

# BIG DATA

# Introdução ao Big Data

Tema da Aula: **Introdução ao Python**

Prof.: **Dino Magri**

## **Coordenação:**

Prof. Dr. Adolpho Walter  
Pimazzi Canton

Profa. Dra. Alessandra de  
Ávila Montini

- Contatos:

- E-mail: [professor.dinomagri@gmail.com](mailto:professor.dinomagri@gmail.com)
- Twitter: [https://twitter.com/prof\\_dinomagri](https://twitter.com/prof_dinomagri)
- LinkedIn: <http://www.linkedin.com/in/dinomagri>
- Site: <http://www.dinomagri.com>

## Coordenação:

Prof. Dr. Adolpho Walter  
Pimazzi Canton

Profa. Dra. Alessandra de  
Ávila Montini

# Currículo

- **(2014-Presente)** – Professor no curso de Extensão, Pós e MBA na Fundação Instituto de Administração (FIA) – [www.fia.com.br](http://www.fia.com.br)
- **(2018-Presente)** – Pesquisa e Desenvolvimento de Big Data e Machine Learning na Beholder (<http://beholder.tech> )
- **(2013-2018)** – Pesquisa e Desenvolvimento no Laboratório de Arquitetura e Redes de Computadores (LARC) na Universidade de São Paulo – [www.larc.usp.br](http://www.larc.usp.br)
- **(2012)** – Bacharel em Ciência da Computação pela Universidade do Estado de Santa Catarina (UDESC) – [www.cct.udesc.br](http://www.cct.udesc.br)
- **(2009/2010)** – Pesquisador e Desenvolvedor no Centro de Computação Gráfica – Guimarães – Portugal – [www.ccg.pt](http://www.ccg.pt)
- **Lattes:** <http://lattes.cnpq.br/5673884504184733>

## Material das aulas

- Caso esteja utilizando seu próprio computador, realize o download de todos os arquivos e salve na **Área de Trabalho** para facilitar o acesso.
  - Lembre-se de instalar os softwares necessários conforme descrito no documento de Instalação (**InstalaçãoPython3v1.2.pdf**).
- Nos computadores da FIA os arquivos já estão disponíveis, bem como a instalação dos softwares necessários.

# Conteúdo da Aula

- Objetivo
- Funções
- Classes e Objetos
- Exercícios

# Conteúdo da Aula

- **Objetivo**
- Funções
- Classes e Objetos
- Exercícios

# Objetivo

- O objetivo dessa aula é **introduzir conceitos básicos sobre** a linguagem de programação **Python** para Big Data.



# Conteúdo da Aula

- Objetivo
- **Funções**
- Classes e Objetos
- Exercícios

# Funções

- Até agora, vimos diversos tipos de dados, atribuições, comparações e estruturas de controle.
- Em termos simples, uma função agrupa um conjunto de comandos e expressões para que seja possível rodar mais de uma vez dentro de um programa.
- Funções são a melhor alternativa ao famoso **copiar** e **colar**.

# Funções

- Funções também são as estruturas mais básicas do Python que possibilita maximizar a reutilização de código.
- A ideia de uma função é **dividir para conquistar**, onde:
  - Um problema é dividido em diversos subproblemas
  - As soluções dos subproblemas são combinadas na solução do problema maior.
- Esses subproblemas têm o nome de funções.

# Funções

- Python tem diversas funções embutidas, que estão prontas para serem utilizadas.
- Uma lista completa pode ser visualizar em <https://docs.python.org/3/library/functions.html>
- **Já utilizamos algumas delas! Quais?**

# Funções Embutidas

Built-in Functions				
<code>abs()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>all()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>any()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>ascii()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bin()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>bool()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	
<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>	

# Funções

- Além das funções que já estudamos, iremos estudar as seguintes:
  - open
  - sorted
  - zip
  - isinstance

# Funções

open

 Abra o arquivo "**aula2-parte1-funções-embutidas.ipynb**"

# open

- A função `open`, permite abrir um arquivo para leitura e escrita.

```
>>> arq = open(nome_arquivo, modo)
```

- Sendo que o modo pode ser:
  - `r` : abre o arquivo para leitura.
  - `w` : abre para escrita o arquivo (se o arquivo já existir seu conteúdo será apagado)
  - `a` : abre o arquivo para escrita e adiciona novos dados no final do arquivo.
  - `+` : pode ser lido e escrito simultaneamente



## open

- O método `write` deve ser utilizado se o arquivo foi aberto para escrita. O código abaixo abre (cria, se não existir) o `arquivo.txt` para escrita e adiciona 4 linhas:

```
>>> arq = open("arquivo.txt", "w")
>>> for i in range(1, 5):
    arq.write('{} Escrevendo em
arquivo\n'.format(i))
>>> arq.close()
```

# open

- O método `read` carrega todo o conteúdo do arquivo em uma única string.

```
>>> arq = open("arquivo.txt", "r")
```

```
>>> texto = arq.read()
```

```
>>> print(texto)
```

```
1. Escrevendo em arquivo
```

```
2. Escrevendo em arquivo
```

```
3. Escrevendo em arquivo
```

```
4. Escrevendo em arquivo
```

```
>>> arq.close()
```

## open

- O método `readlines` salva todo o conteúdo do arquivo em uma lista, onde cada linha do arquivo será um elemento da lista.

```
>>> arq = open("arquivo.txt", "r")
>>> texto = arq.readlines()
>>> print(texto)
['1. Escrevendo em arquivo\n', '2. Escrevendo em arquivo\n',
'3. Escrevendo em arquivo\n', '4. Escrevendo em arquivo\n']
>>> arq.close()
```

# Funções

sorted

 Abra o arquivo "**aula2-parte1-funções-embutidas.ipynb**"

# sorted

- A função `sorted`, permite ordenar os elementos em uma ordem específica (Crescente ou Decrescente).

```
>>> sorted(obj_iteravel, key, reverse)
```

- Os parâmetros possíveis:
  - `obj_iteravel` – sequência (string, tuplas, listas) ou coleções (set, dicionário).
  - `reverse` (opcional) – Se `True`, a lista é ordenada em ordem decrescente.
  - `key` (opcional) – função que serve como chave para realizar a comparação na hora de ordenar.

# Funções

zip

 Abra o arquivo "**aula2-parte1-funções-embutidas.ipynb**"

# zip

- A função `zip`, permite criar um objeto iterável que agrega elementos de duas ou mais estruturas.

```
>>> zip(*iteradores)
```

- O parâmetro:
  - `iteradores` – pode ser string, listas, set, dicionário, entre outros.
- O retorno da função `zip()` retorna um novo iterador com as tuplas criadas agregando os elementos das estruturas utilizadas.

# Funções

## isinstance

 Abra o arquivo "**aula2-parte1-funções-embutidas.ipynb**"



# isinstance

- A função `isinstance`, permite criar um objeto iterável que agrega elementos de duas ou mais estruturas.

```
>>> isinstance(objeto, class_or_tuple)
```

- O parâmetro:
  - `objeto` – o objeto que se deseja verificar.
  - `class_or_tuple` – classe, tipo ou tupla de classes e tipos.
- O retorno da função `isinstance()` retorna verdadeiro (`True`) se o objeto é uma instância ou subclasse. Retorna falso (`False`) caso contrário.

# Funções

# Funções

- Além das funções já existentes no Python, podemos criar novas funções.
- As novas funções que iremos criar funcionam da mesma forma que as funções embutidas do Python (como é feito a chamada, os parâmetros utilizados e resultados de retorno).
- Funções possibilitam capturar a computação realizada e tratá-la como **primitiva**.

# Funções

- Como exemplo, queremos que a variável  $z$  seja o máximo de dois números ( $x$  e  $y$ ).
- Um programa simples seria:

```
>>> if x > y:
    z = x
else:
    z = y
```

# Funções

- A ideia da função é encapsular essa computação dentro de um escopo que pode ser tratado como primitiva.
  - Sendo que os detalhes internos estão escondidos dos usuários.
  - Para utilizá-las, basta chamar o nome da função e fornecendo os parâmetros necessários.
- Uma função tem **3 partes importantes**:
  - Nome, parâmetros e corpo da função

# Funções

```
def <nome> ( <parametros> ):  
    <corpo da função>
```

- `def` é uma palavra chave e serve para definir uma função.
- `<nome>` é qualquer nome aceito pelo Python.
- `<parametros>` é a quantidade de parâmetros que será passado para a função (pode ser nenhum).
- `<corpo da função>` contém o código da função.

# Funções

- Voltando ao nosso exemplo, podemos reescrever:

```
def maximo(x, y):
```

```
    if x > y:
```

```
        z = x
```

```
    else:
```

```
        z = y
```

- Ótimo **temos** uma **função** e podemos reaproveitá-la.
- Porém, para tratá-la como primitiva precisamos retornar o valor, desta forma, utilizamos o comando `return`

# Funções

- Voltando ao nosso exemplo, podemos reescrever:

```
def maximo(x, y):
```

```
    if x > y:
```

```
        return x
```

```
    else:
```

```
        return y
```

- Agora sim! Já podemos reaproveitar nossa função!

- **E como podemos fazer isso?**



# Funções

- Voltando ao nosso exemplo, podemos reescrever:

```
def maximo(x, y):
```

```
    if x > y:
```

```
        return x
```

```
    else:
```

```
        return y
```

Desta forma,  
o retorno é  
atribuído na  
variável z

Todas as expressões são avaliadas, e caso não se encontre correspondência, é retornado o valor None.

Ou até que encontre a palavra especial **return**

- Agora podemos chamar a função:

```
>>> z = maximo(3, 4)
```

```
>>> print(z)
```

```
4
```

Quando chamamos a função `maximo(3, 4)` estamos definindo que `x = 3` e `y = 4`.

# Funções

- Os parâmetros são passados por posição, porém é possível defini-los explicitamente.
- Considere a seguinte função:

```
def funcao1(seq1, seq2):  
    res = []  
    for x in seq1:  
        if x in seq2:  
            res.append(x)  
    return res
```

O que faz essa função?

# Funções

- A variável `res` dentro da `funcao1` é conhecida como **variável local**.
- **Quais outras variáveis também são locais dentro da `funcao1`?**
  - Os argumentos são passados por atribuição, logo `seq1` e `seq2` também são.
  - `x` utilizado dentro do laço também é uma variável local.
- Quando uma variável é definida fora das funções (`def`), ela é **global** para todo o arquivo.

# Funções

```
nome = "Pedro"
```

```
idade = 30
```

```
def funcao2():
```

```
    res = []
```

```
    x = 20
```

- Quais variáveis são locais e globais?

# Argumentos

- Existem diversas maneiras de passar argumentos para uma função.

Veremos algumas delas:

Sintaxe	Descrição
<code>def func(nome)</code>	Argumento normal, corresponde a qualquer valor passado por posição ou nome.
<code>def func(nome=valor)</code>	Valor padrão é pré-definido se não for passado na chamada.
<code>def func(*nome)</code>	Corresponde e coleta argumentos posicionais restantes em uma tupla.
<code>def func(**nome)</code>	Corresponde e coleta os argumentos de palavras-chave restantes em um dicionário.

# Argumentos

Sintaxe	Descrição
<code>def func(nome)</code>	Argumento normal, corresponde a qualquer valor passado por posição ou nome.

```
>>> def f(a, b, c):  
    print(a, b, c)  
  
>>> f(1, 2, 3)  
1 2 3  
  
>>> f(c=1, a=3, b=2)  
3 2 1  
  
>>> f(1, c=3, b=2)  
1 2 3
```

# Argumentos

Sintaxe	Descrição
<code>def func(nome=valor)</code>	Valor padrão é pré-definido se não for passado na chamada.

```
>>> def f(a, b=2, c=3):
    print(a, b, c)

>>> f(1)
1 2 3

>>> f(1, 3)
1 3 3

>>> f(1, b=7)
1 7 3
```

# Argumentos

Sintaxe	Descrição
<code>def func(*nome)</code>	Corresponde e coleta argumentos posicionais restantes em uma tupla.

```
>>> def f(*args):  
    print(args)
```

```
>>> f()  
  
>>> f(1)  
  
(1,)  
  
>>> f(1, 2, 'a', [True, 0])  
  
(1, 2, 'a', [True, 0])
```



# Argumentos

Sintaxe	Descrição
<code>def func (**nome)</code>	Corresponde e coleta os argumentos de palavras-chave restantes em um dicionário.

```
>>> def f (**args) :  
        print (args)
```

```
>>> f ()  
  
{ }
```

```
>>> f (a=1, b=2)  
  
{ 'a': 1, 'b': 2 }
```

```
>>> f (nome='Maria', idade=30,  
altura=1.65)  
  
{ 'nome': 'Maria', 'idade': 30,  
'altura': 1.65 }
```

# Argumentos



Abra o arquivo **"aula2-parte2-funcoes.ipynb"**

# Conteúdo da Aula

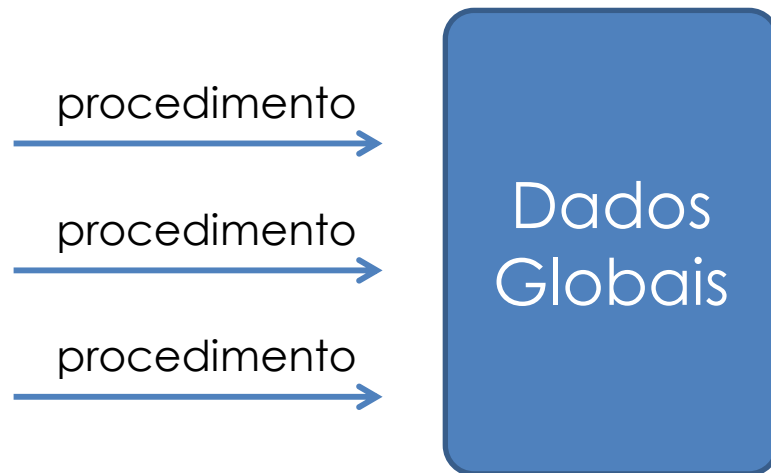
- Objetivo
- Funções
- **Classes e Objetos**
- Exercícios

# Classes e Objetos

- Os sistemas atuais tem uma complexidade muito alta.
- Quanto maior o sistema fica, mais complexo e mais suscetíveis a erros.
- Com isso, para atingir a qualidade e produtividade necessária é importante **REUTILIZAR** códigos.

# Classes e Objetos

- Um paradigma é uma forma de abordar um problema.
- Até agora utilizamos o **paradigma estruturado**:
  - Sequência
  - Decisão
  - Repetição



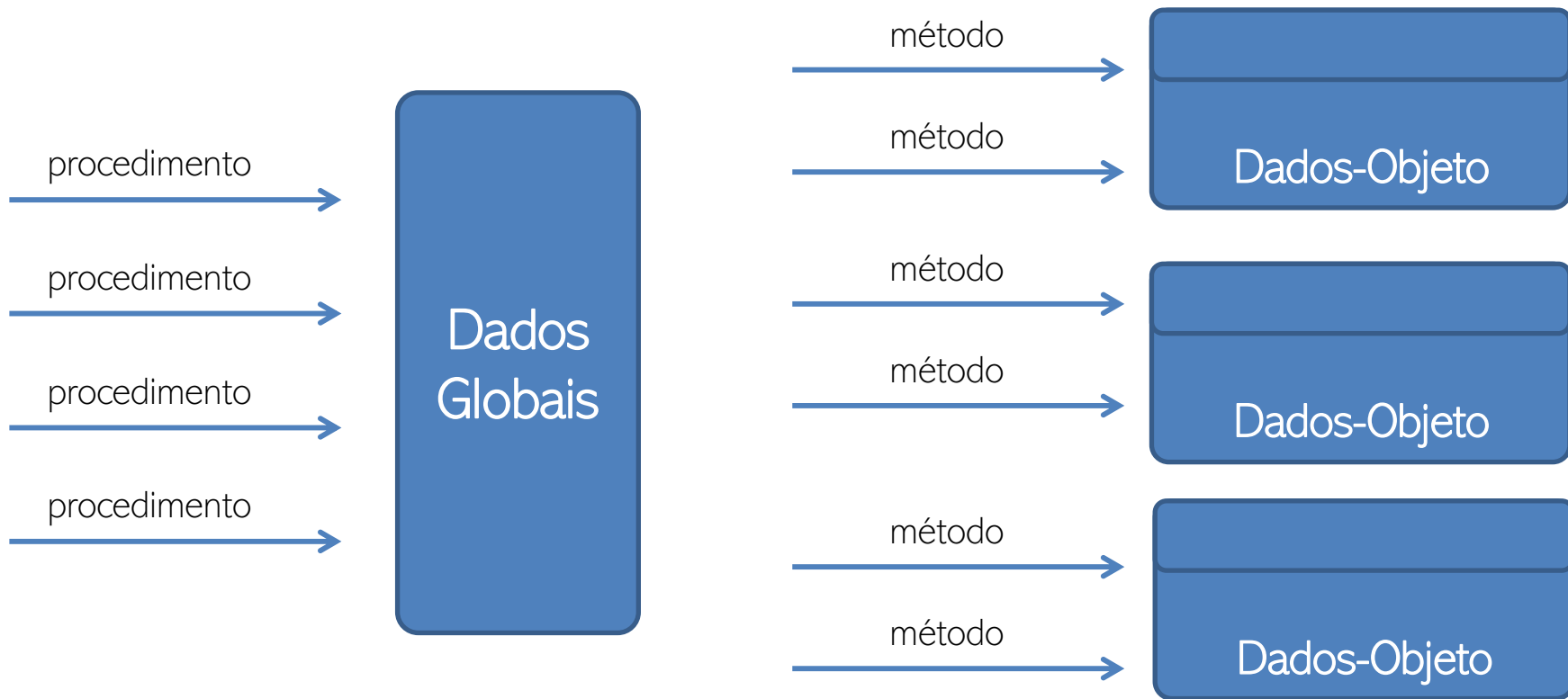
# Classes e Objetos

- Hoje em dia o paradigma estruturado foi superado pelo **paradigma orientado a objetos**.
- Alan Kay formulou a chamada analogia biológica:
  - "Como seria um sistema de software que funcionasse como um ser vivo?"
- Cada célula interage com outras células através do envio de mensagens para realizar um objetivo comum
  - Cada célula se comporta como um unidade autônoma.

# Classes e Objetos

- De uma forma mais geral, Alan Kay pensou em como construir um sistema de software a partir de agentes autônomos que interagem entre si.
- Com isso, estabeleceu os princípios da orientação a objeto.
- **Programação orientada a objetos** consiste em utilizar objetos computacionais para implementar as funcionalidades de um sistema.
- Tudo é um **OBJETO**!

# Classes e Objetos





# Classes e Objetos

- Objetos são entidades que possuem dados e instruções sobre como manipular estes dados.
- Objetos estão ligados à solução do problema.

Exemplo de Software	Objetos
Software Gráfico	Círculos, Linhas, Quadrados, entre outros
Software Banco de Dados	Tabelas, Linhas, Campos, entre outros
Software Comercial	Pedidos, Produtos, Clientes

# Classes e Objetos

- Um programa é uma **coleção de objetos** dizendo uns aos outros o que fazer.
- Para fazer uma requisição a um objeto envia-se **uma mensagem** para este objeto.
- Uma mensagem é uma chamada de um **método** pertencente a um objeto particular.

# Classes e Objetos

- **Todo objeto tem um tipo.**
- Cada objeto é uma instância de uma classe, onde a classe define um tipo.
  - **Classe Cachorro, objeto Jack.**



# Classes e Objetos

- Podemos descrever o cachorro Jack por seus **atributos** físicos:
  - É grande
  - Sua cor principal é dourado
  - Olhos pretos
  - Rabo grande



# Classes e Objetos

- Podemos descrever algumas **ações** do cachorro Jack:
  - Balança o rabo
  - Foge e deita quando leva bronca
  - Late quando ouve um barulho
  - Atende quando o chamamos pelo nome



# Classes e Objetos

- Podemos representar o cachorro Jack:
- Propriedades (atributos):
  - `cor_corpo` : dourado
  - `cor_olhos` : preto
  - `altura` : 58 cm
  - `peso` : 30 kg
  - `idade` : 6
- Métodos (ações):
  - `balançar_rabo()`
  - `latir()`
  - `correr()`
  - `deitar()`
  - `sentar()`

## Instância Jack



Cachorro
cor_corpo : dourado
cor_olhos : preto
altura : 58.4
peso : 30.0
idade : 6
balancar_rabo()
latir()
correr()
deitar()
recuperar_idade()

## Classes e Objetos

### Classe



Cachorro
cor_corpo : string
cor_olhos : string
altura : float
peso : float
idade : int
balancar_rabo()
latir()
correr()
deitar()
recuperar_idade()

## Instância Luna

Cachorro
cor_corpo : preto
cor_olhos : preto
altura : 25.5
peso : 7.5
idade : 3
balancar_rabo()
latir()
correr()
deitar()
recuperar_idade()

Instância  
Jack



Cachorro
cor_corpo : <b>dourado</b>
cor_olhos : <b>preto</b>
altura : <b>58.4</b>
peso : <b>30.0</b>
idade : <b>6</b>
balancar_rabo()
latir()
correr()
deitar()
recuperar_idade()

# Classes e Objetos

## Classe



Instância  
Luna

Cachorro
cor_corpo : <b>preto</b>
cor_olhos : <b>preto</b>
altura : <b>25.5</b>
peso : <b>7.5</b>
idade : <b>3</b>
balancar_rabo()
latir()
correr()
deitar()
recuperar_idade()

- **ATRIBUTOS** descrevem as características das instâncias de uma classe.
- Seus valores definem o estado do objeto.
- O estado de um objeto pode mudar ao longo de sua existência.
- A identidade de um objeto, contudo nunca muda.



## Instância Jack

### Cachorro

cor_corpo	: <b>dourado</b>
cor_olhos	: <b>preto</b>
altura	: <b>58.4</b>
peso	: <b>30.0</b>
idade	: <b>6</b>
balancar_rabo()	
latir()	
correr()	
deitar()	
recuperar_idade()	



6

## Classes e Objetos

### Classe

- **MÉTODOS** representam o comportamento das instâncias de uma classe.
- Correspondem às ações das instâncias de uma classe.

`luna.recuperar_idade()`

`jack.recuperar_idade()`



3

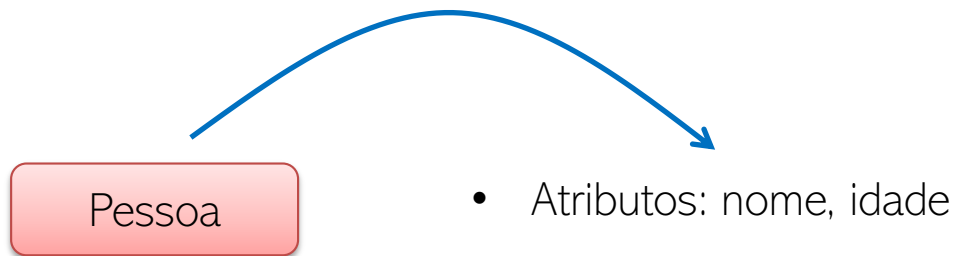
## Instância Luna

### Cachorro

cor_corpo	: <b>preto</b>
cor_olhos	: <b>preto</b>
altura	: <b>25.5</b>
peso	: <b>7.5</b>
idade	: <b>3</b>
balancar_rabo()	
latir()	
correr()	
deitar()	
recuperar_idade()	

# Classes e Objetos

- Como vimos a classe fornece um conjunto de comportamentos na forma de métodos (funções), com implementações que são comuns a todas as instâncias dessa classe.



# Classes e Objetos



- Uma classe serve como o principal meio de abstração na programação orientada à objeto.
- A classe fornece um conjunto de comportamentos na forma de métodos (funções), com **implementações que são comuns a todas as instâncias** dessa classe.

# Classes e Objetos

```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade
```

- Para criar uma classe em Python, temos as seguintes regras:
  - Deve iniciar com a palavra **class** seguida do nome da classe com a primeira letra em maiúscula.
  - Deve terminar com dois pontos no final da linha para abrir o bloco de código.
  - Toda classe associa atributos e métodos numa só estrutura. **Um objeto é uma variável cujo tipo é uma classe, ou seja, um objeto é um instância de uma classe.**
  - Deve conter os métodos que serão utilizados.

# Construtor da classe

```
class Pessoa:  
    def __init__(self, nome, idade):  
        self.nome = nome  
        self.idade = idade
```

 Abra o arquivo **"aula2-parte3-classes.ipynb"**

# Construtor da classe

```
class Pessoa(object):  
    def __init__(self, nome, idade):  
        self.nome = nome  
        self.idade = idade
```

- Quando instanciamos a classe, internamente, é realizado uma chamada do método especial `__init__` que serve como construtor da classe. A quantidade de parâmetros definidos no `__init__` deve ser passado como parâmetro quando a classe for instanciada.
- A principal responsabilidade é **estabelecer o estado do novo objeto** com a instância apropriada das variáveis.

# Construtor da classe

```
class Pessoa(object):  
    def __init__(self, nome, idade):  
        self.nome = nome  
        self.idade = idade
```

- **O self, identifica a instância em que um método é invocado.**
- Todos os métodos de instâncias **devem declarar o self** como primeiro parâmetro!
- Todos os acessos a atributos (inclusive métodos) das instâncias devem ser feitos via **referência explícita a self**.

# Construtor da classe

- Como podemos instanciar e estabelecer esse novo objeto?

```
>>> maria = Pessoa()
```

```
>>> maria = Pessoa('Maria', 30)
```

- Como podemos recuperar o nome e a idade?

```
>>> maria.nome
```

```
>>> maria.idade
```

- E se for preciso alterar a idade para 31?

```
>>> maria.idade = 31
```



## Exercício de 5 minutos

- Modifique a classe Pessoa adicionando os seguintes atributos.
- Apenas os atributos nome e idade são obrigatórios.
- O método `recuperar_info()` deve imprimir todos os atributos.
- Os métodos `falar()` e `andar()` retornam as seguintes strings:
  - `"{} está falando"`
  - `"{} está andando"`

Pessoa
nome : str
idade : int
peso : float
altura : float
<b>falar()</b>
<b>andar()</b>
<b>recuperar_info()</b>

 Abra o arquivo **"aula2-parte3-classes.ipynb"**

## Exercício

- Para testar, realize a instância da classe Pessoa para os objetos (variáveis) maria e pedro.

Instância - maria

Pessoa
nome : Maria
idade : 25
peso : 65.5
altura : 1.72

Instância - pedro

Pessoa
nome : Pedro
idade : 30
peso : 85.5
altura : 1.83

# Exercício



Resposta em "**aula2-parte3-classes-gabarito.ipynb**"

# Herança

- O modelo de hierarquia é muito útil no desenvolvimento de software, pois promove a **reutilização de código**.
- Na programação orientada a objetos, esse modelo é conhecido como **Herança**.
- Isto permite que **uma nova classe seja definida com base em uma classe existente como o ponto de partida**.

# Herança

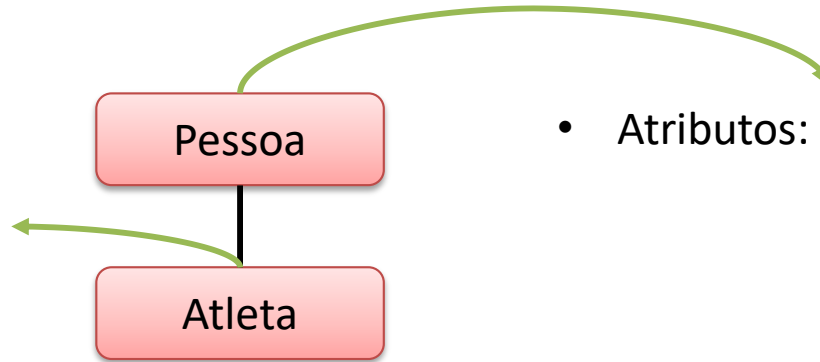
- Na POO, uma classe existente, pode ser descrita como:
  - Classe Base
  - Classe Pai
  - Classe Filha

# Herança

- Uma classe filha pode **especializar** um comportamento existente, fornecendo uma nova aplicação que substitui um método existente.
- E também pode **estender** a classe pai, fornecendo novos métodos.

# Herança

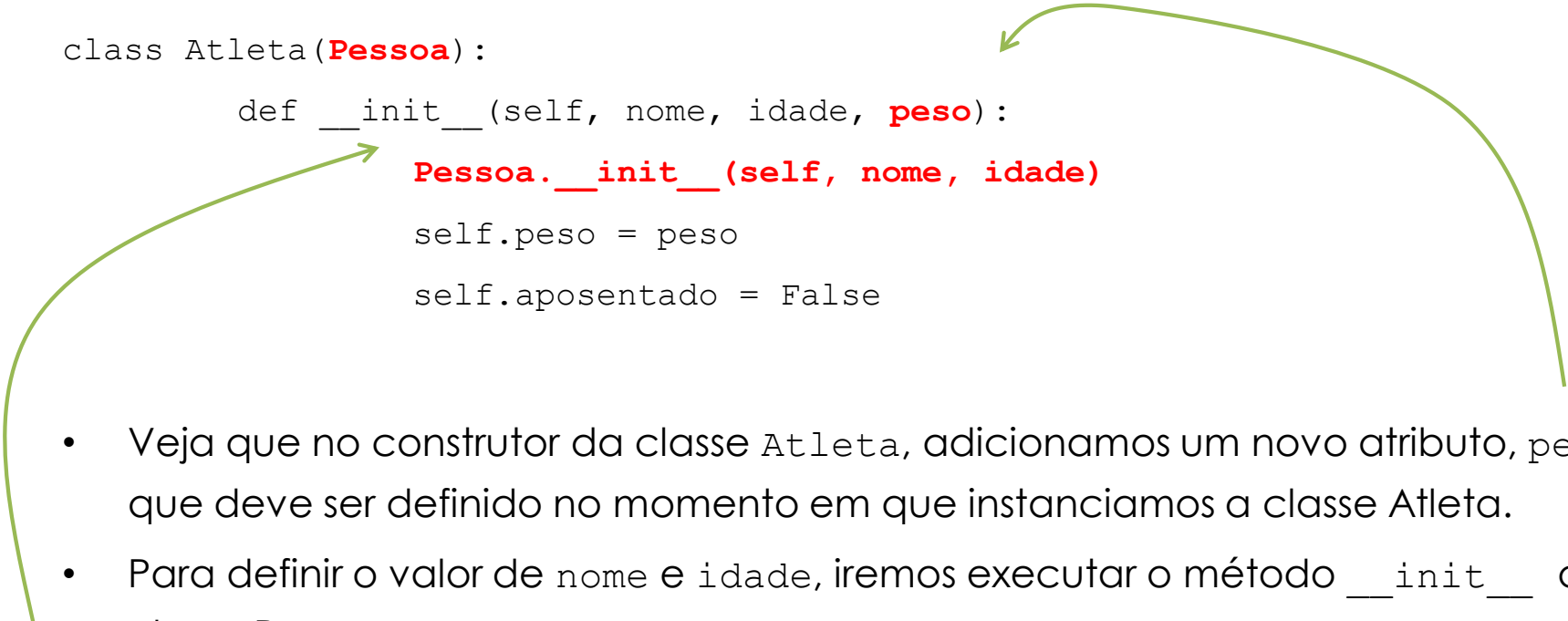
- Atributos: peso, aposentado
- Métodos: aquecer, aposentar



- Atributos: nome, idade

# Herança

```
class Atleta(Pessoa):  
    def __init__(self, nome, idade, peso):  
        Pessoa.__init__(self, nome, idade)  
        self.peso = peso  
        self.aposentado = False
```



- Veja que no construtor da classe `Atleta`, adicionamos um novo atributo, `peso`, que deve ser definido no momento em que instanciamos a classe `Atleta`.
- Para definir o valor de `nome` e `idade`, iremos executar o método `__init__` da classe `Pessoa`.



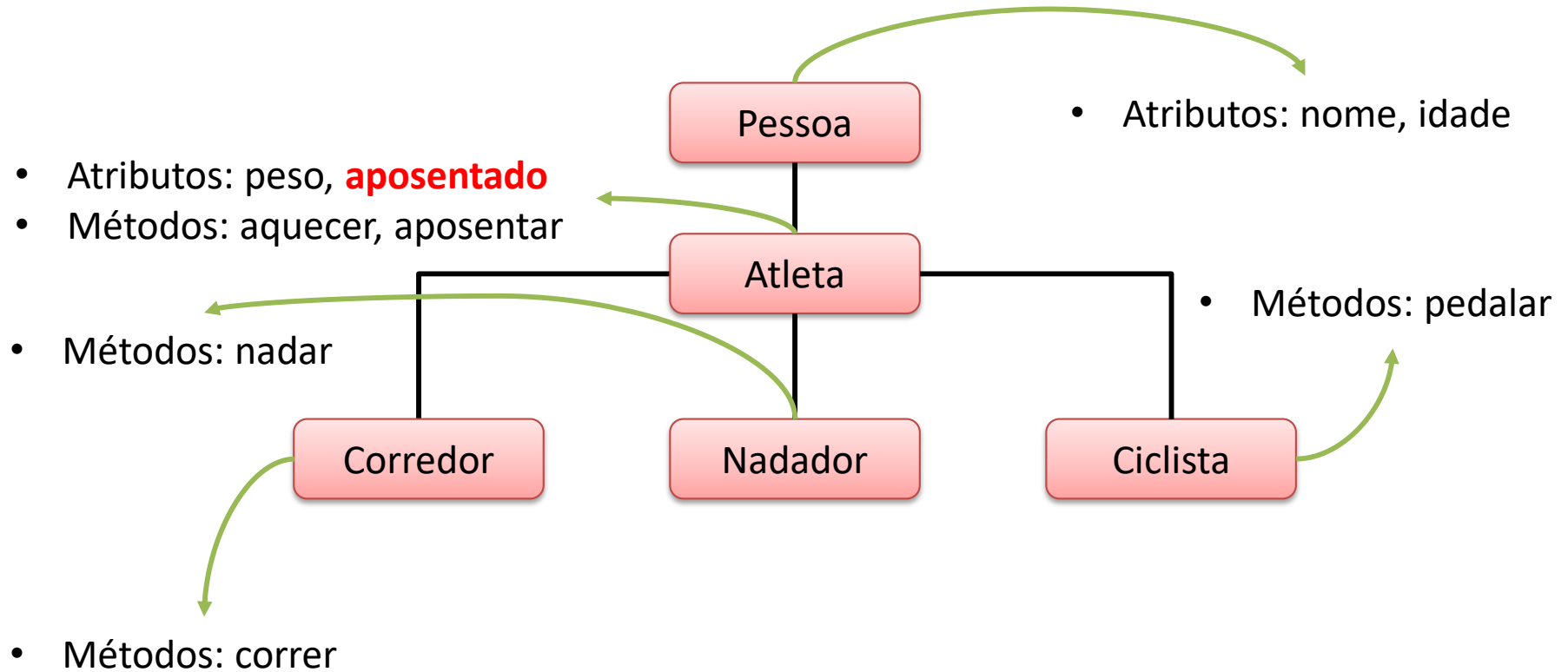
# Herança

```
class Atleta(Pessoa):  
    def __init__(self, nome, idade, peso):  
        Pessoa.__init__(self, nome, idade)  
        self.peso = peso  
        self.aposentado = False  
  
    def aquecer(self):  
        print("Atleta Aquecido")  
  
    def aposentar(self):  
        self.aposentado = True
```



Abra o arquivo "aula2-parte4-classes-herança.ipynb"

# Herança



# Herança

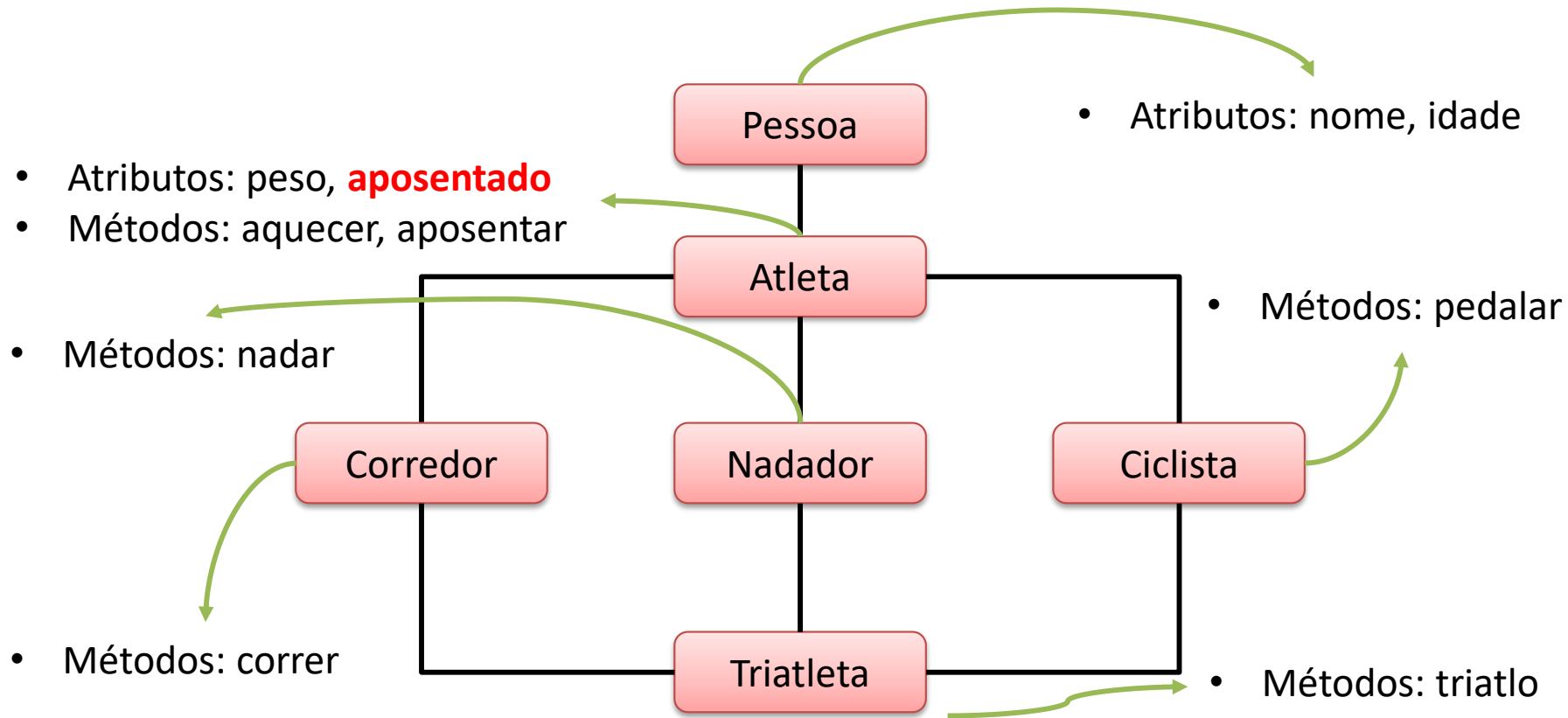
```
class Corredor(Atleta):  
    def correr(self):  
        print("Corredor correndo")
```

```
class Nadador(Atleta):  
    def nadar(self):  
        print("Nadador nadando")
```

```
class Ciclista(Atleta):  
    def pedalar(self):  
        print("Ciclista pedalando")
```

- As três classes Corredor, Nadador e Ciclista têm apenas um método cada e sem nenhum construtor.
- Porém todas herdam os atributos de **Atleta** que por sua vez herdam os atributos de **Pessoa**.

# Herança Múltipla



# Herança Múltipla

- É possível herdar atributos e métodos de múltiplas classes.
- Esse mecanismo é chamado de herança múltipla.
- **Por que quando instanciamos a classe Triatleta temos que definir os valores das variáveis nome, idade e peso?**

# Herança Múltipla

```
class Triatleta(Corredor, Nadador, Ciclista):  
    def triatlo(self):  
        print("Iniciar prova")
```

 Abra o arquivo "**aula2-parte4-classes-herança.ipynb**"

## Exercício de 5 minutos

- Crie uma instancia da classe Pessoa com o nome de **joao**.
- Crie uma instancia da classe Atleta com o nome de **anderson**.
- Crie uma instancia da classe Corredor com o nome de **vanderlei**.
- Crie uma instancia da classe Triatleta com o nome de **silvia**.
- Análise o comportamento da criação e teste os métodos existentes em cada uma das instâncias criadas.

# Conteúdo da Aula

- Objetivo
- Funções
- Classes e Objetos
- **Exercícios**



# Exercícios

1) Escreva uma função chamada `carregar_arquivo`, que lê o conteúdo do arquivo `dados.txt`. Esse arquivo contém uma palavra em cada linha. Para cada linha lida, deve-se adicionar a palavra em uma lista. Ao final do código deve-se retornar a lista criada contendo todas as palavras. Lembre-se de remover o `\n`.

```
>>> def carregar_arquivo(nome_arquivo):
```



Abra o arquivo **"aula2-parte5-exercicios.ipynb"**

## Exercícios

2) Crie uma função chamada **remover\_repetidos**. Essa função deve receber uma lista como parâmetro. Deve-se remover todos as palavras repetidas. Utilize uma lista auxiliar para facilitar. Ao final retorne a lista.

```
>>> def remover_repetidos(dados):
```

## Exercícios

3) Agora, crie uma função chamada **verificar\_repetidos**. Essa função irá receber duas lista como parâmetro. Uma com todas as palavras lidas do arquivo e outra sem as palavras repetidas. Verifique a quantidade de vezes que as palavras aparecem. Ao final imprima a lista de palavras e a quantidade de vezes de cada palavra.

```
>>> def verificar_repetidos(dados, dados_unicos):
```

```
Saída:
```

```
Palavra1 - 10
```

```
Palavra2 - 1
```

# Exercícios

4) Modifique a classe **Pessoa** vista em aula:

- a) Crie um método para calcular a idade em meses, chamado `calcular_meses`.
- b) Instancie a classe com os seguintes argumentos
  - Nome: 'João Silva'
  - Idade: 42
- c) Imprima a seguinte frase:
  - "Y tem X meses de vida", onde Y é o nome e X é o calculo da idade em meses.

# Exercícios

## 5) Implemente a classe Funcionário

- Exemplo de Uso:

```
>>> pedro = Funcionario("Pedro", 7000)
```

```
>>> pedro.aumentar_salario(10)
```

```
>>> pedro.salario
```

```
7700
```

Funcionário
nome : str
salario : float
<b><code>__init__(nome, salario)</code></b>
<b><code>aumentar_salario(percentual)</code></b>

# Exercícios

6) Implemente a classe Aluno. Regras:

- Quando o método estudar(*qtde\_horas*) for executado deve-se **acrescentar** a *qtde\_horas* no atributo *tempo\_sem\_dormir*)
- Quando o método dormir(*qtde\_horas*) for executado deve-se **reduzir** a *qtde\_horas* do atributo *tempo\_sem\_dormir*)
- Crie um código de teste da classe, criando um objeto da classe aluno e utilize os métodos estudar e dormir. Ao final dos testes, imprima a quantidade de horas que o aluno está sem dormir.

Aluno
nome : str
tempo_sem_dormir : int
<b>__init__(nome)</b>
<b>estudar(qtde_horas)</b>
<b>dormir(qtde_horas)</b>

# Exercícios



Resposta em "**aula2-parte5-exercicios-gabarito.ipynb**"

## Referências Bibliográficas

- **Use a Cabeça! Python** – Paul Barry - Rio de Janeiro, RJ: Alta Books, 2012.
- **Use a Cabeça! Programação** – Paul Barry & David Griffiths – Rio de Janeiro RJ: Alta Books, 2010.
- **Aprendendo Python: Programação orientada a objetos** – Mark Lutz & David Ascher – Porto Alegre: Bookman, 2007



# Referências Bibliográficas

- **Python for kids – A playful Introduction to programming** – Jason R. Briggs – San Francisco – CA: No Starch Press, 2013.
- **Python for Data Analysis** – Wes McKinney – USA: O'Reilly, 2013.
- **Python Cookbook** – David Beazley & Brian K. Jones – O'Reilly, 3th Edition, 2013.
- As referências de links utilizados podem ser visualizados em <http://urls.dinomagri.com/refs>