

Aula 2 - processamento de dados com numpy e pandas

Jayme Anchante

23 de fevereiro de 2021

montando ambiente

software

- ▶ git: versionamento de código
- ▶ anaconda: ambientes isolados, reproducibilidade
- ▶ jupyter: interface gráfica

numpy

pacotes que usam numpy

Quantum Computing



QuTiP
PyQuil
Qiskit

Statistical Computing



Pandas
statsmodels
Seaborn

Signal Processing



SciPy
PyWavelets

Image Processing



Scikit-image
OpenCV

Graphs and Networks



NetworkX
graph-tool
igraph
PyGSP

Astronomy Processes



AstroPy
SunPy
SpacePy

Cognitive Psychology



PsychoPy

Bioinformatics



BioPython
Scikit-Bio
PyEnsembl

Bayesian Inference



PyStan
PyMC3

Mathematical Analysis



SciPy
SymPy
cvxpy
FEniCS

Simulation Modeling



PyDSTool

Multi-variate Analysis



PyChem

Geographic Processing



Shapely
GeoPandas
Folium

Interactive Computing

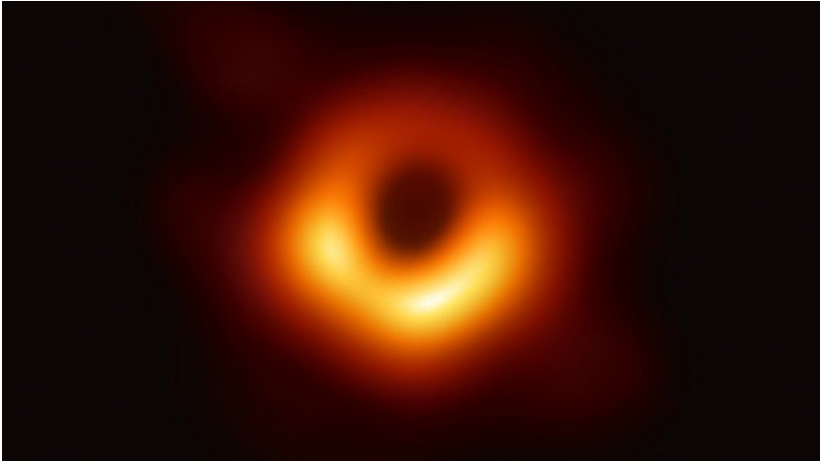


Jupyter
IPython
Binder

The fundamental package for scientific computing with Python, [NumPy website](#)

fronteira da ciência

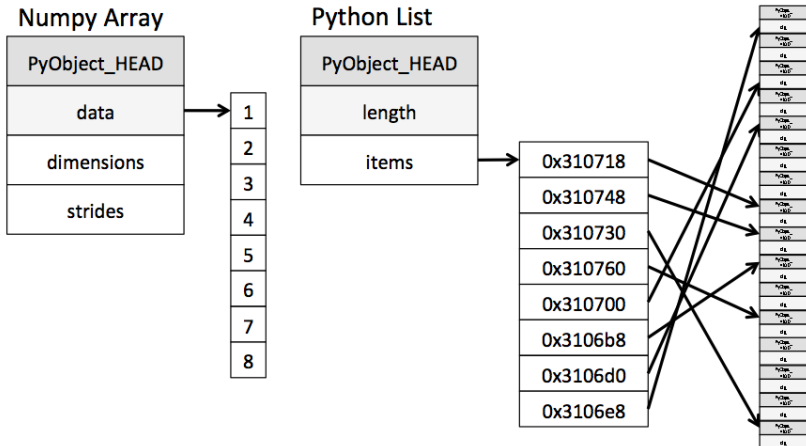
Case Study: First Image of a Black Hole



listas vs arrays vs numpy arrays

- ▶ listas aceitam qualquer tipo de dados (flexibilidade)
- ▶ arrays tem tipo fixo (armazenamento eficiente)
- ▶ numpy arrays tem tipo fixo e otimizações (armazenamento e cálculo eficiente)

lista vs numpy array



criando arrays

```
1  np.zeros(10)
2  np.ones((2,2))
3  np.full((3,1), 3.14)
4  np.arange(5)
5  np.linspace(0, 1, 5)
```

gerador de números pseudo-aleatórios

```
1 np.random.RandomState(42)
2 np.random.seed(42)
3 np.random.<tab>
```

acessando elementos

```
1  x1 = np.random.randint(10, size=6)      # 1 dim
2  x2 = np.random.randint(10, size=(3, 4)) # 2 dim
3  x1[0]
4  x1[-2]
5  x2[0, 0]
6  x2[2, -1]
7  x2[0, 0] = 12
```

fatiamiento de elementos

```
1  # x[start:stop:step]
2  x1[:5]    # primeiros cinco elementos
3  x1[::2]   # cada dois elementos
4  x1[::-1]  # inversão dos elementos
```

cópia de objetos

```
1  x2_sub = x2[:2, :2]
2  x2_sub[0, 0] = 99
3  # o que aconteceu com x2?
4  x2_sub_copy = x2[:2, :2].copy() # fazendo uma cópia
5  x2_sub_copy[0, 0] = 42
6  # o que aconteceu com x2?
```

reformatação de objetos

```
1  x = np.array([1, 2, 3])
2  x.reshape((1, 3)) # row vector via reshape
3  x[np.newaxis, :] # row vector via newaxis
4  x.reshape((3, 1)) # column vector via reshape
5  x[:, np.newaxis] # column vector via newaxis
```

junção e separação de objetos

```
1  np.concatenate
2  np.vstack
3  np.hstack
4  np.split
5  np.vsplit
6  np.hsplit
```

operações com numpy

```
1  def compute_reciprocals(values):
2      output = np.empty(len(values))
3      for i in range(len(values)):
4          output[i] = 1.0 / values[i]
5      return output
6  big_array = np.random.randint(1, 100, size=1000000)
7
8  %timeit compute_reciprocals(big_array)
9
10 %timeit 1.0 / big_array
```


operações com numpy

```
1  # operações agregação
2  x = np.arange(1, 6)
3  np.add.reduce(x)           # soma dos elementos
4  np.add.accumulate(x)      # soma acumulada
5  np.multiply.outer(x, x)   # produto cartesiano
```

sumarizando np.array

```
1  a = np.random.random(100)
2  sum(a)
3  np.sum(a)
4  a.max()
5  # soma com dados faltantes
6  np.nansum(a)
```

máscaras np.array

```
1  a = np.random.random(10)
2  a > 5
3  a[a>5]
4  a[(a>5) & (a<7)]
5  # suporte ao /or e ~/not
```

dado o seguinte np.array

```
1 a = np.arange(25).reshape(5,5)
```

1. Retorne os valores pares positivos menores que 14.
2. Qual a média da segunda coluna?
3. Qual a soma da quarta linha?
4. Separe a última linha e transforme em um vetor coluna.
5. Salve o objeto contendo o np.array em disco.

pandas

história

Iniciado por [Wes McKinney](#) em 2008 quando ele trabalhava no mercado financeiro

Começou como uma implementação em Python da API de dataframe do R

Código aberto em 2009 e posterior apoio pela [NumFocus](#)

pd.Series

```
1 import pandas as pd
2 data = pd.Series([0, 2, 4, 6])
3 # valores e índice são np.array
4 data.values
5 data.index
6 # acessando elementos pelo índice
7 data[-1]
8 data[:2]
```

pd.Series vs np.array

Uma das grandes diferenças está no índice. Ele pode ser não numérico, não sequencial.

```
1 data = pd.Series([0.25, 0.5, 0.75, 1.0],  
2                   index=['a', 'b', 'c', 'd'])  
3 data['b']
```


pd.Series vs dict

```
1  population_dict = {'California': 38332521,  
2                      'Texas': 26448193,  
3                      'New York': 19651127,  
4                      'Florida': 19552860,  
5                      'Illinois': 12882135}  
6  population = pd.Series(population_dict)  
7  population['California':'Illinois']
```

pd.DataFrame

Se o `pd.Series` pode ser comparados a um vetor unidimensional, o `pd.DataFrame` pode ser comparado a uma matrix bidimensional.

O `pd.DataFrame` é como uma sequencia de `pd.Series` que compatilham o mesmo índice.

```
1 area_dict = {'California': 423967, 'Texas': 695662,  
2             'New York': 141297, 'Florida': 170312,  
3             'Illinois': 149995}  
4 area = pd.Series(area_dict)
```

pd.DataFrame a partir de pd.Series

```
1  states = pd.DataFrame({'population': population,  
2                          'area': area})  
3  states  
4  states.index  
5  states.columns
```

construção de pd.DataFrame

```
1  # a partir de series
2  pd.DataFrame(population, columns='population')
3  # a partir de listas
4  data = [{'a': i, 'b': 2 * i} for i in range(3)]
5  pd.DataFrame(data)
6  # preenchimento com nan
7  pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
8  # a partir de np.array
9  pd.DataFrame(np.random.rand(3, 2),
10               columns=['foo', 'bar'],
11               index=['a', 'b', 'c'])
```

índices em pandas

Os índices são como arrays, porém são imutáveis

```
1 ind = pd.Index([2, 3, 5, 7, 11])
2 ind[::2]
3 # tentando colocar novo valor
4 ind[1] = 0
```

índices como sets

Os índices são otimizados para joins e outras operações, baseado na lógica de sets.

```
1 indA = pd.Index([1, 3, 5, 7, 9])
2 indB = pd.Index([2, 3, 5, 7, 11])
3 indA & indB  # intersection
4 indA | indB  # union
5 indA ^ indB  # symmetric difference
```

índices como dicts

```
1 data = pd.Series([0.25, 0.5, 0.75, 1.0],
2                  index=['a', 'b', 'c', 'd'])
3 data['b']          # seleção valor
4 'a' in data        # como dict
5 data.keys()
6 list(data.items())
7 data.to_dict()
8 data['e'] = 1.25 # inserção de nova chave
```

pd.Series como vetor unidimensional

```
1 data['a':'c'] # índice explícito
2 data[0:2]     # índice implícito
3 data[(data > 0.3) & (data < 0.8)] # mask
```


exercícios

1. Crie uma série cujo índice são nomes e cujos valores são idades (use suas informações, de seus amigos e familiares)
2. Teste se 'João' está nos nomes
3. Retorne a última idade
4. Retorne as idades maiores que 65
5. Retorne as idades maiores que 18 e menos que 35

indexadores: loc, iloc, ix

```
1 data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
2
3 data[1]    # índice explícito
4 data[1:3]  # índice implícito
5 # explícito com loc
6 data.loc[1]
7 data.loc[1:3]
8 # implícito com iloc
9 data.iloc[1]
10 data.iloc[1:3]
```

> “Sempre” usar .loc!

dataframe como dict

```
1 data = pd.DataFrame({'area':area, 'pop':pop})
2 # selecionar uma coluna
3 data['area']           # como dict
4 data.area             # como atributo
5 data.area is data['area'] # teste equivalência
6 data.pop is data['pop']   # pop é uma método do obj, perigo
7 data['density'] = data['pop'] / data['area']
```

> Operações usar [column], para atribuições usar .loc!

dataframe como vetor bidimensional

```
1 data.values      # valores brutos
2 data.T          # transposição
3 data.values[0]   # acessando 1a linha
4 data['area']     # acessando coluna
5 data.loc[
6     data.density > 100,
7     ['pop', 'density']]
8                     # select pop, density where density>100
```

operações em dataframe

```
1 df = pd.DataFrame(np.random.randint(0, 10, (3, 4)),  
2                   columns=['A', 'B', 'C', 'D'])  
3 np.sin(df * np.pi / 4) # operações com np e pd
```

dados faltantes

Duas estratégias principais:

- ▶ usando um indicador (V/F) de presença de dados faltantes
- ▶ usando um valor reservado para representar um dado faltante; e.g. -9999 or NaN

Como pandas segue numpy, não existe a noção de NA fora do tipo ponto flutuante

Existem dois valores reservados: NaN (numpy) e None (python)

None

```
1  vals1 = np.array([1, None, 3, 4])
2  vals1 # inferência de tipo é python object

1  # ineficiência da operação em object
2  for dtype in ['object', 'int']:
3      print("dtype =", dtype)
4      %timeit np.arange(1E6, dtype=dtype).sum()
5      print()

1  vals1.sum()
```

NaN

```
1  vals2 = np.array([1, np.nan, 3, 4])
2  vals2.dtype
```

```
1  1 + np.nan
2  vals2.sum(), vals2.min(), vals2.max()
3  np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)
```


pandas: None e NaN

None e NaN são intercambiáveis em pandas

```
1  pd.Series([1, np.nan, 2, None])
2
3  x = pd.Series(range(2), dtype=int)
4  x      # int
5  x[0] = None
6  x      # float
```

operações em nulos

```
1 data = pd.Series([1, np.nan, 'hello', None])
2 data.isnull()
3 data.notnull()
4 data.dropna()
5 data.fillna()
```

> Não fazer comparações diretas como `data == np.nan`!

combinação de dados com numpy: concat

```
1 linha = [1,2,3]
2 np.concatenate([linha, linha, linha])

1 matriz = [[1, 2], [3, 4]]
2 np.concatenate([matriz, matriz])
```

combinação de dados com pandas: concat

```
1  ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
2  pd.concat([ser1, ser1])
3  pd.concat([ser1, ser1], axis=1) # ou axis='columns'
4  pd.concat([ser1, ser1], ignore_index=True)

1  def d(): return np.random.randint(1, 10, (5,2))
2  df1 = pd.DataFrame(d(), columns=['a', 'b'])
3  df2 = pd.DataFrame(d(), columns=['a', 'c'])
4  pd.concat([df1, df2])
5  pd.concat([df1, df2], axis=1)
6  df1.append(df2)
```

combinação de dados: merge

```
1 pd.merge(df1, df2)
2 pd.merge(df1, df2, on='a')           # explicitando chave
3 pd.merge(df1, df2, how='outer')
```

agregação e agrupamento

Vamos usar um dataset do pacote seaborn

```
1 import seaborn as sns
2 planets = sns.load_dataset('planets')
3 planets.shape
4 planets.head()

1 planets.describe()
2 planets.mean()
3 # quantidade de planetas descobertos / ano
4 planets.groupby('year')['number'].sum()
5 # mediana período orbitas / método
6 planets.groupby('method')['orbital_period'].median()
```

vetorização de operações com apply

Aplicação de uma função genérica específica em python puro

```
1 def add2(x):  
2     return x + 2  
3 df = pd.DataFrame(d(), columns=['col1', 'col2'])  
4 df.apply(add2)  
5 df['col1'].apply(add2)
```

trabalhando com texto

```
1 data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
2 [s.capitalize() for s in data]
3 names = pd.Series(data)
4 names.str.capitalize()
```


mais recursos

Livro [Python para Análise de dados](#) do autor do pacote pandas

Vídeos nas conferências PyCon, SciPy e PyData podem ser encontrados no [PyVideo](#)

exercícios

Usando a base de planetas:

1. Mostre os métodos e distâncias após o ano de 2010
2. Calcule o período orbitas vezes massa dividido pela distância
3. Quantos valores nulos existem em cada coluna?
4. Quantos planetas foram descobertos por cada método.
5. Remova os espaços do texto e coloque em caixa baixa a coluna método.

exercícios de casa

lista

Usando a base de dados de [antibióticos](#) (clique no link para acessar os dados), responda:

1. Leia a base e atribua a um objeto chamado `df`
2. Quantas bactérias do tipo “Streptococcus” existem?
3. Qual o maior e menor valor de neomicina? E a qual bactéria estão associados?
4. Quantas bactérias existem por tipo de grama?
5. Crie uma nova coluna chamada “valor” sendo a penicilina vezes a estreptomicina dividido pela neomicina.
6. Salve os dados com essa nova coluna num arquivo chamado “antibioticos.csv” sem o índice e com separador de “;”.