Referencie a hodnoty

- Rozlišujeme:
  - Volanie podľa hodnoty
  - Volanie podľa referencie
- Volanie podľa hodnoty: odovzdáva sa kópia originálnej premennej, pôvodná hodnota premennej sa nemení
- Volanie podľa referencie: odovzdáva sa vlastná premenná, hodnota pôvodnej premennej môže byť zmenená
- v Pythone existuje niečo odlišné: poslanie referencie objektu (pass-by-object-reference)
- Závisí, či premenná je **mutable** alebo **immutable**
- Ak sú premenné mutable, Python hodnotu pôvodnej premennej zmení, pre immutable hodnota pôvodnej premennej zostane zachovaná, nezmenená

```python
# int, float, stringy sú immutable typy, nevieme zmeniť hodnotu
# volanie na základe hodnoty, vytvorí sa kópia pôvodnej premennej
def change_string(x):
    x = "Joe Quenneville"

c = "Patrick Kane"
# Funkcia nezmení hodnotu pôvodnej premennej
change_string(c)

# Vráti Patrick Kane, lebo string je immutable
print(c)


# Python zmení hodnotu mutable objektov
# list, dictionary sú mutable typy
def add_item(item):
    item.append("Chicago Blackhawks player")

my_list = ["Brandon Saad", 20, True]
add_item(my_list)

# Python aktualizuje objekt list, lebo je mutable
# Pridá Chicago Blackhawks player na koniec zoznamu
print(my_list)
```

Výnimky (exceptions)

- **Chyba** – kód, ktorý nie je validný, alebo nedáva zmysel, po spustení programu, Python interpreter vypíše error, a nepokračuje ďalej
- Žiaden kód nie je bezchybný
- Ošetrovanie chýb nazývame **Error handling**
- Chyby je dobré ošetrovať
- Na ošetrovanie chýb existujú sada príkazov **try** a **except**
- Ak nastane chyba v **try** časti, ktorá je definovaná v časti except, Python vyhodí chybu (raise error), ktorá sa nachádza v časti except
- Do **try** časti píšeme kód, ktorý chceme spustiť
- Do časti **except** píšeme kód, ktorý sa spustí, ak nájde v try bloku chybu definovanú v časti except
- Ošetrovanie – čakanie na chybu nastáva len v bloku **try** !
- **Neodporúčané** testovať akúkoľvek chybu v bloku **try** !
- Testovať len na chybu, ktorú očakávame
- V časti except vieme zachytávať viac ako 1 chybu

```
try:
  code
  code
  code
except Exception:
  error  handling
```

Výnimky (exceptions)

```python
# Zachytiť vieme ktorúkoľvek chybu,
# nemusíme ju vždy definovať
# Je dobrým zvykom zachytávať presne každú chybu
my_array = [1102,50, 85, 48]
try:
    my_array[206] # index neexistuje
# nie je použité "e"
except Exception:
    print("Nastala chyba, skúste ešte raz")


# "As e" v sebe drží tzv. objekt výnimky
# A volá sa tento objekt výnimky
# Ak zachytávame každú chybu, vieme potom zistiť, o aku chybu išlo
my_array = [1102,50, 85, 48]
try:
    my_array[102] # index neexistuje
except Exception as e:
    print("Nastala chyba, skúste ešte raz:")
    # Výpis objektu výnimky
    print(e)
```

```python
# Zachytiť môžeme aj viac chýb súčasne,
# ak špecifikujeme typy chýb
def division_nums(num1, num2):
    try:
        result = num1 / num2
    except ZeroDivisionError:
        print("Delenie nulou ! Vraciam 0")
        result = 0
    except TypeError:
        print("Do funkcie posielajte len čísla ! Vraciam 0")
        result = 0
    return result
    # zachytenie akejkoľvek inej chyby
    except Exception as e:
        print("Nastala chyba, skúste ete raz")
        print(e)

print(division_nums("Tomas", "Tatar"))
print(division_nums(21, 0))
print(division_nums(28, 6))
```

Objektovo orientované programovanie (OOP)

- Python je objektovo orientovaný jazyk
- v Pythone je všetko **objekt**
- Objekt je jav z reálneho sveta
- Trieda je akási šablóna tohto reálneho javu
- V Pythone označujeme triedy **class,** ktoré tvoria „tie“ akési šablóny pre vytvorenie objektu
- V triedach definujeme **atribúty** (premenné), **metódy** (funkcie), z ktorých inicializujeme objekt s danými vlastnosťami a správaním
- **Atribúty** reprezentujú stav objektu
- **Metódy** definujú správanie objektu
- Trieda môže byť: **Player** s atribútmi meno, priezvisko, rok narodenia, výška, pozícia, klub, góly, asistencie. Metódy triedy môžu byť: vypočítaj vek hráča, vypočítaj body hráča, zobraz detail hráča
- Vlastnosti OOP
  - **Modularita**: OOP umožňuje rozdelenie kódu objektu a správania
  - **Rozšíriteľnosť**: objekt vie byť ľahko rozšíriteľný o nové atribúty a metódy
  - **Znovapoužiteľnosť**: objekty môžu byť použité kdekoľvek v aplikácii, niekoľkokrát
- Vďaka týmto vlastnostiam objektovo orientované programy sú **ľahšie** udržiavateľné a aktualizovateľné
- OOP programy sú zvyčajne **pomalšie** ako procedurálne (viac inštrukcií pre spracovanie)

Objektovo orientované programovanie (OOP)

- **Konštruktor** je jedna z hlavných metód, ktorá sa volá pri vytváraní objektu. Táto metóda je definovaná v triede a je možné ju použiť na inicializáciu základných premenných
- V Pythone sa označuje konštruktor **__init__** ktorá má niekoľko parametrov, kde prvý je povinný, tzv. self, ktorý vytvára akúsi prázdnu inštanciu triedy

- Cez **self** pristupujeme k inštancii objektu, k premenným a k metódam objektu

- Cez premenné nastavujeme stav objektu
- Pomocou funkcií, alebo metód zabezpečujeme správanie objektu

Objektovo orientované programovanie (OOP)

Class atribúty

- **Class atribúty** sa viažu ku konkrétnej triede a existujú i mimo triedy a nie je nutné pre nich vytvárať objekt
- Sú to atribúty, ktoré existujú pre všetky objekty triedy s určitou default hodnotu. Default hodnota v príklade je „NHL“

Príklad vráti:

Name: Tomas Tatar 21

Number: 21

NHL

```
class Player:

    # Class premenná
    class_my_variable = "NHL"

    def __init__(self, name_p, surname_p, number):
        self.name = name_p
        self.surname = surname_p
        self.number = number

    def show_player(self):
        print(f"Player: {self.name} {self.surname} {self.number}")


p1 = Player("Tomas", "Tatar", 21)
p1.show_player()
p2 = Player.class_my_variable
print(p2)
```

Class metóda

- **Class metódy** sa viažu ku konkrétnej triede a existujú mimo triedy
- Označujú sa dekorátorom (Dekorátor je funkcia, ktorá obaľuje inú funkciu, označuje sa @)
- Nepotrebujeme vytvárať inštanciu triedy, aby sme k ním pristupovali
- Class metóda na vstupe dostáva triedu, preto zväčša používame označenie cls, vie pristúpiť k objektu triedy, vie interagovať s jej funkciami, ale hlavne s atribútmi triedy
- Použitie: pri overloadovaní, ak napríklad nepoznám pre konštruktor vek, ale poznám rok narodenia

Príklad vráti:

2022 – 1991 = **31**

```python
from datetime import date

class Player:

    class_my_variable = "NHL"

    def __init__(self, name_p, age):
        self.name = name_p
        self.age = age

    # Class metóda
    @classmethod
    def get_age(cls, name, year):
        return cls(name, date.today().year - year)


p1 = Player.get_age("Tomas Tarar", 1991)
print(p1.age)
```