KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI
www.inf.upol.cz

Jan Laštovička
jan.lastovicka@upol.cz
www.inf.upol.cz/lide/jan-lastovicka
17. listopadu 12, 771 46 Olomouc

Základy programování pro IT 1

5. Pole

verze z 22. října 2024

1 Příkaz návratu

Zjednodušíme příkaz definující funkci:

```
def function(parameter1, ..., parametern):
    program
```

a zavedeme **příkaz návratu**:

```
return expression
```

Příkaz návratu se smí vyskytovat pouze v těle funkce a vykoná se tak, že se ukončí vykonávání těla funkce, kde se nachází, a funkce vrátí hodnotu výrazu *expression*. Například absolutní hodnotu čísla můžeme definovat takto:

```
def get_abs(number):
    if number >= 0:
        return number
    else:
        return -number
```

Pokud by se při volání funkce vykonalo celé její tělo, pak funkce vrací speciální hodnotu `None` reprezentující **prázdnotu**. Hodnota `None` je zvláštní tím, že ji Shell netiskne přímo, ale musíme použít příkaz tisku:

```
>>> None
>>> print(None)
None
```

Mimochodem `print` je vestavěná funkce, která sama vrací hodnotu `None`. Jak se můžeme přesvědčit:

```
>>> print(print(None))
None
None
>>>
```

Vestavěné funkce definuje interpret. Nemusíme, a většinou ani nemůžeme, je definovat. Můžeme je rovnou používat. Názvy vestavěných funkcí prostředí IDLE tiskne fialovou barvou.

Příklad funkce, která vrací `None`:

```
def print_range(m, n):
    for i in range(m, n):
        print(i)
```

Například:

```
>>> print_range(3, 6)
3
4
5
>>>
```

2 Pole

Hodnotě, která může obsahovat hodnoty, říkáme **složená hodnota**. Příkladem složených hodnot jsou **pole**[1]. Hodnoty obsažené v poli se nazývají jeho **prvky**. Pole určuje pořadí svým prvkům. Máme tedy první prvek, druhý prvek a tak dále. Prvky v poli se mohou opakovat. Například číslo jedna může být prvním i třetím prvkem pole. Zatím budeme uvažovat pouze pole, kde prvky jsou čísla.

Základní způsob, jak vytvořit pole, je určit jeho prvky. Vezměme hodnoty *val1*, *val2*, ..., pak

```
[val1, val2, ...]
```

je výraz, jehož hodnota je pole. Hodnoty *val1*, *val2*, ... se stanou prvky pole. Pořadí prvků bude zachováno. Tedy *val1* bude první prvek pole, *val2* druhý a tak dále. Například:

[1] V Pythonu se pole nazývá anglicky *list*, což bychom přeložili jako *seznam*. Důvod, proč se odchylujeme od standardního názvu je ten, že seznam v Pythonu je implementovaný jako pole a seznam je název běžně používaný pro jinou datovou strukturu (spojový seznam).

```
[1, 2, 1, 3]
```

je pole s prvky 1, 2, 1 a 3. Stejné pole můžeme získat i takto:

```
>>> [(1 * 1), (3 - 1), (2 // 2), ((1 + 1) + 1)]
[1, 2, 1, 3]
```

Vnější závorky výrazů budeme, jak jsme zvyklí, vynechávat:

```
>>> [1 * 1, 3 - 1, 2 // 2, (1 + 1) + 1]
[1, 2, 1, 3]
```

Počet prvků pole se nazývá **délka pole**. Předchozí pole má délku čtyři. Pole, které má délku nula se nazývá **prázdné pole**:

```
[]
```

Toto jsou čtyři pole délky jedna: `[3]`, `[1]`, `[2]` a `[1]`.

Další možnost, jak vytvořit pole, je spojit dvě pole. **Spojením** polí *`array1`* a *`array2`* vznikne pole obsahující nejprve prvky pole *`array1`* a poté prvky pole *`array2`*. Spojení polí uskutečníme použitím operátoru plus:

```
(array1 + array2)
```

Hodnotou výrazu je pole, které vznikne spojením polí *`array1`* a *`array2`*. Například:

```
>>> [1, 2] + [1, 3]
[1, 2, 1, 3]
```

Vidíme, že operátor `+` se používá jak na sčítání čísel, tak na spojování polí.

Délku pole získáme vestavěnou funkcí `len` jednoho parametru. Například:

```
>>> len([1, 2, 1, 3])
4
>>> len([])
0
>>> len([3])
1
```

Pokud *array* je pole délky *array_length* a *index* je nezáporné číslo menší než *array_length*, pak prvkem pole *array* na **indexu** *index* je *index* plus první prvek pole *array*. Tedy uvažujeme-li pole `[1, 2, 1, 3]`, pak prvek na indexu 0 je číslo 1 (první prvek), na indexu 1 je číslo 2 (druhý prvek), na indexu 2 je číslo 1 (třetí prvek) a konečně na indexu 3 je číslo 3 (čtvrtý a poslední prvek).

Prvek pole na určitém indexu získáme následujícím operátorem **indexace**. Hodnotou výrazu

```
array[index]
```

je prvek pole *array* na indexu *index*. Například:

```
>>> s = [1, 2, 1, 3]
>>> s[0]
1
>>> s[1]
2
>>> s[2]
1
>>> s[3]
3
```

Dvě pole *array1* a *array2* jsou si **rovny**, pokud mají stejnou délku *array_length* a pro každé nezáporné číslo *index* menší než *array_length* platí, že prvek pole *array1* na indexu *index* se rovná prvku pole *array2* na indexu *index*. Zkráceně můžeme říci, že dvě pole se rovnají, pokud mají stejnou délku a rovnají se prvky na odpovídajících si indexech.

Například pole `[1, 2, 3]` se nerovná poli `[1, 2]`, protože nemají stejnou délku. Ani pole `[1, 2, 3]` se nerovná poli `[1, 2, 2]`. Zde se sice rovnají jejich délky, ale prvek prvního pole na indexu dva se nerovná prvku druhého pole na indexu dva: $3 \neq 2$. Pole `[1, 2]` a `[1, 2]` se rovnají, protože mají stejnou délku dva a prvek prvního pole na indexu nula se rovná prvku druhého pole na indexu nula ($1 = 1$) a podobně pro prvky na indexu jedna.

Zajímavé je, že nám vychází, že prázdný pole je rovno prázdnému poli. Oba totiž mají nulovou délku a platí, že pro každé *index* < 0 se rovnají prvky na indexu *index*. Žádné nezáporné číslo menší než nula totiž neexistuje a podmínka je triviálně splněna. Nebo-li porovnáváme-li odpovídající si prvky prázdných polí nemusíme dělat vůbec nic, protože žádné prvky nemají.

Operátor `==` také rozhoduje rovnost polí. Tedy podmínka

```
(array1 == array2)
```

je splněna, právě když jsou si pole *array1* a *array2* rovny. Například:

```
>>> [1, 2, 3] == [1, 2]
False
>>> [1, 2, 3] == [1, 2, 2]
False
>>> [1, 2] == [1, 2]
True
>>> [] == []
True
```

V případě, že chceme nějaký program vykonat pro každý prvek pole, můžeme použít následující zkratku. Příkaz opakování tvaru:

```
for variable in array_expr:
    program
```

je zkratkou za:

```
array = array_expr
for index in range(len(array)):
    variable = array[index]
    program
```

kde *index* a *array* jsou libovolné jinde nepoužívané proměnné. Například

```
for el in [0, 2, 4]:
    print(el)
```

je zkratkou za:

```
array = [0, 2, 4]
for i in range(len(array)):
    el = array[i]
    print(el)
```

Tedy:

```
>>> for el in [0, 2, 4]:
        print(el)


0
2
4
```

Otázky a úkoly k semináři

1. Co je hodnotou výrazu `[1 + 1] + [1 + 1]`?
2. Napište funkci `array_range`, která očekává dvě čísla m a n a vrací pole všech čísel větších nebo rovno než m a menších než n. Prvky jsou uspořádané vzestupně a v poli se neopakují. Například:

   ```
   >>> array_range(3, 6)
   [3, 4, 5]
   >>> array_range(6, 3)
   []
   ```

3. Napište funkci `get_digits`, která pro číslo vrátí pole jeho číslic. Například:

   ```
   >>> get_digits(1213)
   [1, 2, 1, 3]
   ```

4. Napište funkci `get_dividers`, která vrátí pole dělitelů zadaného čísla. Například:

   ```
   >>> get_dividers(6)
   [1, 2, 3, 6]
   ```

5. Co je hodnotou následujícího výrazu?

   ```
   [1 + 2][4 % 2]
   ```

6. Napište funkci `count_item`, která vrátí počet výskytů prvku v poli:

   ```
   >>> count_item(1, [1, 2, 3, 1])
   2
   >>> count_item(5, [1, 2, 3, 1])
   0
   ```

7. Napište funkci `array_sum`, která sečte prvky zadaného pole. Pro prázdné pole vraťte nulu. Například:

```
>>> array_sum([1, 2, 1, 3])
7
>>> array_sum([3])
3
>>> array_sum([])
0
```

8. Napište funkci `array_mult`, která pro zadané pole vrátí součin jeho prvků. Pro prázdné pole vraťte jedničku. Například:

```
>>> array_mult([3, 7])
21
>>> array_mult([7])
7
>>> array_mult([])
1
```

9. Napište funkci `are_arrays_equal`, která očekává dvě pole jako argumenty a rozhodne, zda se pole rovnají. Ve funkci nesmíte použít operátor `==` k porovnávání polí. Například:

```
>>> are_arrays_equal([1, 2, 3], [1, 2])
False
>>> are_arrays_equal([1, 2, 3], [1, 2, 2])
False
>>> are_arrays_equal([1, 2], [1, 2])
True
>>> are_arrays_equal([], [])
True
```

10. Pole nazveme palindrom, pokud se rovná svému převrácení. Například `[1, 2, 1]` je palindrom, protože převrácení `[1, 2, 1]` je opět `[1, 2, 1]`, ale `[1, 2, 3]` palindrom není, protože se nerovná převrácenému poli `[3, 2, 1]`. Napište funkci `is_array_palindrom`, která rozhodne, zda je pole palindrom:

```
>>> is_array_palindrom([1, 2, 1])
True
>>> is_array_palindrom([1, 2, 3])
False
```

11. Prvočíselným rozkladem kladného čísla ***`number`*** rozumíme pole prvočísel uspořádané podle velikosti takové, že součin jeho prvků je ***`number`***. Například prvočíselný rozklad čísla 21 je `[3, 7]`, čísla 90 je `[2, 3, 3, 5]`, čísla 7 je `[7]` a nakonec čísla 1 je `[]`. Prvočíselný rozklad budeme dále nazývat pouze rozklad. Napište funkci `prime_factorization`, která pro číslo vrátí jeho rozklad.

```
>>> prime_factorization(21)
[3, 7]
>>> prime_factorization(90)
[2, 3, 3, 5]
>>> prime_factorization(7)
[7]
>>> prime_factorization(1)
[]
```

12. Napište funkci `get_number_from_digits`, která pro pole číslic vrátí číslo jimi určené. Například:

```
>>> get_number_from_digits([1, 2, 1, 3])
1213
```

13. Napište funkci `remove`, která očekává pole a hodnotu. Funkce vrátí pole stejné jako pole, které obdržela, až na to, že vrácené pole nebude obsahovat žádný výskyt zadané hodnoty. Volně můžeme říci, že funkce odstraní všechny výskyty zadané hodnoty z pole. Například:

```
>>> remove(1, [2, 1, 1, 3])
[2, 3]
>>> remove(4, [2, 1, 1, 3])
[2, 1, 1, 3]
>>> remove(4, [])
[]
```

14. Napište funkci `array_square`, která pro pole čísel vrátí pole tvořené čtverci prvků zadaného pole. Pořadí prvků v původním poli je stejné jako pořadí jejich čtverců ve vráceném poli. Volně můžeme říci, že funkce vrátí pole čtverců zadaných čísel. Například:

```
>>> array_square([1, 2, 3])
[1, 4, 9]
>>> array_square([])
[]
```

15. Napište funkci `reverse_array`, která pro pole vrátí pole s prvky v opačném pořadí. Například:

```
>>> reverse_array([1, 2, 3])
[3, 2, 1]
```

16. Pole ***array1*** nezveme **podpolem** pole ***array2***, pokud pole ***array1*** vznikne vynecháním některých (nebo žádných) prvků pole ***array2***. Například `[1, 2, 2]` je podpole `[2, 1, 1, 2, 4, 2]`, protože z `[2, 1, 1, 2, 4, 2]` můžeme vynechat jednu jedničku, čtyřku a první dvojku a obdržíme pole `[1, 2, 2]`,

ale `[1, 1, 2]` ani `[2, 1, 2]` není podpole pole `[1, 2, 2]`. Napište funkci `is_subarray`, která pro dvě zadané pole *`array1`* a *`array2`* rozhodne, zda je *`array1`* podpolem *`array2`*. Například:

```
>>> is_subarray([1, 2, 2], [2, 1, 1, 2, 4, 2])
True
>>> is_subarray([1, 1, 2], [1, 1, 2])
True
>>> is_subarray([1, 1, 2], [1, 2, 2])
False
>>> is_subarray([2, 1, 2], [1, 2, 2])
False
```

17. Jsou dány dvě pole *`array1`* a *`array2`* se vzestupně uspořádanými prvky. Najděte nejdelší pole, který je podpolem jak pole *`array1`* tak pole *`array2`*. Hledáme tedy nejdelší společné podpole obou polí. Například `[1, 2, 4]` je nejdelší společné podpole polí `[1, 1, 2, 3, 4]` a `[1, 2, 2, 4, 4]`.