Statická metóda

- **Statická metóda** nepotrebuje self ani triedu ako parameter. Nevie interagovať s funkciami alebo atribútmi v triede
- Používa sa pre jednoduché funkcie, jednoduché výpočty
- Nepotrebujeme vytvárať inštanciu triedy, aby sme k statickej metóde pristúpili

Príklad vráti:

67

```python
class Player:

    class_my_variable = "NHL"

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


    # Static metóda
    @staticmethod
    def get_points(goals, assists):
        return goals + assists

p1 = Player.get_points(25, 42)
print(p1)
```

Dedenie

- Veľmi dôležitá a kľúčová oblasť v OOP
- Dedenie nám umožňuje definovať triedu, ktorá dedí všetky metódy a vlastnosti z inej triedy
- Rozlišujeme: **rodičovská** a **dcérska** trieda (potomok)
- **Rodičovská trieda** je trieda, z ktorej sa dedí, ktorá sa tiež nazýva základná trieda
- **Dcérska trieda** je trieda, ktorá dedí z inej triedy, ktorá sa tiež nazýva odvodená trieda alebo potomok
- Zápis: class DcerskaTrieda(RodicovskaTrieda):
- V konštruktore dcérskej triedy voláme konštruktor rodičovskej triedy pre zachovanie parametrov (ak ho nezavoláme, môžeme prepísať zdedené atribúty)
- Volaním **super()** funkcie nemusíme používať názov rodičovskej triedy, automaticky dedí metódy a atribúty zo svojej nadradenej triedy. Vynechávame self (pretože posiela sa objekt triedy Person)

Príklad vráti:

Person: Marian Hossa

Number: 81

```python
class Person:
    def __init__(self, first_name, last_name):
        self.name = first_name
        self.surname = last_name

    def show_person(self):
        print(f"Person: {self.name} {self.surname}")

class Player(Person):
    def __init__(self, first_name, last_name, number):
        # Môžme vynechať
        # Person.__init__(self, first_name, last_name)
        super().__init__(first_name, last_name)
        self.number = number

    def show_person(self):
        # Dedí správanie z triedy Person
        super().show_person()
        print(f"Number: {self.number}")

p2 = Player("Marian", "Hossa", 81)
p2.show_person()
```

Viacúrovňové dedenie

- Metódou **isistance()**, ktorá vráti True alebo False zisťujeme, či daná inštancia triedy je inštanciou porovnávanej triedy.
- V príklade inštancie p2 a t1 sú inštanciami triedy Person, ale inštancia triedy Person p1 nie je inštanciou triedy Player

kľúčové slovo **pass používame, ak telo triedy je prázdne

Príklad vráti:

Person: Joe Quenneville

Number: 1

True

True

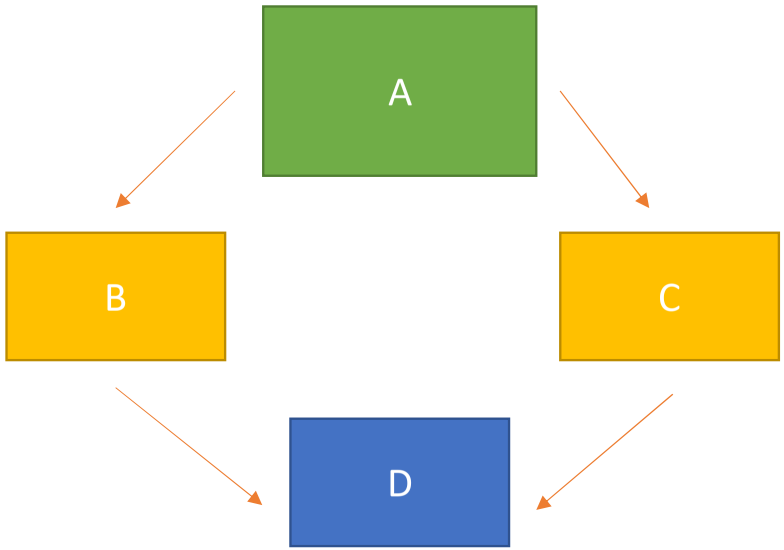
False

True

```python
class Person:
    def __init__(self, first_name, last_name):
        self.name = first_name
        self.surname = last_name

    def show_person(self):
        print(f"Person: {self.name} {self.surname}")


class Player(Person):
    def __init__(self, first_name, last_name, number):
        super().__init__(first_name, last_name)
        self.number = number

    def show_person(self):
        super().show_person()
        print(f"Number: {self.number}")

class Trainer(Player):
    pass

p1 = Person("Tomas", "Tatar")
p2 = Player("Marian", "Hossa", 81)
t1 = Trainer("Joe", "Quenneville", 1)
t1.show_person()

print(isinstance(p2, Person))
print(isinstance(p2, Player))
print(isinstance(p1, Player))
print(isinstance(t1, Person))
```

Viacnásobné dedenie

- Umožňuje špecifikovať viac ako jedného rodiča pre potomka
- Ak dedíme od nejakej triedy, potom dedíme aj od rodičovskej triedy danej rodičovskej triedy
- **Zápis**: class DcerskaTrieda(RodicovskaTrieda1, RodicovskaTrieda2, ...):
- Pri viacnásobnom dedení dochádza k tzv. diamantovému problému, napr. triedy B a C, dedia od triedy A. Trieda D dedí od tried B, C. Čo môže viesť ku komplikáciam v dedení. Napríklad čo ak 2 triedy majú rovnakú metódu, return_name()
- Riešenie: **linearizácia** – berie sa prvá trieda zľava, a z nej daná metóda s rovnakým názvom



Viacnásobné dedenie

- Trieda D najskôr dedí od C, potom od B, použitá **linearizácia**
- Vo **výsledku sa vypíše** „South“, potom „North“, následne „Eastern“
- Posledný výpis vráti postupnosť dedenia v rámci tried, funkcia *mro()*:
- *„<class '__main__.D'>, <class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>] “*
- Najskôr sa pozeráme do tela vlastnej triedy, potom do triedy C, potom A, na záver triedy B.

```
class A:
    division = "Est"

class B:
    division = "West"

    def __init__(self, club):
        self.club = "Boston"

class C(A):
    division = "South"
    def __init__(self, conference):
        self.conference = "Eastern"

class D(C, B):
    division = "North"

    def get_division(self):
        print(super().division)

div1 = D("No division")
# Vypíše zdedené division
div1.get_division()

# Vypíše vlastné division
print(div1.division)

# Vypíše zdedené conference
print(div1.conference)

print(D.mro())
```

Overloading vs. Overriding

Overloading:

- umožňuje použiť 1 metódu s rôznym počtom parametrov
- Použitie tzv. optional parametrov s default hodnotou napr. None
- Optional parametre sa píšu na záver v tele parametrov

Overriding:

- Prepísanie metód vrátane konštruktora
- Najčastejšie sa stretávame v dedení
- Dcérske metódy vedia zmeniť správanie zdedenej metódy
- Takáto metóda musí mať rovnaký názov ako zdedená metóda

Výstup z Overloadingu (objekt p1):

- Not enough identifiers
- Person: Tatar
- Not enough identifiers

Výstup z Overridignu (objekt p2):

- Changed method print_person()

```python
class Person:
    def __init__(self, first_name, last_name):
        self.name = first_name
        self.surname = last_name

    def print_person(self, surname=None, name=None):
        if name and surname:
            print(f"Person: {self.name} {self.surname}")
        elif surname:
            print(f"Person: {self.surname}")
        else:
            print(f"Not enough identifiers")


class Player(Person):
    def print_person(self):
        print(f"Changed method print_person()")


p1 = Person("Tomas", "Tatar")
p1.print_person()
p1.print_person("Tatar")
p1.print_person("", "Tomas")

p2 = Player("Marian", "Hossa")
p2.print_person()
```