slovem `def` může být získán z paměti.

Podívej se na příklad:

```
1 def secti_dve_cisla(a,b):  
2     return a + b
```

Sekvence čísel a písmen, která následuje po 'function secti_dve_cisla at' je ve skutečnosti adresa v paměti, kde se objekt funkce nachází. Jméno funkce ukazuje na tuto adresu. Když napišeme jméno funkce, objekt, který Python získá z této adresy, může být přidělen jiné proměnné.

Podívej se na příklad:

```
1 # Ziskani funkce
2 print(secti_dve_cisla)
```

```
<function secti_dve_cisla at 0x7f655a7b0ae8>
```

```
1 # Prideleni funkce do promenne
2 nova_promenna = secti_dve_cisla
```

Proměnná `nova_promenna` odkazuje na vše, na co odkazuje proměnná `secti_dve_cisla`. Funkce `nova_promenna` nyní funguje stejně jako `secti_dve_cisla`. Jména obou proměnných odkazují na stejný objekt funkce, tím pádem mohou být vložena do jiné funkce.

```
1 # Zavolani nove_pridano a ulozeni vysledku
2 vysledek = nova_promenna(5,7)
3
4 # Tisk vysledku
5 print(vysledek)
```

```
12
```

```
1 def pozdrav():
2     print('Hello')
3
4 def obal(fce):
5     return fce()
```

Funkce `obal()` zavolá objekt funkce a vrátí jeho výsledek.

```
1 # Pouziti funkce 'obal'
2 obal(pozdrav)
```

```
Hello
```

Funkce vs. metody

PYTHON #2: FUNKCE A SMÝČKY / ÚVOD DO FUNKCÍ / FUNKCE / FUNKCE VS. METODY

Nejčastěji používáme dva typy funkcí.

Funkce

Některé funkce **stojí samostatně**, stejně jako třeba built-in funkce `sum()`.

```
>>> sum((1,2,3))
6
```

Metody

Nicméně některé funkce jsou **připoutány k jinému objektu pomocí operátoru tečka**: `.`:

```
>>> 'abcd'.upper() )
'ABCD'
```

Druhý typ funkcí se nazývá metody. Ty vykonávají své operace (pomocí operátoru tečky) na objektech, ke kterým patří.

Například metoda `.upper()` nemá nic vloženo mezi kulaté závorky. To proto, že metoda vykonává operaci na stringu, ke kterému je připojena. Jediným vstupem této metody je daný string.

Funkce v praxi

Funkce dělají kód lepším, srozumitelnějším a čitelnějším pro čtenáře. Jsou užitečné, protože:

1. šetří místo **opakováným použitím** kódu,
2. poskytují prostředky **abstrakce**.

Znovupoužití kódu

Funkce umožňují opakování spuštění kódu v průběhu programu. Nemusíme kopírovat a vkládat ten samý kód znova a znova. Funkce slouží jako boxy pro zpracování kódu. Přijmou data, spustí kód, který může zpracovat nebo jinak použít vstupy, a mohou také vrátit výsledek.

Abstrakce

Funkce by se měly soustředit na jeden úkol a tím pádem by neměly být příliš dlouhé. To zaručuje lepší čitelnost a snadnější údržbu kódu do budoucna. Funkce by neměly provádět více než 1 úkol. Měly by být designovány tak, aby reprezentovaly pouze 1 úkol. Musíme se jen zamyslet, jaký úkol mám funkce udělat. A poté jen vložit potřebné prvky do funkce.

Příklad

Chceme vytvořit program, který spočítá body mass index (BMI) nějakého člověka. Rádi bychom přivedli váhu z liber na kilogramy a výšku z palců na metry.

Celý program bychom si měli rozdělit do následujících kroků:

1. převod výšky,
2. převod váhy,
3. spočítání BMI (vydělit váhu v kg výškou na druhou).

To znamená, že potřebujeme funkcionalitu ne pouze na spočítání BMI, ale také pro převod z liber a palců.

```

1 def na_kg(vaha):
2     return vaha * 0.45
3
4 def na_metry(vyska):
5     return vyska * 0.025
6
7 def bmi(vaha, vyska):
8     return na_kg(vaha) / na_metry(vyska) **2

```

```

1 # Zavolani funkce na bmi
2 moje_bmi = bmi(125, 63)
3
4 # Tisk vysledku
5 print(moje_bmi)

```

22.67573696145124

SPECIÁLNÍ ZNAKY STRING

Přehled

Speciální znaky mají v Pythonu speciální význam. Například tabulátor slouží k ukončení řádku, návratu na začátek řádku, apod. Zde je krátký seznam některých užitečných speciálních znaků a jejich zápis:

- **\n ASCII Linefeed/Newline** (LF) - posun o řádek,
- **\r Carriage Return** (CR) - posun na začátek řádku,
- **\t Horizontal Tab** (TAB) - horizontální odsazení,
- **\b ASCII backspace** (BS) - návrat o 1 znak zpět

- **\f** ASCII formfeed (FF) - posun na další stránku.

Escaping

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / SPECIÁLNÍ ZNAKY STRING / ESCAPING

Než se vrhneme na jednotlivé speciální znaky, měli bychom zmínit **backslash** (obrácené lomítko), které jim předchází. Možná víš, že všechny speciální znaky v přechozí podkapitole zahrnovaly znak backslash ****. Backslash signalizuje, že následující znak by měl být interpretován speciálním způsobem.

Jak ale můžeme Pythonu říct, aby vytisknul backslash jako takový? V tomto případě musíme přidat další backslash:

```
1 # Tisk znaku backslash
2 print('\\')
```

```
\
```

Tisk samostatného znaku backslash by nám vrátil chybu v pohledu **SyntaxError**.

```
1 # Chybny tisk znaku backslash
2 print('\\')
```

```
File "<stdin>", line 1
    print('\\')
          ^
SyntaxError: EOL while scanning string literal
```

Vložení backslash před jiný backslash nebo úplně jiný znak, který má speciální význam pro Python, nazýváme **escaping** - escapování (eskejpcování). **Escaping** zaručuje, že znak, který následuje za **backslash**, bude vytisknutý.

Escaping a uvozovky

Kdybychom chtěli zahrnout jednoduché uvozovky, které se nachází ve stringu, který je uzavřen jinými jednoduchými uvozovkami, potřebovali bychom backslash před danou uvozovkou.

```
1 # Tisk znaku jednochuche uvozovky
2 print('It\'s time for breakfast')
```

```
It's time for breakfast
```

Pokud **neunikneme** (escape) jednoduchým uvozovkám, dostaneme **SyntaxError**.

```
1 # Chybny tisk znaku jednochuche uvozovky
2 print('It's time for breakfast')
```

```
File "<stdin>", line 1
    print('It's time for breakfast')
          ^
SyntaxError: invalid syntax
```

Jiné řešení by bylo uzavřít string dvojitými uvozovkami. Potom bychom nemuseli *uniknout* jednoduchým uvozovkám uvnitř našeho stringu a kód by vypadal mnohem lépe.

```
1 # Pouziti dvojitych uvozovek
2 print("It's time for breakfast")
```

```
It's time for breakfast
```

Vše, co je uvedeno výše, platí i pro dvojité uvozovky uvnitř stringu. Kdybychom to chtěli udělat, **unikli** bychom dvojitým uvozovkám například tak, že bychom použili jednoduché uvozovky:

```
1 # Pouziti dvojitych uvozovek
2 print('It's time for breakfast')
```

```
It's time for breakfast
```

Lineteed / Newline

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / SPECIÁLNÍ ZNAKY STRING / LINEFEED / NEWLINE

Znak **newline** řekne Pythonu, že má **pokračovat na dalším řádku**. Jeho zkratka může být buď **NL**, nebo **LF** (linefeed). Běžně se zapisuje takto - "**n**". Ve Windows se znak newline používá společně s **carriage return** - "**\r**". Použitím tohoto znaku ukončíme řádek a kurzor se posunu na začátek rádu dalšího - "**\r\n**".

```
1 # Pouziti newline
2 print('This is the first line\nThis is the second line')
```

```
This is the first line
This is the second line
```

Trocha historie

CR = Carriage Return a LF = Line Feed. Dva výrazy, jejichž kořeny sahají až ke starým psacím strojům. LF posunul papír nahoru, ale horizontálně zůstal na stejně pozici. CR posunul jezdce ("**carriage**") zpět, aby další znak, který se napiše, byl úplně napravo na papíru, ale na stejném řádku.

CR+LF dělal obojí. Tzn. vrátil se na začátek nového řádku. Jak šel čas a lidé začali přecházet na počítače, designéři OS se rozhodli používat pouze jeden znak. Většina moderních textových editorů a textově orientovaných aplikací automaticky detekuje konce řádku daného souboru.

Carriage return

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / SPECIÁLNÍ ZNAKY STRING / CARRIAGE RETURN

Znak **carriage return** řekne Pythonu, aby se vrátil na začátek aktuálního řádku, aniž by se posunul o řádek niže. Zkratkou se zapisuje **CR** a v kódu vypadá takto: "**\r**". Název **carriage** (jezdce) vychází z jezdce u tiskárny.

```
1 # Pouziti carriage return
2 print('This is the first line\rThis is the second line')
```

```
This is the second line
```

V příkladu výše je string '**This is the first line**' vytisknut první, ale carriage return vráti kurzor (jezdce u tiskárny) na začátek stejného řádku. Druhý string '**This is the second line**' je potom vytisknut přes string první. Proto nevidíme první větu.

Tabulátor

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / SPECIÁLNÍ ZNAKY STRING / TABULÁTOR

Tabulátor říká Pythonu, aby přidal několik mezer (většinou 4) na místo, kde se nachází. Zkráceně se zapisuje **TAB** a v kódu vypadá takto: "**\t**".

```
1 # Pouziti tabulatoru
2 print('This is the first line\tThis is the second line')
```

```
This is the first line    This is the second line
```

Raw string

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / SPECIÁLNÍ ZNAKY STRING / RAW STRING

Už víme, že pokud vytiskneme zároveň znaky '**\n**', '**\r**', '**\t**', dostaneme se na začátek nového řádku a kurzor se posune o několik mezer doprava. Z úvodní podkapitoly (escaping) bychom si mohli odvodit, že když chceme vytisknout dané znaky, museli bychom použít jeden extra **backslash** - zpětné lomítko.

```
1 # Tisk speciálních znaku
2 print('\\\n', '\\r', '\\t')
```

```
\n \r \t
```

Kód se zpětným lomítkem nevypadá úplně pěkně. Řešením této situace je tzv. **raw string**. Raw string se vytvoří tak, že před náš string napišeme písmeno **r**:

```
1 # Pouziti raw string
2 print(r'\n, \r, \t')
```

```
\n, \r, \t
```

Speciální znaky nejsou interpretovány jako speciální. Jsou vytisknuty tak, jak jsou napsány.

```
1 # Pouziti raw string - dalsi priklad
2 print(r'First line \n Second Line \t tabbed part')
```

```
First line \n Second Line \t tabbed part
```

Něco navíc: Backspace, Formfeed

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / SPECIÁLNÍ ZNAKY STRING / NĚCO NAVÍC: BACKSPACE, FORMFEED

Backspace smaže předcházející znak. Zkratkou se zapisuje **BS** a v kódu takto: "**\b**". Příklad níže smaže písmeno **e** ze slova **line**:

```
1 # Pouziti backspace
2 print('This is the first line\bThis is the second line')
```

```
This is the first linThis is the second line
```

Formfeed přeskocí na další 'stránku'. Zkratkou se zapisuje **FF** a v kódu takto: "**\f**". Formfeed se běžně používal pouze jako oddělovač stránek. Nyní se používá také jako oddělovač sekcí. V textových editorech se tento znak používá, pokud vložíš 'ukončení stránky'. Zde je bohužel těžké demonstrovat jeho účinek.

```
1 # Pouziti formfeed
2 print('This is the first line\fThis is the second line')
```

```
This is the first line
This is the second line
```

PYTHON KNIHOVNA

Python moduly

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / PYTHON KNIHOVNA / PYTHON MODULY

Kód můžeme zabalit do funkce a použít ho později. Stejný princip můžeme uplatnit i u větší jednotky - Python souboru. Python soubory (soubory s koncovkou **.py**) se také nazývají moduly.

Moduly mohou obsahovat skripty, které můžeme spustit jako programy. To ale neplatí vždy. Některé moduly totiž obsahují pouze definice funkce (nebo jiné Python výrazy). K těmto funkcím potom můžeme přistoupit z jiného skriptu, a tím použít daný modul spíše jako **zásoobárnu funkcionality** než jako program.

Výhody modulů

1. Nemusíme všechno programovat sami,
2. moduly zpřehledňují kód,
3. části kódu můžeš využít vícekrát.

Základní moduly

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / PYTHON KNIHOVNA / ZÁKLADNÍ MODULY

Instalace Pythonu v sobě zahrnuje **knihovnu základních Python modulů**. Ty obsahují užitečné funkce, které spadají do různých oblastí programování (networking, datové typy, práce se soubory a další). Moduly Python knihovny poskytují standardní řešení pro mnoho problémů, které se mohou vyskytnout v každodenním programování.

Přehled modulů, který jsou v této knihovně dostupné, můžeme najít v [Python dokumentaci](#).

Existují ale také **knihovny třetí strany**, které poskytují dodatečné funkcionality pro Python.

Importování

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / PYTHON KNIHOVNA / IMPORTOVÁNÍ

Abychom mohli přistoupit k této funkcionalitě, musíme **importovat modul**. V Pythonu importujeme pomocí klíčového slova **import**. Za něj následuje jméno souboru, jehož funkcionalitu bychom rádi nahráli do našeho programu. Importování modulů zapisujeme vždy **na začátku** našich Python skriptů.

V následující sekci se zatím naučíme pracovat se **základním Python modulem** - **random**. Modul **random** nahrajeme následovně:

```
import random
```

Dále se o modulech učíme více do hloubky v našem pokročilejším Python kurzu. Zde je pouze krátký úvod do této problematiky, abychom lépe pochopili kontext, ve kterém mohou funkce operovat.

MODUL RANDOM

Úvod k modulu random

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / MODUL RANDOM / ÚVOD K MODULU RANDOM

Modul **random** používáme ve třech případech:

- pokud chceme vygenerovat náhodné číslo v určitém rozmezí (range),
- když chceme promíchat prvky v sekvenci (jako v balíčku karet),
- jestliže chceme zvolit prvek ze sekvence.

Modul random můžeme nahrát do našeho skriptu následovně: **import random**

Zápis importování by měl být vždy na úplném začátku našeho skriptu. Python nám sice dovolí zapsat importování kdekoli, ale zápis importování na prvním místě v kódu je nepřesaná a respektovaná domluva mezi programátory. Je důležité, aby jiní programátoři, kteří možná budou náš kód číst, věděli, jakou funkcionalitu používáme.

Nyní budeme chtít volat různé funkce z modulu random. Abychom to mohli udělat, budeme muset říct Pythonu, ve kterém modulu by měl hledat. Uděláme to tak, že specifikujeme **jméno modulu**, za kterým následuje **tečka** a **jméno funkce**.

Příklad volání funkce **randrange()** z modulu: **random.randrange(1,100)**

V následující podkapitole si vysvětlíme, jak jednotlivé funkce fungují.

Generování náhodného čísla

Importovali jsme modul random na začátku našeho skriptu: `import random`

Dále bychom chtěli **vygenerovat náhodné číslo mezi 0.0 a 1.0**. Uděláme to pomocí funkce `random()` z modulu `random`.

Funkci zavoláme tak, že specifikujeme jméno modulu a jméno funkce: `random.random()`

Příklad:

```
>>> random.random()
0.21908985054910168
>>> random.random()
0.5183769210052581
>>> random.random()
0.8788703883338108
```

Tvoje úloha

- Importuj modul random,
- pomocí funkce `random()` z modulu `random` vygeneruj číslo mezi 0.0 a 100.0.

```
1 # Import modulu random
2 import random
3
4 # Generovani (a tisk) nahodneho cisla mezi 0.0 a 100.0
5 print(random.random() * 100)
6
7
```

SPUSTIT ZNOVU

16.49504942203294

Řešení

V rozbalovacím boxu najdeš řešení úlohy s vysvětlením.

Zobrazit řešení

```
1 # Import modulu random
2 import random
3
4 # Generovani (a tisk) nahodneho cisla mezi 0.0 a 100.0
5 print( random.random() * 100 )
```

Promíchání prvků v sekvenci

Příklad použití:

```
1 # Nas list
2 lst = [45, 21, 53, 1, 213, 43, 42, 85]
3
4 # Promichani a tisk listu
5 random.shuffle(lst)
6 print(lst)
```

```
[42, 21, 53, 213, 85, 43, 1, 45]
```

V příkladu výše se list `lst` náhodně promíchal. Tato funkcionality se hodí, když chceme například vytvořit karetní hru. List `lst` by mohl reprezentovat balíček karet. **Vstupní sekvence musí být možné změnit - musí mít datový typ, který je měnitelný (mutable).** Z tohoto důvodu jsme použili list.

Zvolení náhodné sekvence

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / MODUL RANDOM / ZVOLENÍ NÁHODNÉ SEKVENCE

Řekněme, že chceme vytvořit hru, kde budeme potřebovat vybrat náhodný prvek ze sbírky - například **hra, ve které hráč hádá jméno**. V takové hře bychom normálně **zvolili náhodné slovo ze sbírky slov**. K provedení náhodného výběru ze sbírky můžeme použít funkci `choice()`.

Její celý zápis vypadá následovně: `random.choice(sekvence)`

Příklad:

```
1 # Slova
2 slova = ['hello', 'new', 'write', 'car', 'notebook']
3
4 # Nahodny vyber slova
5 slovo = random.choice(slova)
6
7 # Hadani uzivatelem
8 tip = input('Hadej slovo: ')
9
10 # Vysledek
11 vysledek = 'uhodnuto' if tip == slovo else 'neuhodnuto'
12 print(vysledek)
```

```
Hadej slovo: dsfsf
neuhodnuto
```

Funkce `random.choice()` vrátí náhodný prvek ze sekvence, která není prázdná. Pokud máme prázdnou sekvenci, dostane chybové hlášení v podobě `IndexError`:

```
1 # Slova
2 slova = []
3
4 # Nahodny vyber slova
5 slovo = random.choice(slova)

Traceback (most recent call last):
  File "C:/Users/...", line 7, in <module>
    slovo = random.choice(slova)
  File "C:/Users/...", line 258, in choice
    raise IndexError('Cannot choose from an empty sequence') from None
IndexError: Cannot choose from an empty sequence
```

Zvolení náhodného čísla (čísel) v rozmezí (range) ...

V naší hře bychom také mohli chtít získat náhodné číslo v určitém rozmezí. Tahle funkcionality se nám může hodit, když chceme například **generovat id čísla našeho zaměstnance**. K tomu můžeme použít `randrange()` nebo `randint()` funkci:

1. `random.randrange(start, stop, krok)`

Vrátí číslo `n` ze sekvence mezi `start` a `stop`, **bez čísla stop**. Některá čísla ve vstupní sekvenci mohou být přeskočena hodnoty `krok` (například když bychom chtěli každé druhé číslo, krok se bude rovná 2)

2. `random.randint(start, stop)`

Vrátí číslo `n`, které je větší, nebo rovno `start` a menší, nebo rovno `stop`.

Rozdíl mezi `randint()` a `randrange()` spočívá v tom, že `randrange` nikdy jako výsledek nevrátí číslo na pozici `stop`. Funkce `randint()` také nemá parametr `krok`. Nemůžeme tedy přeskakovat čísla.

V příkladu níže kontrolujeme, jestli se námi vygenerované id nachází v naší databázi. Pokud ne, přidáme ho:

```
1 # Databaze
2 ids = ['X5235', 'X6752']
3
4 # Vygenerovani nahodnenho cisla mezi 1000 a 9999
5 id = random.randint(1000, 9999)
6
7 # Pridani nami vegenerovaneho id, pokud uz není v databazi
8 if 'X' + str(id) not in ids:
9     ids.append('X' + str(id))
10
11 # Tisk
12 print(ids)
```

```
['X5235', 'X6752', 'X5890']
```

Nakonec bychom mohli chtít náhodnou sbírku z jiné větší sbírky. V tom případě bychom použili funkci `sample()`.

3. `random.sample(population, k)`

Ta bere dva argumenty - původní sbírku a délku našeho výběru pro novou sbírku.

```
>>> lst = [5,6,8,13,2,6]
>>> random.sample(lst, 3)
[2, 13, 6]
```

Nově vygenerovanou sbírku musíme uložit do listu! Délka nově generované sbírky nesmí být větší než délka naší původní sekvence. V opačném případě dostaneme chybu v podobně `ValueError`.

```
>>> lst = [5,6,8,13,2,6]
>>> random.sample(lst, 7)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib64/python3.7/random.py", line 321, in sample
    raise ValueError("Sample larger than population or is negative")
ValueError: Sample larger than population or is negative
```

Funkce `sample` se nám také hodí, když chceme zamíchat neměnný datový typ. Uděláme to tak, že do argumentu pro délku nové sbírky vložíme délku původní sbírky. Jak? Pomocí built-in funkce `len()` - `sample(sbirka, len(sbirka))`.

```
>>> tup = (5,6,8,13,2,6)
>>> random.sample(tup, len(tup))
(5, 13, 6, 2, 8, 6)
```

K PROCVIČENÍ

Úkol 9: Min a Max

SPUSTIT ÚKOL

Představ si, že built-in funkce `min()` a `max()` chybí. Zvládneš je vytvořit?

⌚ 15 min. ⚡ střední

Úkol 10: Find

SPUSTIT ÚKOL

Zkus vytvořit svou vlastní `my_find()` po příkladu string metody `.find()`. Funkce by měla najít shodu ve stringu.

⌚ 10 min. ⚡ těžší

Úkol 11: All a Any

SPUSTIT ÚKOL

Vytvoř funkce, které budou imitovat built-in funkce `all()` a `any()`.

⌚ 10 min. ••• lehké

Úkol 12: Reversed

SPUSTIT ÚKOL

Vytvoř funkci, která bude imitovat built-in funkci `reversed()`.

⌚ 10 min. ••• střední

Úkol 13: Sum

SPUSTIT ÚKOL

Vytvoř funkci, která bude imitovat built-in funkci `sum()`.

⌚ 10 min. ••• těžší

Úkol 14: Count

SPUSTIT ÚKOL

Vytvoř funkci, která zjišťuje počet výskytů daného prvku.

⌚ 10 min. ••• těžší

Úkol 15: Mean

SPUSTIT ÚKOL

Vytvoř funkci, která spočítá průměrnou hodnotu pro danou sekvenci číselných hodnot.

⌚ 10 min. ••• těžší

Úkol 16: GCD

SPUSTIT ÚKOL

Napiš program, který najde největšího společného dělitele.

⌚ 20 min. ••• těžší

KVÍZ

Proměnné 2.0

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / KVÍZ / PROMĚNNÉ 2.0

1/15

Co se nám uloží do proměnné `c` v následujícím výrazu?: `a, b, c = 1, 2, 3`

A. 3

B. 2

C. Takový výraz není povolen - SyntaxError

D. 1,2,3

Funkce

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / KVÍZ / FUNKCE

1/15

Co se vytiskne do **příkazového řádku**, pokud vložíme následující příkaz?: `print`

A. Vytiskne se nový řádek

B. Dostaneme SyntaxError

C. Vytiskne se prázdný string

D. Vrátí se nám objekt funkce

Moduly úvod

PYTHON #2: FUNKCE A SMYČKY / ÚVOD DO FUNKCÍ / KVÍZ / MODULY ÚVOD

1/5

Jak nahrát modul `random` do našeho skriptu?

A. include random

B. import random

C. load random

D. use random

DALŠÍ LEKCE

>_ Terminál

