



SÜLEYMAN DEMİREL ÜNİVERSİTESİ
SİBER GÜVENLİK LABORATUVARI

Python Metaprogramming

Mehmet GÜRDAL

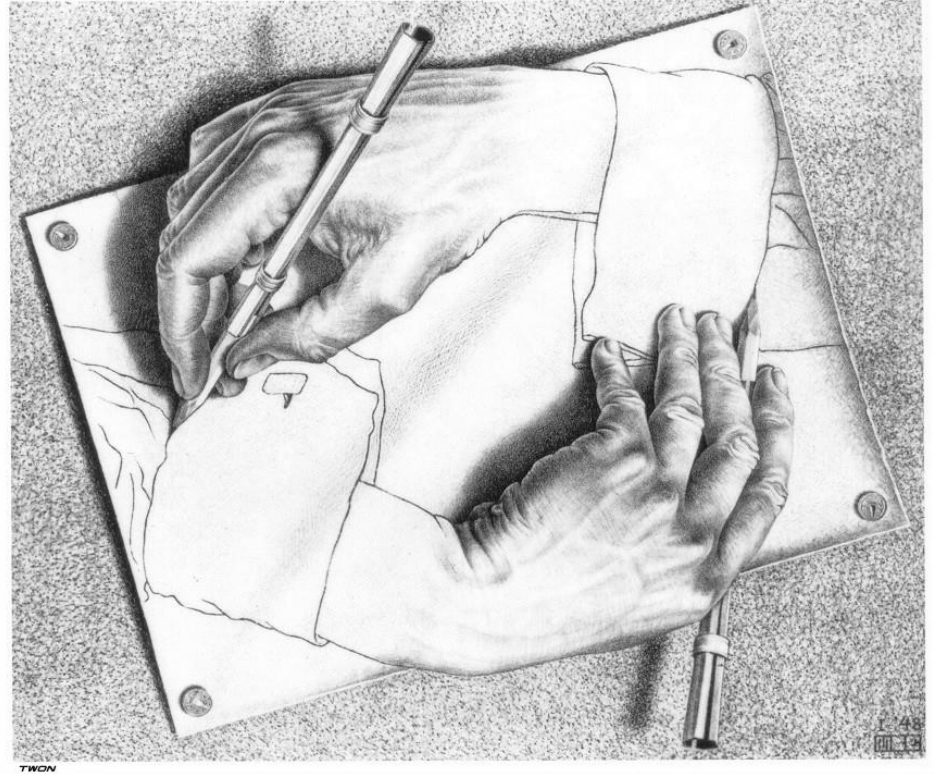
Diğer programları tasarlayan, oluşturan, analiz eden okuyan ve dönüştüren, hatta çalışma esnasında kendini değiştiren program.

Development süresini kısaltır

Yazılan satır sayısını azatır

Esneklik katar

Program yeniliklere ve değişikliklere açık olur



```
class User:

    def add(self, name):
        return name

    def delete(self, name):
        return name

    def update(self, name):
        return name

    def delete(self, name):
        return name

    .
    .
```





```
import datetime, timeit
```

```
def add(self, name):
```

```
    start = timeit.timeit()
```

```
    try:
```

```
        return name
```

```
    finally:
```

```
        stop = timeit.timeit()
```

```
        print("{} fonksiyonu {} tarihinde çalıştırıldı ve {}sn sürdü".format(
```

```
            "add",
```

```
            datetime.date.today(),
```

```
            start- stop
```

```
        ))
```

```
    .  
    .  
    .
```

Decorator

```
def log(func):
```

```
    def wrapper(*args, **kwargs):
```

```
        start = timeit.timeit()
```

```
        try:
```

```
            return func(*args, **kwargs)
```

```
        finally:
```

```
            stop = timeit.timeit()
```

```
            print("{} fonksiyonu {} tarihinde çalıştırıldı ve {}sn sürdü.".format(
```

```
                func.__qualname__,
```

```
                datetime.date.today(),
```

```
                start- stop
```

```
            ))
```

```
    return wrapper
```



```
def log(func):
```

```
    def wrapper(*args, **kwargs):
```

Eski fonksiyonun yerini tutacak bir fonksiyon tanımla

```
        start = timeit.timeit()
```

```
        try:
```

```
            return func(*args, **kwargs)
```

Asıl fonksiyonu çalıştır ve oluşan değeri döndür

```
        finally:
```

```
            stop = timeit.timeit()
```

```
            print("{} fonksiyonu {} tarihinde çalıştırıldı ve {}sn sürdü".format(
```

```
                func.__qualname__,
```

```
                datetime.date.today(),
```

```
                start- stop
```

```
            ))
```

```
    return wrapper
```

Oluşturulan yeni fonksiyonu döndür



```
def log(func=None, *, prefix=""):
    if func is None: return partial(log, prefix=prefix)
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = timeit.timeit()
        try:
            return func(*args, **kwargs)
        finally:
            stop = timeit.timeit()
            print("{}{} fonksiyonu {} tarihinde çalıştırıldı ve {}sn sürdü.".format(
                prefix,
                func.__qualname__,
                datetime.date.today(),
                start- stop
            ))
    return wrapper
```

Decorator'e parametre ekleme

```
def log(func=None, *, prefix=""):
    if func is None: return partial(log, prefix=prefix)
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = timeit.timeit()
        try:
            return func(*args, **kwargs)
        finally:
            stop = timeit.timeit()
            print("{}{} fonksiyonu {} tarihinde çalıştırıldı ve {}sn sürdü.".format(
                prefix,
                func.__qualname__,
                datetime.date.today(),
                start- stop
            ))
    return wrapper
```

Func ile ilgili bilgileri wrapper'a aktar

Decorator'e parametre ekleme

```
def log(func=None, *, prefix=""):
```

```
    if func is None: return partial(log, prefix=prefix)←
```

```
    @wraps(func)←
```

```
    def wrapper(*args, **kwargs):
```

```
        start = timeit.timeit()
```

```
        try:
```

```
            return func(*args, **kwargs)
```

```
        finally:
```

```
            stop = timeit.timeit()
```

```
            print("{}{} fonksiyonu {} tarihinde çalıştırıldı ve {}sn sürdü.".format(
```

```
                prefix,
```

```
                func.__qualname__,
```

```
                datetime.date.today(),
```

```
                start- stop
```

```
            ))
```

```
    return wrapper
```

Eğer decorator, fonksiyon şeklinde kullanılmadıysa fonksiyonmuş gibi çalıştır

Func ile ilgili bilgileri wrapper'a aktar

Decorator'e parametre ekleme

```
def log(func=None, *, prefix=""):
```

```
    if func is None: return partial(log, prefix=prefix) ←
```

```
    @wraps(func) ←
```

```
    def wrapper(*args,
```

```
        start = timeit.timeit()
```

```
        try:
```

```
            return func(*
```

```
        finally:
```

```
            stop = timeit.timeit()
```

```
            print("{}{} fonksiyonu {} tarihinde çalıştırıldı ve {}sn sürdü.".format(
```

```
                prefix,
```

```
                func.__qualname__,
```

```
                datetime.date.today(),
```

```
                start- stop
```

```
            ))
```

```
    return wrapper
```



fonksiyon şeklinde
fonksiyonmuş gibi çalıştır

gileri wrapper'a aktar

```
@log  
def add(x, y):  
    return x + y
```

```
@log  
def delete(x, y):  
    return x - y
```

```
@log(prefix="****")  
def update(x, y):  
    return x * y
```

```
@log(prefix="****")  
def get(x, y):  
    return x // y
```

TEKRAR!

```
def clslog(cls):  
    for key, val in vars(cls).items():  
        if callable(val):  
            setattr(cls, key, log(val))  
    return cls
```

```
@clslog
```

```
class User:  
    def add(self, name, passwd):  
        pass
```




```
def clslog(cls):  
    for key, val in vars(cls).items():  
        if callable(val):  
            setattr(cls, key, log(val))  
    return cls
```

Class'ı parametre olarak al

```
@clslog  
class User:  
    def add(self, name, passwd):  
        pass
```

```
def clslog(cls):  
    for key, val in vars(cls).items():  
        if callable(val):  
            setattr(cls, key, log(val))  
    return cls
```

Class'ı parametre olarak al

İçerisinde tanımlı değişkenlere ve fonksiyonlara eriş

```
@clslog
```

```
class User:
```

```
    def add(self, name, passwd):  
        pass
```

```
def clslog(cls):  
    for key, val in vars(cls).items():  
        if callable(val):  
            setattr(cls, key, log(val))  
    return cls  
  
@clslog  
class User:  
    def add(self, name, passwd):  
        pass
```

Class'ı parametre olarak al

İçerisinde tanımlı değerlere eriş

Eğer fonksiyonsa
Fonksiyonu loglanmış haliyle değiştir

```
@clslog  
class A:  
    pass
```

```
@clslog  
class B:  
    pass
```

```
@clslog  
class C:  
    pass
```

```
@clslog  
class D:  
    pass
```

TEKRAR!



@clslog



pass

Metaclass

```
a = 5
```

```
type(a)    # int
```

```
type(int)  # type
```

```
type(str)  # type
```

```
type(object) # type
```



```
# type('name', (), {})  name, bases, classdict
```

```
User = type('User', (object,), {'name':'mehmet', 'email':'@gmail.com'})
```

```
u = User()
```

```
u.name
```

```
'mehmet'
```

```
class generic_type(type):  
    def __new__(cls, name, parents, clsdict):  
        clsobj = super().__new__(cls, name, parents, clsdict)  
        new = clslog(clsobj)  
        return new
```

```
class User(metaclass=generic_type):
```

```
    .  
    .  
    .
```



```
class generic_type(type):  
    def __new__(cls, name, parents, clsdict):  
        new = super().__new__(cls, name, parents, clsdict)  
        new = clslog(new)  
        return new
```

Class oluşturmak için gerekli bilgileri al

```
class User(metaclass=generic_type):
```

```
    .  
    .  
    .
```

```
class generic_type(type):  
    def __new__(cls, name, parents, clsdict):  
        new = super().__new__(cls, name, parents, clsdict)  
        new = clslog(new)  
        return new
```

Class oluşturmak için gerekli bilgileri al

Type'ı kullanarak yeni bir class oluştur

```
class User(metaclass=generic_type):
```

```
    .  
    .  
    .
```

```
class generic_type(type):  
    def __new__(cls, name, parents, clsdict):  
        new = super().__new__(cls, name, parents, clsdict)  
        new = clslog(new)  
        return new
```

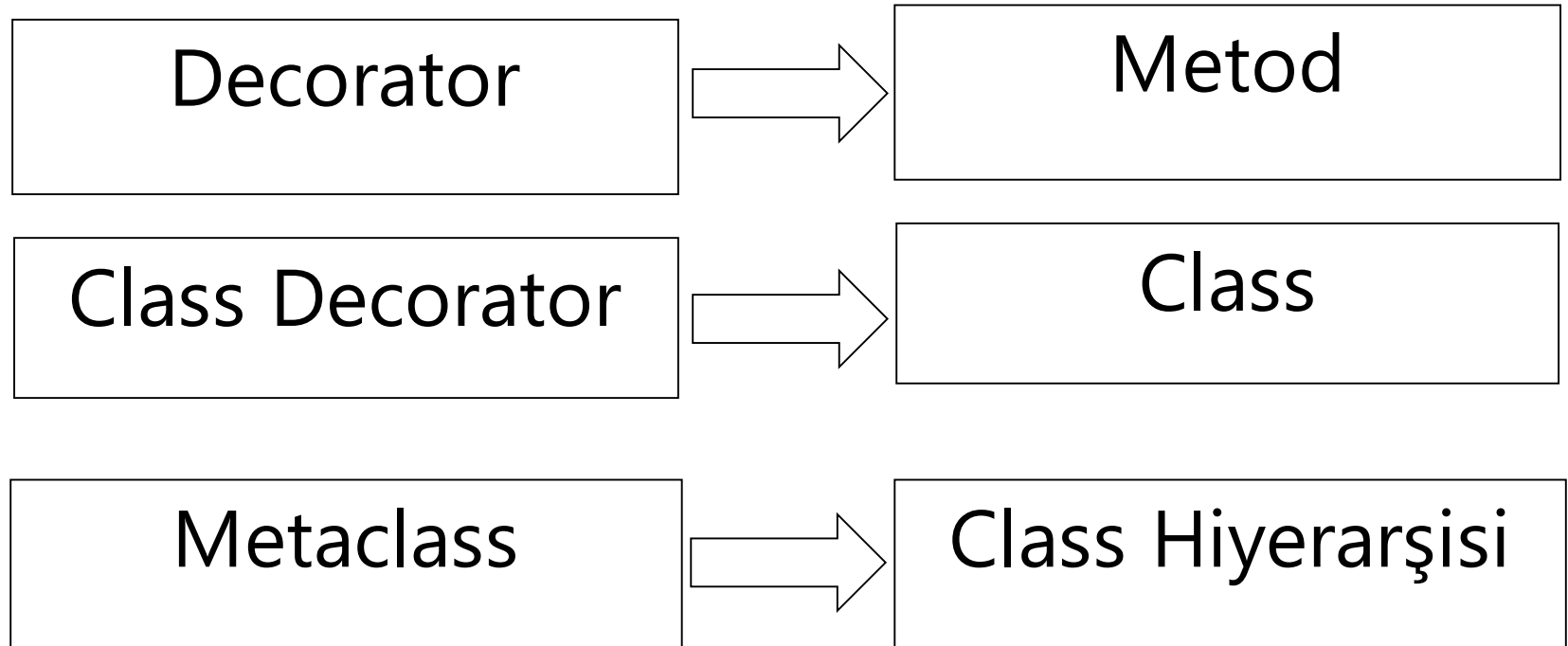
Class oluşturmak için gerekli bilgileri al

Type'ı kullanarak yeni bir class oluştur

Oluşan class'ı loglanmış haliyle değiştir

```
class User(metaclass=generic_type):
```

```
    .  
    .  
    .
```






```
class User:
```

```
    def __init__(self, name, email, age, passwd):  
        self.name = name  
        self.email = email  
        self.age = age  
        self.passwd = passwd
```

```
class Point:
```

```
    def __init__(self, x, y):  
        self.x = x  
        self.y = y
```

```
class Connection:
```

```
    def __init__(self, user, hostname, port):  
        self.user = user  
        self.hostname = hostname  
        self.port = port
```

```
class Struct:
    _fields = []
    def __init__(self, *args, **kwargs):
        for key, value in zip(self._fields, args):
            setattr(self, key, value)

class User(Struct):
    _fields = ['name', 'email', 'age', 'passwd']
```

struct adında genel bir class tanımla
_fields değişkenleri tutacak

```
class Struct:
    _fields = []
    def __init__(self, *args, **kwargs):
        for key, value in zip(self._fields, args):
            setattr(self, key, value)

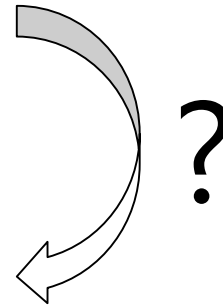
class User(Struct):
    _fields = ['name', 'email', 'age', 'passwd']
```

struct adında genel bir class tanımla
_fields değişkenleri tutacak

__init__'e parametre olarak girilenler ile
_fields listesindeki elemanları birleştirip
class'a değişken olarak ata

```
u = User('mehmet', '@hotmail.com', 32, passwd='toor')
```

```
{'age': 32, 'email': '@hotmail.com', 'name': 'mehmet'}
```



```
from inspect import Signature, Parameter
```

```
def make_signature(fields):
```

```
    return Signature(Parameter(name, Parameter.POSITIONAL_OR_KEYWORD) for name in fields)
```

```
class Struct:
```

```
    __signature__ = make_signature([])
```

```
    def __init__(self, *args, **kwargs):
```

```
        bound = self.__signature__.bind(*args, *kwargs)
```

```
        for key, value in bound.arguments.items():
```

```
            setattr(self, key, value)
```

```
class User(Struct):
```

```
    __signature__ = make_signature(['name', 'email', 'age', 'passwd'])
```

```
u = User('mehmet', '@hotmail.com', age=32, passwd="toor")
```



```
from inspect import Signature, Parameter
```

```
def make_signature(fields):
```

```
    return Signature(Parameter(name, Parameter.POSITIONAL_OR_KEYWORD) for name in fields)
```

```
class Struct:
```

```
    __signature__ = make_signature([])
```

```
    def __init__(self, *args, **kwargs):
```

```
        bound = self.__signature__.bind(*args, *kwargs)
```

```
        for key, value in bound.arguments.items():
```

```
            setattr(self, key, value)
```

```
class User(Struct):
```

```
    __signature__ = make_signature(['name', 'email', 'age', 'passwd'])
```

```
u = User('mehmet', '@hotmail.com', age=32, passwd="toor")
```

Normal bir listeden Signature üret

```
from inspect import Signature, Parameter
```

```
def make_signature(fields):
```

```
    return Signature(Parameter(name, Parameter.POSITIONAL_OR_KEYWORD) for name in fields)
```

```
class Struct:
```

```
    __signature__ = make_signature([])
```

```
    def __init__(self, *args, **kwargs):
```

```
        bound = self.__signature__.bind(*args, *kwargs)
```

```
        for key, value in bound.arguments.items():
```

```
            setattr(self, key, value)
```

Normal bir listeden Signature üret

Oluşturulan Signature'a fonksiyondaki argümanları bağla

```
class User(Struct):
```

```
    __signature__ = make_signature(['name', 'email', 'age', 'passwd'])
```

```
u = User('mehmet', '@hotmail.com', age=32, passwd="toor")
```



```
from inspect import Signature, Parameter
```

```
def make_signature(fields):
```

```
    return Signature(Parameter(name, Parameter.POSITIONAL_OR_KEYWORD) for name in fields)
```

```
class Struct:
```

```
    __signature__ = make_signature([])
```

```
    def __init__(self, *args, **kwargs):
```

```
        bound = self.__signature__.bind(*args, *kwargs)
```

```
        for key, value in bound.arguments.items():
```

```
            setattr(self, key, value)
```

```
class User(Struct):
```

```
    __signature__ = make_signature(['name', 'email', 'age', 'passwd'])
```

```
u = User('mehmet', '@hotmail.com', age=32, passwd="toor")
```

Normal bir listeden Signature üret

Oluşturulan Signature'a fonksiyondaki argümanları bağla

Signature'daki parametrelerle oluşan değişkenleri class'a ekle

```
class User(Struct):  
    __signature__ = make_signature(['name', 'email', 'age', 'passwd'])  
  
class Point(Struct):  
    __signature__ = make_signature(['x', 'y'])  
  
class Connection(Struct):  
    __signature__ = make_signature(['user', 'hostname', 'port'])
```

```
class User(Struct):  
    __signature__ = make_signature(['name', 'email', 'age', 'passwd'])
```

```
class Point(Struct):  
    __signature__ = make_signature(['x', 'y'])
```

```
class Connection(Struct):  
    __signature__ = make_signature(['user', 'hostname', 'port'])
```

Sadeleştirilebilir mi?

META-PROGRAMMING



IN PYTHON

```
class struct_meta(type):  
    def __new__(cls, name, bases, clsdict):  
        clsobj = super().__new__(cls, name, bases, clsdict)  
        sign = make_signature(clsobj._fields)  
        setattr(clsobj, "__signature__", sign)  
        return clsobj
```

```
class Struct(metaclass=struct_meta):  
    _fields = [] #__signature__ = make_signature([])  
    def __init__(self, *args, **kwargs):  
        bound = self.__signature__.bind(*args, *kwargs)  
        for key, value in bound.arguments.items():  
            setattr(self, key, value)
```

```
class struct_meta(type):
    def __new__(cls, name, bases, clsdict):
        clsobj = super().__new__(cls, name, bases, clsdict)
        sign = make_signature(clsobj._fields)
        setattr(clsobj, "__signature__", sign)
        return clsobj
```

Class içindeki `_fields` listesinden
signature oluştur

```
class Struct(metaclass=struct_meta):
    _fields = [] #__signature__ = make_signature([])
    def __init__(self, *args, **kwargs):
        bound = self.__signature__.bind(*args, *kwargs)
        for key, value in bound.arguments.items():
            setattr(self, key, value)
```

```
class struct_meta(type):  
    def __new__(cls, name, bases, clsdict):  
        clsobj = super().__new__(cls, name, bases, clsdict)  
        sign = make_signature(clsobj._fields)  
        setattr(clsobj, "__signature__", sign)  
        return clsobj
```

Class içindeki `_fields` listesinden
signature oluştur

Class'ın `__signature__`'ını yeni
signature ile değiştir

```
class Struct(metaclass=struct_meta):  
    _fields = [] #__signature__ = make_signature([])  
    def __init__(self, *args, **kwargs):  
        bound = self.__signature__.bind(*args, **kwargs)  
        for key, value in bound.arguments.items():  
            setattr(self, key, value)
```

```
class struct_meta(type):
```

```
    def __new__(cls, name, bases, attrs):  
        clsobj = super().__new__(cls, name, bases, attrs)  
        sign = meta.sign(clsobj)  
        setattr(clsobj, 'sign', sign)  
        return clsobj
```

```
class Struct(metaclass=struct_meta):  
    __fields__ = []  
    def __init__(self, *args, **kwargs):  
        bound = self.__fields__  
        for key, value in kwargs.items():  
            setattr(self, key, value)
```



esinden

yeni

Descriptor

class Descriptor:

```
def __init__(self, name=None):
```

```
    self.name = name
```

```
def __set__(self, instance, value):
```

```
    print("set", value)
```

```
def __delete__(self, instance):
```

```
    print("del", self.name)
```

class Descriptor:

```
def __init__(self, name=None):  
    self.name = name  
  
def __set__(self, instance, value):  
    print("set", value)  
  
def __delete__(self, instance):  
    print("del", self.name)
```

Manipüle edeceğimiz değişkeni tuttuk

class Descriptor:

```
def __init__(self, name=None):
```

```
    self.name = name
```

```
def __set__(self, instance, value):
```

```
    print("set", value)
```

```
def __delete__(self, instance):
```

```
    print("del", self.name)
```

Manipüle edeceğimiz değişkeni tuttuk

Değiştirme ve silme özelliklerini yeniden tasarladık

```
class Descriptor:
```

```
    def __init__(self, name=None):
```

```
        self.name = name
```

```
    def __set__(self, instance, value):
```

```
        instance.__dict__[self.name]=value
```

```
    def __delete__(self, instance):
```

```
        del instance.__dict__[self.name]
```

```
class User(Struct):
```

```
    _fields = ['name', 'email', 'age', 'passwd']
```

```
    name = Descriptor('name')
```

```
    email = Descriptor('email')
```

```
    age = Descriptor('age')
```

```
    passwd = Descriptor('passwd')
```

```
class Typed(Descriptor):  
    ty = object  
    def __set__(self, instance, value):  
        if not isinstance(value, self.ty):  
            raise TypeError("Expected {}", self.ty)  
        return super().__set__(instance, value)
```

```
class Integer(Typed):    ty = int
```

```
class String(Typed):    ty = str
```

```
class Float(Typed):    ty = float
```

```
class Typed(Descriptor):
```

```
    ty = object
```

```
    def __set__(self, instance, value):
```

```
        if not isinstance(value, self.ty):
```

```
            raise TypeError("Expected {}", self.ty)
```

```
        return super().__set__(instance, value)
```

Descriptordaki `__set__` metodunu ezdik

```
class Integer(Typed):    ty = int
```

```
class String(Typed):    ty = str
```

```
class Float(Typed):    ty = float
```

```
class Typed(Descriptor):
```

```
    ty = object
```

```
    def __set__(self, instance, value):
```

```
        if not isinstance(value, self.ty):
```

```
            raise TypeError("Expected {}", self.ty)
```

```
        return super().__set__(instance, value)
```

Descriptor'daki `__set__` metodunu ezdik

Girilen değerin bizim belirlediğimiz tipte olup olmadığını sorguladık.

```
class Integer(Typed):    ty = int
```

```
class String(Typed):    ty = str
```

```
class Float(Typed):    ty = float
```



```
class User(Struct):  
    _fields = ['name', 'email', 'age', 'passwd']  
    name = String('name')  
    email = String('email')  
    age = Integer('age')  
    passwd = Integer('passwd')
```

```
class User(Struct):  
    _fields = ['name', 'email', 'age', 'passwd']  
    name = String('name')  
    email = String('email')  
    age = Integer('age')  
    passwd = Integer('passwd')
```

```
from collections import OrderedDict

class struct_meta(type):

    @classmethod
    def __prepare__(cls, name, bases):
        return OrderedDict()

    def __new__(cls, clsname, bases, clsdict):
        fields = [key for key, val in clsdict.items()
                   if isinstance(val, Descriptor)]
        for name in fields:
            clsdict[name].name = name

        clsobj = super().__new__(cls, clsname, bases, dict(clsdict))
        sign = make_signature(fields)
        setattr(clsobj, "__signature__", sign)
        return clsobj
```

```
from collections import OrderedDict
```

```
class struct_meta(type):
```

```
    @classmethod
```

```
    def __prepare__(cls, name,
```

```
        return OrderedDict()
```

```
    def __new__(cls, clsname,
```

```
        fields = [key for key, val
```

```
            if isinstance(val, D
```

```
        for name in fields:
```

```
            clsdict[name].name
```

```
        clsobj = super().__new__(
```

```
        sign = make_signature(
```

```
        setattr(clsobj, "__signa
```

```
        return clsobj
```



```
from collections import OrderedDict
```

```
class struct_meta(type):
```

```
    @classmethod
```

```
    def __prepare__(cls, name, bases):
```

```
        return OrderedDict()
```

Class oluşturulurken kullanılan sözlüğü üreten metod

```
    def __new__(cls, clsname, bases, clsdict):
```

```
        fields = [key for key, val in clsdict.items()
```

```
                    if isinstance(val, Descriptor)]
```

```
        for name in fields:
```

```
            clsdict[name].name = name
```

```
    clsobj = super().__new__(cls, clsname, bases, dict(clsdict))
```

```
    sign = make_signature(fields)
```

```
    setattr(clsobj, "__signature__", sign)
```

```
    return clsobj
```

```
from collections import OrderedDict
```

```
class struct_meta(type):
```

```
    @classmethod
```

```
    def __prepare__(cls, name, bases):
```

```
        return OrderedDict()
```

Class oluşturulurken kullanılan sözlüğü üreten metod

```
    def __new__(cls, clsname, bases, clsdict):
```

```
        fields = [key for key, val in clsdict.items()
```

```
                    if isinstance(val, Descriptor)]
```

```
        for name in fields:
```

```
            clsdict[name].name = name
```

Descriptor tipinde tanımlı değişkenlerden
signature oluştururken kullandığımız fields'i
oluştur

```
    clsobj = super().__new__(cls, clsname, bases, dict(clsdict))
```

```
    sign = make_signature(fields)
```

```
    setattr(clsobj, "__signature__", sign)
```

```
    return clsobj
```

```
from collections import OrderedDict
```

```
class struct_meta(type):
```

```
    @classmethod
```

```
    def __prepare__(cls, name, bases):
```

```
        return OrderedDict()
```

Class oluşturulurken kullanılan sözlüğü üreten metod

```
    def __new__(cls, clsname, bases, clsdict):
```

```
        fields = [key for key, val in clsdict.items()
```

```
                    if isinstance(val, Descriptor)]
```

Descriptor tipinde tanımlı değişkenlerden
signature oluştururken kullandığımız fields'i
oluştur

```
        for name in fields:
```

```
            clsdict[name].name = name
```

name = String('name')

```
    clsobj = super().__new__(cls, clsname, bases, dict(clsdict))
```

```
    sign = make_signature(fields)
```

```
    setattr(clsobj, "__signature__", sign)
```

```
    return clsobj
```

```
class User(Struct):  
    name = String()  
    email = String()  
    age = Integer()  
    passwd = Integer()
```

```
class Point(Struct):  
    x = Float()  
    y = Float()
```

```
class Connection(Struct):  
    user = String()  
    hostname = String()  
    port = Integer()
```


Teşekkürler

Soru - Cevap

Kodlar ve Sunum
github.com/Anti-code/pymeta