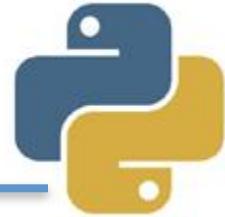


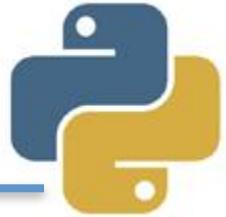
Capítulo 03



Orientação a Objetos em Python



Márcio Palheta, M.Sc.
marcio.palheta@gmail.com

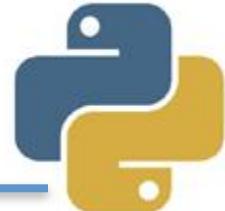


Apresentação

- Programador desde 2000
- Professor de programação desde 2009
- Mestre pelo ICOMP/UFAM – 2013
- Fundador da Buritech – 2014
- Doutorando pelo ICOMP/UFAM
- Pesquisador das áreas: Banco de Dados, Recuperação da Informação, Big Data, Mineração de dados e Aprendizado de Máquina

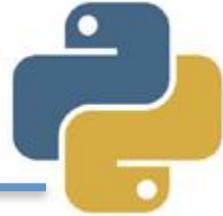


Agenda



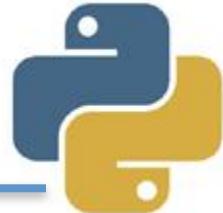
- Revisão da aula anterior
 - Revendo a programação estruturada
 - Pensamento Orientado a Objetos
 - Diferença entre Classes e objetos
 - Atributos e Métodos
 - Relacionamentos entre classes
-

Revisão



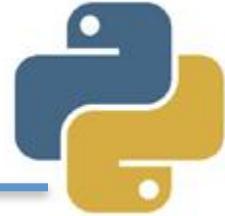
- Declaração e tipos de variáveis
 - Atribuição e conversão de tipos
 - Blocos de comandos e a Indentação
 - Estruturas condicionais
 - Laços de repetição
 - Listas, tuplas e dicionários
-

O que temos por aqui?



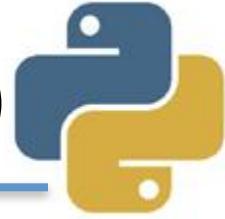
- O que é Orientação a Objetos?
 - Como utilizar?
 - Entendendo as definições de Classes, objetos, atributos e métodos
 - Como fica a memória, quando rodamos um programa OO ?
-

Orientação a Objetos – OO



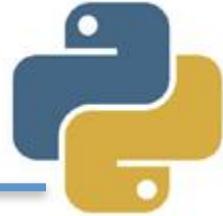
- Em **python**, **não existem** variáveis de **tipos primitivos**. Tudo é objeto
 - Diferente da programação estruturada, em OO **juntamos dados e operações** em um mesmo lugar (objeto)
 - As **definições** de quais dados e operações um objeto possui são feitas em **classes**
-

Benefícios da abordagem OO

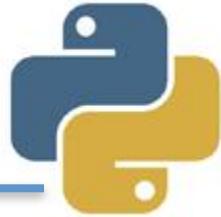


- **Modularidade**: Uma vez criado, um objeto pode ser passado por todo o sistema
 - **Encapsulamento**: Detalhes de implementação ficam ocultos externamente ao objeto
 - **Reuso**: Uma vez criado, um objeto pode ser utilizado em outros programas
 - **Manutenibilidade**: manutenção é realizada em pontos específicos do seu programa
-

Objetos



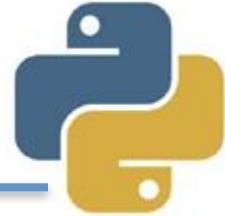
- São “COISAS” do mundo real, que possuem dados (**atributos**) e realizam ações(**métodos**) relevantes
 - O estado de um objeto pode variar de acordo com o tempo de execução de um programa
 - São oriundos (**instâncias**) de classes;
-



Objetos

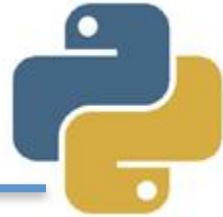
Objeto	Estado	Comportamento
Pessoa	Nome, idade, RG	Falar, andar, respirar
Cachorro	Nome, raça	Latir, correr
Conta bancária	Agência, número	Creditar, debitar
Carro	Cor, marca, modelo	Acelerar, frear, abastecer;

Classes



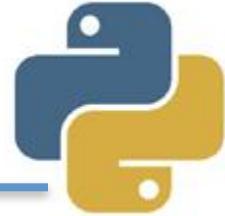
- A **classe** é o conjunto de especificações de atributos e métodos que um objeto possui.
 - Todo **objeto** é criado (ou **instanciado**) a partir de uma classe, respeitando suas regras
 - Um **objeto** é uma **instância** de uma classe
-

Classes e objetos



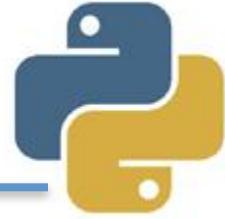
Documento	Documento01	Documento02
Foto:	Img01.jpg	Img02.jpg
Código:	123456	654321
Nome:	Joao	Maria
Nascimento:	10/05/1980	30/06/1990

Pensando em sistema



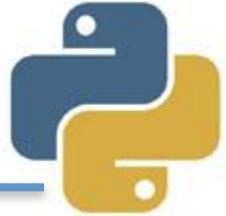
- Imagine que você é programador de uma **instituição financeira** e precisa criar um sistema para cadastro de **contas bancárias** de clientes
 - Consegues perceber que a **conta bancária** é uma **entidade** importante para este contexto?
-

Pensando na conta bancária



- Quais **características** da conta bancária são **importantes** para o nosso projeto?
 - Número da conta, Agência, Nome do Cliente e Saldo
 - Quais **operações** poderíamos realizar em nossas contas bancárias?
 - Sacar, depositar, transferir, ver saldo
-

Pensando na modelagem

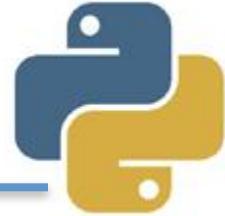


ContaBancaria

- numero : String
- agencia : String
- nome_cliente : String
- saldo : double

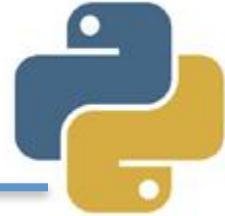
+ sacar(valor : double) : boolean
+ depositar(valor : double) : void
+ transferir(valor : double, destino : ContaBancaria) : boolean

Antes de continuar

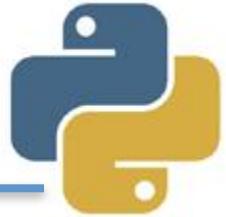


- Nos capítulos anteriores, programamos usando o **interpretador** do python e os **notebooks** do jupyter
 - Agora, vamos começar a **modularizar** nossos programas, criando arquivos com **finalidades distintas**
 - Hora de **mudar** de local de edição
-

Ambiente de desenvolvimento

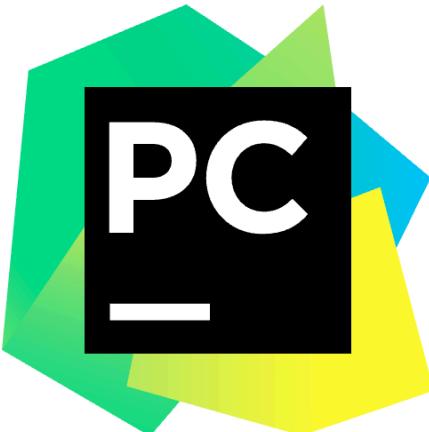


- O python **não requer** uma IDE específica, deixando essa escolha a cargo do **programador**
 - As mais **comuns**:
 - Notepad++, Sublime text
 - Canopy, Pydev, PyCharm
 - Neste curso, usaremos o **Pycharm**
-



Baixe e instale

- <https://www.jetbrains.com/pycharm/download>



Version: 2017.1.2
Build: 171.4249.47
Released: April 27, 2017

[System requirements](#)
[Installation Instructions](#)

Download PyCharm

[Windows](#) macOS [Linux](#)

Professional

Full-featured IDE
for Python & Web
development

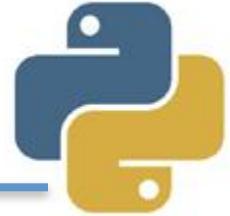
[DOWNLOAD](#)
Free trial

Community

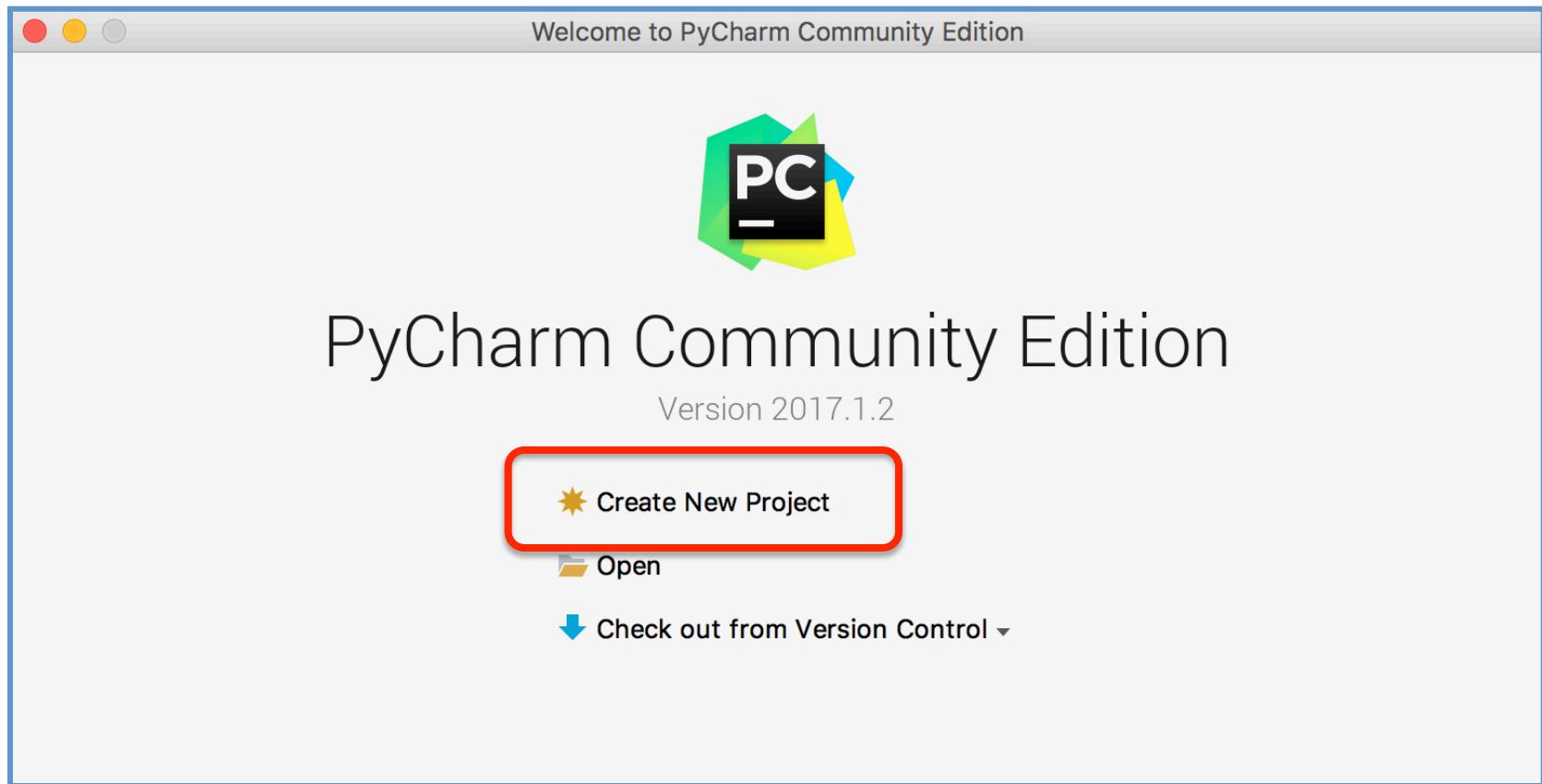
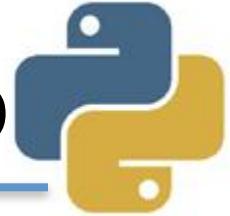
Lightweight IDE
for Python & Scientific
development

[DOWNLOAD](#)
Free, open-source

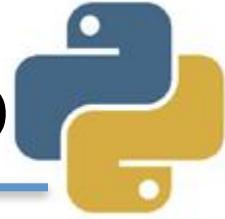
Execute o PyCharm



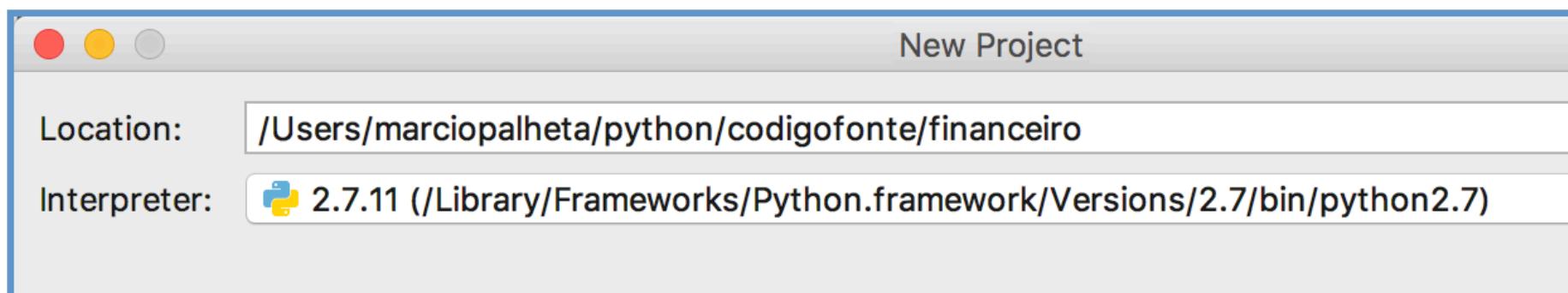
Vamos criar nosso 1º Projeto



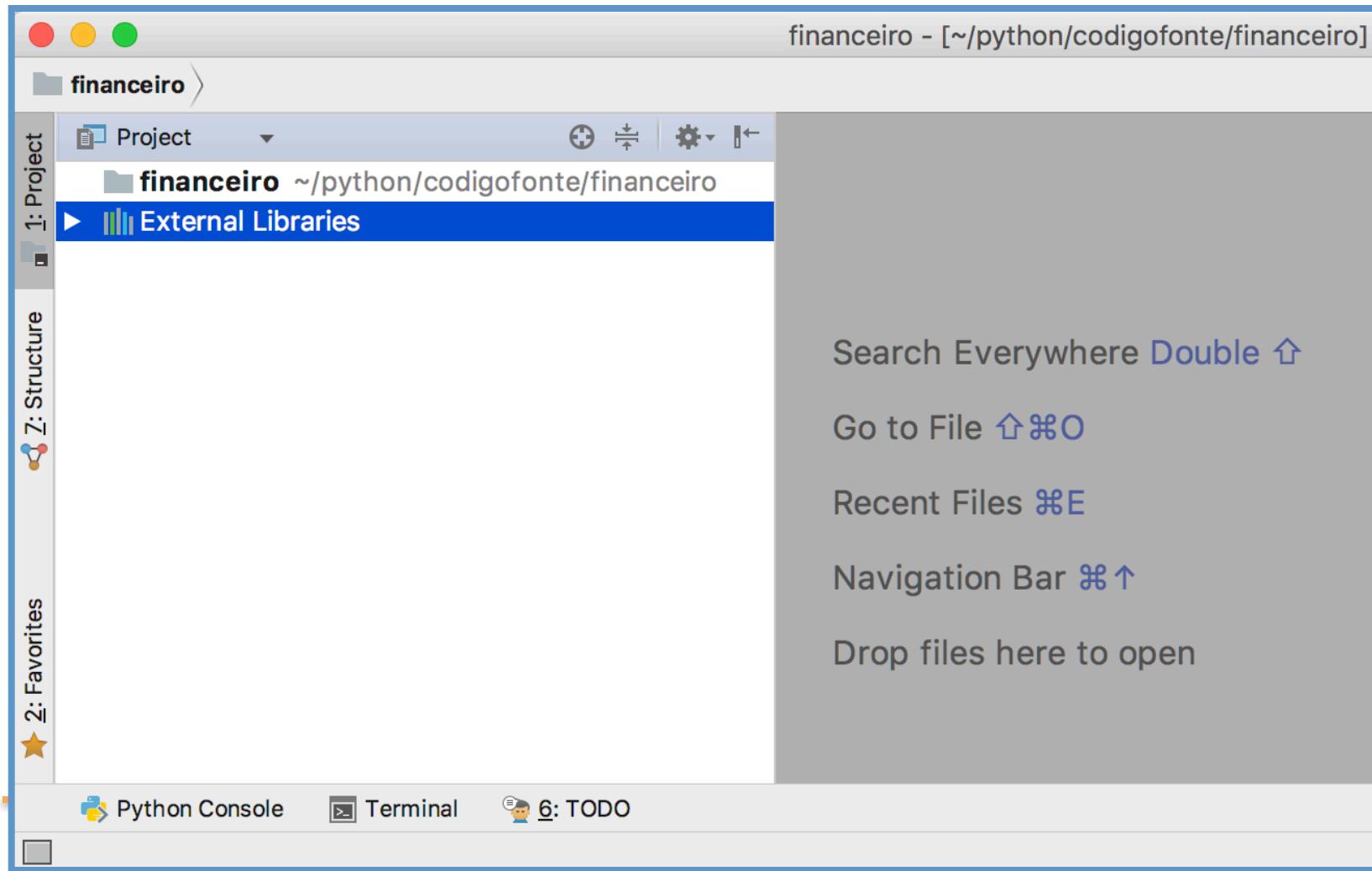
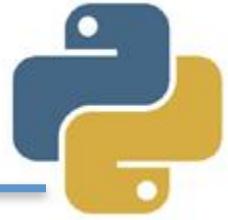
Dados do projeto - financeiro



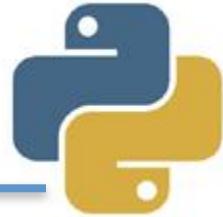
- **Location:** diretório para armazenamento dos arquivos do projeto
- **Interpreter:** Versão do python
- Depois de configurar, clique em **Create**



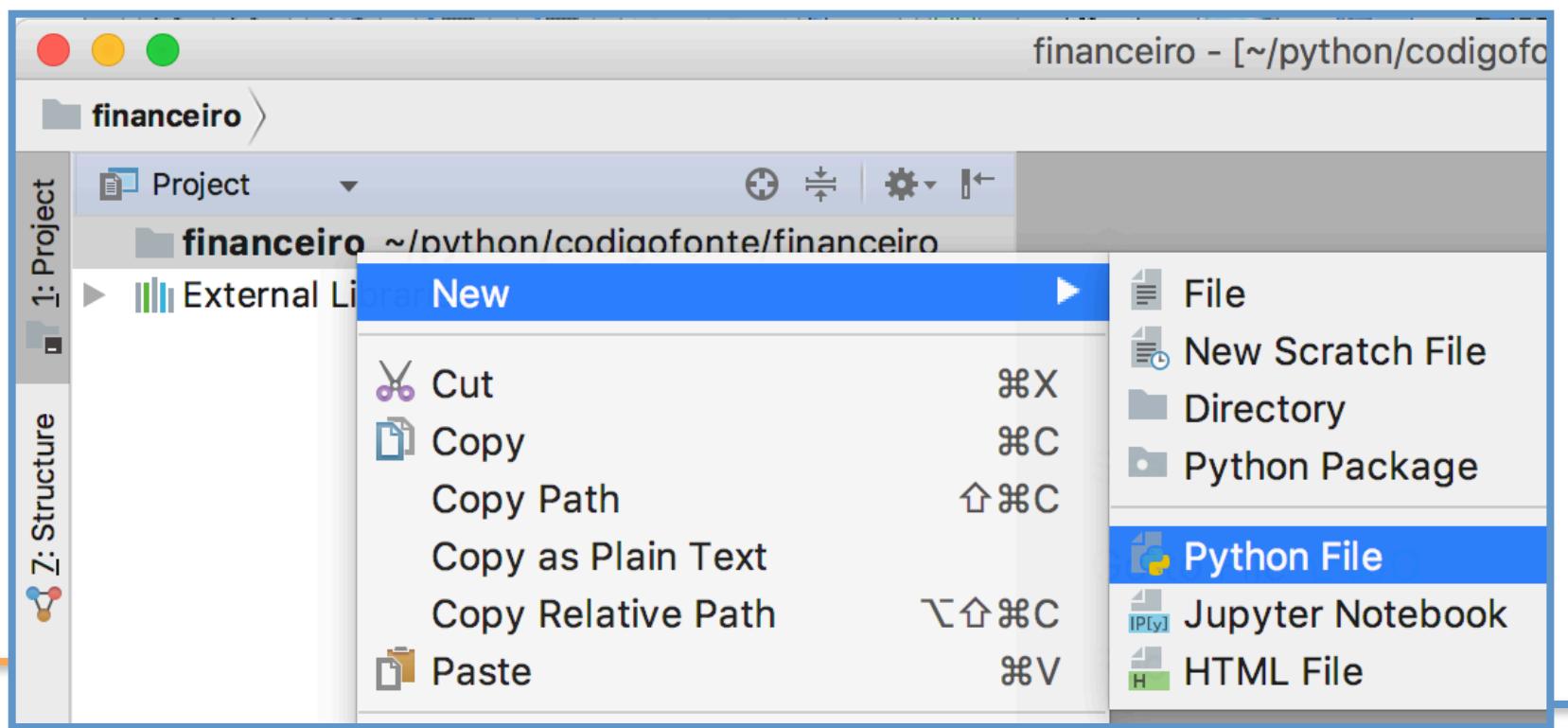
Bem vindo(a)!



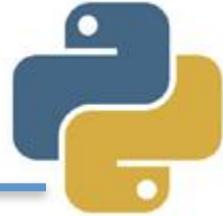
Primeiro arquivo



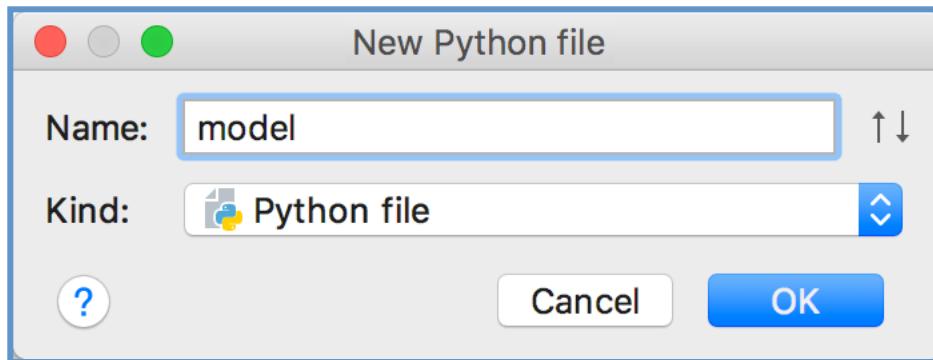
- Clique com o **botão direito** no projeto e selecione **New / Python File**



O arquivo `model.py`

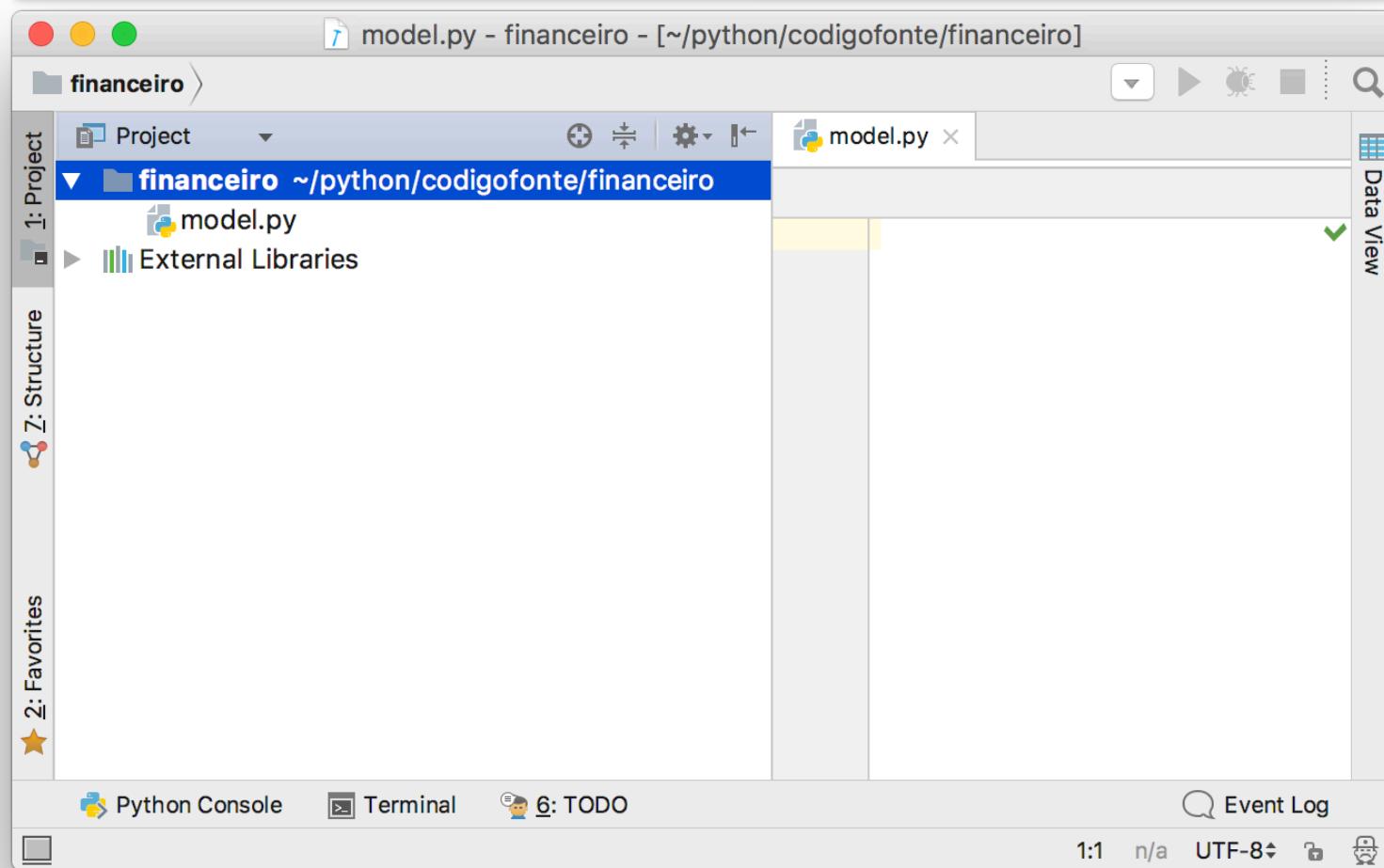
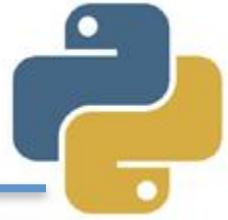


- Novo arquivo: `model.py`

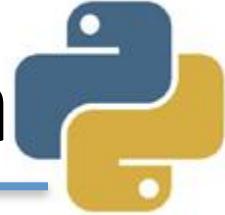


- Em python, é comum que as classes que representam entidades fiquem em um arquivo chamado `model.py`
-

Depois de tudo feito

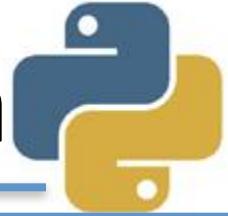


Exercício 01 – Conta bancária



- Agora que nossa IDE está instalada e que criamos o nosso projeto financeiro, chegou a hora de transformar o diagrama de classes em código fonte
 - Abra o arquivo model.py e crie a classe ContaBancaria
-

Exercício 01 – Conta bancária

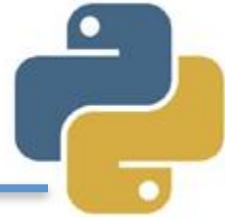


model.py ×

ContaBancaria

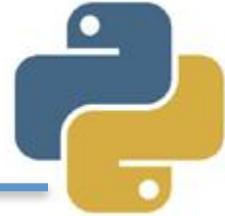
```
1 # -*- coding: UTF-8 -*-
2
3 class ContaBancaria(object):
4     def __init__(self):
5         self.agencia = None
6         self.numero = None
7         self.nome_cliente = None
8         self.saldo = 0.0
9
```

Por dentro do código



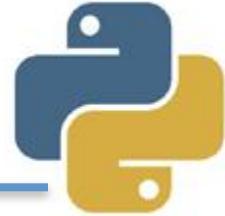
- Acabamos de criar nossa **1^a classe** Python
 - Por ora, precisamos saber apenas que:
 - **class** – indica a criação de uma classe
 - **__init__** – método para inicialização de objeto
 - **self** – objeto recebido como parâmetro
 - **self.variavel** – declaração de atributos
-

Criação e uso de Objetos



- Agora que definimos nossa classe, podemos **criar nossos Objetos** do tipo **ContaBancaria**, usando a **sintaxe**:
 - **novo_objeto = ContaBanaria()**
 - Para acesso aos **atributos** da conta, podemos usar o operador “.”(**ponto**)
-

Exercício 02 – Usando a conta



- Crie um novo arquivo `teste_conta.py`
 - Para usar a definição de conta bancária, precisamos `importar` o arquivo `model.py`
 - Crie um novo objeto
 - Informe os `dados do objeto`
 - Imprima os valores informados
 - Execute o arquivo `teste_conta.py`
-

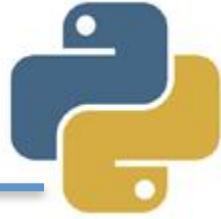
Exercício 02 – Usando a conta



```
model.py x teste_conta.py x

1 # -*- coding: UTF-8 -*-
2
3 from model import ContaBancaria
4
5 conta = ContaBancaria()
6
7 conta.agencia = "0233"
8 conta.numero = "1234-5"
9 conta.nome_cliente = "Maria Jose"
10 conta.saldo = 1500.0
11
12 print "Cliente " + conta.nome_cliente
13 print "Agência %s e Conta %s" % (conta.agencia, conta.numero)
14 print "Saldo "+str(conta.saldo)
15
```

Exercício 02 – Usando a conta



```
model.py x teste_conta.py x

1 # -*- coding: UTF-8 -*-
2
3 from model import ContaBancaria
4
5 conta = ContaBancaria()
6
7 conta.agencia =
8 conta.numero = "1234-5"
9 conta.nome_cliente = "Maria Jose"
10 conta.saldo = 1500.0
11
12 print "Cliente " + conta.nome_cliente
13 print "Agência %s e Conta %s" % (conta.agencia, conta.numero)
14 print "Saldo "+str(conta.saldo)
15
```

A red rectangular box highlights the line "# -*- coding: UTF-8 -*-". A blue bracket-like shape originates from the start of the line "print "Agência %s e Conta %s" % (conta.agencia, conta.numero)" and points towards the text "Definição de encoding" which is enclosed in a blue rounded rectangle.

Definição de encoding

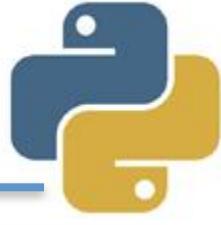
Exercício 02 – Usando a conta



```
model.py x teste_conta.py x
1 # -*- coding: UTF-8 -*-
2
3 from model import ContaBancaria
4
5 conta = ContaBancari
6
7 conta.agencia =
8 conta.numero =
9 conta.nome_cliente
10 conta.saldo = 1500.0
11
12 print "Cliente " + conta.nome_cliente
13 print "Agência %s e Conta %s" % (conta.agencia, conta.numero)
14 print "Saldo "+str(conta.saldo)
15
```

A red rectangular box highlights the line `from model import ContaBancaria`. A blue callout bubble points to this box with the text **Importação da classe ContaBancaria**.

Exercício 02 – Usando a conta

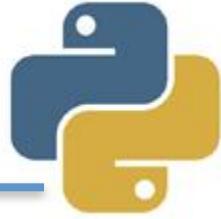


```
model.py x teste_conta.py x

1 # -*- coding: UTF-8 -*-
2
3 from model import ContaBancaria
4
5 conta = ContaBancaria()
6
7 conta.agencia = "02"
8 conta.numero = "123"
9 conta.nome_cliente
10 conta.saldo = 1500
11
12 print "Cliente "+conta.nome_cliente
13 print "Agência %s e Conta %s" % (conta.agencia, conta.numero)
14 print "Saldo "+str(conta.saldo)
15
```

A red rectangular callout box highlights the line of code `conta = ContaBancaria()`. A blue line connects this box to a blue rounded rectangular callout box containing the text **Criação do objeto ContaBancaria**.

Exercício 02 – Usando a conta

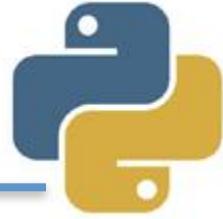


```
model.py  teste_conta.py
```

```
1 # -*- coding: UTF-8 -*-
2
3 from model import ContaBancaria
4
5 conta = ContaBancaria()
6
7 conta.agencia = "0233"
8 conta.numero = "1234-5"
9 conta.nome_cliente = "Maria Jose"
10 conta.saldo = 1500.0
11
12 print "Cliente " + conta.nome_cliente
13 print "Agência %s e Conta %s" % (conta.agencia, conta.numero)
14 print "Saldo "+str(conta.saldo)
15
```

Atualização de dados da instância

Exercício 02 – Usando a conta

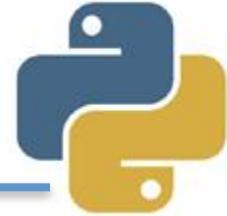


```
model.py  teste_conta.py
```

```
1 # -*- coding: UTF-8 -*-
2
3 from model import ContaBancaria
4
5 conta = ContaBancaria()
6
7 conta.agencia = "0233"
8 conta.numero = "1234-5"
9 conta.nome_cliente = "Maria"
10 conta.saldo = 1500.0
11
12 print "Cliente " + conta.nome_cliente
13 print "Agência %s e Conta %s" % (conta.agencia, conta.numero)
14 print "Saldo "+str(conta.saldo)
```

Acesso a dados da instância

Executando o script de teste



- Diferente do JAVA, não precisamos de uma classe para realizar os testes
 - Criamos um **script python** simples, para a execução dos testes
 - Clique com o botão direito no arquivo **teste_conta.py**
 - Selecione a opção **Run teste_conta**
-

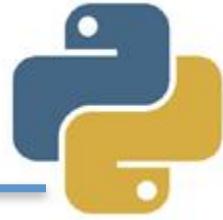
Executando teste_conta.py



```
model.py x teste_conta.py x

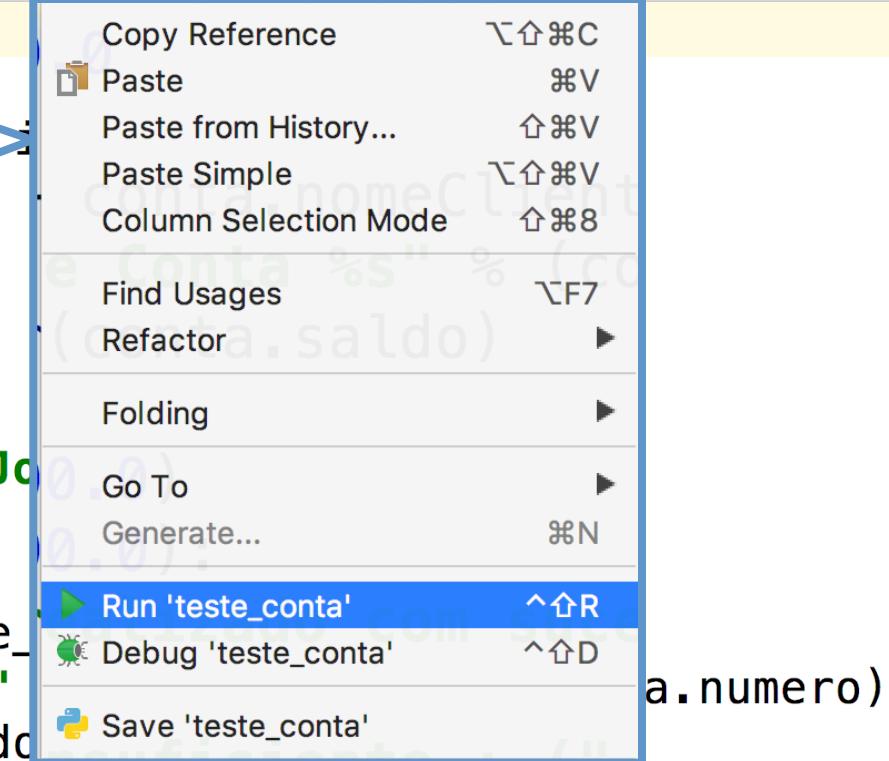
1 # -*- coding: UTF-8 -*-
2
3 from model import ContaBancaria
4
5 conta = ContaBancaria()
6
7 conta.agencia = "0233"
8 conta.numero = "1234-5"
9 conta.nome_cliente = "Maria Jose"
10 conta.saldo = 1500.0
11
12 print "Cliente " + conta.nome_cliente
13 print "Agência %s e Conta %s" % (conta.agencia, conta.numero)
14 print "Saldo "+str(conta.saldo)
15
```

Executando teste_conta.py

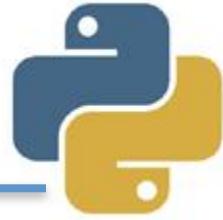


Clique com o botão
direito no arquivo

```
1 conta = ContaBancaria()
2
3 conta.agencia = "0233"
4 conta.numero = "1234-5"
5 conta.nome_cliente = "Maria Jo"
6 conta.saldo = 1500.0
7
8 print "Cliente " + conta.nome_
9 print "Agência %s e Conta %s"
10 print "Saldo "+str(conta.saldo)
11
12
13
14
15
```



Executando teste_conta.py



Clique com o **botão direito** no arquivo

```
1 conta = ContaBancaria()
2
3 conta.agencia = "0233"
4 conta.
5 conta.
6 conta.
7 conta.
8
9
10 print
11 print
12 print "Saldo "+str(conta.saldo)
13
14
15
```

Seleciona a
opção **Run**

- Copy Reference ⌘C
- Paste ⌘V
- Paste from History... ⌘V
- Paste Simple ⌘V
- Column Selection Mode ⌘8
- Find Usages ⌘F7
- Refactor ►
- Folding ►
- Go To ►
- Generate... ⌘N
- ▶ Run 'teste_conta' ⌘R
- ▶ Debug 'teste_conta' ⌘D
- ▶ Save 'teste_conta'



Resultado no Console

The screenshot shows the PyCharm IDE interface with the following details:

- Project:** financeiro (~/python/codigoforum)
- Files:** model.py, teste_cliente.py, teste_conta.py (selected)
- External Libraries:** None
- Code in teste_conta.py:**

```
7 conta.agencia = "0233"
8 conta.numero = "1234-5"
9 conta.nome_cliente = "Maria Jose"
10 conta.saldo = 1500.0
11
12 print "Cliente " + conta.nome_cliente
13 print "Agência %s e Conta %s" % (conta,
14 print "Saldo "+str(conta.saldo))
```
- Run Output:**

```
/Library/Frameworks/Python.framework/Versions/2.7/bin/python /Users/luiz/PycharmProjects/financeiro/teste_conta.py
Cliente Maria Jose
Agência 0233 e Conta 1234-5
Saldo 1500.0

Process finished with exit code 0
```



Resultado no Console

The screenshot shows the PyCharm IDE interface. The project navigation bar at the top lists 'Project' and files 'model.py', 'teste_cliente.py', and 'teste_conta.py'. The file 'teste_conta.py' is currently selected and highlighted in blue. The code editor displays the following Python script:

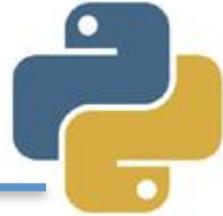
```
7 conta.agencia = "0233"
8 conta.numero = "1234-5"
9 conta.nome_cliente = "Maria Jose"
10 conta.saldo = 1500.0
11
12 print "Cliente " + conta.nome_cliente
13 print "Agência %s e Conta %s" % (conta,
14 print "Saldo "+str(conta.saldo))
```

The 'Run' toolbar at the bottom has a 'Run' button and the text 'teste_conta'. The run output window shows the execution results:

```
/Library/Frameworks/Python.framework/Versions/2.7/bin/python /Applications/PyCharm.app/Contents/helpers/pydev/_pydev_bundle/pydev_ipython_console.py
Client Maria Jose
Agência 0233 e Conta 1234-5
Saldo 1500.0
```

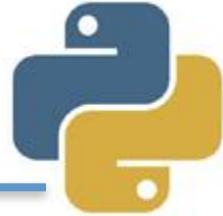
A red rectangular box highlights the output text 'Client Maria Jose', 'Agência 0233 e Conta 1234-5', and 'Saldo 1500.0'. At the bottom of the run output, the message 'Process finished with exit code 0' is displayed.

Métodos

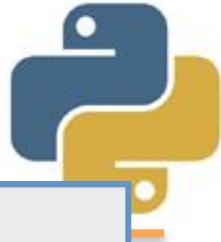


- Representam **ações** que objetos podem realizar **sobre** seus **atributos**
 - São equivalentes às **funções** e **procedimentos** da programação estruturada
 - Costumam ser definidos **após** a **declaração** de atributos
-

Métodos

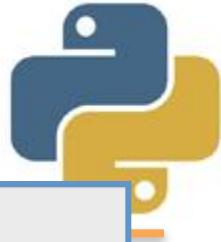


- Vamos pensar nos **métodos** da nossa Conta Bancária
 - Qual é a **lógica** necessária para a realização de **depósito** e **saque** da conta?
 - Altere a classe **ContaBancaria**, do arquivo **model.py**, e implemente os métodos **sacar(valor)** e **depositar(valor)**
-



Exercício 03 – depositar()

```
model.py x teste_conta.py x
ContaBancaria
1 # -*- coding: UTF-8 -*-
2
3 class ContaBancaria(object):
4     def __init__(self):
5         self.agencia = None
6         self.numero = None
7         self.nomeCliente = None
8         self.saldo = 0.0
9
10    def depositar(self, valor):
11        self.saldo += valor
```



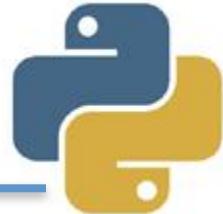
Exercício 03 – depositar()

```
model.py x teste_conta.py x
```

ContaBancaria

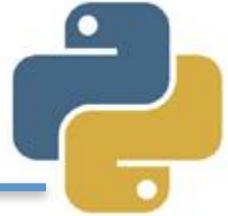
```
1 # -*- coding: UTF-8 -*-
2
3 class ContaBancaria(object):
4     def __init__(self):
5         self.agencia = None
6         self.numero = None
7         self.nomeCliente = None
8         self.saldo = 0.0
9
10    def depositar(self, valor):
11        self.saldo += valor
```

Exercício 04 – sacar()



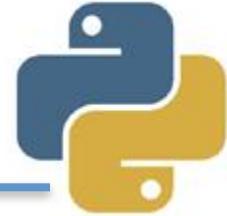
```
10     def depositar(self, valor):  
11         self.saldo += valor  
12  
13     def sacar(self, valor):  
14         if valor <= self.saldo:  
15             self.saldo -= valor  
16             return True  
17         return False  
18
```

Exercício 04 – sacar()



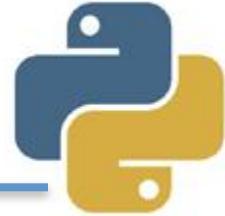
```
10     def depositar(self, valor):  
11         self.saldo += valor  
12  
13     def sacar(self, valor):  
14         if valor <= self.saldo:  
15             self.saldo -= valor  
16             return True  
17         return False  
18
```

Novos métodos, novos teste



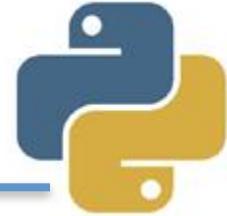
- Vamos **testar** nossos **novos métodos**?
 - Atualize o arquivo **teste_conta.py**
 - Faça um **depósito** de R\$ 500,00
 - Tente realizar um **saque** de R\$ 2.500,00
 - Exiba mensagens de **feedback** para o usuário, informando o **resultado** da operação de saque
-

Exercício 05: teste de métodos



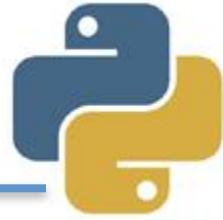
```
11  
12     print "Cliente " + conta.nomeCliente  
13     print "Agência %s e Conta %s" % (conta.agencia,  
14     print "Saldo "+str(conta.saldo)  
15  
16 #Teste dos metodos depositar() e sacar()  
17 conta.depositar(500.0)  
18 if conta.sacar(2500.0):  
19     print "Saque realizado com sucesso :-)"  
20 else:  
21     print "Saldo insuficiente :-(  
22
```

Exercício 05: teste de métodos



```
11  
12     print "Cliente " + conta.nomeCliente  
13     print "Agência %s e Conta %s" % (conta.agencia,  
14     print "Saldo "+str(conta.saldo)  
15  
16 #Teste dos metodos depositar() e sacar()  
17 conta.depositar(500.0)  
18 if conta.sacar(2500.0):  
19     print "Saque realizado com sucesso :-)"  
20 else:  
21     print "Saldo insuficiente :-(  
22
```

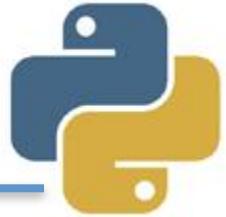
Resultado no console



```
16     #Teste dos métodos depositar() e sacar()
17     conta.depositar(500.0)
18     if conta.sacar(2500.0):
19         print "Saque realizado com sucesso :)"
20     else:
21         print "Saldo insuficiente :-("
22
```

Run teste_conta

```
/Library/Frameworks/Python.framework/Versions/
Cliente Maria Jose
Agência 0233 e Conta 1234-5
Saldo 1500.0
Saldo insuficiente :-(
```



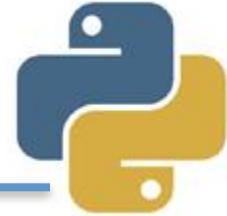
Resultado no console

```
16     #Teste dos métodos depositar() e sacar()
17     conta.depositar(500.0)
18     if conta.sacar(2500.0):
19         print "Saque realizado com sucesso :)"
20     else:
21         print "Saldo insuficiente :-("
22
```

Run teste_conta

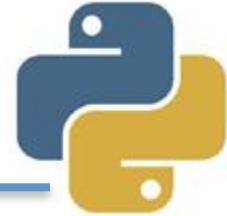
```
/Library/Frameworks/Python.framework/Versions/
Cliente Maria Jose
Agência 0233 e Conta 1234-5
Saldo 1500.0
Saldo insuficiente :-(
```

Trabalhando com referências



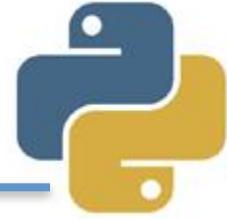
- A partir de uma instrução:
 - `conta` = ContaBancaria()
 - É comum escutarmos a afirmação:
 - “A variável `conta` é um `objeto`”
 - Contudo, a variável `conta` **NÃO** é um objeto.
 - A `conta` é uma **referência** para um objeto
-

Trabalhando com referências

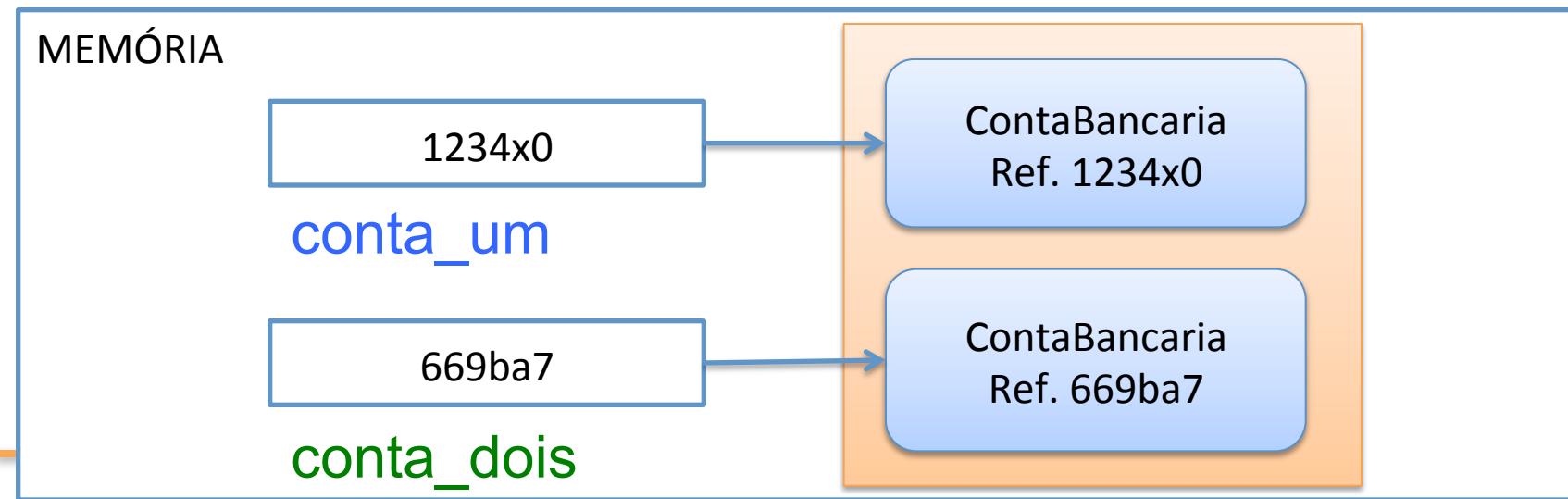


- Variáveis de **referência** guardam os **endereços de memória** onde foram criados os objetos
 - Em determinados pontos do sistema, é possível que **mais de uma variável** de referência aponte para um **determinado objeto**
-

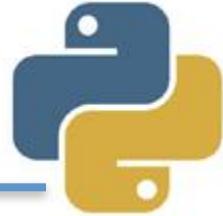
Trabalhando com referências



- Como fica a memória, após a execução de:
- `conta_um = ContaBancaria()`
- `conta_dois = ContaBancaria()`

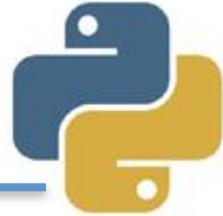


Um objeto, duas referências



- `conta_um = ContaBancaria()`
 - `conta_dois = ContaBancaria()`
 - `conta_tres = conta_um`
-

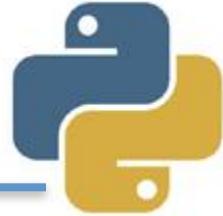
Um objeto, duas referências



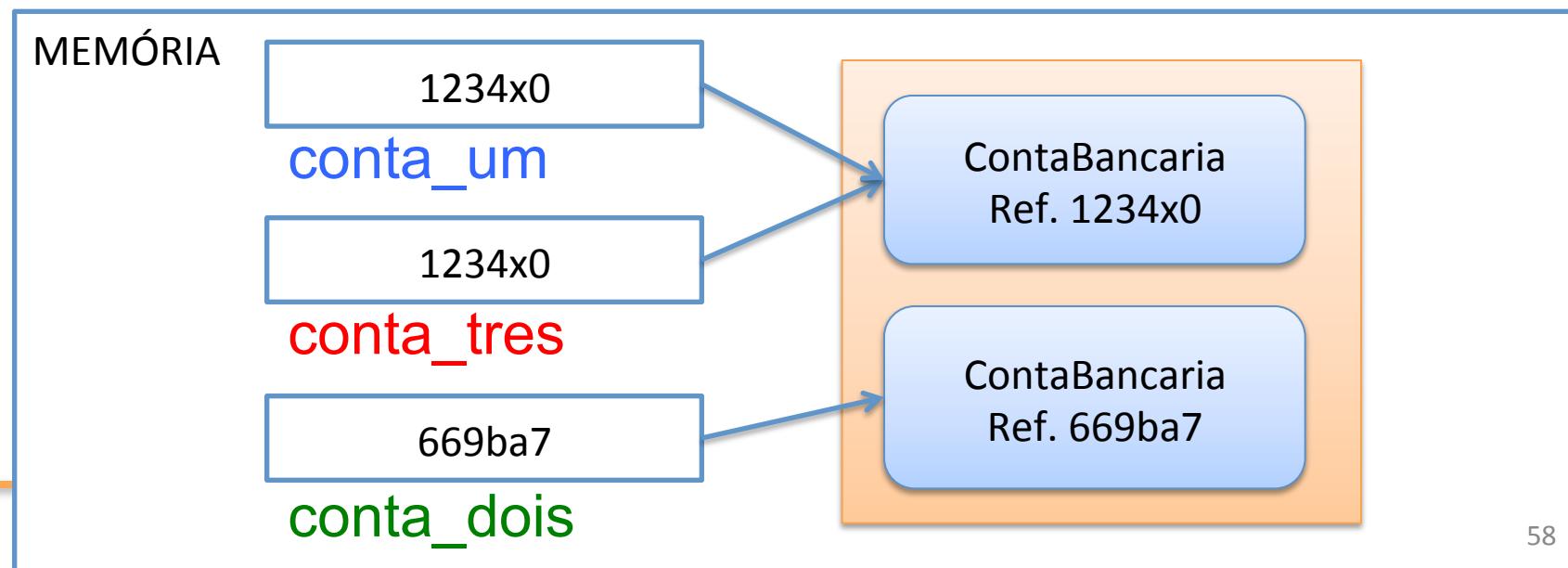
- `conta_um = ContaBancaria()`
- `conta_dois = ContaBancaria()`
- `conta_tres = conta_um`

A variável `conta_tres` é
criada a partir de
`conta_um`

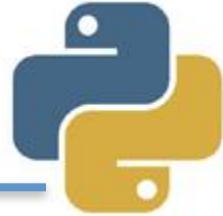
Um objeto, duas referências



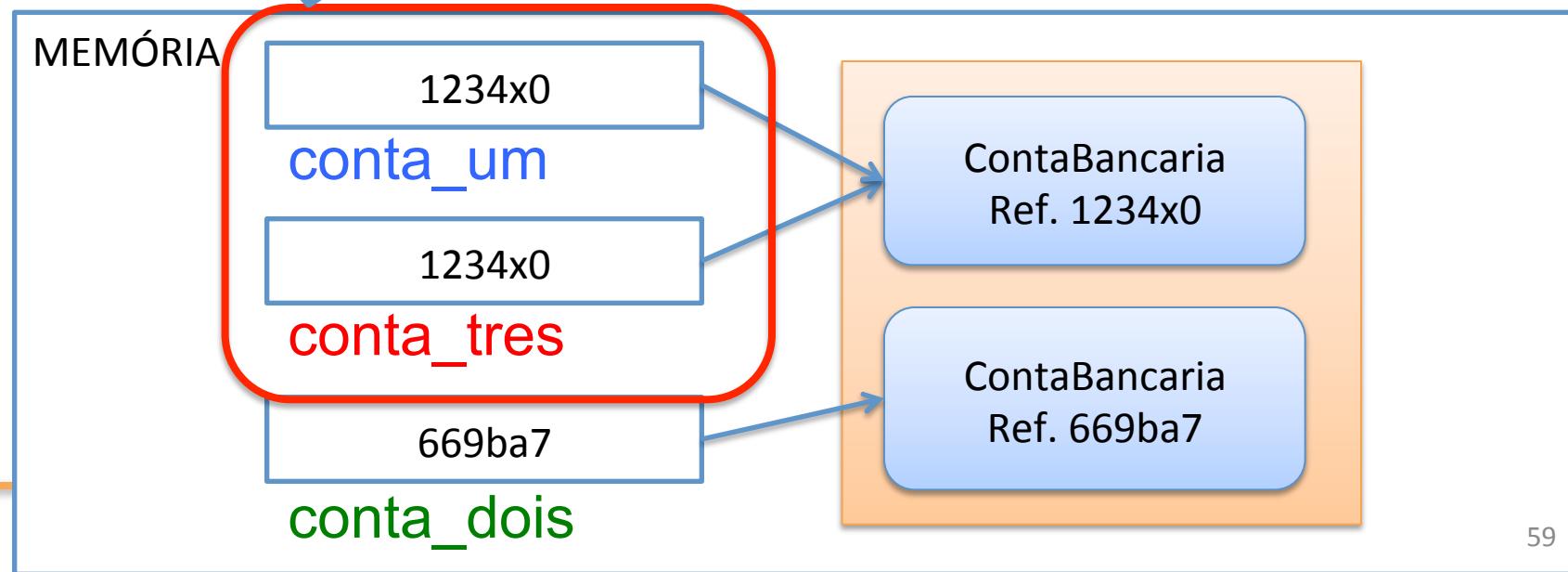
- `conta_um = ContaBancaria()`
- `conta_dois = ContaBancaria()`
- `conta_tres = conta_um`



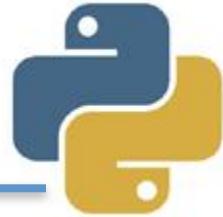
Um objeto, duas referências



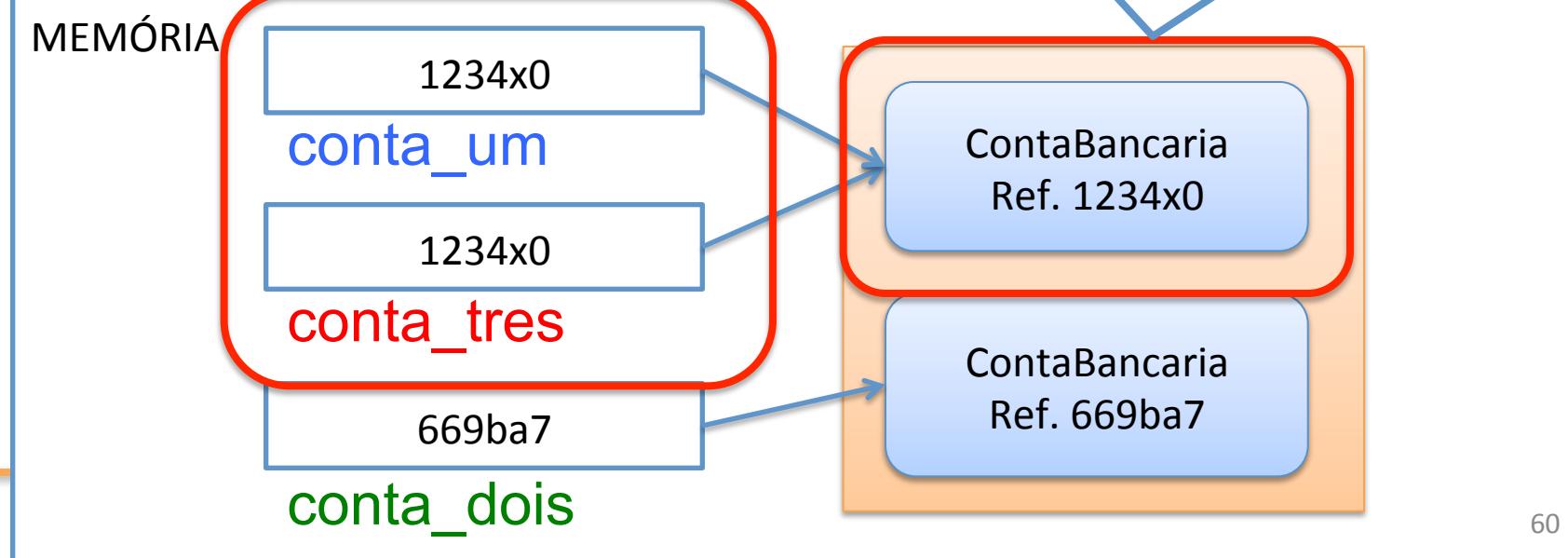
- `conta_um = ContaBancaria()`
- `co` Duas variáveis de `ContaBancaria()` referência...
- `conta_` ~~tres~~ `conta_um`



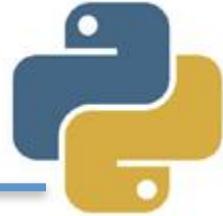
Um objeto, duas referências



- `conta_um = ContaBancaria()`
- `conta_dois = conta_um` Duas variáveis de referência...
- `conta_tres = conta_um` ...apontando para o mesmo objeto

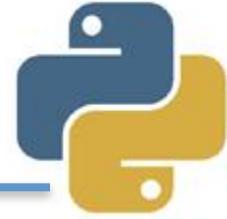


Transferência entre contas



- Agora, precisamos implementar o **método** para **transferência** de valores entre contas bancárias
 - Podemos assumir que o método transferir de uma conta vai receber o **valor a transferir** e a **conta de destino** da transferência
-

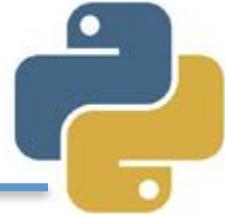
Exercício 06 – Novo método



- Altere a classe **ContaBancaria**

```
13      def sacar(self, valor):  
14          if valor <= self.saldo:  
15              self.saldo -= valor  
16              return True  
17          return False  
18  
19      def transferir(self, valor, destino):  
20          if self.sacar(valor):  
21              destino.depositar(valor)  
22              return True  
23          return False  
24
```

Exercício 06 – Novo método

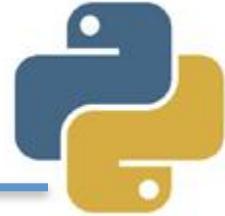


- Altere a classe **ContaBancaria**

```
13     def sacar(self, valor):
14         if valor <= self.saldo:
15             self.saldo -= valor
16             return True
17         return False
18
19     def transferir(self, valor, destino):
20         if self.sacar(valor):
21             destino.depositar(valor)
22             return True
23         return False
24
```

Na classe **ContaBancaria**, inclua o método **transferir()**

Que tal testar o método?



- Chegou a hora de realizarmos **transferências** entre contas bancárias
 - Volte ao arquivo **teste_conta.py**
 - Crie duas contas bancárias: **origem**, com **saldo inicial = 1200** e **destino**, sem saldo
 - **Transfira 500** da conta **origem** para **destino**
-

Exercício 7: transferindo

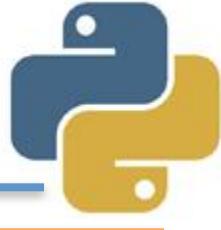


```
model.py x teste_conta.py x

23 #Teste do método de transferencia
24 origem = ContaBancaria()
25 origem.depositar(1200.0)
26 destino = ContaBancaria()
27 print '\nSaldo Inicial Origem: ', origem.saldo
28 print 'Saldo Inicial Destino: ', destino.saldo
29 print 'Transferência ======'
30 origem.transferir(500.0, destino)
31 print 'Saldo Origem: ', origem.saldo
32 print 'Saldo Destino: ', destino.saldo
```

Exercício 7:

Criação da conta de **origem**



```
model.py x teste_conta.py x
23 #Teste do método de transferencia
24 origem = ContaBancaria()
25 origem.depositar(1200.0)
26 destino = ContaBancaria()
27 print '\nSaldo Inicial Origem: ', origem.saldo
28 print 'Saldo Inicial Destino: ', destino.saldo
29 print 'Transferência ======'
30 origem.transferir(500.0, destino)
31 print 'Saldo Origem: ', origem.saldo
32 print 'Saldo Destino: ', destino.saldo
```

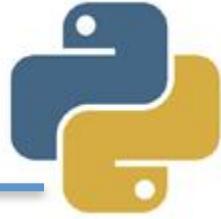


Exercício 7: transferindo

Criação da conta de **destino**

```
model.py x teste_conta.py x
23 #Teste do método
24 origem = ContaBancaria()
25 origem.depositar(1200)
26 destino = ContaBancaria()
27 print '\nSaldo Inicial Origem: ', origem.saldo
28 print 'Saldo Inicial Destino: ', destino.saldo
29 print 'Transferência ======'
30 origem.transferir(500.0, destino)
31 print 'Saldo Origem: ', origem.saldo
32 print 'Saldo Destino: ', destino.saldo
```

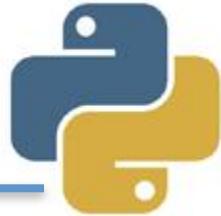
Exercício 7: transferindo



```
model.py x teste_conta.py x

23 #Teste do método
24 origem = ContaBancaria()
25 origem.depositar(500.0)
26 destino = ContaBancaria()
27 print '\nSaldo Inicial Origem: ', origem.saldo
28 print 'Saldo Inicial Destino: ', destino.saldo
29 print 'Transferência =-----'
30 origem.transferir(500.0, destino)
31 print 'Saldo Origem: ', origem.saldo
32 print 'Saldo Destino: ', destino.saldo
```

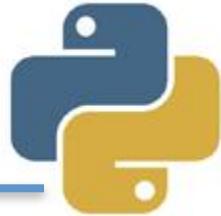
Realização da transferência



Resultado no console

Run teste_conta

```
/Library/Frameworks/Python.framework/  
Cliente Maria Jose  
Agência 0233 e Conta 1234-5  
Saldo 1500.0  
Saldo insuficiente :-(  
  
Saldo Inicial Origem: 1200.0  
Saldo Inicial Destino: 0.0  
Transferência ======  
Saldo Origem: 700.0  
Saldo Destino: 500.0
```

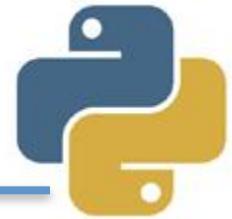


Resultado no console

Run teste_conta

```
/Library/Frameworks/Python.framework/  
Cliente Maria Jose  
Agência 0233 e Conta 1234-5  
Saldo 1500.0  
Saldo insuficiente :-(
```

```
Saldo Inicial Origem: 1200.0  
Saldo Inicial Destino: 0.0  
Transferência =====  
Saldo Origem: 700.0  
Saldo Destino: 500.0
```

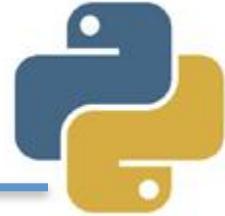


Difícil?

É Naadaaaa

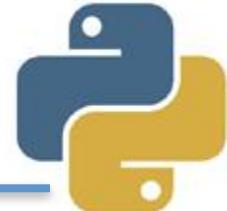


Refletindo a respeito...



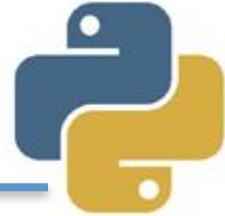
- O que aconteceu com a **conta** que foi passada como **parâmetro** para o método **transferir**?
 - O objeto foi **clonado** ?
 - **Negativo** 😞
 - Passamos a **referência** para a conta de destino. 😊
-

E o tempo passa...



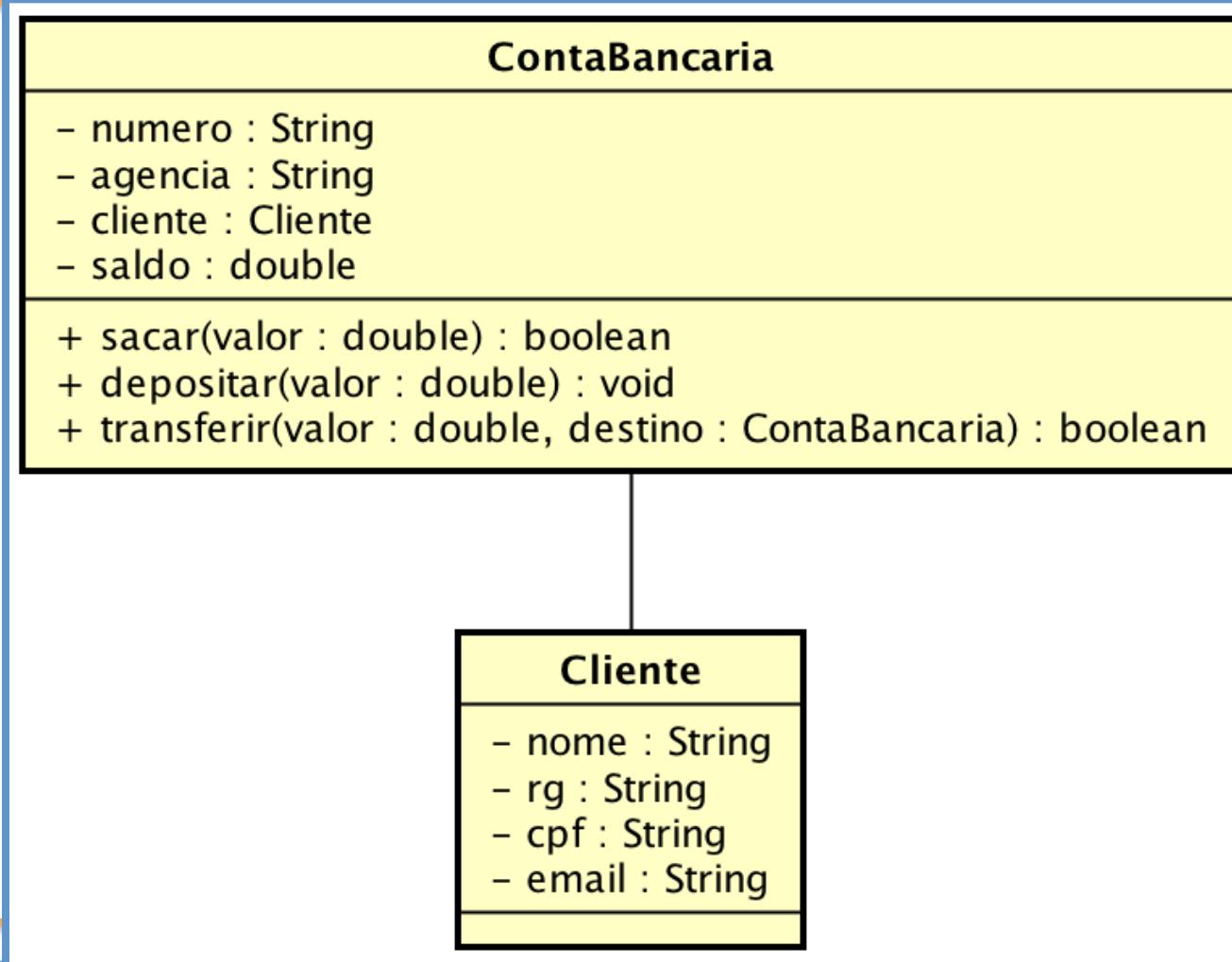
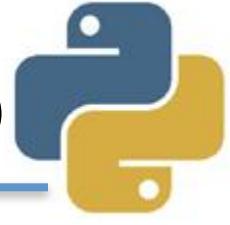
- O tempo passa e surge a necessidade de armazenamento de **novos dados**
 - Agora, precisamos armazenar **RG**, **CPF** e **e-mail** do **cliente**
 - Vamos criar novos atributos na classe **ContaBancaria**?
 - **Negativo** 😞
-

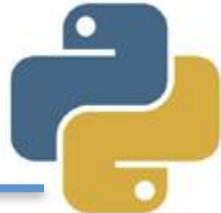
Preocupados com a coesão



- Em OO, uma classe possui **alta coesão** quando tem um **papel bem definido**
 - Os novos dados **não pertencem** à conta bancária.
 - Os novos dados pertencem ao **cliente**
 - Vamos atualizar a **modelagem**
-

Diagrama de classes atualizado





Atualização do código

- Precisamos implementar duas atualizações
 - No arquivo `model.py`, crie a classe `Cliente`, abaixo do método `transferir()` da classe `ContaBancaria`
 - Na `ContaBancaria`, altere o atributo `nome_cliente` para `cliente`
-



Exercício 08: a classe Cliente

```
model.py x
Cliente
19     def transferir(self, valor, destino):
20         if self.sacar(valor):
21             destino.depositar(valor)
22             return True
23         return False
24
25 class Cliente (object):
26     def __init__(self):
27         self.nome = None
28         self.cpf = None
29         self.rg = None
30         self.email = None
31
```

Exercício 08: a classe Cliente



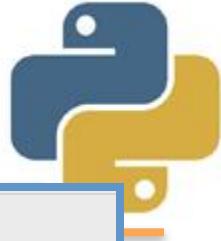
model.py x

Cliente

```
19     def transferir(self, destino):
20         if self.saldo < 10:
21             destino.saldo += self.saldo
22             self.saldo = 0
23             return False
24
25 class Cliente(object):
26     def __init__(self):
27         self.nome = None
28         self.cpf = None
29         self.rg = None
30         self.email = None
```

A nova classe
Cliente

Exercício 09: a classe Conta



```
model.py x
ContaBancaria __init__()
1 # -*- coding: UTF-8 -*-
2
3 class ContaBancaria(object):
4     def __init__(self):
5         self.agencia = None
6         self.numero = None
7         self.cliente = None
8         self.saldo = 0.0
9
10    def depositar(self, valor):
11        self.saldo += valor
12
```

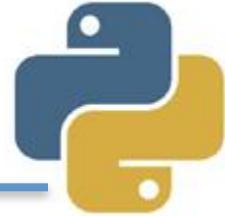
Exercício 09: a classe Conta



```
model.py x
ContaBancaria      init  ()
1   # -*- coding: utf-8 -*-
2
3   class ContaBancaria:
4       def __init__(self):
5           self.agencia = None
6           self.numero = None
7           self.cliente = None
8           self.saldo = 0.0
9
10      def depositar(self, valor):
11          self.saldo += valor
12
```

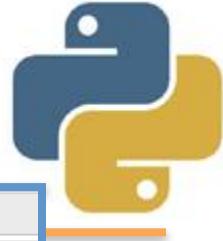
Altere o atributo
nome_cliente
para cliente

Teste da classe Cliente



- Vamos testar a nossa classe `Cliente`
 - Crie o novo arquivo `teste_cliente.py`
 - Importe `ContaBancaria` e `Cliente`
 - Instancie um cliente e uma conta.
 - Atribua o cliente à `conta` criada
 - Imprima os dados do cliente e da conta
-

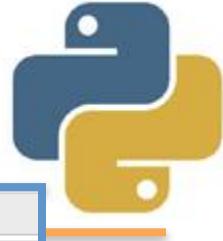
Exercício 10: teste do Cliente



```
teste_cliente.py x

1 # -*- coding: UTF-8 -*-
2 from model import ContaBancaria
3 from model import Cliente
4
5 cliente = Cliente()
6 cliente.nome = "Luiza"
7 cliente.cpf = "123456"
8 cliente.email = "luiza@gmail.com"
9
10 conta = ContaBancaria()
11 conta.saldo = 1000.0
12 conta.cliente = cliente
13
14 print "Saldo: ", conta.saldo
15 print "Cliente.cpf: ", cliente.cpf
16 print "Nome: ", conta.cliente.nome
17 print "E-mail: ", conta.cliente.email
```

Exercício 10: teste do Cliente

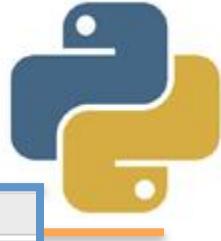


```
teste_cliente.py x

1 # -*- coding: UTF-8 -*-
2 from model import ContaBancaria
3 from model import Cliente
4
5 cliente = Cliente()
6 cliente.nome =
7 cliente.cpf =
8 cliente.email =
9
10 conta = ContaBa
11 conta.saldo = 1
12 conta.cliente = cliente
13
14 print "Saldo: ", conta.saldo
15 print "Cliente.cpf: ", cliente.cpf
16 print "Nome: ", conta.cliente.nome
17 print "E-mail: ", conta.cliente.email
```

Encoding e
Importação de
classes

Exercício 10: teste do Cliente

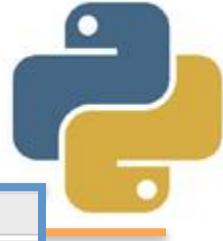


```
teste_cliente.py x

1 # -*- coding: UTF-8 -*-
2 from model import ContaBancaria
3 from model import Cliente
4
5 cliente = Cliente()
6 cliente.nome = "Luiza"
7 cliente.cpf = "123456"
8 cliente.email = "luiza@gmail.com"
9
10 conta = ContaBancaria()
11 conta.saldo = 1000
12 conta.cliente =
13
14 print "Saldo: "
15 print "Cliente: "
16 print "Nome: "
17 print "E-mail: ", conta.cliente.email
```

Criação do objeto
cliente

Exercício 10: teste do Cliente

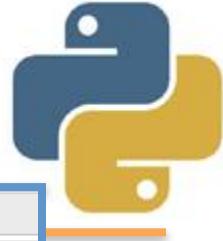


```
teste_cliente.py x

1 # -*- coding: UTF-8 -*-
2 from m
3 from
4
5 clien
6 clien
7 client
8 cliente.emai
9
10 conta = ContaBancaria()
11 conta.saldo = 1000.0
12 conta.cliente = cliente
13
14 print "Saldo: ", conta.saldo
15 print "Cliente.cpf: ", cliente.cpf
16 print "Nome: ", conta.cliente.nome
17 print "E-mail: ", conta.cliente.email
```

Criação do objeto
conta bancária

Exercício 10: teste do Cliente



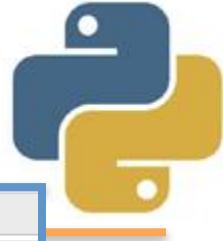
```
teste_cliente.py x

1 # -*- coding: UTF-8 -*-
2 from model import ContaBancaria
3 from model import Cliente
4
5 cliente = Cliente("João", "1234567890123456", "johndoe@example.com")
6 conta = ContaBancaria()
7 conta.saldo = 1000.0
8 conta.cliente = cliente
9
10 print "Saldo: ", conta.saldo
11 print "Cliente.cpf: ", cliente.cpf
12 print "Nome: ", conta.cliente.nome
13 print "E-mail: ", conta.cliente.email
```

Associação do cliente à conta

om"

Exercício 10: teste do Cliente

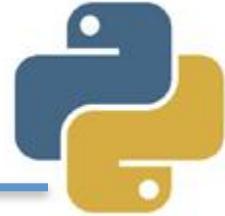


```
teste_cliente.py x

1 # -*- coding: UTF-8 -*-
2 from model import ContaBancaria
3 from model import Cliente
4
5 cliente = Cliente()
6 cliente.nome = "Luiza"
7 cliente.cpf = "123456"
8 clien
9 conta
10 conta
11 conta.cliente
12 conta.cliente
13
14 print "Saldo: ", conta.saldo
15 print "Cliente.cpf: ", cliente.cpf
16 print "Nome: ", conta.cliente.nome
17 print "E-mail: ", conta.cliente.email
```

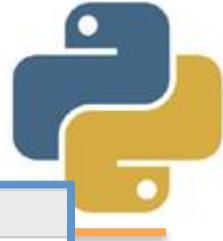
Impressão dos dados

Depois das alterações



- O script `teste_cliente.py` funcionou bem
 - Mas e o script `teste_conta.py`, que usa o atributo `conta.nome_cliente`, que foi alterado para `conta.cliente`?
 - Execute novamente `teste_conta.py`.
 - E aí, o que aconteceu?
 - Por que funcionou ?
-

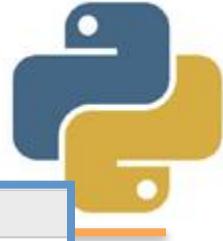
Revendo o código



```
model.py × teste_conta.py × teste_cliente.py ×

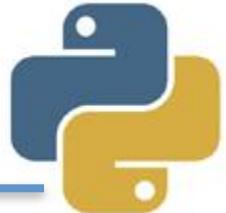
1 # -*- coding: UTF-8 -*-
2
3 from model import ContaBancaria
4
5 conta = ContaBancaria()
6
7 conta.agencia = "0233"
8 conta.numero = "1234-5"
9 conta.nome_cliente = "Maria Jose"
10 conta.saldo = 1500.0
11
12 print "Cliente " + conta.nome_cliente
13 print "Agência %s e Conta %s" % (conta.agencia,
14 print "Saldo "+str(conta.saldo)
15
```

Revendo o código

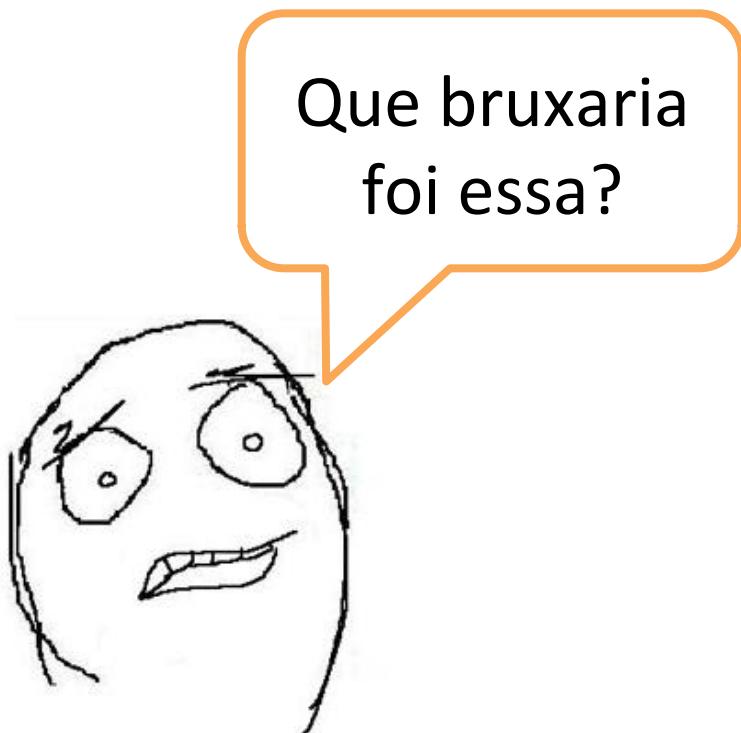


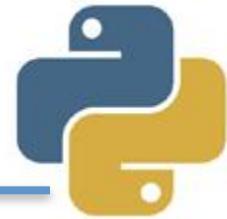
```
model.py x teste_conta.py x teste_cliente.py x
1  #
2  f
3  co
4
5
6
7  conta.agencia = "0233"
8  conta.numero_agencia = "1234-5"
9  conta.nome_cliente = "Maria Jose"
10 conta.saldo = 1500.0
11
12 print "Cliente " + conta.nome_cliente
13 print "Agência %s e Conta %s" % (conta.agencia,
14 print "Saldo "+str(conta.saldo)
15
```

O Script **rodou**, mesmo usando
um **atributo** que **não existe** na
classe ContaBancaria

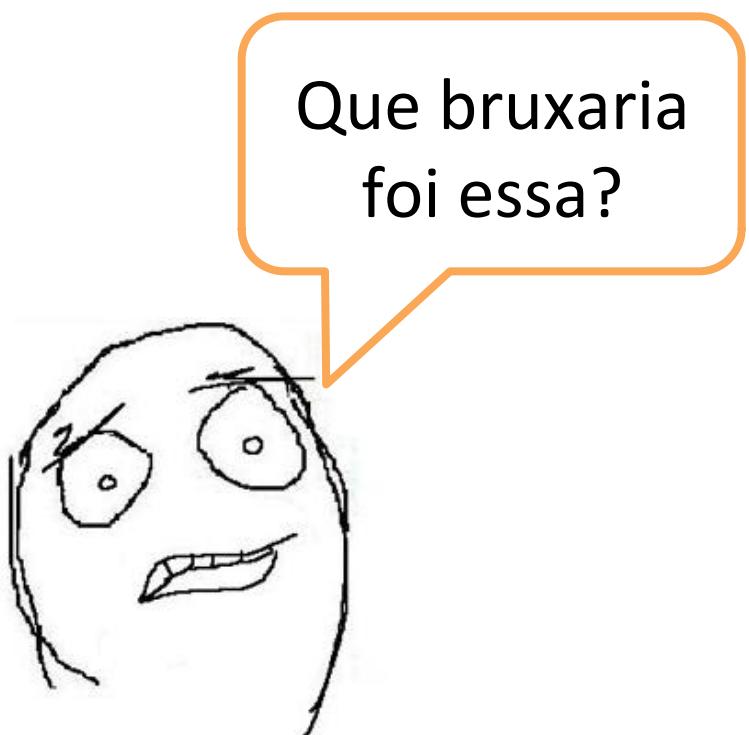


E agora?

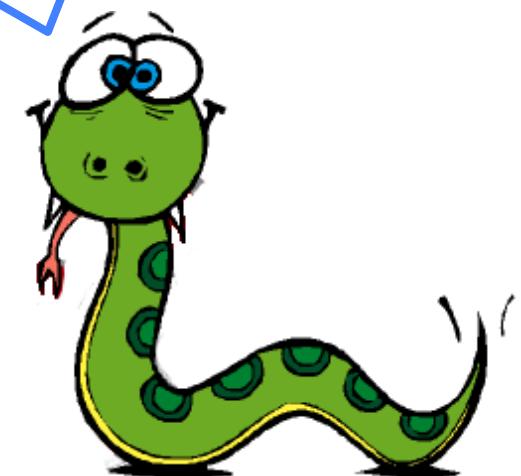


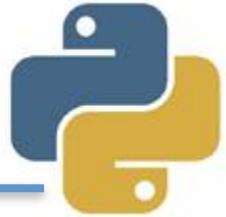


E agora?

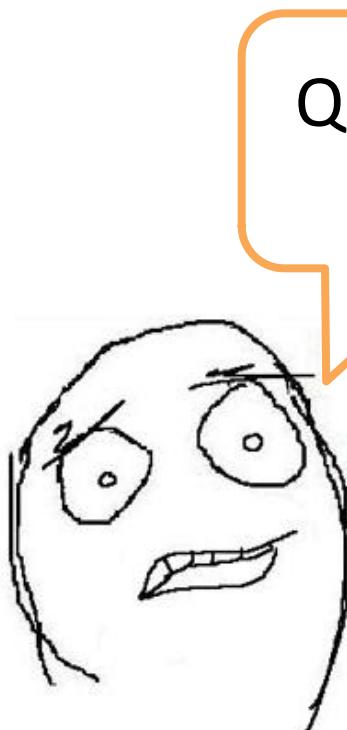


Uma bruxaria
chamada **Dinamismo**





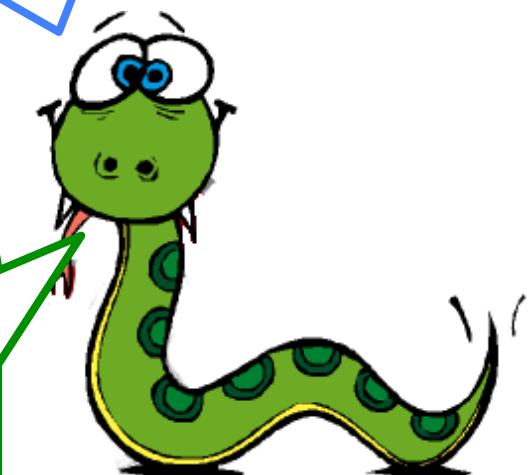
E agora?



Que bruxaria
foi essa?

Uma bruxaria
chamada **Dinamismo**

Em python, nós podemos
adicionar atributos e
métodos em **tempo de**
execução





Imagine o código a seguir

```
Python Console
/Library/Frameworks/Python.framework/Versions/2.7/bin/python2.7
Python 2.7.11 (v2.7.11:6d1b6a68f775, Dec  5 2015, 12:54:16)
In[2]: class Pessoa (object):
...:     def __init__(self, nome):
...:         self.nome = nome
...:
+ In[3]: p1 = Pessoa("Joao")
In[4]: p1.idade = 35
In[5]: print p1.nome, p1.idade
Joao 35
In[6]: p2 = Pessoa("Maria")
In[7]: print p2.nome, p2.idade
Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versions/2.7/lib/
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "<ipython-input-7-5532aa554d8b>", line 1, in <module>
    print p2.nome, p2.idade
AttributeError: 'Pessoa' object has no attribute 'idade'
In[8]:
```



Imagine o código a seguir

```
Python Console
/Library/Frameworks/Python.framework/Versions/2.7/bin/python2.7
Python 2.7.11 (v2.7.11:6d1b6a68f775, Dec  5 2015, 12:54:16)
In[2]: class Pessoa (object):
...:     def __init__(self, nome):
...:         self.nome = nome
...:
In[3]: p1 = Pessoa("Joao")
In[4]: p1.idade = 35
In[5]: print p1.nome, p1.idade
Joao 35
In[6]: p2 = Pessoa("Maria")
In[7]: print p2.nome, p2.idade
Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versions/2.7/lib/
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "<ipython-input-7-5532aa554d8b>", line 1, in <module>
    print p2.nome, p2.idade
AttributeError: 'Pessoa' object has no attribute 'idade'
In[8]:
```

Definimos a classe **Pessoa**, apenas com o atributo nome

Imagine



Criamos `p1`,
adicionamos e usamos o
atributo `idade`

```
Python Console
/Library/Frameworks/Py
Python 2.7.11 (v2.7.11:
In[2]: class Pessoa ():
...:     def __init__(self, nome):
...:         self.nome = nome
...:
In[3]: p1 = Pessoa("Joao")
In[4]: p1.idade = 35
In[5]: print p1.nome, p1.idade
Joao 35
In[6]: p2 = Pessoa("Maria")
In[7]: print p2.nome, p2.idade
Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versions/2.7/lib/
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "<ipython-input-7-5532aa554d8b>", line 1, in <module>
    print p2.nome, p2.idade
AttributeError: 'Pessoa' object has no attribute 'idade'
In[8]:
```



Imagine o código a seguir

```
Python Console
/Library/Frameworks/Python.framework/Versions/2.7/bin/python2.7
Python 2.7.11 (v2.7.11:6d1b6a68f775, Dec  5 2015, 12:54:16)
In[2]: class Pessoa(object):
...:     def __init__(self, nome, idade):
...:         self.nome = nome
...:         self.idade = idade
In[3]: p1 = Pessoa("João", 35)
In[4]: p1.idade =
In[5]: print p1.nome, p1.idade
João 35
In[6]: p2 = Pessoa("Maria")
In[7]: print p2.nome, p2.idade
Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versions/2.7/lib/
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "<ipython-input-7-5532aa554d8b>", line 1, in <module>
    print p2.nome, p2.idade
AttributeError: 'Pessoa' object has no attribute 'idade'
In[8]:
```

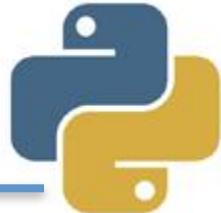
Criamos p2 e tentamos usar o atributo idade



Imagine o código a seguir

```
Python Console
/Library/Frameworks/Python.framework/Versions/2.7/bin/python2.7
Python 2.7.11 (v2.7.11:6d1b6a68f775, Dec  5 2015, 12:54:16)
In[2]: class Pessoa (object):
...:     def __init__(self, nome):
...:         self.
...:
In[3]: p1 = Pessoa(
In[4]: p1.idade = 3
In[5]: print p1.nom
Joao 35
In[6]: p2 = Pessoa("Maria")
In[7]: print p2.nome, p2.idade
Traceback (most recent call last):
File "/Library/Frameworks/Python.framework/Versions/2.7/lib/
exec(code_obj, self.user_global_ns, self.user_ns)
File "<ipython-input-7-5532aa554d8b>", line 1, in <module>
    print p2.nome, p2.idade
AttributeError: 'Pessoa' object has no attribute 'idade'
In[8]:
```

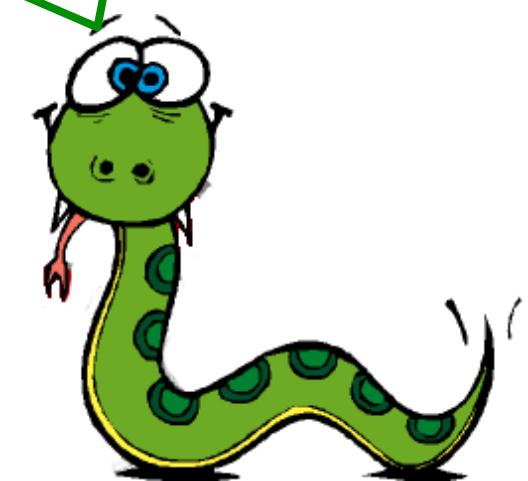
Mas ocorreu um erro, porque
idade só existe em p1

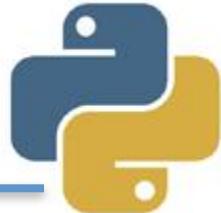


Atribuição de métodos

Também podemos atribuir métodos a objetos, em tempo de execução

```
Python Console
In[11]: def imprimir():
...:     print 'teste'
...: p1.i = imprimir
In[12]: p1.i()
teste
In[13]:
```



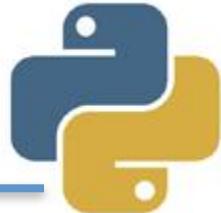


Atribuição de métodos

Definição do método
imprimir

```
Python Console
In[11]: def imprimir():
...:     print 'teste'
...: p1.i = imprimir
In[12]: p1.i()
teste
In[13]:
```

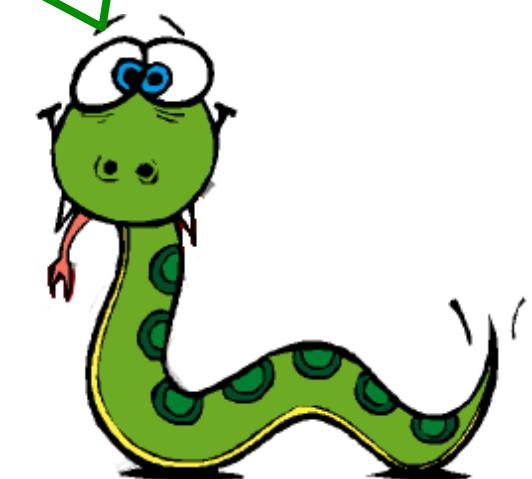


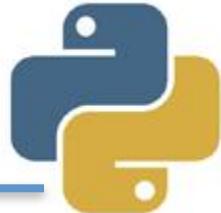


Atribuição de métodos

Atribuição do novo
método a p1

```
Python Console
In[11]: def imprimir():
...:     print 'teste'
...:     p1.i = imprimir
In[12]: p1.i()
teste
In[13]:
```

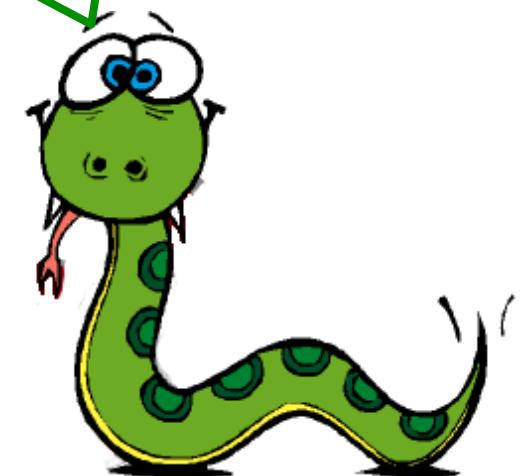


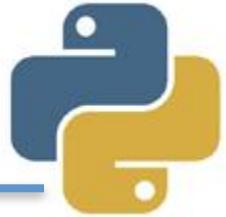


Atribuição de métodos

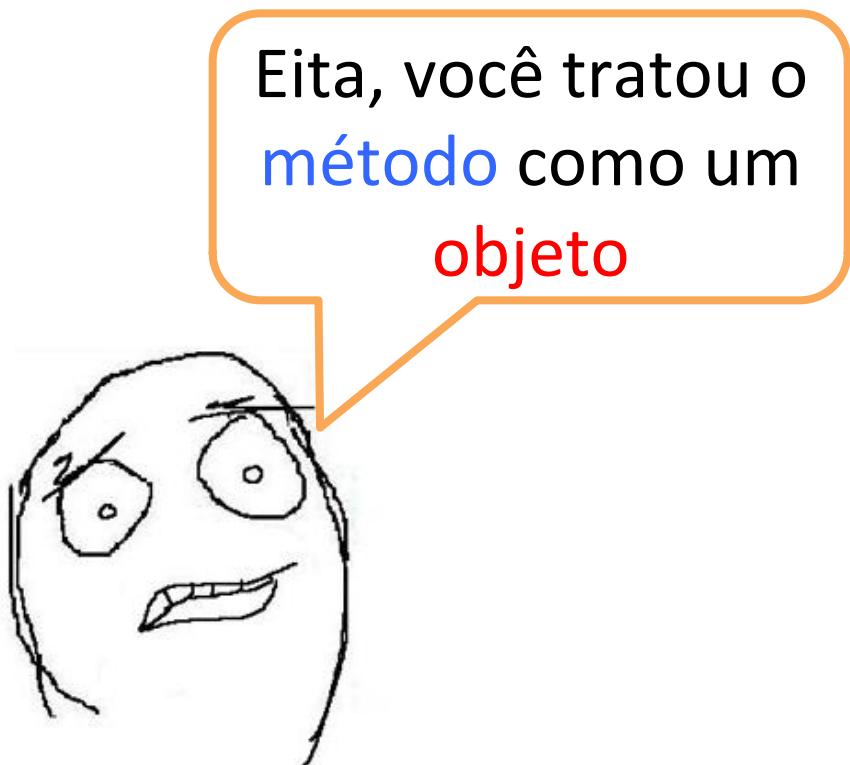
p1 usando seu **novo**
método

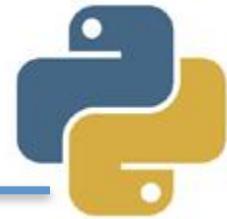
```
Python Console
In[11]: def imprimir():
...:     print 'teste'
...: p1.i = imprimir
In[12]: p1.i()
teste
In[13]:
```



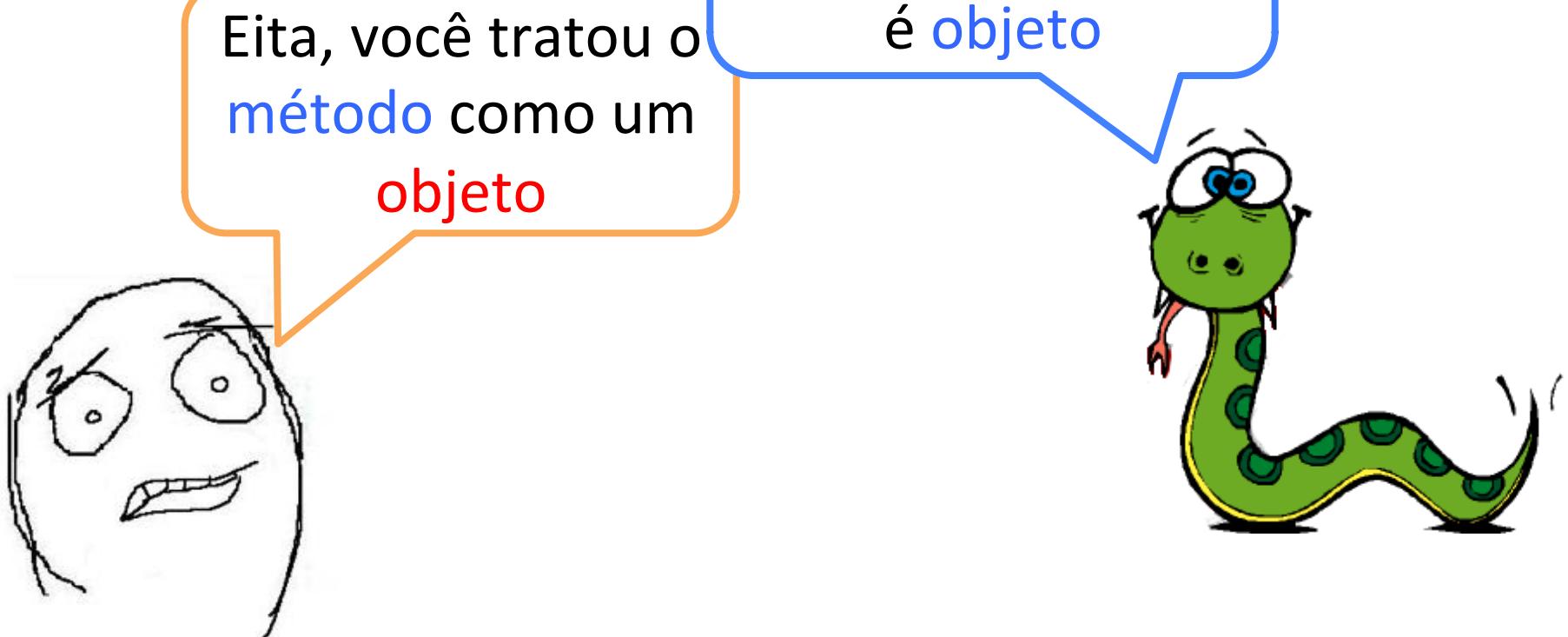


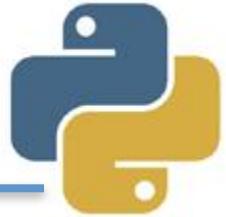
Atribuição de métodos?





Atribuição de métodos?





Atribuição de métodos?

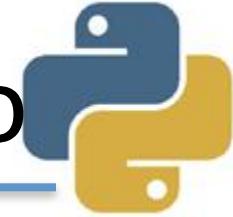
A cartoon illustration featuring a confused-looking person on the left and a green Python snake with large eyes on the right. They are engaged in a conversation via speech bubbles.

Eita, você tratou o
método como um
objeto

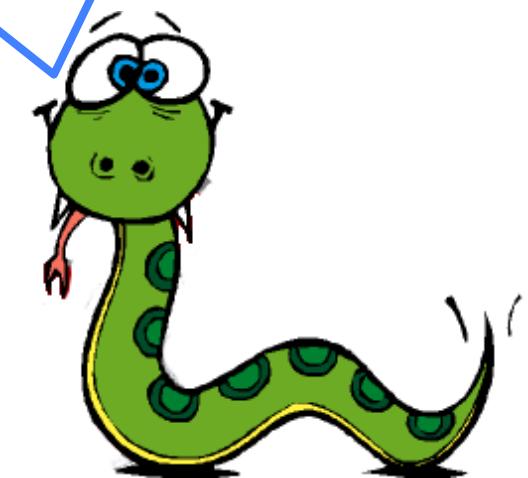
Exatamente!
Em **python**, **TUDO**
é **objeto**

E eu pensando que
isso era só uma
frase de efeito

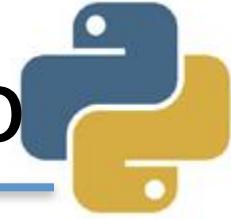
Ainda em tempo de execução



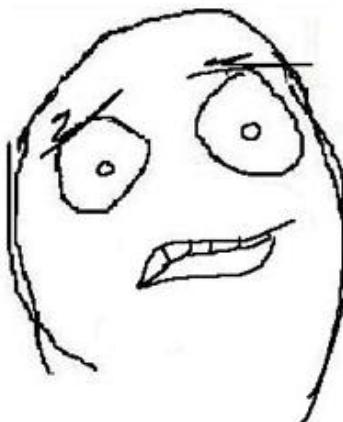
E se eu disser que,
podemos **adicionar**
atributos e **métodos...**



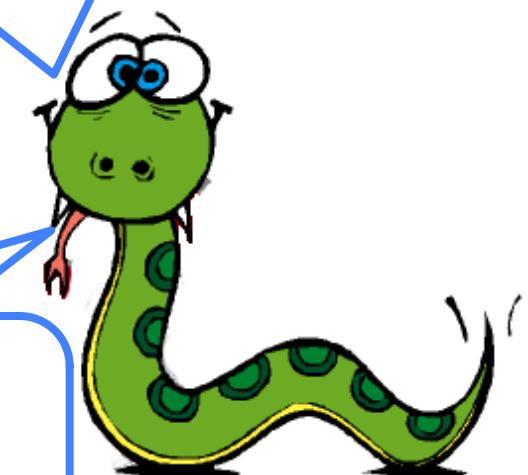
Ainda em tempo de execução



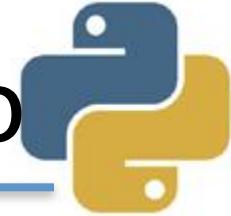
E se eu disser que,
podemos **adicionar**
atributos e **métodos...**



... a uma **classe** em
tempo de execução?



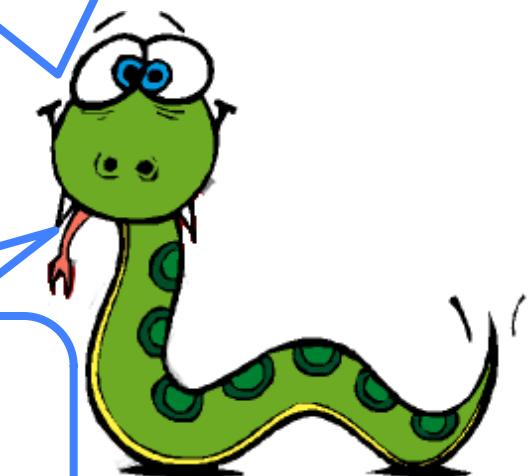
Ainda em tempo de execução

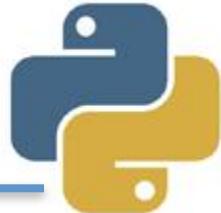


Eu diria:
Shoooow

E se eu disser que,
podemos **adicionar**
atributos e **métodos...**

... a uma **classe** em
tempo de execução?





Exercício 11: implemente

```
class Pessoa (object):
    def __init__(self, nome):
        self.nome = nome
p1 = Pessoa("Joao")
p1.idade = 35
print "nome1", p1.nome, "idade", p1.idade

Pessoa.apelido = "coxinha"
print 'Pessoa.apelido', Pessoa.apelido

def funcao(self):
    print 'Oi,', self.nome, self.apelido

Pessoa.falar = funcao

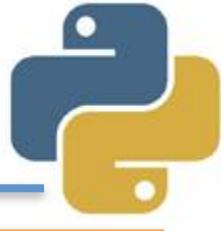
p2 = Pessoa("Maria")
print 'nome2', p2.nome, 'apelido', p2.apelido

p2.falar()

p1.falar()

print 'A classe Pessoa é do tipo', type(Pessoa)

print 'O método funcao é do tipo', type(funcao)
```



Exercício 11: implemente

```
class Pessoa (object):
    def __init__(self, nome):
        self.nome = nome
p1 = Pessoa("Joao")
p1.idade = 35
print "nome1", p1.nome, "idade", p1

Pessoa.apelido = "coxinha"
print 'Pessoa.apelido', Pessoa.apelido

def funcao(self):
    print 'Oi,', self.nome, self.apelido

Pessoa.falar = funcao

p2 = Pessoa("Maria")
print 'nome2', p2.nome, 'apelido', p2.apelido

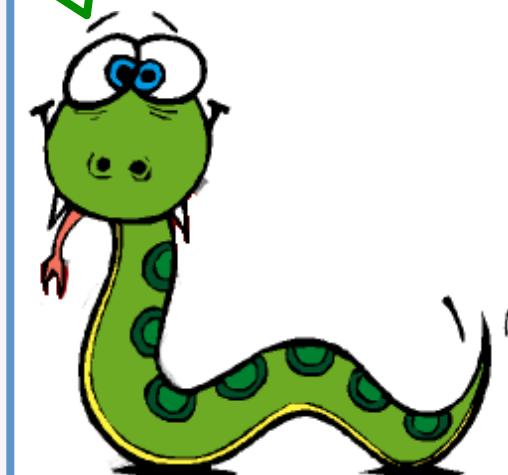
p2.falar()

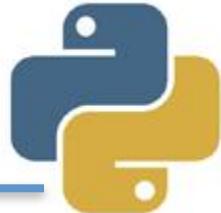
p1.falar()

print 'A classe Pessoa é do tipo', type(Pessoa)

print 'O método funcao é do tipo', type(funcao)
```

Definição da
Classe Pessoa





Exercício 11: implemente

```
class Pessoa (object):
    def __init__(self, nome):
        self.nome = nome
p1 = Pessoa("Joao")
p1.idade = 35
print "nome1", p1.nome, "idade", p1.idade

Pessoa.apelido = "coxinha"
print 'Pessoa.apelido' Pessoa.apelido

def funcao(self):
    print 'Oi,', sel
Pessoa.falar = funca

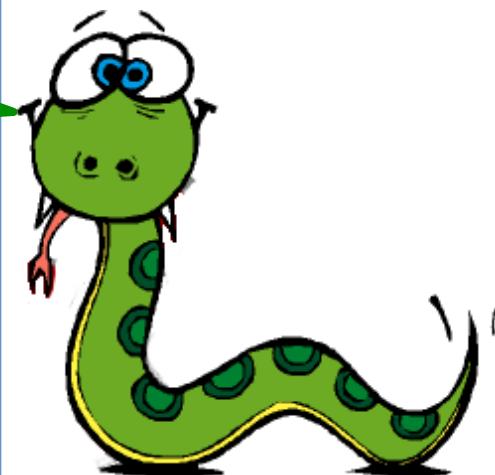
p2 = Pessoa("Maria")
print 'nome2', p2.nome, 'apelido', p2.apelido

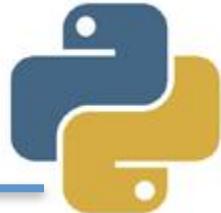
p2.falar()

p1.falar()

print 'A classe Pessoa é do tipo', type(Pessoa)
print 'O método funcao é do tipo', type(funcao)
```

Definição da
variável p1





Exercício 11: implemente

```
class Pessoa (object):
    def __init__(self, nome):
        self.nome = nome
p1 = Pessoa("Joao")
p1.idade = 35
print "nome1", p1.nome, "idade", p1.idade
```

```
Pessoa.apelido = "coxinha"
print 'Pessoa.apelido' Pessoa.apelido
```

```
def funcao(self):
    print 'Oi,', sel
```

```
Pessoa.falar = funca
```

```
p2 = Pessoa("Maria")
print 'nome2', p2.nome, 'apelido', p2.apelido
```

```
p2.falar()
```

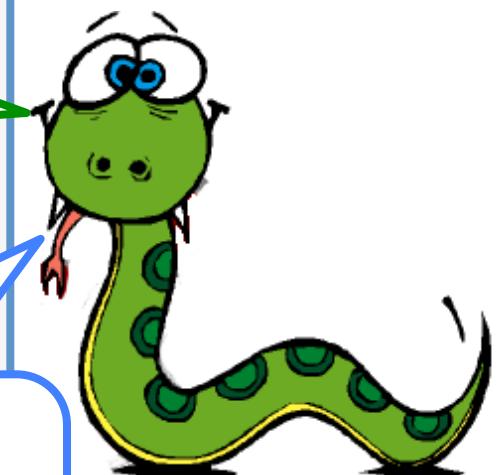
```
p1.falar()
```

```
print 'A classe Pess
```

```
print 'O método funcao é do tipo', type(funcao)
```

Definição da
variável p1

Rpare que p1
cria dinamicamente
o atributo idade





Exercício 11: implemente

```
class Pessoa (object):
    def __init__(self, nome):
        self.nome = nome
p1 = Pessoa("Joao")
p1.idade = 35
print "nome1", p1.nome, "idade", p1.idade

Pessoa.apelido = "coxinha"
print 'Pessoa.apelido', Pessoa.apelido

def funcao(self):
    print 'Oi,', self.nome, self.apelido

Pessoa.falar

p2 = Pessoa()
print 'nome2'

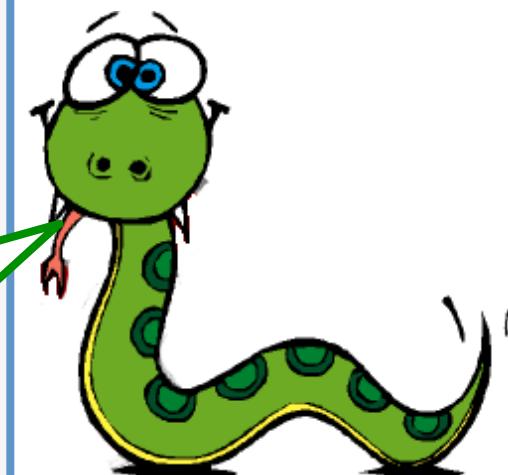
p2.falar()

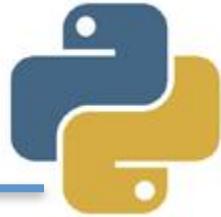
p1.falar()

print 'A classe Pessoa é do tipo', type(Pessoa)

print 'O método funcao é do tipo', type(funcao)
```

A classe **Pessoa**
cria dinamicamente
o atributo **apelido**





Exercício 11: implemente

```
class Pessoa (object):
    def __init__(self, nome):
        self.nome = nome
p1 = Pessoa("Joao")
p1.idade = 35
print "nome1", p1.nome

Pessoa.apelido = "coxa"
print 'Pessoa.apelido', Pessoa.apelido

def funcao(self):
    print 'Oi,', self.nome, self.apelido

Pessoa.falar = funcao

p2 = Pessoa("Maria")
print 'nome2', p2.nome, 'apelido', p2.apelido

p2.falar()

p1.falar()

print 'A classe Pessoa é do tipo', type(Pessoa)

print 'O método funcao é do tipo', type(funcao)
```

Criação e atribuição
dinâmica de método
à classe Pessoa





Exercício 11: implemente

```
class Pessoa (object):
    def __init__(self, nome):
        self.nome = nome
p1 = Pessoa("Joao")
p1.idade = 35
print "nome1", p1.nome, "idade", p1.idade

Pessoa.apelido = "coxinha"
print 'Pessoa.apelido', Pessoa.apelido

def funcao(self):
    print 'Oi,', self.nome, self.apelido

Pessoa.falar = funcao

p2 = Pessoa("Maria")
print 'nome2', p2.nome, 'apelido', p2.apelido

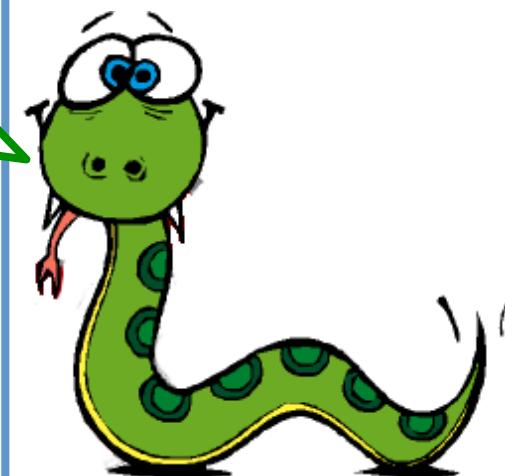
p2.falar()

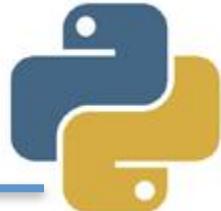
p1.falar()

print 'A classe Pessoa é do tipo', type(Pessoa)

print 'O método funcao é do tipo', type(funcao)
```

Criação da variável p2





Exercício 11: implemente

```
class Pessoa (object):
    def __init__(self, nome):
        self.nome = nome
p1 = Pessoa("Joao")
p1.idade = 35
print "nome1", p1.nome, "idade", p1.idade

Pessoa.apelido = "coxinha"
print 'Pessoa.apelido', Pessoa.apelido

def funcao(self):
    print 'Oi,', self.nome, self.apelido
    self.falar = funcao

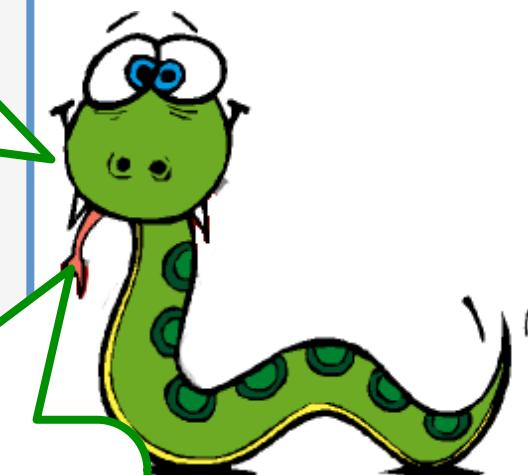
p2 = Pessoa("Maria")
print 'nome2', p2.nome, 'apelido', p2.apelido

p2.falar()

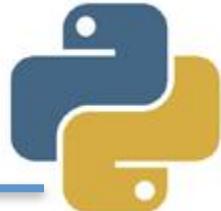
p1.falar()

print 'A classe Pessoa é do tipo', Pessoa.__class__
print 'O método funcao é do tipo', funcao.__class__
```

Criação da variável p2



Note que p2 já usa o atributo apelido



Exercício 11: implemente

```
class Pessoa (object):
    def __init__(self, nome):
        self.nome = nome
p1 = Pessoa("Joao")
p1.idade = 35
print "nome1", p1.nome, "idade", p1.idade

Pessoa.apelido = "Silva"
print 'Pessoa.apelido'

def funcao(self):
    print 'Oi,'

Pessoa.falar = funcao

p2 = Pessoa("Maria")
print 'nome2', p2.nome, 'apelido', p2.apelido

p2.falar()

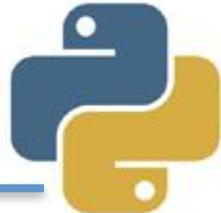
p1.falar()

print 'A classe Pessoa é do tipo', type(Pessoa)

print 'O método funcao é do tipo', type(funcao)
```

Variáveis p1 e p2
usando o novo
método





Exercício 11: implemente

```
class Pessoa (object):
    def __init__(self, nome):
        self.nome = nome
p1 = Pessoa("Joao")
p1.idade = 35
print "nome1", p1.nome, "idade", p1.idade

Pessoa.apelido = "Python"
print 'Pessoa.apelido'

def funcao(self):
    print 'Oi,'

Pessoa.falar = funcao

p2 = Pessoa("Miguel")
print 'nome2', p2.nome

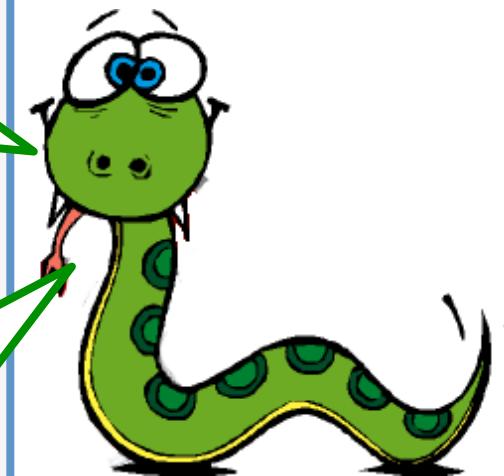
p2.falar()
p1.falar()

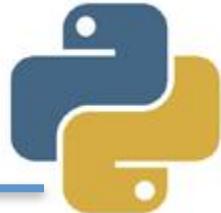
print 'A classe Pessoa é do tipo', type(Pessoa)

print 'O método funcao é do tipo', type(funcao)
```

Variáveis **p1** e **p2**
usando o novo
método

Quando **p1** foi criada,
ainda nem existia o
método **falar()**





Exercício 11: implemente

```
class Pessoa (object):
    def __init__(self, nome):
        self.nome = nome
p1 = Pessoa("Joao")
p1.idade = 35
print "nome1", p1.nome, "idade", p1.idade

Pessoa.apelido
print 'Pessoa.a'

def funcao(self):
    print 'Oi,'

Pessoa.falar = funcao

p2 = Pessoa("Maria")
print 'nome2', p2.nome, 'apelido', p2.apelido

p2.falar()

p1.falar()

print 'A classe Pessoa é do tipo', type(Pessoa)
print 'O método funcao é do tipo', type(funcao)
```

Verificando o **tipo** da
classe e do **método**

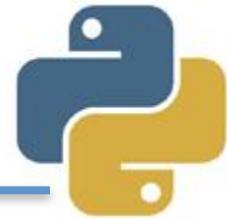




Resultado no notebook

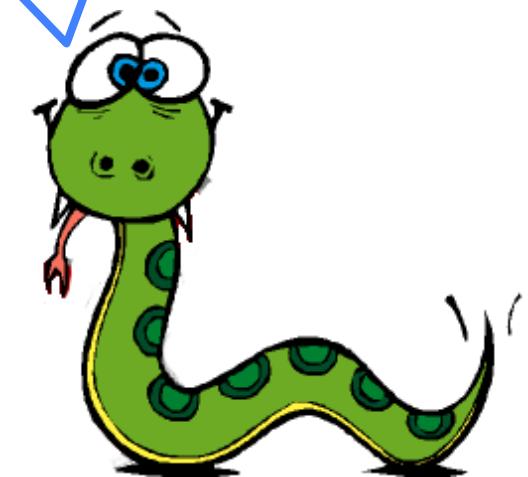
```
print 'A classe Pessoa é do tipo', type(Pessoa)  
print 'O método funcao é do tipo', type(funcao)
```

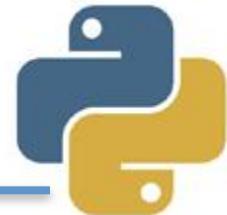
```
nome1 Joao idade 35  
Pessoa.apelido coxinha  
nome2 Maria apelido coxinha  
Oi, Maria coxinha  
Oi, Joao coxinha  
A classe Pessoa é do tipo <type 'type'>  
O método funcao é do tipo <type 'function'>
```



Tudo deu certo

Tudo **funcionou**
perfeitamente

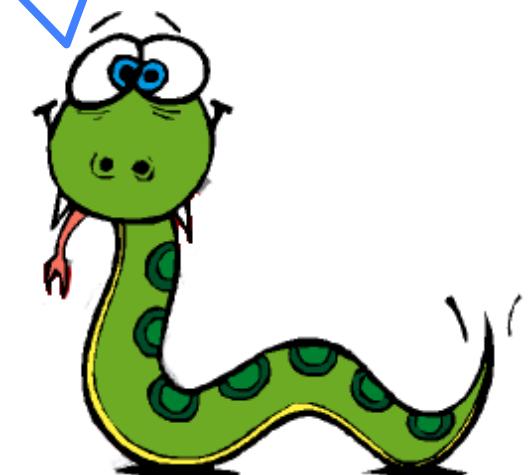
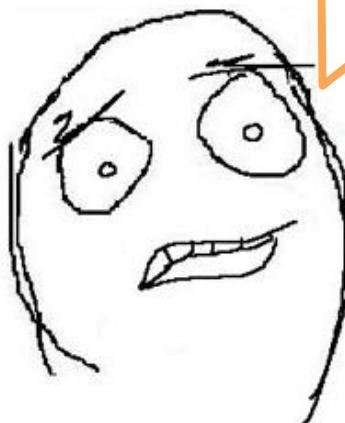


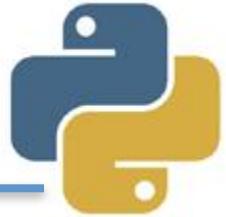


Tudo deu certo

Tudo **funcionou**
perfeitamente

Eu queria **entender**
melhor os **porquês**



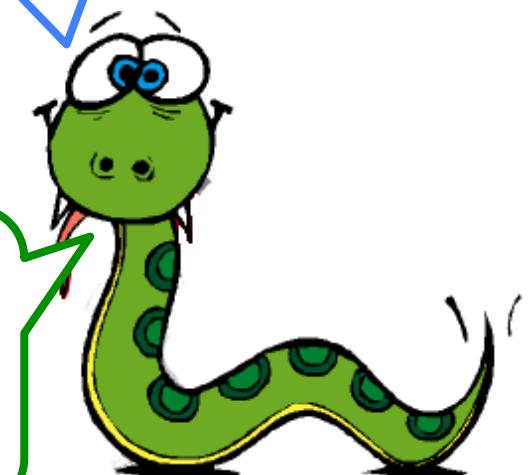
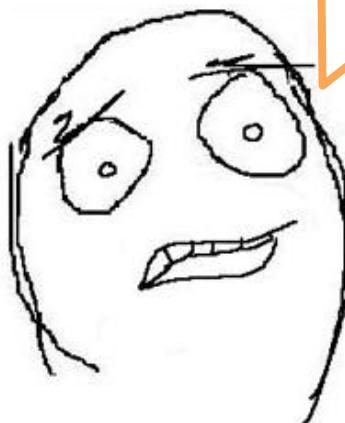


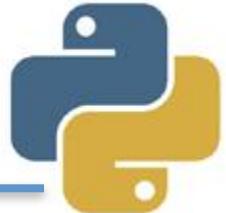
Tudo deu certo

Tudo **funcionou** perfeitamente

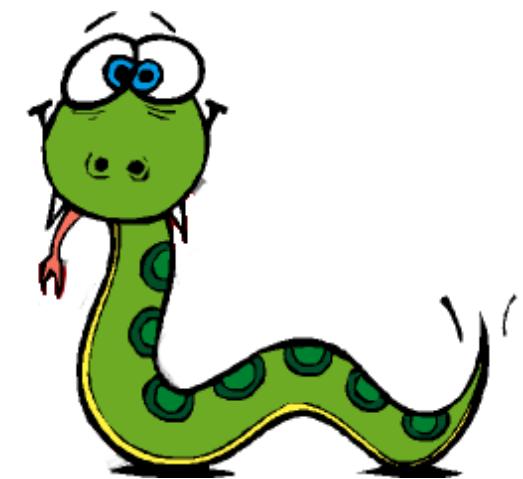
Eu queria **entender** melhor os **porquês**

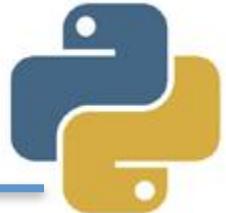
Pra isso, precisaremos de **conceitos** do capítulo de **herança** e **polimorfismo**



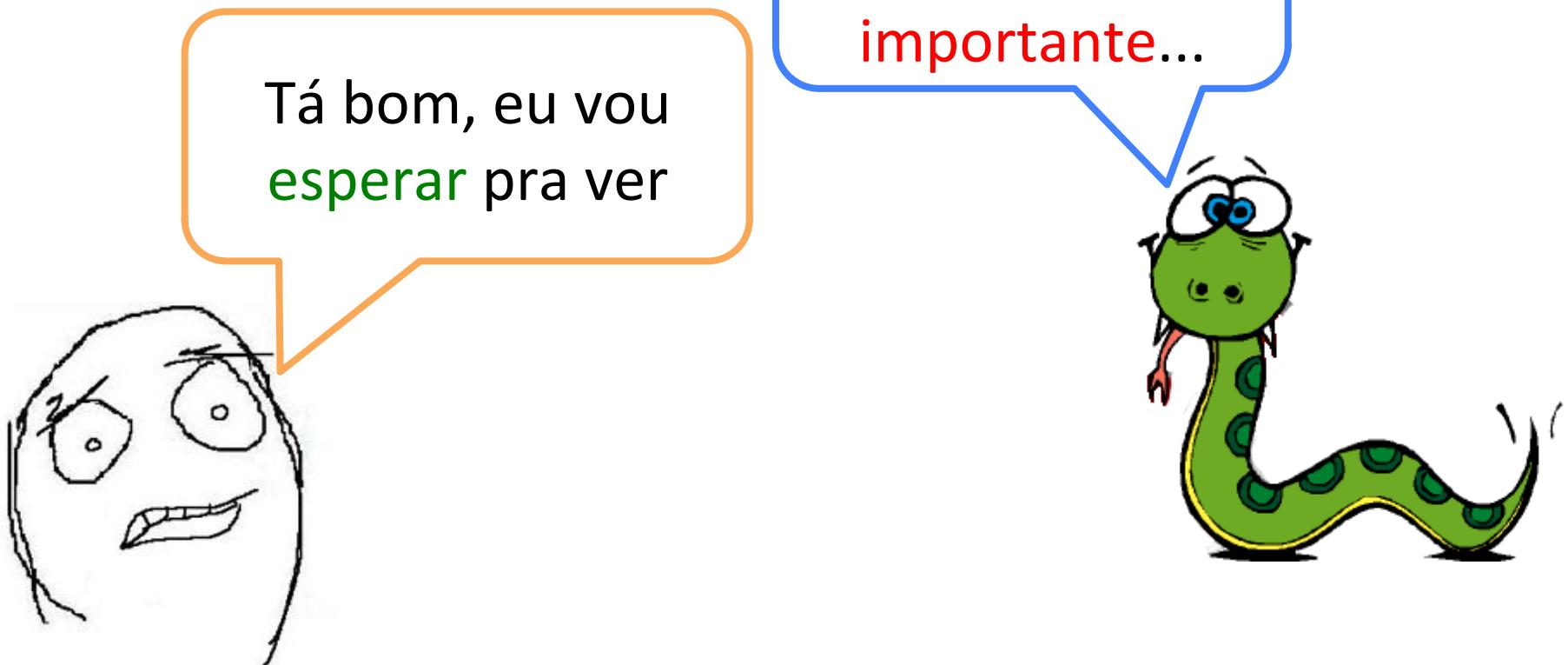


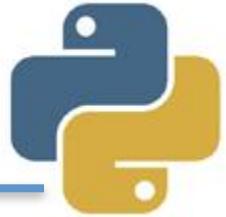
Tudo deu certo



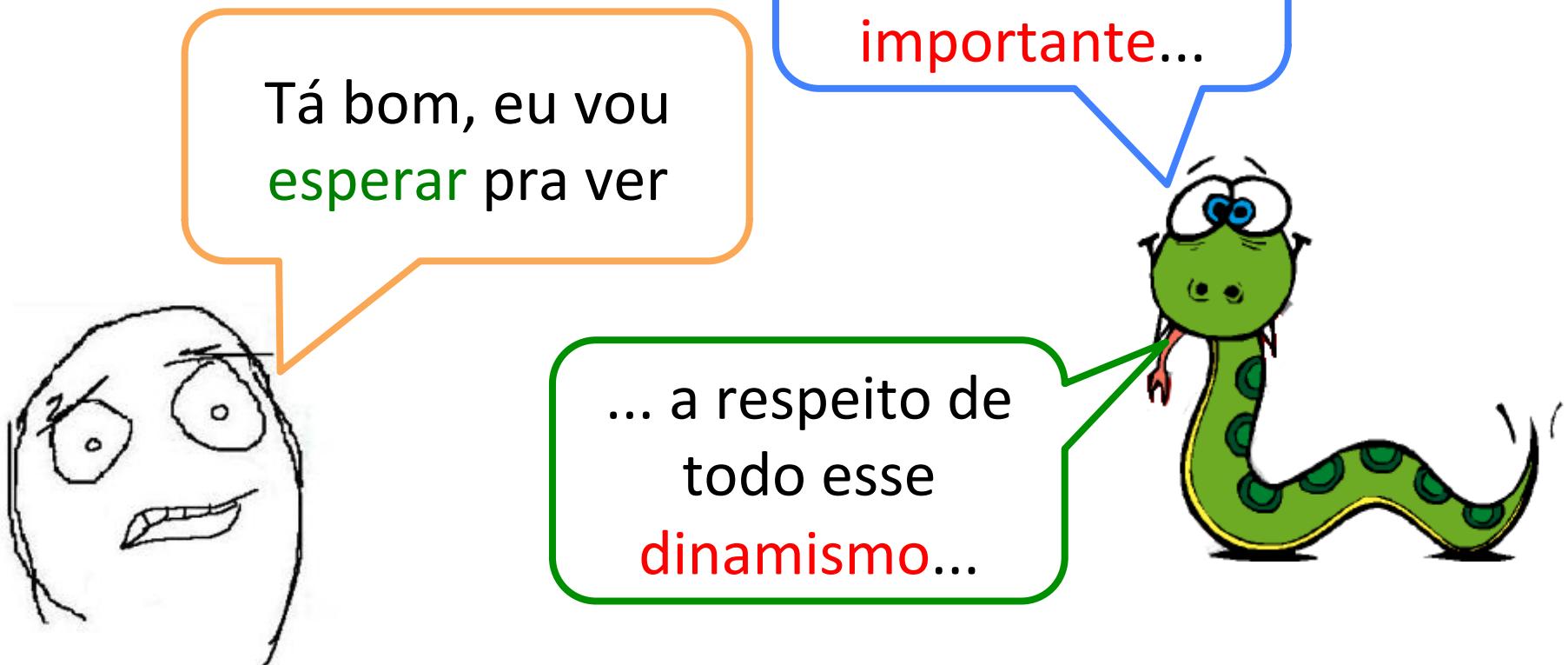


Tudo deu certo

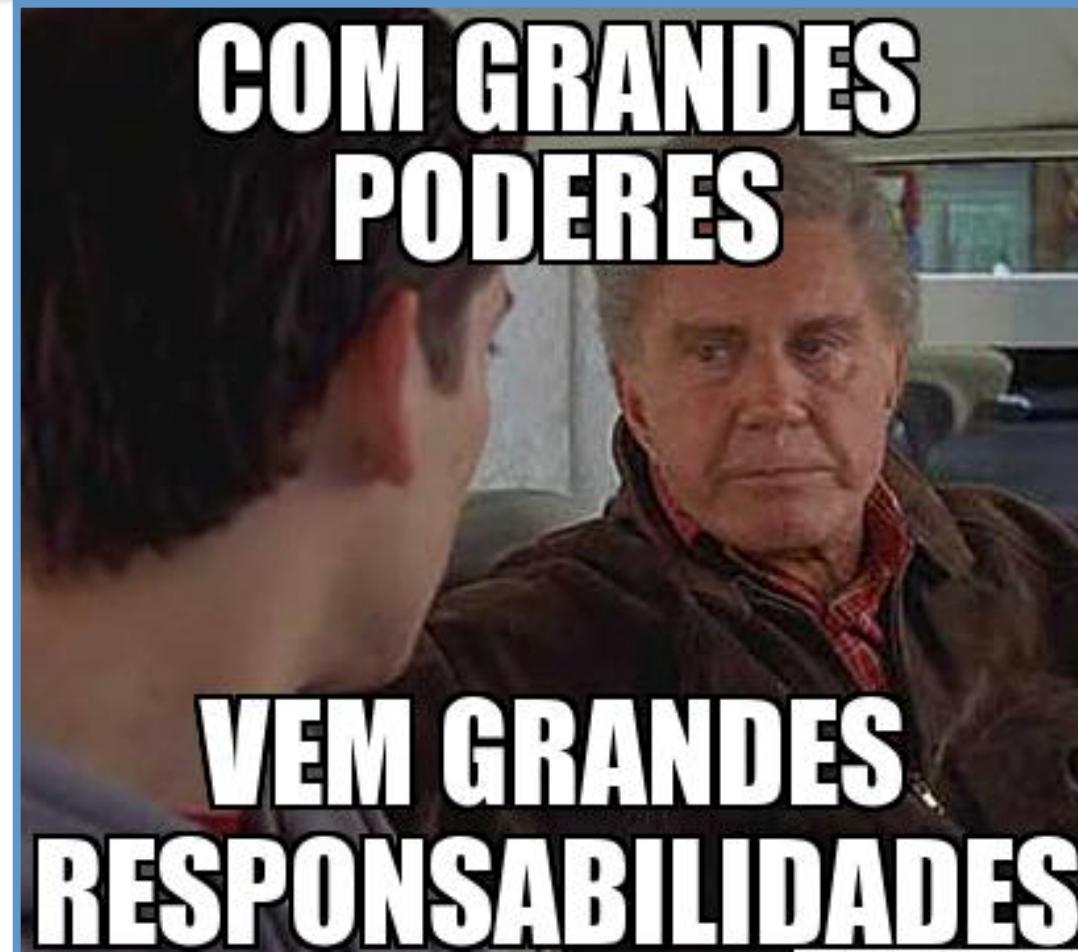
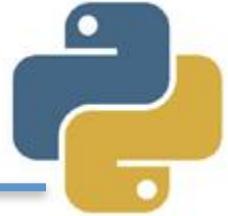




Tudo deu certo



... não esqueça que ...

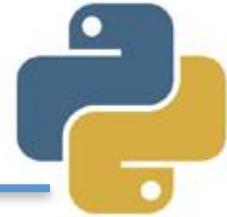




FIM

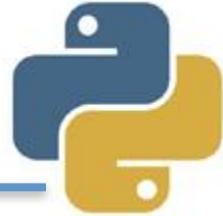


Contatos



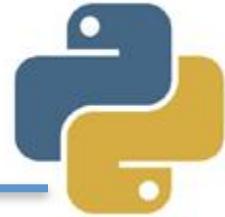
- Márcio Palheta
 - marcio.palheta@gmail.com
 - @marciopalheta
 - <https://sites.google.com/site/marciopalheta/>
-

Bibliografia



- LIVRO: Apress - Beginning Python From Novice to Professional
 - LIVRO: O'Reilly - Learning Python
 - <http://www.python.org>
 - <http://www.python.org.br>
 - Mais exercícios:
 - <http://wiki.python.org.br/ListaDeExercicios>
 - Documentação do python:
 - <https://docs.python.org/2/>
-

Capítulo 03



Orientação a Objetos em Python



Márcio Palheta, M.Sc.
marcio.palheta@gmail.com
