ООП. Процесс создания классов. Декорирование классов. Mixins. Dataclasses.

Преподаватель: Тенигин Альберт Андреевич

Сириус
IT-Колледж

Декорирование функций

```python
from typing import Callable


def decorator(target_func: Callable) -> Callable:
    def new(*args, **kwargs):
        print('decorated')
        return target_func(*args, **kwargs)
    return new


@decorator
def printer(message: str) -> None:
    print(message)


printer('hello, world!')
```

Сириус
IT-Колледж

Декоратор с пробросом аргумента (на самом деле функция, создающая декоратор)

```python
from typing import Callable

def decorator_creator(message: str) -> Callable:
    def decorator(target_func: Callable) -> Callable:
        def new(*args, **kwargs):
            print(message)
            return target_func(*args, **kwargs)
        return new
    return decorator


@decorator_creator('hello')
def printer(message: str) -> None:
    print(message)


printer('world!')
```

Сириус
IT-Колледж

```python
from typing import Callable, Any

class Decorator:
    def __init__(self, target_func: Callable) -> None:
        self.function = target_func

    def __call__(self, *args) -> Any:
        print('decorated')
        return self.function(*args)

@Decorator
def printer(message: str) -> None:
    print(message)


printer('hello, world!')
```

Декорирование классом

Сириус
IT-Колледж

```python
# Декорирование классом с пробросом аргумента
from typing import Callable, Any

class Decorator:
    def __init__(self, message: str) -> None:
        self.message = message

    def __wrap_function(self, target_function: Callable) -> Callable:
        def new(*args, **kwargs) -> Any:
            print(self.message)
            return target_function(*args, **kwargs)
        return new

    def __call__(self, target_function: Callable) -> Callable:
        return self.__wrap_function(target_function)

@Decorator('decorated')
def printer(message: str) -> None:
    print(message)

printer('hello, world!')
```

Декорирование класса

```python
def add_str_dunder(class_: type) -> type:
    def str_method(self) -> str:
        attrs = [f'{key}: {value}' for key, value in self.__dict__.items()]
        return ', '.join(attrs)
    setattr(class_, '__str__', str_method)
    return class_

@add_str_dunder
class Person:
    def __init__(self, name: str, age: int) -> None:
        self.name, self.age = name, age

print(Person('Ivanov', 20))
```

```
name: Ivanov, age: 20
```

Сириус
IT-Колледж

```python
"""Декорирование класса классом (черная магия вне Хогвартса)."""
class StrDunderModifier:
    def __init__(self, class_: type) -> None:
        self.class_ = class_

    def __call__(self, *args, **kwargs) -> type:
        def str_method(self) -> str:
            attrs = [f'{key}: {value}' for key, value in self.__dict__.items()]
            return ', '.join(attrs)
        setattr(self.class_, '__str__', str_method)
        return self.class_(*args, **kwargs)


@StrDunderModifier
class Person:
    def __init__(self, name: str, age: int) -> None:
        self.name, self.age = name, age

print(Person('Ivanov', 20))
```

```
name: Ivanov, age: 20
```

```python
"""Mixin classes."""
class StrMixin:
    def __str__(self) -> str:
        attrs = [f'{key}: {value}' for key, value in self.__dict__.items()]
        return ', '.join(attrs)

class Person(StrMixin):
    def __init__(self, name: str, age: int) -> None:
        self.name, self.age = name, age

class Bike(StrMixin):
    def __init__(self, model: str, volume: float, price: float) -> None:
        self.model, self.volume, self.price = model, volume, price

print(Person('Ivanov', 21))
print(Bike('Kawasaki Versys 650', 649, 1200000.0))
```

```
name: Ivanov, age: 21
model: Kawasaki Versys 650, volume: 649, price: 1200000.0
```

Dataclasses

Dataclasses позволяют заменять шаблонный код простым определением и упрощают процесс создания классов, существенно экономя время разработчика.

Сириус
IT-Колледж

```python
class Message:
    def __init__(self, id_: int, sender: User, recipient: User, content: str) -> None:
        self.id: int = id_
        self.sender: User = sender
        self.recipient: User = recipient
        self.content: str = content
```

Сириус
IT-Колледж

```python
class Message:
    def __init__(self, id_: int, sender: User, recipient: User, content: str) -> None:
        self.__id: int = id_
        self.__sender: User = sender
        self.__recipient: User = recipient
        self.__content: str = content

    @property
    def id(self) -> int:
        return self.__id

    @property
    def sender(self) -> User:
        return self.__sender

    @property
    def recipient(self) -> User:
        return self.__recipient

    @property
    def content(self) -> str:
        return self.__content
```

```python
def __str__(self) -> str:
    classname = self.__class__.__name__
    return f'{classname} id={self.id} from {self.sender} to {self.recipient}: {self.content}'

def __repr__(self) -> str:
    classname = self.__class__.__name__
    result = '{} id={}, from={}, to={}, content={}'
    return result.format(classname, self.id, self.sender, self.recipient, self.content)
```

Сириус
IT-Колледж

```python
def __eq__(self, other: Self) -> bool:
    if not self.__class__ is other.__class__:
        return False
    attrs = 'id', 'sender', 'recepient', 'content'
    return all([getattr(self, attr) == getattr(other, attr) for attr in attrs])

def __ne__(self, other: Self) -> bool:
    return not self.__eq__(other)
```

Сириус
IT-Колледж

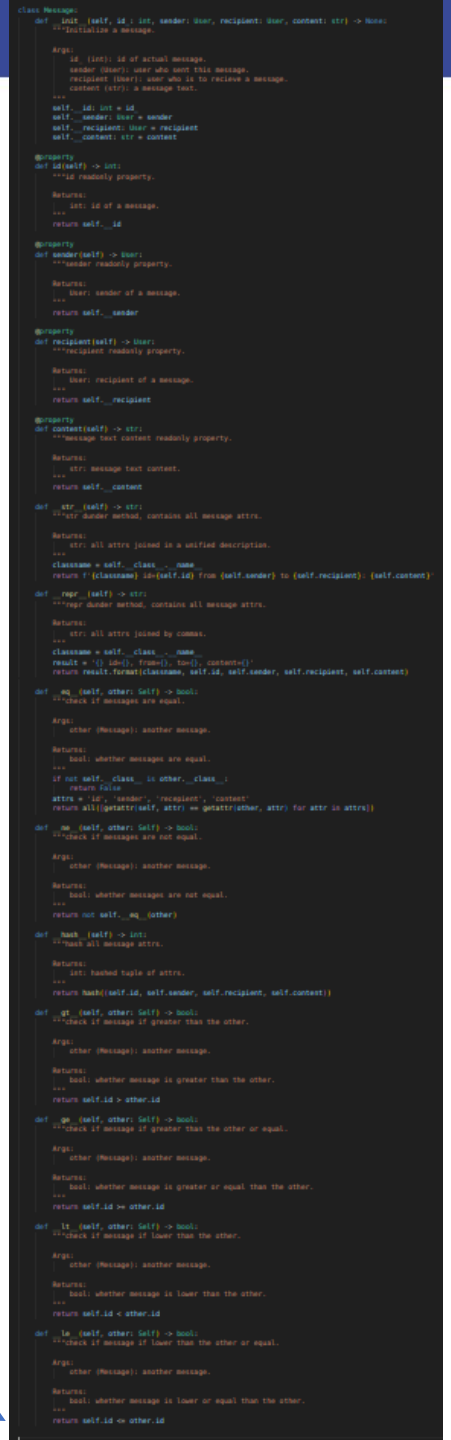
```python
def __hash__(self) -> int:
    return hash((self.id, self.sender, self.recipient, self.content))

def __gt__(self, other: Self) -> bool:
    return self.id > other.id

def __ge__(self, other: Self) -> bool:
    return self.id >= other.id

def __lt__(self, other: Self) -> bool:
    return self.id < other.id

def __le__(self, other: Self) -> bool:
    return self.id <= other.id
```



```python
class Message:
    def __init__(self, id_: int, sender: User, recipient: User, content: str) -> None:
        """Initialize a message.

        Args:
            id_ (int): id of actual message.
            sender (User): user who sent this message.
            recipient (User): user who is to recieve a message.
            content (str): a message text.
        """
        self._id: int = id
        self._sender: User = sender
        self._recipient: User = recipient
        self._content: str = content

    @property
    def id(self) -> int:
        """id readonly property.

        Returns:
            int: id of a message.
        """
        return self._id

    @property
    def sender(self) -> User:
        """sender readonly property.

        Returns:
            User: sender of a message.
        """
        return self._sender

    @property
    def recipient(self) -> User:
        """recipient readonly property.

        Returns:
            User: recipient of a message.
        """
        return self._recipient

    @property
    def content(self) -> str:
        """message text content readonly property.

        Returns:
            str: message text content.
        """
        return self._content

    def __str__(self) -> str:
        """str dunder method, contains all message attrs.

        Returns:
            str: all attrs joined in a unified description.
        """
        classname = self.__class__.__name__
        return f'{classname} id={self.id} from {self.sender} to {self.recipient}: {self.content}'

    def __repr__(self) -> str:
        """repr dunder method, contains all message attrs.

        Returns:
            str: all attrs joined by commas.
        """
        classname = self.__class__.__name__
        result = '{} id={}, from={}, to={}, content={}'
        return result.format(classname, self.id, self.sender, self.recipient, self.content)

    def __eq__(self, other: Self) -> bool:
        """check if messages are equal.

        Args:
            other (Message): another message.

        Returns:
            bool: whether messages are equal.
        """
        if not self.__class__ is other.__class__:
            return False
        attrs = 'id', 'sender', 'recipient', 'content'
        return all([getattr(self, attr) == getattr(other, attr) for attr in attrs])

    def __ne__(self, other: Self) -> bool:
        """check if messages are not equal.

        Args:
            other (Message): another message.

        Returns:
            bool: whether messages are not equal.
        """
        return not self.__eq__(other)

    def __hash__(self) -> int:
        """hash all message attrs.

        Returns:
            int: hashed tuple of attrs.
        """
        return hash((self.id, self.sender, self.recipient, self.content))

    def __gt__(self, other: Self) -> bool:
        """check if message if greater than the other.

        Args:
            other (Message): another message.

        Returns:
            bool: whether message is greater than the other.
        """
        return self.id > other.id

    def __ge__(self, other: Self) -> bool:
        """check if message if greater than the other or equal.

        Args:
            other (Message): another message.

        Returns:
            bool: whether message is greater or equal than the other.
        """
        return self.id >= other.id

    def __lt__(self, other: Self) -> bool:
        """check if message if lower than the other.

        Args:
            other (Message): another message.

        Returns:
            bool: whether message is lower than the other.
        """
        return self.id < other.id

    def __le__(self, other: Self) -> bool:
        """check if message if lower than the other or equal.

        Args:
            other (Message): another message.

        Returns:
            bool: whether message is lower or equal than the other.
        """
        return self.id <= other.id
```

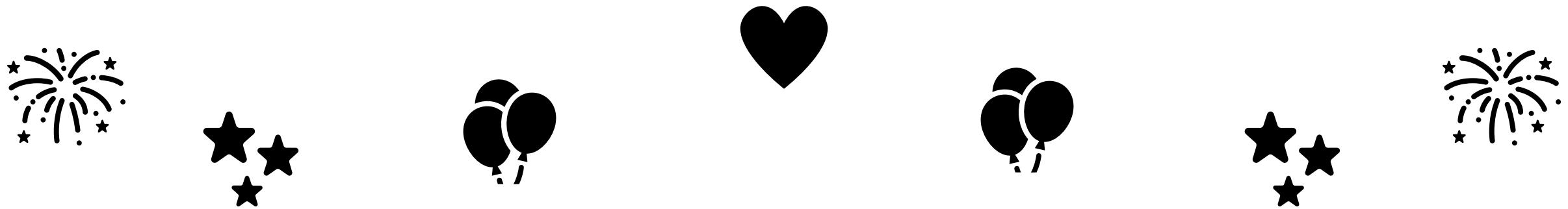
```
messages = [Message(20, User(), User(), 'first'), Message(11, User(), User(), 'second')]
print(*messages, sep=' | ')
```

```
Message id=20 from <some user> to <some user>: first | Message id=11 from <some user> to <some user>: second
```

```
messages = [Message(20, User(), User(), 'first'), Message(11, User(), User(), 'second')]
print(*messages, sep=' | ')
messages.sort()
print(*messages, sep=' | ')
```

```
Message id=20 from <some user> to <some user>: first | Message id=11 from <some user> to <some user>: second
Message id=11 from <some user> to <some user>: second | Message id=20 from <some user> to <some user>: first
```

Сириус
IT-Колледж



```python
from dataclasses import dataclass


@dataclass(frozen=True, order=True, unsafe_hash=True)
class Message:
    id: int
    sender: User
    recipient: User
    content: str
```

```
@dataclasses.dataclass(*, init=True, repr=True, eq=True, order=False,
unsafe_hash=False, frozen=False, match_args=True, kw_only=False, slots=False,
weakref_slot=False)
```

This function is a decorator that is used to add generated special methods to classes, as described below.

Сириус
IT-Колледж

The parameters to `dataclass()` are:

- `init`: If true (the default), a `__init__()` method will be generated.

  If the class already defines `__init__()`, this parameter is ignored.

- `repr`: If true (the default), a `__repr__()` method will be generated. The generated repr string will have the class name and the name and repr of each field, in the order they are defined in the class. Fields that are marked as being excluded from the repr are not included. For example: `InventoryItem(name='widget', unit_price=3.0, quantity_on_hand=10)`.

  If the class already defines `__repr__()`, this parameter is ignored.

- `eq`: If true (the default), an `__eq__()` method will be generated. This method compares the class as if it were a tuple of its fields, in order. Both instances in the comparison must be of the identical type.

  If the class already defines `__eq__()`, this parameter is ignored.

- `order`: If true (the default is `False`), `__lt__()`, `__le__()`, `__gt__()`, and `__ge__()` methods will be generated. These compare the class as if it were a tuple of its fields, in order. Both instances in the comparison must be of the identical type. If `order` is true and `eq` is false, a `ValueError` is raised.

  If the class already defines any of `__lt__()`, `__le__()`, `__gt__()`, or `__ge__()`, then `TypeError` is raised.

- `unsafe_hash`: If `False` (the default), a `__hash__()` method is generated according to how `eq` and `frozen` are set.