Aula 5

Objetivos

- Variaveis de classe
- Herança
- Overriding de métodos
- Variaveis privadas
- Polimorfismo
- Herança múltipla

Herança

```
>>> class Car():
...     pass
...
>>> class Yugo(Car):
...     pass
...
```

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
```

```
>>> class Car():
...     def exclaim(self):
...         print("I'm a Car!")
...
>>> class Yugo(Car):
...     pass
...


>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
>>> give_me_a_car.exclaim()
I'm a Car!
>>> give_me_a_yugo.exclaim()
I'm a Car!
```

Sobreposição (Override) de método

```
>>> class Car():
...     def exclaim(self):
...         print("I'm a Car!")
...
>>> class Yugo(Car):
...     def exclaim(self):
...         print("I'm a Yugo! Much like a Car, but more Yugo-ish.")
...
```

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
```

```
>>> give_me_a_car.exclaim()
I'm a Car!
>>> give_me_a_yugo.exclaim()
I'm a Yugo! Much like a Car, but more Yugo-ish.
```

```
>>> class Person():
...     def __init__(self, name):
...         self.name = name
...
>>> class MDPerson(Person):
...     def __init__(self, name):
...         self.name = "Doctor " + name
...
>>> class JDPerson(Person):
...     def __init__(self, name):
...         self.name = name + ", Esquire"
...
```

```
>>> person = Person('Fudd')
>>> doctor = MDPerson('Fudd')
>>> lawyer = JDPerson('Fudd')
>>> print(person.name)
Fudd
>>> print(doctor.name)
Doctor Fudd
>>> print(lawyer.name)
Fudd, Esquire
```

```
>>> class Car():
...     def exclaim(self):
...         print("I'm a Car!")
...
>>> class Yugo(Car):
...     def exclaim(self):
...         print("I'm a Yugo! Much like a Car, but more Yugo-ish.")
...     def need_a_push(self):
...         print("A little help here?")
...
```

```
>>> class Person():
...     def __init__(self, name):
...         self.name = name
...
```

```
>>> class EmailPerson(Person):
...     def __init__(self, name, email):
...         super().__init__(name)
...         self.email = email
```

```
>>> bob = EmailPerson('Bob Frapples', 'bob@frapples.com')
```

```
>>> bob.name
'Bob Frapples'
>>> bob.email
'bob@frapples.com'
```

Por que não usar a implementação abaixo?

```
>>> class EmailPerson(Person):
...     def __init__(self, name, email):
...         self.name = name
...         self.email = email
```

Parâmetro self

```
>>> car = Car()
>>> car.exclaim()
I'm a Car!
```

```
>>> Car.exclaim(car)
I'm a Car!
```

Métodos podem chamar outros métodos usando o argumento self

```
class Bag:
    def __init__(self):
        self.data = []

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Variaveis de classe e de instância

```
class Dog:

    kind = 'canine'         # class variable shared by all
instances

    def __init__(self, name):
        self.name = name    # instance variable unique to
each instance


>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # shared by all dogs
'canine'
>>> e.kind                  # shared by all dogs
'canine'
>>> d.name                    # unique to d
'Fido'
>>> e.name                    # unique to e
```

```
class Dog:

    tricks = []             # mistaken use of a class
variable

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

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks               # unexpectedly shared by
all dogs
['roll over', 'play dead']
```

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list
for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

Propriedades

```
>>> class Duck():
...     def __init__(self, input_name):
...         self.hidden_name = input_name
...     def get_name(self):
...         print('inside the getter')
...         return self.hidden_name
...     def set_name(self, input_name):
...         print('inside the setter')
...         self.hidden_name = input_name
...     name = property(get_name, set_name)
```

```
>>> fowl = Duck('Howard')
>>> fowl.name
inside the getter
'Howard'
```

```
>>> fowl.name = 'Daffy'
inside the setter
>>> fowl.name
inside the getter
'Daffy'
```

```
>>> fowl.get_name()
inside the getter
'Howard'

>>> fowl.set_name('Daffy')
inside the setter
>>> fowl.name
inside the getter
'Daffy'
```

Decoradores

```
>>> class Duck():
...     def __init__(self, input_name):
...         self.hidden_name = input_name
...     @property
...     def name(self):
...         print('inside the getter')
...         return self.hidden_name
...     @name.setter
...     def name(self, input_name):
...         print('inside the setter')
...         self.hidden_name = input_name
```

```
>>> fowl = Duck('Howard')
>>> fowl.name
inside the getter
'Howard'
>>> fowl.name = 'Donald'
inside the setter
>>> fowl.name
inside the getter
'Donald'
```

```
>>> class Circle():
...     def __init__(self, radius):
...         self.radius = radius
...     @property
...     def diameter(self):
...         return 2 * self.radius
...
```

```
>>> c = Circle(5)
>>> c.radius
5

>>> c.diameter
10

>>> c.radius = 7
>>> c.diameter
14


>>> c.diameter = 20
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Variaveis privadas

- Em Python não existem variaveis instância acessadas exclusivamente dentro do objeto.
- Existe uma convenção que estabelece que nomes que se iniciam com “_” devem ser tratados como nomes não públicos.
- Nomes iniciados com “__” (pelo menos dois _) são substituidos por _classname_name. Onde classname é o nome da classe corrente. (*Name mangling*)

```
class SecretString:
    '''A not-at-all secure way to store a secret string.'''

    def __init__(self, plain_string, pass_phrase):
        self.__plain_string = plain_string
        self.__pass_phrase = pass_phrase

    def decrypt(self, pass_phrase):
        '''Only show the string if the pass_phrase is correct.'''
        if pass_phrase == self.__pass_phrase:
            return self.__plain_string
        else:
            return ''
```

```
>>> secret_string = SecretString("ACME: Top Secret", "antwerp")
>>> print(secret_string.decrypt("antwerp"))
ACME: Top Secret
```

```
>>> print(secret_string.__plain_string)
Traceback( ……
……
Attribute error SecretString has no attribute
_plain_string
```

```
>>> print(secret_string._SecretString__plain_string)
ACME: Top Secret
```

Métodos de instância, de classe e estáticos

```
class MyClass:
    def method(self):
        return 'instance method called‘, self

    @classmethod
    def classmethod(cls):
        return 'class method called', cls

    @staticmethod
    def staticmethod():
        return 'static method called'
```

```
>>> obj = MyClass()
>>> obj.method()
('instance method called', <MyClass instance at 0x101a2f4c8>)
```

```
>>> obj.classmethod()
('class method called', <class MyClass at 0x101a2f4c8>)
```

```
>>> obj.staticmethod()
'static method called'
```

```
>>> MyClass.classmethod()
('class method called', <class MyClass at 0x101a2f4c8>)

>>> MyClass.staticmethod()
'static method called'

>>> MyClass.method()
TypeError: unbound method method() must
    be called with MyClass instance as first
    argument (got nothing instead)
```

```
class Pizza:
    def __init__(self, ingredients):
        self.ingredients = ingredients

    def __repr__(self):
        return f'Pizza({self.ingredients!r})'

>>> Pizza(['cheese', 'tomatoes'])
Pizza(['cheese', 'tomatoes'])
```

```
Pizza(['mozzarella', 'tomatoes'])
Pizza(['mozzarella', 'tomatoes', 'ham', 'mushrooms'])
Pizza(['mozzarella'] * 4)
```

```
class Pizza:
    def __init__(self, ingredients):
        self.ingredients = ingredients

    def __repr__(self):
        return f'Pizza({self.ingredients!r})'

    @classmethod
    def margherita(cls):
        return cls(['mozzarella', 'tomatoes'])

    @classmethod
    def prosciutto(cls):
        return cls(['mozzarella', 'tomatoes', 'ham'])
```

```
>>> Pizza.margherita()
Pizza(['mozzarella', 'tomatoes'])

>>> Pizza.prosciutto()
Pizza(['mozzarella', 'tomatoes', 'ham'])
```

```
import math

class Pizza:
    def __init__(self, radius, ingredients):
        self.radius = radius
        self.ingredients = ingredients

    def __repr__(self):
        return (f'Pizza({self.radius!r}, '
                f'{self.ingredients!r})')

    def area(self):
        return self.circle_area(self.radius)

    @staticmethod
    def circle_area(r):
        return r ** 2 * math.pi
```

```
>>> p = Pizza(4, ['mozzarella', 'tomatoes'])
>>> p
Pizza(4, ['mozzarella', 'tomatoes'])
>>> p.area()
50.26548245743669
>>> Pizza.circle_area(4)
50.26548245743669
```

Observações

- **Métodos de instância** necessitam de um objeto instanciado e pode ser acessado via ponteiro self.
- **Métodos de classe** não necessitam de um objeto instanciado. Não pode acessar a instância.
- **Métodos estáticos** não possuem acesso a classe ou a instância. São como funções regulares que pertencem ao espaço de nomes de uma classe.
- **Métodos de classe e estáticos** servem para comunicar uma intenção de projeto da classe. Server para forçar o desenvolvedor a seguir um determinado design. Traz benefícios para manutenção.

Classes abstratas

```
# the simplest way to write an abstract method in
Python is:

class Pizza(object):
    def get_radius(self):
        raise NotImplementedError
```

oblema: Se for criada uma subclasse de Pizza e for esquecido a implementação
get_radius, o erro somente será descoberto quando o método for executado.

```
import abc

class BasePizza(object):
    __metaclass__  = abc.ABCMeta

    @abc.abstractmethod
    def get_radius(self):
         """Method that should do
something."""
```

```
>>> BasePizza()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class BasePizza with abstract
methods get_radius
```

```
import abc

class BasePizza(object):
    __metaclass__=abc.ABCMeta

    def __init__(self):
         self.ingredients=['cheese']

    @abc.abstractmethod
    def get_ingredients(self):
        """Method should do something"""

class Calzone(BasePizza):
    def get_ingredients(self,with_egg=False):
        if with_egg:
            return self.ingredients+['egg']
        return self.ingredients

x=Calzone()
print(x.get_ingredients(True))
```

```
import abc

class BasePizza(object):
    __metaclass__  = abc.ABCMeta

    @abc.abstractmethod
    def get_ingredients(self):
         """Returns the ingredient list."""

class DietPizza(BasePizza):
    @staticmethod
    def get_ingredients():
        return None
```

```
import abc

class BasePizza(object):
    __metaclass__  = abc.ABCMeta

    ingredients = ['cheese']

    @classmethod
    @abc.abstractmethod
    def get_ingredients(cls):
        """Returns the ingredient list."""
        return cls.ingredients
```

```
import abc

class BasePizza(object):
    __metaclass__  = abc.ABCMeta

    default_ingredients = ['cheese']

    @classmethod
    @abc.abstractmethod
    def get_ingredients(cls):
         """Returns the ingredient list."""
         return cls.default_ingredients

class DietPizza(BasePizza):
    def get_ingredients(self):
        return ['egg'] + super(DietPizza, self).get_ingredients()
```

Referencias

- https://julien.danjou.info/blog/2013/guide-python-static-class-abstract-methods

Exercícios

- Crie um exemplo usando métodos abstratos, métodos estáticos e métodos de classe. O exemplo deve ilustrar as vantagens de cada tipo de método.

Polimorfismo

- Polimorfismo permite que objetos de classes distintas executem implementações distintas para um mesmo método.
- A decisão de qual implementação será ativada é realizada durante a execução. (*Dynamic Binding)*
- Exemplos:

Operador + em Números e Strings.

```
class AudioFile:
    def __init__(self, filename):
        if not filename.endswith(self.ext):
            raise Exception("Invalid file format")

        self.filename = filename

class MP3File(AudioFile):
    ext = "mp3"
    def play(self):
        print("playing {} as mp3".format(self.filename))

class WavFile(AudioFile):
    ext = "wav"
    def play(self):
        print("playing {} as wav".format(self.filename))

class OggFile(AudioFile):
    ext = "ogg"
    def play(self):
        print("playing {} as ogg".format(self.filename))
```

```
>>> ogg = OggFile("myfile.ogg")
>>> ogg.play()
playing myfile.ogg as ogg
>>> mp3 = MP3File("myfile.mp3")
>>> mp3.play()
playing myfile.mp3 as mp3
>>> not_an_mp3 = MP3File("myfile.ogg")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "polymorphic_audio.py", line 4, in __init__
    raise Exception("Invalid file format")
Exception: Invalid file format
```

Python usa *Duck Typing*

```
class FlacFile:
    def __init__(self, filename):
        if not filename.endswith(".flac"):
            raise Exception("Invalid file format")

        self.filename = filename

    def play(self):
        print("playing {} as flac".format(self.filename))
```

Métodos para Sobreposição de Operadores

Method	Overloads	Called for
`__init__`	Constructor	Object creation: `X = Class()`
`__del__`	Destructor	Object redamation
`__add__`	Operator +	`X + Y`, `X += Y`
`__or__`	Operator | (bitwise OR)	`X | Y`, `X |= Y`
`__repr__`,`__str__`	Printing, conversions	`print X`, `repr(X)`, `str(X)`
`__call__`	Function calls	`X()`
`__getattr__`	Qualification	`X.undefined`
`__setattr__`	Attribute assignment	`X.any = value`
`__getitem__`	Indexing	`X[key]`, `for` loops and other iterations if no `__iter__`
`__setitem__`	Index assignment	`X[key] = value`
`__len__`	Length	`len(X)`, truth tests
`__cmp__`	Comparison	`X == Y`, `X < Y`
`__lt__`	Specific comparison	`X < Y` (or else `__cmp__`)
`__eq__`	Specific comparison	`X == Y` (or else `__cmp__`)
`__radd__`	Right-side operator +	`Noninstance + X`
`__iadd__`	In-place (augmented) addition	`X += Y` (or else `__add__`)
`__iter__`	Iteration contexts	`for` loops, `in` tests, list comprehensions, map, others

```
>>> class empty:
...     def __getattr__(self, attrname):
...         if attrname == "age":
...             return 40
...         else:
...             raise AttributeError, attrname
...
>>> X = empty()
>>> X.age
40
>>> X.name
...error text omitted...
AttributeError: name
```