INSTITUTO DE INFORMÁTICA
Universidade Federal de Goiás

Introdução à Python

Altino Dantas

PARTE II

INSTITUTO DE INFORMÁTICA UFG

UFG UNIVERSIDADE FEDERAL DE GOIÁS

Objetivos

- Conhecer o ecossistema SciPy
- Estudar o módulo NumPy

SciPy (Pronúncia “sái pai”)

- As **operações básicas** usadas na programação científica incluem *arrays*, matrizes, integrais, *solvers* de equações diferenciais, estatísticas e muito mais.
- O Python, por padrão, **não possui** nenhuma dessas funcionalidades embutidas, exceto algumas operações matemáticas triviais que só podem lidar com uma variável e não com um *array* ou matriz.

SciPy

- O **SciPy** é um ecossistema baseado em **Python**, com software de código aberto para matemática, ciências e engenharia.

NumPy
Base N-dimensional array package

SciPy library
Fundamental library for scientific computing

Matplotlib
Comprehensive 2D Plotting

IP[y]:
IPython

IPython
Enhanced Interactive Console

Sympy
Symbolic mathematics

pandas
Data structures & analysis

SciPy

- O ecossistema SciPy, uma coleção de software de código aberto para computação científica em Python;
- A comunidade usa e desenvolve;
- Diversas conferências dedicadas à computação científica em Python - SciPy, EuroSciPy e SciPy.in;
- A **biblioteca** SciPy, é um dos componente do ecossistema SciPy e fornece muitas rotinas numéricas.
- A biblioteca SciPy está desenvolvida sobre a base do NumPy

NumPy

- NumPy é o pacote fundamental do Python para **computação científica**;
- Adiciona recursos de matrizes N-dimensionais, operações elemento-a-elemento e operações matemáticas fundamentais como **álgebra linear**;
- *Arrays* em NumPy possuem muitas vantagens sobre lista ou dicionários nativos do Python.

NumPy

- As listas em Python podem armazenar praticamente qualquer tipo de dado;
- Operar sobre um elemento de uma lista só é possível **iterando** sobre a coleção, o que em Python **não é muito eficiente**;
- NumPy permite contornar essa limitação das listas fornecendo um objeto de armazenamento de dados chamado ***ndarray***.

NumPy

- O **ndarray** é semelhante às listas, mas apenas um mesmo tipo de elemento pode ser armazenado em cada coluna;
- Por exemplo, com uma lista do Python, você pode transformar o primeiro elemento em uma lista e o segundo em uma *string* ou dicionário;
- Com os arrays NumPy, você só pode armazenar o mesmo tipo de elemento, por exemplo, todos os elementos devem ser *float*, inteiros ou *strings;*
- Apesar desta limitação, o ndarray é superior quando se trata de **desempenho**, pois as operações são significativamente mais rápidas.

NumPy vs Python List

- Comparando os dois em termos de desempenho;

```python
import numpy as np

# Criar um array com 10^7 elements.
arr = np.arange(1e7)

# Converter o ndarray numa lista
larr = arr.tolist()

# Como lsitas não aceitam um operação "broadcast"
# é necessário criar uma função para simular
def list_times(alist, scalar):
    for i, val in enumerate(alist):
        alist[i] = val * scalar
    return alist
```

```
In [11]: timeit list_times(larr,.5)
973 ms ± 113 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [12]: timeit arr * 0.5
47.6 ms ± 716 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

NumPy ndarray vs matrix

- Operações de Álgebra Linear provavelmente devem ser executadas em **matrix** invés de **ndarray**;
- Objetos **matrix** não utilizam operação *broadcast* padrão de **ndarray**;
- Por exemplo, se a A e B forem do tipo matrix, A * B executa uma multiplicação de matrizes;
- Por outro lado, se A e B forem do tipo ndarray, A * B executa uma multiplicação entre cada $A_{i,j}$ e $B_{i,j}$

Criando *array* e definindo tipo

```
# A partir de uma lista
alist = [1, 2, 3]
arr = np.array(alist)
```

```
# Criando array de 'zeros' tendo n elementos
arr = np.zeros(5)
```

```
# Criando array obedecendo um 'range'
arr = np.arange(100)

# de 10 a 100
arr = np.arange(10,100)


# 200 passos entre 0 e 1
arr = np.linspace(0, 1, 200)
```

Criando *array* e definindo tipo

- Por padrão, **NumPy** define o tipo de acordo com o endereçamento padrão do SO;
- Você pode especificar a precisão de *bits* ao criar *array*, definindo o parâmetro (dtype) para **int**, **numpy.float16**, **numpy.float32** ou **numpy.float64**

```
# Criar um array 5x5 de zeros (uma imagem)
image = np.zeros((5,5))

# Criar um cubo 5x5x5 de 1's
# O método astype() define um array de inteiros
cube = np.zeros((5,5,5)).astype(int) + 1

# Ou ainda mais simples, definindo uma precisão de 16-bit
cube = np.ones((5, 5, 5)).astype(np.float16)
```

Criando *array* e definindo tipo

- Definição do tipo de cada *array:*

```
# Array de zeros do tipo inteiro
arr = np.zeros(2, dtype=int)
arr[0] = 5.98454654315
```

```
arr
```

```
array([5, 0])
```

```
# Array de zeros do tipo float
arr = np.zeros(2, dtype=np.float32)
arr[0] = 5.98454654315
```

```
arr
```

```
array([5.9845467, 0.       ], dtype=float32)
```

Redefinindo o formato

- Se temos uma matriz de 25 elementos, podemos fazer uma matriz de 5 × 5, ou mesmo uma matriz tridimensional a partir de uma matriz plana;

```
# Criando um array com elementos de 0 a 999
arr1d = np.arange(1000)

# Agora remodelando o array para um array 3D de 10x10x10
arr3d = arr1d.reshape((10,10,10))

# O comando reshape pode, alternativamente, ser chamado dessa maneira
arr3d = np.reshape(arr1d, (10, 10, 10))

# Inversamente, podemos "achatar" matrizes
arr4d = np.zeros((10, 10, 10, 10))
arr1d = arr4d.ravel()
```

Record Arrays

- Este é um recurso para adicionar “cabeçalho” a um ***array***;
- Especialmente útil para quando as colunas possuem tipo de dados diferentes;

```
# Criando uma matriz de zeros e definindo tipos de coluna
recarr = np.zeros((2,), dtype=('i4,f4,a10'))

# Definindo valores para adicionar ao array
toadd = [(1,2.,'Hello'),(2,3.,"World")]
recarr[:] = toadd
```

Record Arrays

- Este exemplo ilustra como a *recarray* se parece, mas é difícil ver como podemos popular facilmente esse *array;*
- Felizmente, em Python existe uma função global chamada **zip** que cria uma lista de tuplas como vemos acima para o objeto toadd.

```
# Criando uma matriz de zeros e definindo tipos de coluna
recarr = np.zeros((4,), dtype=('i4,f4,a10'))

# Criando as colunas que serão armazenadas no recarray
col1 = np.arange(4) + 1
col2 = np.arange(4, dtype=np.float32)
col3 = ['Hello', 'World', 'my', 'friend']

# Aqui cria-se uma lista de tupas
toadd = zip(col1, col2, col3)

# Atribui-se os valores ao recarray
recarr[:] = list(toadd)
```

Record Arrays

- Por fim, é possível definir nomes para as colunas e acessá-las por estes:

```
# Atribuindo nomes a cada coluna
# Por padrão chamadas de 'f0', 'f1' e 'f2'.
recarr.dtype.names = ('Integers' , 'Floats', 'Strings')

# Chamando uma coluna pelo título
recarr['Floats']
```

Indexando e fatiando

- Acessar índices de **arrays numpy** é um pouco diferente de listas normais:

```
alist=[[1,2],[3,4]]

# Para retornar o elemento (0,1) seria:
alist[0][1]
```

- Se quisermos retornar a **coluna da direita**, não há maneira trivial de fazer isso com as listas do Python.
- No NumPy, a indexação segue uma sintaxe mais conveniente.

Indexando e fatiando

- Indexação com **numpy**:

```
# Convertendo a lista definida acima em um array
arr = np.array(alist)

# Para retornar o elemento (0,1)...
arr[0,1]

# Agora para acessar a última coluna ...
arr[:,1]

# Analogamente, a linha de baixo pode ser acessada ...
arr[1,:]
```

Indexando e fatiando

- Às vezes, são necessários esquemas de indexação mais complexos, como a indexação condicional;
- O tipo mais comumente usado é **numpy.where ().**

```
# Criando um novo array
arr = np.array([0,3,8,3])

# Criando um array de indíces quando o respectivo
# valor for maior que 2
index = np.where(arr > 2)

print(index)

# Criando o array a partir dos índices
new_arr = arr[index]

print(new_arr)
```

Indexando e fatiando

- É possível remover índices específicos;
- Para fazer isso, você pode usar **numpy.delete ()**;
- As variáveis de entrada necessárias são o *array* e os índices que se deseja remover.

```
# We use the previous array
new_arr = np.delete(arr, index)
print(new_arr)
```

```
[0]
```

```
index = arr > 2
print(index)

new_arr = arr[index]
print(new_arr)
```

```
[False  True  True  True]
[3 8 3]
```

Declarações Booleanas com NumPy

- As declarações booleanas são comumente usadas em combinação com o operador **and** e o operador **or;**

- Em **arrays** NumPy, é possível usar **&** e **|** para comparações rápidas de valores booleanos.

```
# Criando uma imagem
img1 = np.zeros((20, 20)) + 3
img1[4:-4, 4:-4] = 6
img1[7:-7, 7:-7] = 9
# Gráfico A

# filtrando todos os valores maiores que 2 e menores que 6.
index1 = img1 > 2
index2 = img1 < 6
compound_index = index1 & index2

# O compound_index poderia ser obtido por:
compound_index = (img1 > 2) & (img1 < 6)
img2 = np.copy(img1)
img2[compound_index] = 0
# Gráfico B

# Making the boolean arrays even more complex
index3 = img1 == 9
index4 = (index1 & index2) | index3
img3 = np.copy(img1)
img3[index4] = 0
# Gráfico C
```

Que tal um *plot*?

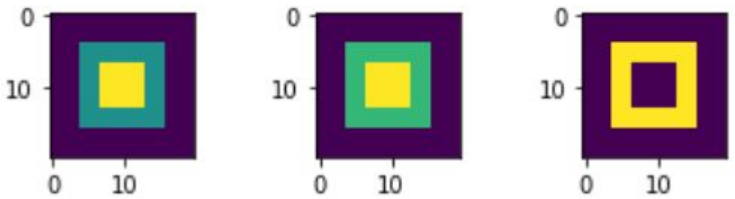
```
from matplotlib import pyplot as plt

plt.subplot(331)
plt.imshow(img1, interpolation='nearest')

plt.subplot(332)
plt.imshow(img2, interpolation='nearest')

plt.subplot(333)
plt.imshow(img3, interpolation='nearest')

plt.show()
```



Exercício

- Gere um vetor unidimensional com 1000 valores entre 0 e 225;
- Gere dois vetores A e B, preenchidos com zeros e uns, respectivamente. Os dois novos vetores devem possuir o formato (10,50);
- Para o vetor A jogue os valores do **primeiro vetor** que sejam menores ou iguais a 112,5; os demais valores coloque em **B**;
- Salve em um vetor **M,** unidimensional, as média dos valores das colunas de A;
- Atualize o vetor B, adicionando 0 nas posições correspondentes às duas primeiras linhas com todas as colunas, e nas duas primeiras colunas com todas as linhas.

Exercícios para diversão

http://www.labri.fr/perso/nrougier/teaching/numpy.100

https://www.machinelearningplus.com/python/101-numpy-exercises-python

Obrigado

altinobasilio@inf.ufg.br

Dúvidas ou sugestões?