[!important] Obecná část o objektově orientovaném programování je k nalezení v několika separátních souborech 1) [[Objektově orientované programování - třídy a objekty, zprávy a metody]] 2) [[Základy objektového programování - třídy, objekty zasílání zpráv]] 3) [[Principy objektového programování - zapouzdření, polymorfismus a dědičnost]] 4) [[Přehled a základní rysy programovacích paradigmat - funkcionální, procedurální, objektové]] ## 1. Objektově orientované programování [!info] **OOP** … Programovací paradigma, které zapouzdřuje vlastnosti a funkcionalitu do individuálních objektů

- Objekt reprezentující řetězec `"Hello World"`, obsahuje rovněž funkcionalitu `"Hello World".split()`

Třídy

- Vytvoření uživatelsky definovaných tříd (následně pak objektů) budeme v tuto chvíli chápat hlavně jako tvorbu vlastních datových struktur s navázanou funkcionalitou
- Tuto možnost bychom měli využít pouze v případě, že již definované struktury nejsou dostatečné

  [!info] **Třída** (příkaz `class`) tedy slouží k vytváření uživatelsky definovaných datových struktur. Určují jak má výsledná datová struktura vypadat a fungovat. Na základě *třídy* (předpis) můžeme vytvářet jednotlivé *instance třídy* (objekty).

  Např. Můžeme si představit již dobře známý *seznam*. Jeden konkrétní *seznam* je instancí (objektem) třídy *seznam*, která popisuje jak seznamy vypadají a fungují. *Třídy* jsou tedy **obecným předpisem**, *objekty* pak **konkrétní entity** vytvořené na základě tohoto předpisu.

  [!important] Třídy umisťujeme do modulů stejného názvu

```
# PEP8 - název třídy používá CapWords konvenci
# definice prázdné třídy v souboru "credit_account.py"
class CreditAccount:
    pass
```

Metody a vlastnosti

- Funkcím které jsou s třídou úzce spjaty říkáme *metody*
- Pro třídy i metody se používá obdobná konvence docstringů ##### Metoda `.__init__()`
- Jedná se o konstruktor => nastavuje **počáteční stav** (initial state)
- Může obsahovat libovolný počet parametrů, prvním musí však vždy být `self` ##### Vlastnosti třídy
- Můžeme je využívat napříč všemi instancemi dané třídy

```python
class CreditAccount:
    """Account with stored credits."""

    max_balance = 1000 # vlastnost třídy

    def __init__(self, owner, initial_credits=0):
        """Creates credit account with given owner and initial credits.

        Args:
            owner: owner of the account
            initial_credits (optional): credit balance. Defaults to 0.
        """

        self.owner = owner
        self.balance = initial_credits

    # jeden prázdný řádek
    def transfer_to(self, other, value):
        """Transfer money into another account. Negative balance is allowed.

        Args:
            other: Target of money transfer.
            value: Amount of money to be transfered.
        """

        self.balance -= value
        other.balance += value
```

Přístup k vlastnostem objektu

- Narozdíl od ostatních jazyků Python přistupuje k vlastnostem **přímo** (přes tečkovou notaci) čili nevytváříme tzv. *gettery* ani *settery*
- Existuje však způsob jak udělat metody "privátní" (využíváme k rozdělení kompilkované metody do více jednodušší, do kterých však nechceme dovolit zasahovat uživateli)

```python
class TestClass:
    def __init__(self):
        self._private_data = []

    # "privátní metoda" (většinou se jedná o pomocné metody)
    def _reverse_order(self):
        pass

    # použité v jiné metodě téže třídy
    def reverse(self):
```

```
        return self._reverse_order()
```

Dědičnost

- Jeden z hlavních konceptů OOP
- Zjednodušené vysvětlení metody `super()` - umožňuje přístup k metodám z tříd od kterých dědíme

```
class Account:
    """Represents an account."""

    def __init__(self, owner, initial_balance=0):
        """Creates account with given owner and initial balance.

        Args:
            owner: owner of the account
            initial_balance (optional): initial balance. Defaults to 0.
        """

        self.owner = owner
        self.balance = initial_balance

    def transfer_to(self, other, value):
        """Transfer money into another account. Negative balance is allowed.

        Args:
            other: Target of money transfer.
            value: Amount of money to be transfered.
        """

        self.balance -= value
        other.balance += value
```

```python
from datetime import datetime
from account import Account


class CreditAccount(Account):
    """Represents a credit account."""

    def __init__(self, owner, initial_balance=0):
        # vyvolání konstruktoru předka
        super().__init__(owner, initial_balance=initial_balance)

        # dodané vlastnosti
        self.expiration = datetime.now() + datetime.timedelta(days=365)
```

```
    # dodaná funkcionalita
    def expires_soon(self):
        """Checks if accounts validate within 30 days."""
        return datetime.now() + datetime.timedelta(days=30) >= self.expiration
```

NamedTuple

- Možnost využít pro jednodušší strukturovaná data

```
from collections import namedtuple

Person = namedtuple("Person", ["name", "phone", "email"])

owner = Person("Lukas Novak", "723812052", "novak@gmail.com")

owner.name
owner.phone
owner.email

# funguje jako klasický tuple
assert owner.name == owner[0]
```

2. Dunder metody

> [!info] **Dunder metody** …. speciální metody sloužící k implementaci podpory pro vestavěné funkce Pythonu a jinou rozšiřující funkcionalitu (např. `__init__` jakožto konstruktor)
>
> **Přehled** zde - Způsob implementace je vždy podobný (respektive je na nás jak ho vnitřně provedeme) #### String reprezentace - Pomocí *dunder metod* `__repr__` a `__str__` implementujeme `repr()` a `str()`

```
def __repr__(self):
    return f"CreditAccount({self.owner}, {self.balance})"

def __str__(self):
    return self.__repr__()

# zkouška
print(credit_account_1)
repr(credit_account_1)
```

Převod na jiné datové typy

```
__bool__     # chování funkce bool()
__complex__  # chování funkce complex()
```

```
__int__       # chování funkce int()
__float__     # chování funkce float()
__hash__      # chování funkce hash()
```

Unární číselné operátory

```
__abs__  # chování funkce abs()
__neg__  # chování unárního minus
__pos__  # chování unárního plus
```

Porovnávání

```
__lt__  # chování <
__le__  # chování <=
__eq__  # chování ==
__ne__  # chování !=
__gt__  # chování >
__ge__  # chování >=
```

Aritmetické operátory

```
__add__       # chování +
__sub__       # chování -
__mul__       # chování *
__truediv__   # chování /
__floordiv__  # chování //
__mod__       # chování %
__divmod__    # chování divmod()
__pow__       # chování ** nebo pow()
__round__     # chování round()
```

- V případě, že pro nějaký typ není metoda implementována je nutné vracet konstantu `NotImplemented`
- Aritmetické operátory je možné implementovat pro **"oba směry"**
- V situaci kdy vyhodnocujeme $x+y$ Python hledá implementaci `x.__add__` nebo `y.__radd__`

Kombinované operátory přiřazení s aritmetickými operacemi

- Je běžné (ne však nutné), aby výsledná metoda vracela `self`

```
__iadd__       # chování +=
__isub__       # chování -=
__imul__       # chování *=
__itruediv__   # chování /=
__ifloordiv__  # chování //=
__imod__       # chování %=
__ipow__       # chování **=
```

Bitové operátory

```
__invert__ # chování ~
__lshift__ # chování <<
__rshift__ # chování >>
__and__ # chování &
__or__ # chování |
__xor__ # chování ^
```

Emulace kolekcí

- Důležité, dostaneme se k nim později

```
__index__ # chování převodu na integer například při slicingu
__len__ # chování len()
__getitem__ # chování x[20]
__setitem__ # chování x[20] = 2
__delitem__ # chování del x[20]
__contains__ # chování in
```

- Funkcionalita se dá samozřejmě použít i uvnitř tříd

Vestavěné funkce

- Dunder metody představují elegantní řešení pro implementaci podpory vestavěných funkcí.
- Většinou je tedy lepší využívat implementace těchto metod pro podporu `len()` než implementování vlastní metody `object.length()`
- Výjimkou může být například pomyslná třída `Vector`
  - Pro výpočet délky vektoru implementujeme vlastní metodu
  - Metodu `len()` totiž používáme v kontextu kolekcí ke zjištění počtu prvků

Pořadí definic metod

- **Neexistuje** žádný jeden správný způsob v
- Populární možnost je následující

```
class MyClass:
    # Dunder metohods

    # @staticmethod a @classmethod

    # @property

    # _private_method(self)

    # public_method(self)
```