Universidade Federal do Piauí - UFPI

Sistemas de Informação

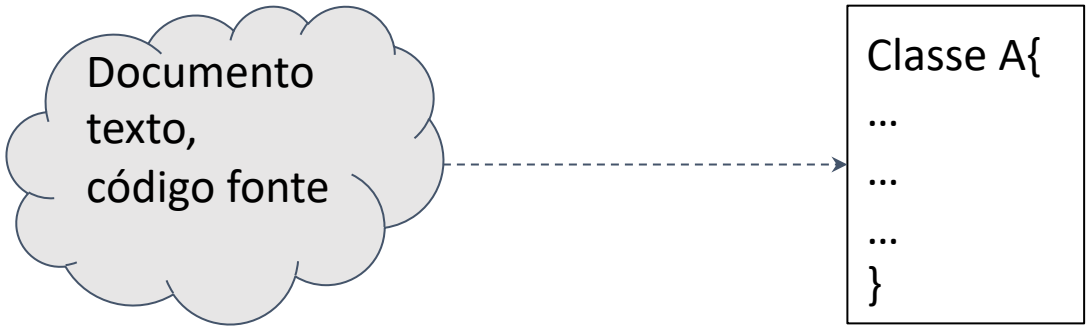
Programação Orientada a Objetos I- POO

Introdução a orientação a objetos

Prof. Flávio Araújo - UFPI - Picos PI

Classes e Objetos

Uma classe é uma estrutura que abstrai um conjunto de objetos com características similares. Uma classe define o comportamento de seus objetos - através de métodos - e os estados possíveis destes objetos - através de atributos.



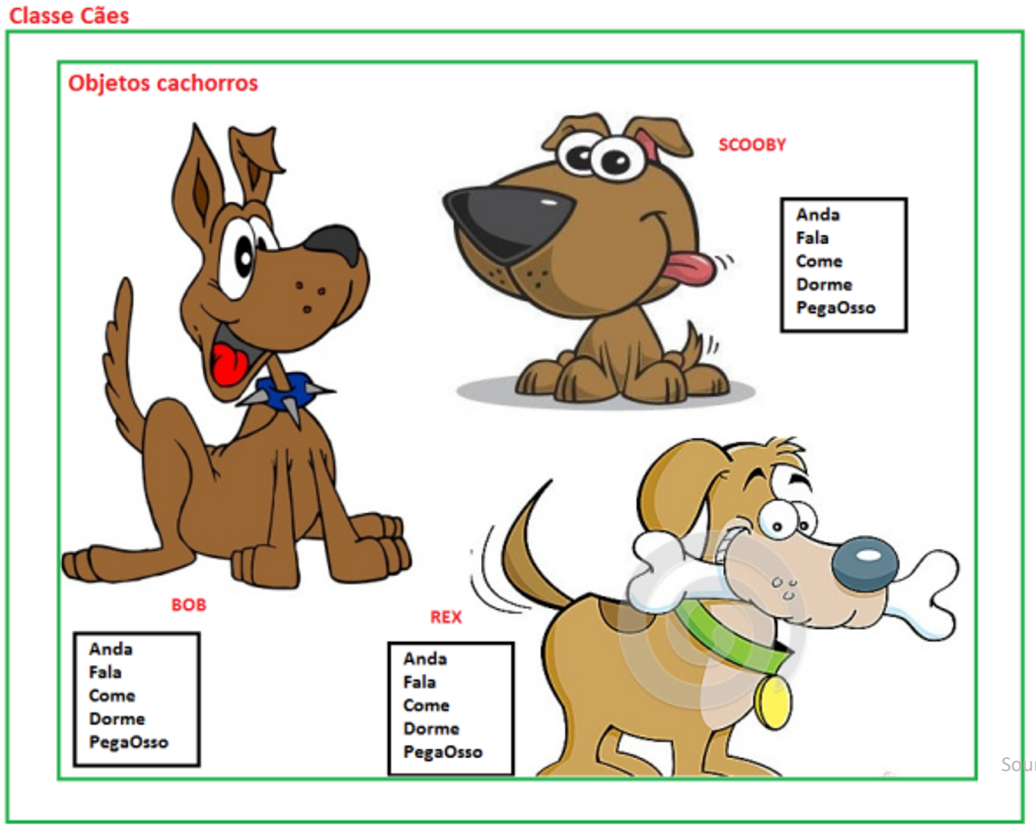
Objeto

Um **objeto**, em programação orientada a objetos, é uma instância (ou seja, um exemplar) de uma classe.

Em POO, um objeto:

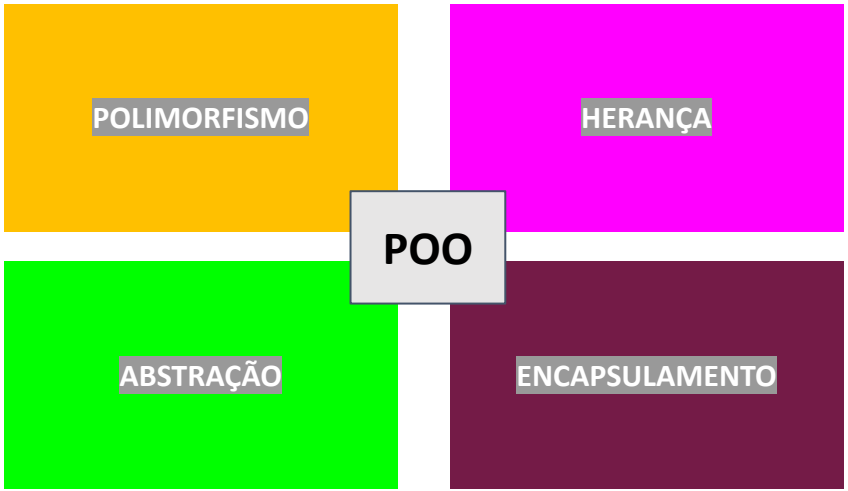
- ocupa memória
- é criado
- seus métodos ocupam CPU

Orientação a Objetos



Source: http://tiny.cc/q03kbz

Programação Orientada a Objetos



Orientação a Objetos

```
class Conta:
    pass
```

```
>>> from conta import Conta
>>> conta = Conta()
>>> type(conta)
<class 'conta.Conta'>
```

```
class Conta:
    def __init__(self, numero, titular, saldo, limite):
        self.numero = numero
        self.titular = titular
        self.saldo = saldo
        self.limite = limite
```

```
>>> from conta import Conta
>>> conta = Conta()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: __init__() missing 4 required positional arguments: 'numero', 'titular', 'saldo', and 'lim
ite
```

```
>>> conta = Conta('123-4', 'João', 120.0, 1000.0)
```

Métodos

```
class Conta:

    # método __init__() omitido

    def deposita(self, valor):
        self.saldo += valor

    def saca(self, valor):
        self.saldo -= valor

    def extrato(self):
        print("numero: {} \nsaldo: {}".format(self.numero, self.saldo))
```

```
>>> from conta import Conta
>>>
>>> conta = Conta('123-4', 'João', 120.0, 1000.0)
>>> conta.deposita(20.0)
>>> conta.extrato()
numero: '123-4'
saldo: 140.0
>>> conta.saca(15)
>>> conta.extrato()
numero: '123-4'
saldo: 125.0
```

Atividade

- Construa a classe Conta com:
  - os métodos deposita, saca e extrato;
  - os atributos numero, titular, saldo e limite.

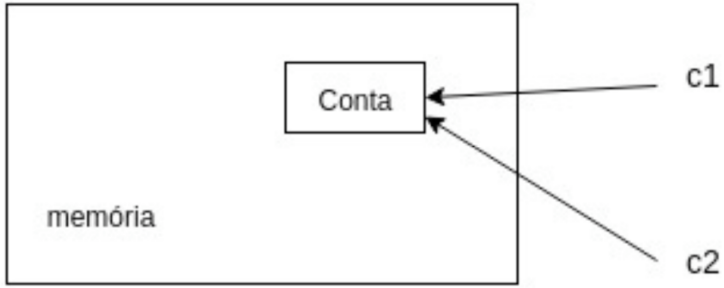
Métodos com Retorno

```
def saca(self, valor):
    if (self.saldo < valor):
        return False
    else:
        self.saldo -= valor
        return True
```

```
>>> from conta import Conta
>>> minha_conta.saldo = 1000
>>> consegui = minha_conta.saca(2000)
>>> if(consegui):
...     print(“consegui sacar”)
... else:
...     print(“não consegui sacar”)
>>>
'não consegui sacar'
```

Objetos são acessados por **Referência**

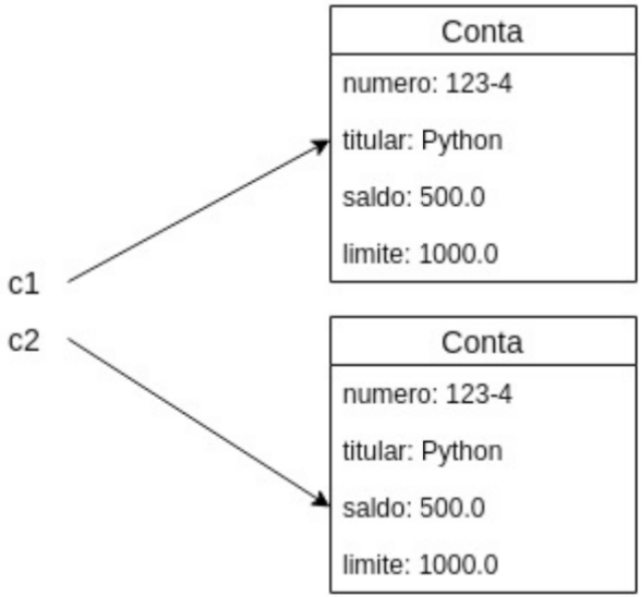
```
>>> from conta import Conta
>>> c1 = Conta('123-4', 'João', 120.0, 1000.0)
>>> c2 = c1
>>> c2.saldo
120.0
>>> c1.deposita(100.0)
>>> c1.saldo
220.0
>>> c2.deposita(30.0)
>>> c2.saldo
250.0
>>> c1.saldo
250.0
```




```
>>> id(c1) == id(c2)
True
>>> c1 == c2
True
```

Objetos são acessados por **Referência**

```
>>> c1 = Conta("123-4", "Python", 500.0, 1000.0)
>>> c2 = Conta("123-4", "Python", 500.0, 1000.0)
>>> if(c1 == c2):
...     print(“contas iguais”)
>>>
```



Método Transfere

```
class Conta:

    # código omitido

    def transfere(self, destino, valor):
        retirou = self.saca(valor)
        if (retirou == False):
            return False
        else:
            destino.deposita(valor)
            return True
```

Atividade

- Adicione a função transfere na sua conta;
- Faça testes para transferir entre contas, com e sem saldo suficiente.

Atributos com valor padrão

```
class Conta:

    def __init__(self, numero, titular, saldo, limite=1000.0):
        self.numero = numero
        self.titular = titular
        self.saldo = saldo
        self.limite = limite
```

```
>>> conta = Conta('123-4', 'joão', 120.0)
```

Atributos como objetos de outras classes- Agregação

```
class Cliente:

    def __init__(self, nome, sobrenome, cpf):
        self.nome = nome
        self.sobrenome = sobrenome
        self.cpf = cpf

class Conta:

    def __init__(self, numero, cliente, saldo, limite):
        self.numero = numero
        self.titular = cliente
        self.saldo = saldo
        self.limite = limite


>>> from conta import Conta, Cliente
>>> cliente = Cliente('João', 'Oliveira', '1111111111-1')
>>> minha_conta = Conta('123-4', cliente, 120.0, 1000.0)
```

Atividade

- Edite seu projeto e crie a classe Cliente com nome, sobrenome e cpf;
- Toda conta agora deverá receber um cliente como parâmetros;

Tudo é Objeto

```
>>> type(conta.numero)
<class 'str'>
>>> type(conta.saldo)
<class 'float'>
>>> type(conta.titular)
<class '__conta__.Cliente'>
```

COMPOSIÇÃO

```
import datetime

class Historico:

    def __init__(self):
        self.data_abertura = datetime.datetime.today()
        self.transacoes = []

    def imprime(self):
        print("data abertura: {}".format(self.data_abertura)
        print("transações: ")
        for t in self.transacoes:
            print("-", t)


class Conta:

    def __init__(self, numero, cliente, saldo, limite=1000.0
        self.numero = numero
        self.cliente = cliente
        self.saldo = saldo
        self.limite = limite
        self.historico = Historico()
```

```
class Conta:

    #código omitido

    def deposita(self, valor):
        self.saldo += valor
        self.historico.transacoes.append("depósito de {}".format(valor))

    def saca(self, valor):
        if (self.saldo < valor):
            return False
        else:
            self.saldo -= valor
            self.historico.transacoes.append("saque de {}".format(valor))

    def extrato(self):
        print("numero: {} \nsaldo: {}".format(self.numero, self.saldo))
        self.historico.transacoes.append("tirou extrato - saldo
        de {}".format(self.saldo))

    def transfere_para(self, destino, valor):
        retirou = self.saca(valor)
        if (retirou == False):
            return False
        else:
            destino.deposita(valor)
            self.historico.transacoes.append("transferencia de {}
            para conta {}".format(valor, destino.numero))
            return True
```

Testando...

```
$python3.6
>>> from conta import Conta, Cliente
>>> cliente1 = Cliente('João', 'Oliveira', '11111111111-11')
>>> cliente2 = Cliente('José', 'Azevedo', '222222222-22')
>>> conta1 = Conta('123-4', cliente1, 1000.0)
>>> conta2 = Conta('123-5', cliente2, 1000.0)
>>> conta1.deposita(100.0)
>>> conta1.saca(50.0)
>>> conta1.transfere_para(conta2, 200.0)
>>> conta1.extrato
numero: 123-4
saldo: 850.0
>>> conta1.historico.imprime()
data abertura: 2018-05-10 19:44:07.406533
transações:
- depósito de 100.0
- saque de 50.0
- saque de 200.0
- transferencia de 200.0 para conta 123-5
- tirou extrato - saldo de 850.0
>>> conta2.historico.imprime()
data abertura: 2018-05-10 19:44:07.406553
transações:
- depósito de 200.0
```

Atividade

- Inclua a classe Histórico na sua conta;
- Faça testes com depósitos, transferências e saques.

Modificador de Acesso

- Em Python utilizamos “__” para modificar o acesso do atributo. Em Java, por exemplo, utilizamos **private.**

```
class Pessoa:

    def __init__(self, idade):
        self.__idade = idade
```

```
>>> pessoa = Pessoa(20)
>>> pessoa.idade
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Pessoa' object has no attribute 'idade'
```

- Em Python nenhum atributo é verdadeiramente privado... Podemos acessa o atributo idade da seguinte forma:

```
>>> p._Pessoa__idade
```

- **Entretanto, isso é considerado uma má pratica!**

Modificador de Acesso

- Entretanto existe uma convenção em Python que todo atributo com ‘_’ é um atributo privado e não deve ser acessado fora da classe.

```
def __init__(self, idade):
    self._idade = idade
```

Get e Set

- Visto que nossos atributos são privados e só devemos acessar os mesmos dentro da classe, é necessário criar modos de acessar esses atributos;

- A maioria das linguagens de programação utilizam *getters* e *setters*, para acessar e modificar atributos, respectivamente.

```
class Conta:

    def __init__(self, titular, saldo):
        self._titular = titular
        self._saldo = saldo
def get_saldo(self):
    return self._saldo

def set_saldo(self, saldo):
    self._saldo = saldo

def get_titular(self):
    return self._titular

def set_titular(self, titular):
    self._titular = titular
```

... Outra solução em Python

- Um método que é usado para obter um valor (o *getter*) é decorado com @property;
  - colocamos essa linha diretamente acima da declaração do método que recebe o nome do próprio atributo.
- O método que tem que funcionar como *setter* é decorado com @saldo.setter.
- Podemos chamar esses métodos sem os parênteses, como se fossem atributos públicos;
- É uma forma mais elegante de encapsular nossos atributos.

```
class Conta:

    def __init__(self, saldo=0.0):
        self._saldo = saldo

    @property
    def saldo(self):
        return self._saldo

    @saldo.setter
    def saldo(self, saldo):
        if(self._saldo < 0):
            print("saldo não pode ser negativo")
        else:
            self._saldo = saldo
```

```
>>> conta = Conta(1000.0)
>>> conta.saldo = -300.0
"saldo não pode ser negativo"
```

Atividade

- Crie os métodos de controle de acesso para os atributos da conta.
- **Obs: o atributo saldo não deve ser incluído, pois o mesmo deve ser atualizado através dos métodos de sacar e depositar!**

Atributos da Classe

- Para criar um atributo que controle o total de contas criadas é necessário que o “total de contas” seja um atributo da classe Conta, e não de um objeto em particular.

```
class Conta:

    total_contas = 0

    def __init__(self, saldo):
        self._saldo = saldo
        Conta.total_contas += 1
```

```
>>> c1 = Conta(100.0)
>>> c1.total_contas
1
>>> c2 = Conta(200.0)
>>> c2.total_contas
2
```

```
>>> Conta.total_contas
2
```

Atributos da Classe

- Para controlar o acesso ao atributo, vamos adicionar o “_”

```
class Conta:

    _total_contas = 0
```

- Entretanto....

```
>>> Conta.total_contas
Traceback (most recent call last):
  File <stdin>, line 23, in <module>
    Conta.total_contas
AttributeError: 'Conta' object has no attribute 'total_contas'
```

Criando um get...

- Funciona quando chamamos este método por um instância, mas quando fazemos *Conta.get_total_contas()* o interpretador reclama pois não passamos a instância:

```
class Conta:

    _total_contas = 0

    # __init__ e outros métodos

    def get_total_contas(self):
        return Conta._total_contas
```

```
>>> c1 = Conta(100.0)
>>> c1.get_total_contas()
1
>>> c2 = Conta(200.0)
>>> c2.get_total_contas()
2
>>> Conta.get_total_contas()
Traceback (most recent call last):
  File <stdin>, line 17, in <module>
    Conta.get_total_contas()
TypeError: get_total_contas() missing 1 required positional argument: 'self'
```

Continuando...

```
>>> c1 = Conta(100.0)
>>> c2 = Conta(200.0)
>>> Conta.get_total_contas(c1)
2
```

- O código acima funciona, mas não é a maneira correta de se fazer.
- E se tirar o self do método?

```
def get_total_contas():
    return Conta._total_contas
```

```
>>> c1 = Conta(100.0)
>>> c1.get_total_contas()
Traceback (most recent call last):
  File <stdin> in <module>
    c1.get_total_contas()
TypeError: get_total_contas() takes 0 positional arguments but 1 was given
```

Solução: @staticmethod

```
@staticmethod
def get_total_contas():
    return Conta._total_contas
```

```
>>> c1 = Conta(100.0)
>>> c1.get_total_contas()
1
>>> c2 = Conta(200.0)
>>> c2.get_total_contas()
2
>>> Conta.get_total_contas()
2
```

Atividade

- Crie o contador de contas na sua classe Conta.

Slots

- Mas como Python é uma linguagem dinâmica, nada impede que usuários de nossa classe Conta criem atributos em tempo de execução, fazendo, por exemplo:

```
>>> conta.nome = "minha conta"
```

- Esse código não acusa erro e nossa conta fica aberta a modificações ferindo a segurança da classe;
- Para evitar isso podemos utilizar uma variável embutida no Python chamada __slots__ que pode guardar uma lista de atributos da classe definidos por nos:

```
class Conta:

    __slots__ = ['_numero', '_titular', '_saldo', '_limite']

    def __init__(self, numero, titular, saldo, limite=1000.0):
        # inicialização dos atributos
```

Atividade

- Crie Slots em sua classe Conta;

Atividade

- Similar à classe Conta criada em sala de aula, crie uma classe Fotografia.
- A classe Fotografia terá os seguintes atributos:
  - Foto -> recebe str com endereço da imagem (o atributo deverá armazenar a imagem e não o endereço dela, dica: leia a imagem com: **from skimage.io import imread**)
  - Fotógrafo -> Pessoa com Nome, CPF, Endereço e telefone
  - Data -> data que a fotografia foi obtida
  - Proprietário -> Pessoa
  - Quantidade de Fotos -> contador para quantidade de objetos Fotografia criados
- A classe Fotografia terá os seguintes métodos:
  - Mostrar Fotografia (dica: **from matplotlib.pyplot import imshow**) dica2 para pausar a exibição da imagem: **from matplotlib.pyplot import show.** Use imshow, depois show.
  - Propriedades da Fotografia: tamanho da Fotografia em pixels (dica: **fotografia.shape**), fotógrafo, data
  - Métodos para alterar e acessar atributos.
- Crie Slots