Python Básico

Para desfrutar dessas aulas de Python, siga as instruções:

1. Você pode fazer o download e instalar uma distribuição de Python no seu computador. Uma opção é a distribuição gratuita Anaconda Scientific Python. Existem outras opções como Canopy.
2. Você pode rodar Python na nuvem usando Wakari.

Se você decidir trabalhar com sua instalação local de Python, você deverá navegar até um terminal e digitar:

```
ipython notebook
```

Você verá uma janela do navegador como essa aqui. Crie um novo notebook Python 3. Vamos trabalhar!

Bibliotecas

Python é uma linguagem *open-source* de alto nível. Mas o **Mundo Python** é habitado por muitos pacotes e bibliotecas que fornecem operações muito úteis como por exemplo: trabalhar com listas, arrays, plotar funções e muito mais. Nós podemos importar as bibliotecas para tornar nossos programas mais poderosos.

OK! Vamos começar importando algumas biliotecas que podem nos ajudar. Primeiro: vamos importar nossa biblioteca favorita (**Numpy**), que nos fornece operações com *arrays* muito úteis (similares ao MATLAB). Usaremos isso MUITO! a segunda biblioteca que precisamos é a **Matplotlib**, uma biblioteca para criar gráficos 2D.

Execute as primeiras linhas:

In [1]:

```
# <-- comentário em python são escritos utilizando o símbolo sustenido (par
a músicos) ou jogo da velha (para gamers)

import numpy                    # nós importamos a biblioteca dos arrays
from matplotlib import pyplot    # importa a biblioteca de fazer gráficos
```

Nós estamos importanto uma biblioteca chamada `numpy` e estamos importando um módulo chamado `pyplot` de uma grande biblioteca chamada `matplotlib`.

Para usar uma função que pertence a uma dessas bibliotecas, nós temos que informar ao Python onde encontrar a função. Para isso, cada nome de função é escrita seguindo o nome de sua biblioteca, com um ponto no meio.

Se queremos usar a função Nympy `linspace()`, que cria um array com números igualmente espaçados entre um início e um fim, devemos escrever:

In [2]:

```
myarray = numpy.linspace(0, 5, 10)
print(myarray)
```

```
[ 0.          0.55555556  1.11111111  1.66666667  2.22222222  2.77777778
  3.33333333  3.88888889  4.44444444  5.        ]
```

Se nós *não* colocarmos o prefixo `numpy` à função `linspace()`, **Python vai informar um erro**, porque ele não sabe onde encontrar essa função. Tente isso:

In [3]:

```
myarray = linspace(0, 5, 10)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-ed3ba806937a> in <module>()
----> 1 myarray = linspace(0, 5, 10)

NameError: name 'linspace' is not defined
```

Estilo de importação (import style)

Você verá frequentemente trechos de códigos com as seguintes linhas

```
import numpy as np
import matplotlib.pyplot as plt
```

Do que se trata todo esse negócio de `import-as`? É uma maneira de criar um 'atalho' para a biblioteca Numpy e módulo pyplot. Você verá frequentemente esse tipo de uso, mas por enquanto preferimos usar as importações *explícitas*. Acreditamos que ajuda a manter o código mais legível.

dica profissional:

As vezes, você verá pessoas importando uma biblioteca completa usando um atalho para ela (como `from numpy import *`). Isso ajuda a reduzir a digitação, mas pode ser perigoso e o seu código pode gerar confusões as vezes. É melhor começar com bons hábitos desde o começo!

Para aprender novas funções disponíveis, visite a página NumPy Reference. Se você tem proficiência em `Matlab`, existe uma página wiki que pode ser bastante útil para você: NumPy for Matlab Users

Variáveis

Em Python não é necessário declarar os tipos de variáveis, como em C ou Fortran e outras linguagens. Apenas de um valor a uma variável e o Python entende o que você quer:

In [5]:

```
a = 5      # a é o número inteiro 5 (integer)
b = 'cinco' # b é a junção de letras para a palavra 'cinco' (string)
```

```
c = 5.0    # c é o número real 5 (floating point)
```

Pergunte ao Python para te informar qual tipo de variável ele atribuiu para as suas variáveis:

In [6]:

```
type(a)
```

Out[6]:

```
int
```

In [7]:

```
type(b)
```

Out[7]:

```
str
```

In [8]:

```
type(c)
```

Out[8]:

```
float
```

Espaços brancos em Python

Python usa indentação e espaços brancos para agrupar trechos de códigos dentro de loops. Para ver o contraste, se escrevermos um pequeno loop na linguagem C, você poderá usar:

```
for (i = 0, i < 5, i++){
   printf("Olá! \n");
}
```

Em Python não é necessário usar chaves como em C, nosso amigo Python usa indentação em vez sisso; então o mesmo programa acima é escrito em Python da seguinte forma:

In [17]:

```
for i in range(5):
    print("Olá \n")
```

```
Olá

Olá

Olá

Olá

Olá
```

Você percebeu o uso da função `range()`? Ela é uma função "nativa" do Python que nos dá uma

lista baseado em uma progressão aritmética.

Se você tiver aninhado loops `for`, é necessário adicionar uma indentação extra para o loop interno, veja:

In [16]:

```
for i in range(3):
    for j in range(3):
        print(i, j)

    print("Esse texto é dentro no loop i, mas não dentro do loop j")
```

```
0 0
0 1
0 2
Esse texto é dentro no loop i, mas não dentro do loop
j
1 0
1 1
1 2
Esse texto é dentro no loop i, mas não dentro do loop
j
2 0
2 1
2 2
Esse texto é dentro no loop i, mas não dentro do loop
j
```

Cortando arrays

Em NumPy, você pode olhar para porções de arrays da mesma forma que em MATLAB, com alguns segredos extras. Vamos considerar um array de valores de 1 a 5:

In [19]:

```
myvals = numpy.array([1, 2, 3, 4, 5])
myvals
```

Out[19]:

```
array([1, 2, 3, 4,
5])
```

Pytho usa índices começando em zero (como C), o que é uma coisa boa. Sabendo disso, vamos olhar para o primeiro e último elemento do array que criamos acima:

In [20]:

```
myvals[0], myvals[4]
```

Out[20]:

```
(1, 5)
```

Existem 5 elementos no array `myvals`, mas se nós tentarmos olhar para `myvals[5]`, o interpretador do Python ficará triste e **irá nos informar um erro**, pois `myvals[5]` está chamando

um elemento não existente no array, o 6º elemento.

In [21]:

```
myvals[5]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-21-6cc4d3ae83cd> in <module>()
----> 1 myvals[5]

IndexError: index 5 is out of bounds for axis 0 with size 5
```

Arrays também podem ser *cortados* e ainda separar uma parte dos seus valores. Vamos ver os primeiros 3 elementos,

In [22]:

```
myvals[0:3]
```

Out[22]:

```
array([1, 2,
3])
```

Note aqui que o corte é inclusivo no começo e exclusivo no final, então o comando acima nos dá os valores de `myvals[0]`, `myvals[1]` e `myvals[2]`, mas não `myvals[3]`.

Atribuindo variáveis array

Uma das pequenas peculiaridades/características estranhas de Python que as vezes confude as pessoas acontece quando nós atribuímos e comparamos arrays de valores. Veja um exemplo rápido. Começamos definindo um array 1-D chamado a:

In [23]:

```
a = numpy.linspace(1,5,5)
```

In [24]:

```
a
```

Out[24]:

```
array([ 1.,  2.,  3.,  4.,  5.]
)
```

OK, temos um array a, com os valores 1 até o 5. Vamos fazer uma cópica desse array, chamada b, veja:

In [25]:

```
b = a
```

In [26]:

```
b
```

Out[26]:

```
array([ 1.,  2.,  3.,  4.,  5.]
)
```

Ótimo. Então a tem os valores de 1 a 5 e o mesmo ocorre com b. Agora que temos um backup de a, vamos mudar seus valores sem se preocupar em perder dados (ou pelo menos pensamos que não perderemos nada!).

In [27]:

```
a[2] = 17
```

In [28]:

```
a
```

Out[28]:

```
array([  1.,   2.,  17.,   4.,   5.]
)
```

Aqui, o terceiro elemento de a foi mudado para 17. Vamos checar b?

In [29]:

```
b
```

Out[29]:

```
array([  1.,   2.,  17.,   4.,   5.]
)
```

E é assim que as coisas podem dar MUITO errado! Quando você usa atribuições do tipo `a = b`, ao invés de fazer uma cópia de todos os valores de `a` no novo array `b`, o Python apenas cria uma um atalho chamado `b`, e informa a ele para direcionar para `a`. Portanto, se mudarmos o valor de `a`, então `b` vai refletir essa mudança (tecnicamente, isso é chamado de *atribuição por referência*). Se você quer criar uma cópia verdadeira de um array, você deverá informar o Python para criar uma cópia de `a` da seguinte maneira:

In [30]:

```
c = a.copy()
```

Agora, nós podemos tentar mudar um valor em a e ver se as mudanças também ocorreram em c.

In [31]:

```
a[2] = 3
```

In [32]:

```
a
```

Out[32]:

```
array([ 1.,  2.,  3.,  4.,  5.]
```

```
)
```

In [33]:

```
c
```

Out[33]:

```
array([  1.,   2.,  17.,   4.,   5.]
)
```

OK, isso funcionou! Se a diferença entre `a = b` e `a = b.copy()` não está clara, por gentileza leia tudo novamente.
