- Umožňuje přidávat k funkci další funkcionalitu (její "obalení")
- **Funkce vyššího řádu**: funkce, která vrací funkci nebo ji bere jako argument
- Syntaktický cukr

```
@decorator
def my_func():
    print("KMI/JP")
```

- Demonstrací jednoduché funkcionality může být následující

```
# dekorátor
def my_decorator(func):
    """Decorator description."""
    def wrapper():
        print("Something before function call")
        func()
        print("Something after function call")

    return wrapper

# definovaná funkce
def my_function():
    print("Super function!")

# volání původní funkce
my_function()

# obalení funkce dekorátorem
my_function_decorated = my_decorator(my_function)

# volání modifikované funkce
my_function_decorated()
```

- Používání dekorátorů může způsobit na první pohled řadu komplikací, viz dále
- Pokud dekorovaná funkce přijímá argumenty, případně má nějakou hodnotu vracet

```
# 01 - podporujeme předání argumentů vnitřní funkci
def do_twice(func):
    """Calls a function twice."""
    def wrapper(*args, **kwargs):
        func(*args, **kwargs)
        func(*args, **kwargs)

    return wrapper
```

```python
# příklad
@do_twice
def multiply_list(list_, by):
    """Multiply items from list by given value.

    Args:
        list_: list to be multiplied
        by: value by which items are multiplied

    Returns:
        multiplied list
    """

    result = []

    for item in list_:
        result.append(item * by)

    return result

multiply_list([1, 2, 3], 5)

# 02 - zajistíme správné vrácení hodnoty upravením dekorátoru
def do_twice(func):
    """Calls a function twice."""
    def wrapper(*args, **kwargs):
        func(*args, **kwargs)
        return func(*args, **kwargs)

    return wrapper
```

- Dále tzv. *zachování identity* funkce (např. dosažitelnost docstringu)

```python
# řešení je použití dekorátoru 'functools.wraps'

import functools

def do_twice(func):
    """Calls a function twice."""
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        func(*args, **kwargs)
        return func(*args, **kwargs)

    return wrapper
```

- Dekorátory je možné zanořovat (vrstvit), avšak vždy záléží na pořadí ###

Použití
- Pro timming funkce (měření doby běhu), logování, autorizace či autentizace

Dekorátory tříd

- Některé běžně používané jsou již "zabudované" v Pythonu (`@classmethod`, `@staticmethod`, ...)
- Kromě dekorování metod třídy, můžeme dekorovat i celou třídu

```
@dataclass
class PlayingCard:
    rank: str
    suit: str
```

- Tento přístup nedekoruje každou metodu třídy nýbrž se dekorátor aplikuje pouze při vytváření instance třídy

Speciální dekorátory ve třídách

`@property`

- Jelikož nepoužíváme klasické gettery a settery může nastat situace, kdy budeme chtít dělat "něco navíc" při nastavení hodnoty (např. kontrola záporné hodnoty)

```
# opět třída CreditAccount

@property
    def balance(self):
        return self._balance

@balance.setter
def balance(self, new_balance):
    """Sets new value of balance, new value cannot be negative."""
    if new_balance < 0:
        raise ValueError("Balance cannot be negative number!")

    self._balance = new_balance

@balance.deleter
def balance(self):
    self._balance = 0
```

`@classmethod`

- Tento dekorátor lze použít v situaci kdy je nutné metodám předat odkaz na celou třídu

- Demonstrovat jej můžeme na příkladu metod `from_*`, tedy metod které umí vytvořit instanci třídy různými způsoby

```
@classmethod
def from_csv(cls, input_string, separator=","):
    """Creates CreditAccount class from csv string"""
    owner, initial_credits = input_string.split(separator)

    return cls(owner, int(initial_credits))
```

`@staticmethod`

- Naopak tento dekorátor nemá přístup ke své třídě

```
@staticmethod
def credit_to_money(credit, exchange_rate):
    """Calculates money value of credits.

    Args:
        credit: amount of credits
        exchange_rate: how many money per one credit

    Returns: money value
    """

    return credit * exchange_rate
```