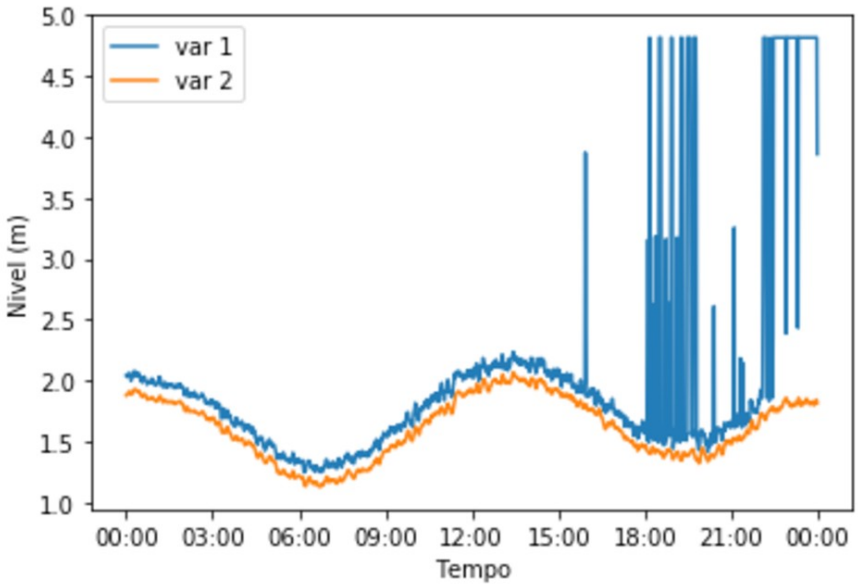
Analisando dados com Python

Fazendo um gráfico de marés observadas pelo RPMG/IBGE.

Parte 1

V2. 2020.01.22



Carlos A.F. Schettini

Laboratório de Oceanografia Costeira e Estuarina

LOCostE / FURG

O Python e o Jupyter já devem estar adequadamente instalados.

Objetivos:

1 Descarregar dados de repositório

2 Carregar no Python/Jupyter

3 Preparar as variáveis

4 Visualizar graficamente

Parte 1 – Dados de maré de um dia para Imbituba

Em termos de análise de dados oceanográficos (físicos) no que diz respeito à visualização, eu colocaria que analisar dados de maré está no nível 'iniciante', pois é uma variável em função do tempo. Depois vem variações espaciais (distribuições em longitude e latitude) ou espaço-temporais (tempo-profundidade, ou outra...), em um nível intermediário. No o nível avançado estamos falando de matrizes 3-D, geralmente de dados de modelos ou sequências de imagens de satélite. E acima disso tem o nível 'guru' que é quando trabalhamos com medições móveis, onde x,y,z e t são variáveis, como campanhas com ADCP móvel. Isso na área da oceanografia física. Creio que só na geofísica (sísmica/sonografia) tenha coisas mais complicadas...

Dados de maré são registros do nível do mar medidos em intervalos regulares de tempo com algum tipo de eliminação de dados de alta frequência gerados por ondas de vento. A eliminação pode ser por um filtro mecânico (o sensor fica em um tubo conectado ao ambiente por um poro que limita a entrada e saída de água) ou digital (com o registro de dados em intervalos de 1 s (1 Hz → ciclos por segundo), a partir dos quais é obtido um valor médio de 1 minuto ou mais, em intervalos de 10 minutos ou mais.

Há dúzias de locais na internet onde é possível obter dados de maré para muitos locais do mundo, especialmente nos EUA e Europa. Aqui no Brasil nós temos algumas opções. Uma delas é o Instituto Brasileiro de Geografia e Estatística (IBGE) que mantém a Rede Maregráfica Permanente para Geodéisa (RMPG). Acessem o site em leiam o conteúdo, em especial do item 'Sobre a publicação'.

Uma vez no site, investigando as possilidades, tentativa e erro, vai descobrir como baixar dados. Para este exemplo eu baixei um arquivo 'zip' com o nome 'IMB201014'. Geralmente os nomes de arquivos baixados de repositórios são em código. Neste, 'IMB' é a abreviação de Imbituba, onde está a estação que eu escolhi para baixar os dados, e '201014' possivelmente indica os dois últimos dígitos do ano, mês e dia. Digo possivelmente pois não tenho certeza uma vez que não abri os dados.

Uma vez que baixou o arquivo no seu computador uma cópia do arquivo está salva na memória sólida e em um algum diretório (pasta!) do sistema operacional (Windows 10? Downloads?). Eu recomendo criar um diretório de trabalho onde irá adicionar e organizar suas coisas. Por exemplo, eu tenho um diretório no meu computador como 'c:\guto', e aí eu tenho sub-diretórios para outras coisas. Isso é importante porque eu sei (quase sempre!) exatamente estão as coisas. Crie seu diretório (ex: c:\fulano), e crie um sub-diretório específico para esta atividade (ex: c:\fulano\mare_ibge), e mova o arquivo para lá. Ao dar nome para diretórios e sub-diretórios, recomendo fortemente não usar espaços em branco, não começar nomes com números e nem usar caracteres especiais (ç, ã, é, ü, etc.). Há programas que não 'entendem' isso, então evite problemas futuros!

Gerenciamento de arquivos no seu computador é MUITO IMPORTANTE! Pra não dizer, básico... Não deixe que o sistema operacional faça isso por você! Para isso eu gosto de usar o programa Total Commander (https://www.ghisler.com/), que é gratuito e ótimo para isso. Ele sempre mostra sempre a pasta de origem e de destino uma do lado da outra. E com muitas funcionalidades que geralmente são difíceis de achar no Explorer do Windows. Recomendo!

Voltando ao arquivo, que já movi para o meu diretório... ele está compactado como 'zip'. O Total Commander facilmente me mostra o conteúdo dos arquivos dentro dele. Neste caso há somente o arquivo 'IMB201013.TXT'. Para extrair o arquivo pelo Total Commander basta copiá-lo (tecla de função F5) para o diretório de destino, que é o próprio diretório que está o arquivo 'ZIP'.

A extensão 'TXT' sugere que é um arquivo no padrão ASCII (American System Code for Information Interchange). Se realmente for, eu posso abrir com qualquer editor de texto, como o Bloco de Notas (Figura 1). O meu sistema associa a extensão 'TXT' diretamente com o Bloco de Notas, o que permite que com um duplo clique do mouse ele abra automaticamente. Se o seu sistema for diferente, uma solução é abrir o Bloco de Notas e depois procurar e abrir arquivo. Feito isso, veremos a janela do Bloco de Notas com o conteúdo do arquivo (Figura 1)

Figura 1: Janela do programa Bloco de Notas com o conteúdo do arquivo 'IMB201013.TXT'.

O que vejo? Números, barras '/', símbolos de dois pontos ':' e vírgulas ','. Para analisar dados, números são bons, mas tudo o mais é ruim! E, em termos computacionais, o separador decimal é o ponto '.', e não a vírgula! Esse último detalhe causa certo incômodo, pois o sistema operacional adapta muitos aplicativos (Excel, Word, ...) para usar o padrão regional, que no Brasil é a vírgula. Para quem vai se dedicar à análise de dados, uma coisas a fazer é mudar isso no seu sistema operacional e determinar que o separador decimal seja o ponto '.', lá nas configurações regionais! Ocasionalmente usamos o Excel para verificar algum arquivo antes de ir para o Python, e se o Excel usar o separador ',', vai dar confusão! Então, novamente, se poupe de problemas futuros.

Essa primeira inspeção dos dados é importante. Agora sabemos que estão em padrão ASCII, pois abriu no Bloco de Notas, e tem essas coisas lá... Não procurei, mas imagino que no site do IBGE deva ter algum metadado (que descreve os dados) descrevendo o padrão utilizado. Mas aqui é óbvio que primeiro vem a data '14/10/2020' seguido da hora '00:00:00' e de outras duas colunas que devem ser dados de nível da água (maré!) talvez em dois níveis de referência (ache os metadados e confirme!). As colunas estão separadas por espaço, e isto é uma informação importante. Poderia ser por vírgula ou ponto e vírgula ou outra coisa... A data pode ser uma fonte de problema, pois não há um padrão internacional. Intuitivamente, aqui parece ser dia/mês/ano, mas como sabemos que não é mês/dia/ano? Porque não temos 14 meses, lógico! Mas e se fosse? 8/9/2020? E 12/11/09? Lembre-se que nos EUA o padrão comum é usar o mês antes do dia 'March 10', e eles usam 03/10, o que para nós pareceria 3 de outubro. Por isso os metadados são muito importantes, e também, muitas vezes, um cabeçalho! Como padrão pessoal, eu uso 2020 10 14, ano, mês dia, e separado por espaço. Ou 20201014 no nome de um arquivo.

No bloco de notas também é interessante, como é a primeira vez que está trabalhando com estes dados, de dar uma olhada em todo arquivo até o final, para ver se todo ele está neste padrão. Normalmente sim, mas sempre pode haver uma surpresa...

Ok. Temos um arquivo ASCII, sabemos exatamente onde ele está e temos uma ideia do seu conteúdo (números e símbolos). Podemos ir ao Jupyter. Isso é outra coisa legal do Total Commander é que ele abre o 'prompt' diretamente na pasta que está. Em 'Commands' use 'Open command prompt window', e abrirá a janela de comando na pasta de trabalho. Então inicie uma instância do Jupyter (digitando '`jupyter lab`' na linha de comando) (Figura 2). No Jupyter,

comece um novo notebook do python clicando no ícone do python que está abaixo de ‘Notebook’ (Figura 3).

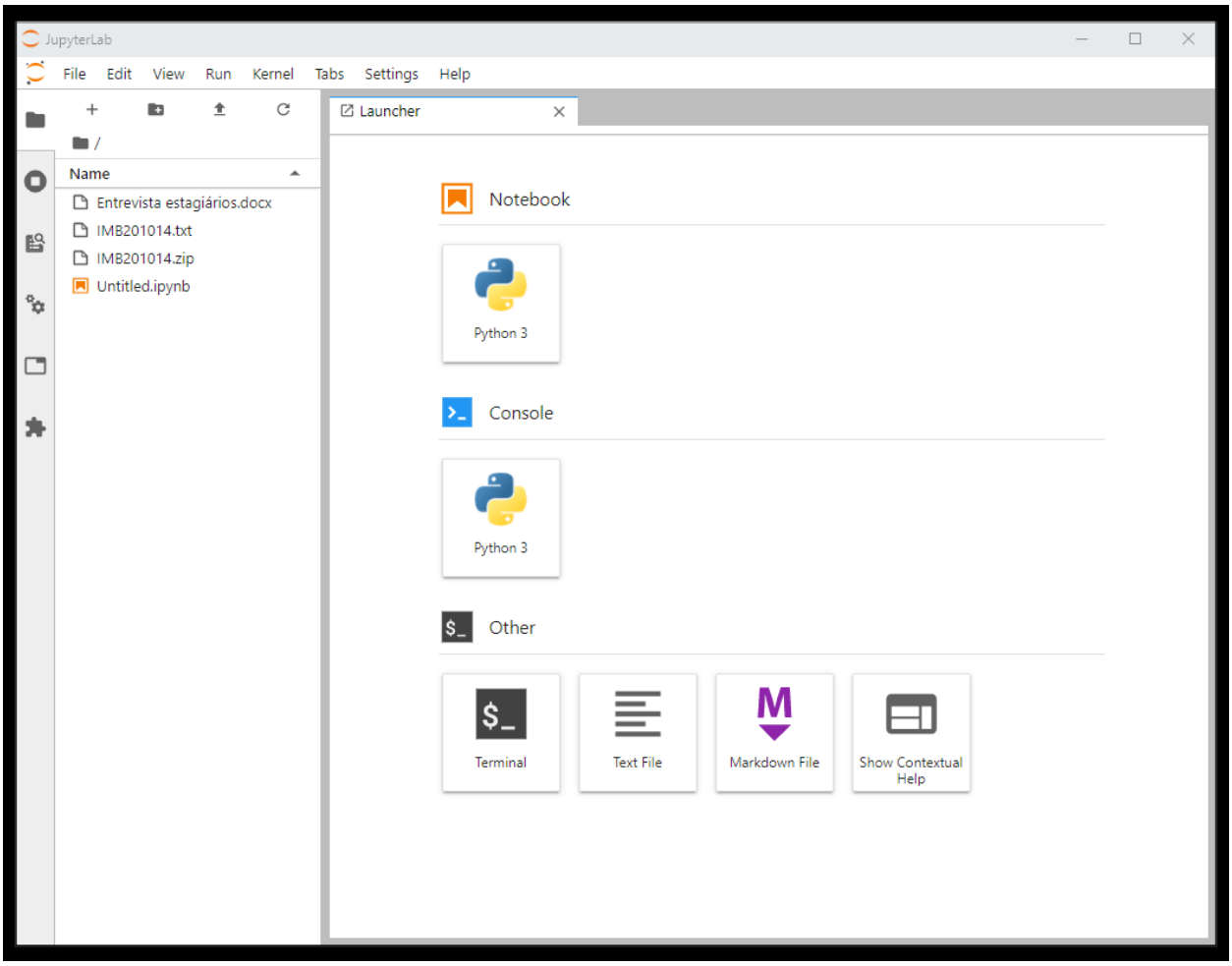


Figura 2: Janela inicial do Jupyter Lab. Isto é uma janela do Google Chrome, mas não deve ser idêntica à sua. A minha foi configurada para abrir sem as opções de endereço e ferramentas do Google Chrome. Se gostou, procure nos fóruns para descobrir como deixar assim.

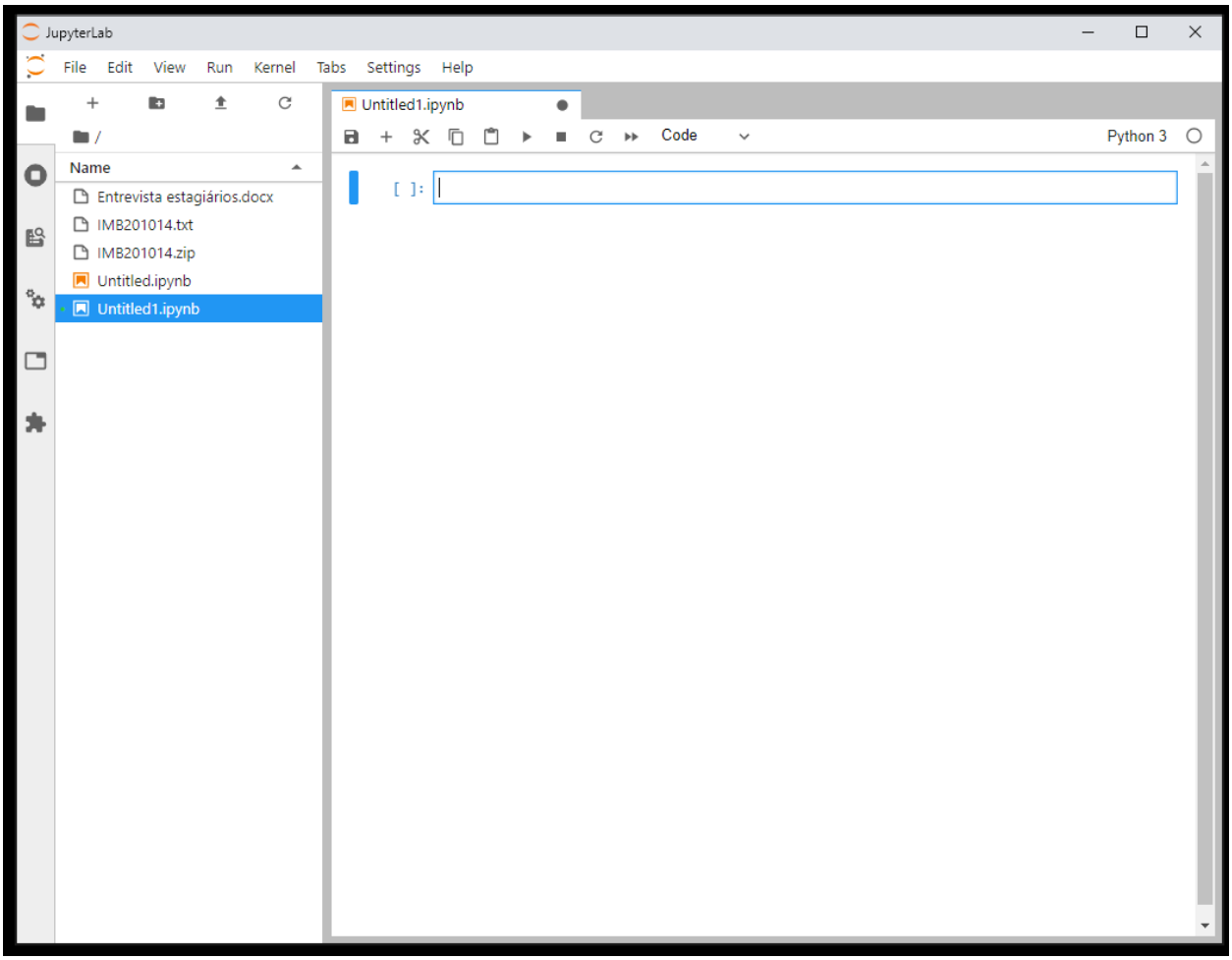


Figura 3: Janela do Jupyter Lab com um notebook novo.

Assistam tutoriais no Youtube para aprender o básico do Jupyter Lab. Há dúzias... assista mais de um e mais de uma vez aquele que achou melhor. E com o uso frequente isso se torna natural!

Pra começar, mude a primeira célula para Markdown, e coloque um cabeçalho (Figura 4). Cabeçalhos são importantes! Indique inicialmente do que se trata o notebook, digo, o tema principal. E depois em letras menores documente. Data, projeto, quem fez. Isso é essencial

especialmente quando esse notebook irá passar por outras mãos, e inclusive pelas suas próprias daqui a um ano ou dois. Isso vai ajudar a ‘entrar no clima’ do que vem abaixo!

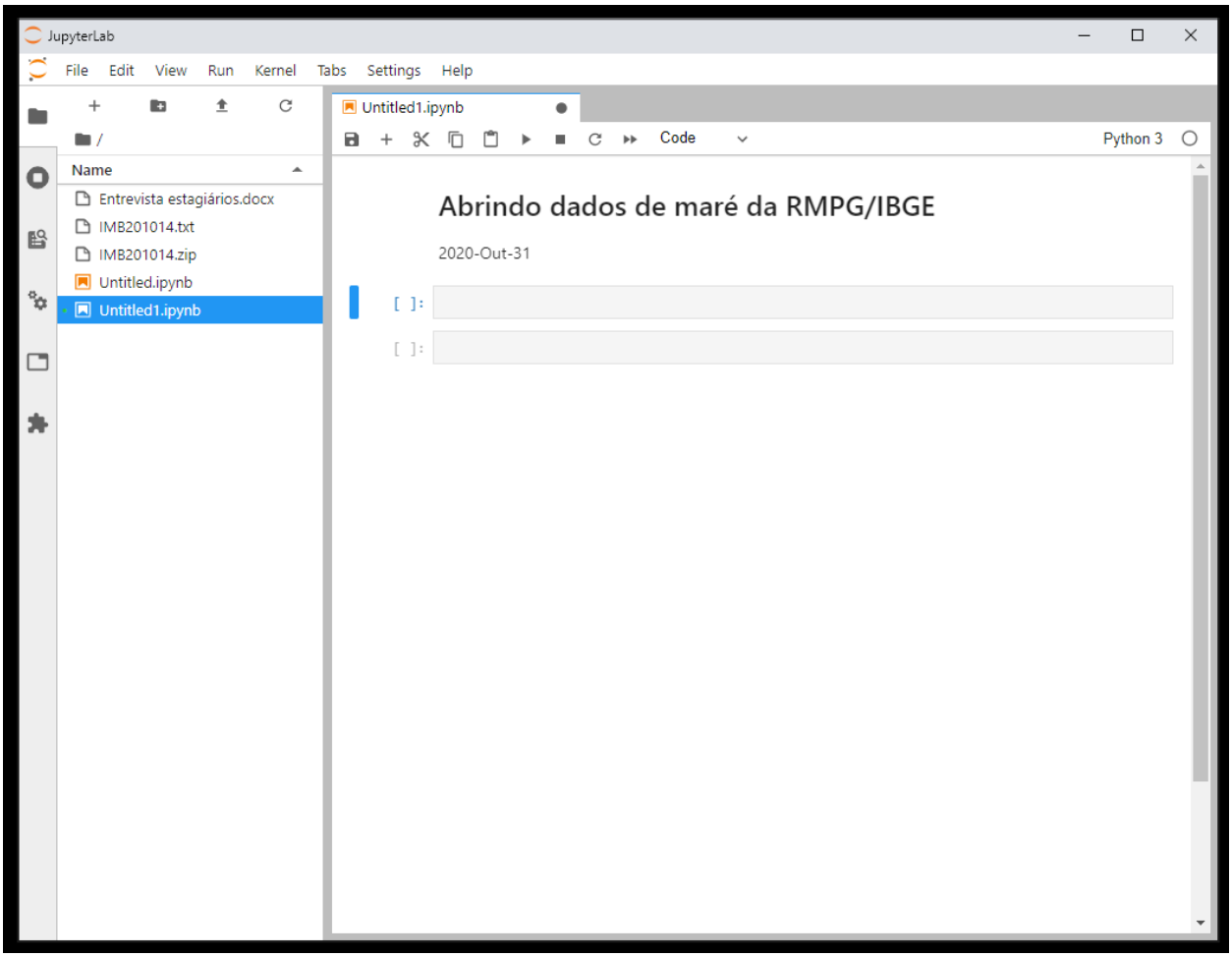


Figura 4: Exemplo de cabeçalho em Markdown no notebook.

Em seguida vamos carregar os pacotes datetime, numpy e matplotlib.pyplot na célula seguinte (Código 1)

```
[2]: import datetime
     import numpy as np
     import matplotlib.pyplot as plt
```

Código 1: Célula com importação de pacotes.

Porque estes pacotes? [Assista alguns tutoriais sobre pacotes no python.] O numpy é que dá o carácter matemático/algébrico ao python. O matplotlib é que permite fazer gráficos. O datetime ajuda a trabalhar com o tempo. Saber trabalhar com o tempo é extremamente importante para quem trabalhar com variáveis temporais (nós!), e é mais complicado do que tu pode inicialmente imaginar! Voltaremos a este assunto adiante.

Agora temos que decidir a abordagem que adotaremos para abrir este arquivo. Não há uma solução única... se procurar nos fóruns (StackOverflow, Github, youtube, etc...) vai achar soluções diferentes, usando funções e módulos diferentes.

Em termos computacionais, eu tenho que separar os elementos da data e da hora, bem como substituir a vírgula por ponto. Dá pra fazer no Excel? Dá. Mas nosso objetivo é fazer no python. A primeira vez é lenta, e seria mais fácil no Excel e talvez até fosse o caso se quiséssemos usar só um dia de dados. Mas se eu quiser juntar um ano de dados, teria que fazer isso 365 vezes no Excel. Fica melhor então automatizar o processo, e aí que entra a vantagem da programação.

Então, como Jack, por partes... embora o texto fique longo, eu vou descrever como eu estou fazendo para vocês terem ideia do processo! Eu já fiz isso mas não lembro exatamente como. Lembro que tinha que usar um comando '`with`' e ler linha por linha e depois separar o conteúdo. Então, pra começar, pesquisamos no Google com os termos: **'Python read file line by line with'** [Obs: procurar em português talvez funcione... mas o mundo da computação é

prioritariamente em inglês!]. Dos resultados, geralmente eu opto primeiro para ver o que tem no Stackoverflow na primeira opção. Ver as datas das perguntas é bom porque as vezes uma pergunta/resposta de 2010 pode não funcionar na versão atual! Procure as respostas mais recentes, do tipo, 2015 em diante. Mas isso não é uma regra! Entrei em *https://stackoverflow.com/questions/3277503/how-to-read-a-file-line-by-line-into-a-list*

Temos a pergunta **How to read a file line-by-line into a list?** que deve dar alguma luz. Na verdade eu não dou muito importância para o que quem está perguntado escreveu, mas para as respostas, mas não só a primeira pois as outras podem ter a solução mais apropriada do que precisamos. As vezes é a última e menos votada que fornece o que precisamos. E ali está o que precisamos (Figura 5).

Using `with` and `readlines()` :

```
with open('file.txt') as f:
    lines = f.readlines()
```

Figura 5: Extração de trecho do Stackoverflow.

Podemos copiar/colar este pedaço de código e colar no nosso notebook. Intuição e inglês ajudam... substituimos **'`filename`'** pelo nosso arquivo e em ao invés de usar **'`lines`'** vou usar **'`linhas`'** para facilitar. Mudar o nome do arquivo é obrigatório, mas mudar de *lines* para *linhas* é apenas estético. 6 = ½ dúzia. Lembrando aqui que a indentação é necessária! O python usa isso para saber o que está ***dentro*** do '`with`'. Dentro, aqui, refere-se ao código que executado pela operação '`with`'. E usamos também esta expressão ***dentro*** para o '`for`', '`while`' e '`if`'.

Neste caso, o arquivo será aberto (`open`) e atribuído à variável (objeto) '`f`' (de file!). E, dentro do '`with`', a variável (objeto) '`linhas`' irá conter o conteúdo de '`f`' inserido pelo método '`readlines()`', que faz isso linha por linha (Código 2).

```
[3]: with open('imb201014.txt') as f:

         linhas = f.readlines()
```

Código 2: Abrindo um arquivo e lendo linha a linha.

Executo a célula (rodo! = pressiono junto as teclas <ctrl> + <enter>). Aparentemente não aconteceu nada, mas o número entre [ ] ao lado da célula apareceu ou mudou. Significa que não deu erro, o que é bom sinal. Adicione uma nova célula de código, e use o comando '`who`' (seguido de <ctrl> + <enter>!). Vai aparecer as variáveis do ambiente, como os apelidos dos módulos que carregamos (np, plt) e outras coisas. Mas entre estas outras coisas estão 'f' e 'linhas', que são variáveis criadas na execução do código. Se usar o comando '`whos`', é parecido, mas aparece também o tipo e informações adicionais. Note que para a variável 'linhas' aparece 'n=1444', e isto indica que a variável 'linhas' contém 1444 coisas. Se usar a função '`len()`', de length, vai indicar a mesma coisa. O número 1440 lhe diz alguma coisa? Os dados são para um dia, e 1440/24 = 60, ou, um dado por minuto durante 24 horas.

Dando um '`print(linhas)`' irão ver que os dados estão lá, porém todos juntos. Já é algo. Reparem que cada linha dos dados do arquivo está entre aspas '' o que indica que está sendo tratada como uma 'string', e não como números ('float' ou 'int'), e separado da próxima linha por vírgula. Notem que no final de cada sequencia de caracteres que forma uma linha tem

um ‘\n’. Isso é um indicador de final de linha, e por vezes é um causador de problema! Ok, mas como separar tudo? Agora temos que pegar linha por linha, ou seja, uma nova variável, e a partir dela ir quebrando a string para pegar os números que nos interessam. Para isso, fazemos uma repetição (looping) com o ‘`for`’, que precisa de uma variável auxiliar que ira assumir o conteúdo de cada elemento que está na variável ‘`linhas`’ (Código 3). O ‘x = 1’ é porque tem que ter algo dentro do ‘`for`’, senão dá erro. Nesse código, será atribuído o valor 1 à variável x 1440 vezes.

```
[18]: for linha in linhas:
          x = 1
```

Código 3: Uso elementar do looping 'for'

Então, fazendo por partes (Ripper, 1888), vamos pegar só a primeira linha. Fazemos o processo com ela. Depois, quando o assunto estiver dominado, fazemos com as demais. Par isso podemos interromper o ‘`for`’ usando uma condicionante ‘`if`’ e um contador ‘`c`’ (Código 4).

```
[21]: c = 0
      for linha in linhas:
          if c == 0:
              break
          c += 1   # isso é igual a   c = c + 1
```

Código 4: O loop com contador interrompido com break.

Eu crio uma variável de contagem ‘c’, e atribuo o valor zero. Dentro do looping eu atribuo a soma de 1 em cada ciclo (c += 1). Se não colocar o teste para interromper o resultado será c = 1440. Para interromper, eu faço o teste com uma condicionante, que se verdadeira, executará o que está dentro do teste. Neste caso, a palavra-chave ‘`break`’. E, muito importante, a notação. Na maior parte das linguagens o símbolo ‘=’ é uma atribuição (ex: x = 2), e a identidade é designada como ‘==’ (dois = juntos), as vezes chamado de *absolutamente igual*. E, (mais um) diferente da maioria das linguagens, o Python não indica explicitamente o fim do looping ou do teste! Isso fica a carga da indentação! Isso torno o cógido mais limpo e fácil de entender.

Como o ‘`for`’ rodou só uma vez, então ‘`linha`’ contem o conteúdo do primeiro elemento de ‘`linhas`’. Em uma outra célula podemos explorar o conteúdo de ‘linha’, e saberemos o que tem nela, que tipo de variável (uma string) e quantos caracteres tem (32) (Codigo 5).

```
[24]: print(linha)
      print(type(linha))
      print(len(linha))
```

```
14/10/2020 00:00:00 2,042 1,880

<class 'str'>
32
```

Código 5: Conteúdo e características de 'linha'.

Até aqui demos dois passos muito importantes. Primeiro, conseguimos carregar o conteúdo do arquivo no Python/Jupyter. E segundo, conseguimos pegar uma linha do arquivo. Mudando o teste, poderíamos selecionar qualquer linha. Experimente.

Uma vez que já sabemos como pegar uma linha somente, podemos separar o seu conteúdo. Como teremos uma ação de separar, vamos ter que colocar o resultado em algum lugar. Então criamos outra variável, '`linha_quebrada`' para isso. Vamos usar método do Python '`split()`' [o que seria um método do Python?] para separar a string. Lembrando aqui que para acessar a documentação e ajuda, em uma célula em branco podemos escrever '`split()? +<shift>+<enter>`'. A string será separada usando os espaços vazios para delimitar sub-conjuntos (Código 6).

```
[26]: c = 0

for linha in linhas:

    linha_quebrada = linha.split()

    if c == 0:

        break

    c += 1   # isso é igual a   c = c + 1

print(linha_quebrada)
print(type(linha_quebrada))
print(len(linha_quebrada))
```

```
['14/10/2020', '00:00:00', '2,042', '1,880']
<class 'list'>
4
```

Código 6: Separando o conteúdo de 'linha' com split().

Enquanto '`linha`' é uma string, '`linha_quebrada`' é uma lista que tem 4 elementos, sendo cada elemento uma string. E podemos pegar cada elemento separadamente (Código 7)

```
[31]: print(linha_quebrada[0])
      print(linha_quebrada[1])
      print(linha_quebrada[2])
      print(linha_quebrada[3])
```

```
14/10/2020
00:00:00
2,042
1,880
```

Código 7: Elementos de 'linha_quebrada'

Pegar um subconjunto de um conjunto é chamado de fatiamento (*slicing*). Cada elemento da lista tem um índice que indica seu endereço dentro da lista, começando por 0, 1, 2, etc... Isso é uma coisa que pode confundir um pouco, pois o Python usa como primeiro índice o zero (0), enquanto que outras linguagens (R, Matlab) usam o um (1) . Tipo, se eu quiser pegar o 4º elemento da string, o seu índice é 3!

Agora podemos criar outras variáveis para conter as quatro strings de ‘linha_quebrada’. Mas fazendo isso, já podemos ir adiante e usar o método ‘`split()`’ para separar os elementos das strings que contém a data e a hora, e o método ‘`replace()`’para trocar a vírgula pelo ponto. As variáveis ‘`ldata`’ e ‘`lhora`’ (o ‘l’ de lista) agora contém listas de strings, e as variáveis ‘`var1`’ e ‘`var2`’ contém uma única string, agora com ‘.’ no lugar da ‘,’ (Código 8)

```
[36]: c = 0

      for linha in linhas:

          linha_quebrada = linha.split()

          ldata = linha_quebrada[0].split('/')
          lhora = linha_quebrada[1].split(':')
          var1 = linha_quebrada[2].replace(',', '.')
          var2 = linha_quebrada[3].replace(',', '.')

          if c == 0:

              break

          c += 1   # isso é igual a   c = c + 1

      print(ldata)
      print(lhora)
      print(var1, type(var1))
      print(var2, type(var2))
```

```
['14', '10', '2020']
['00', '00', '00']
2.042 <class 'str'>
1.880 <class 'str'>
```

Código 8: Separando o conteúdo de 'linha_quebrada'.

Tocando o barco... agora temos que separar mais uma vez os elementos de data e hora. E, também informar ao Python que isto tudo são números e não strings! Em computação, por uma questão de eficiência, as variáveis que contém números podem ser do tipo inteiro ‘int’ ou do tipo ponto flutuante ‘float’, que são simplesmente os números com um ‘.’ flutuando (!) em algum lugar, para não complicar o assunto. Temos criar variáveis para cada coisa! E quando fizermos isso já podemos fazer a conversão. As strings que contém as informações de data e hora serão transformadas em ‘int’, e as variáveis 1 e 2 em ‘float’. E, tendo já tudo isso separado, podemos montar uma nova linha (depois de desmontar tudo?) com os dados numéricos (Código 9)!

```
[46]: c = 0

for linha in linhas:

    linha_quebrada = linha.split()

    ldata = linha_quebrada[0].split('/')
    lhora = linha_quebrada[1].split(':')
    var1 = linha_quebrada[2].replace(',', '.')
    var2 = linha_quebrada[3].replace(',', '.')

    dia = int(ldata[0])
    mes = int(ldata[1])
    ano = int(ldata[2])
    hora = int(lhora[0])
    minuto = int(lhora[1])
    segundo = int(lhora[2])
    var1 = float(var1)
    var2 = float(var2)

    nova_linha =[ano, mes, dia, hora, minuto, segundo, var1, var2]

    if c == 0:

        break

    c += 1   # isso é igual a   c = c + 1

print(nova_linha)
```

```
[2020, 10, 14, 0, 0, 0, 2.042, 1.88]
```

Código 9: Shiva! Desmontando e remontando!

Ok. Mais um passo importante concluído. Agora, temos que fazer isso para todas as linhas do arquivo de entrada. Isto implica também que teremos que armazenar o que for feito. Cada looping criará uma 'nova_linha', mas isso vai acontecer apagando a 'nova_linha' do looping anterior, e no final teremos apenas a última 'nova_linha'. Criamos então uma lista vazia antes do looping 'guarda_linhas', e no final do processamento usamos o método '`append()`' que ira incrementar o conteúdo de 'guarda_linhas' com o conteúdo de 'nova_linha' que será atualizado a cada ciclo. Mas antes de abrir a porteira, vamos verificar o que vai acontecer quando mudarmos o teste de 0 para 1 e executar com 2 linhas (Código 10).

```
[52]: c = 0
guarda_linhas=[]

for linha in linhas:

    linha_quebrada = linha.split()

    ldata = linha_quebrada[0].split('/')
    lhora = linha_quebrada[1].split(':')
    var1 = linha_quebrada[2].replace(',', '.')
    var2 = linha_quebrada[3].replace(',', '.')

    dia = int(ldata[0])
    mes = int(ldata[1])
    ano = int(ldata[2])
    hora = int(lhora[0])
    minuto = int(lhora[1])
    segundo = int(lhora[2])
    var1 = float(var1)
    var2 = float(var2)

    nova_linha =[ano, mes, dia, hora, minuto, segundo, var1, var2]

    guarda_linhas.append(nova_linha)

    if c == 1:

        break

    c += 1   # isso é igual a   c = c + 1

print(guarda_linhas)
print(type(guarda_linhas))
```

```
[[2020, 10, 14, 0, 0, 0, 2.042, 1.88], [2020, 10, 14, 0, 1, 0, 2.032, 1.885]]
<class 'list'>
```

Código 10: Testando o código.

O resultado é uma lista de listas. Mas o mais importante é que agora são listas de números! Não precisamos mais do contador nem do teste, e o código funcional fica assim (Código 11):

```
[56]: guarda_linhas=[]

for linha in linhas:

    linha_quebrada = linha.split()

    ldata = linha_quebrada[0].split('/')
    lhora = linha_quebrada[1].split(':')
    var1 = linha_quebrada[2].replace(',', '.')
    var2 = linha_quebrada[3].replace(',', '.')

    dia = int(ldata[0])
    mes = int(ldata[1])
    ano = int(ldata[2])
    hora = int(lhora[0])
    minuto = int(lhora[1])
    segundo = int(lhora[2])
    var1 = float(var1)
    var2 = float(var2)

    nova_linha =[ano, mes, dia, hora, minuto, segundo, var1, var2]

    guarda_linhas.append(nova_linha)
```

Código 11: Código funcional.

Percebam que o processo não é direto... a gente vai verificando cada passo o que está sendo produzido, e quando necessário adicionamos coisas que serão removidas depois. O que é ***EXTREMAMENTE, IMPERIOSAMENTE, ULTRAMEGA IMPORTANTE***, é que temos certeza do que teremos no final. E isso é totalmente responsabilidade sua! Você fez o código!

Até agora usamos somente o Python nativo, e nada dos pacotes que carregamos inicialmente. Aqui entra o numpy e seus módulos que tornam muito mais fácil trabalhar com números. Primeiramente convertemos nossa lista de listas em um objeto ARRANJO (array) do numpy. O jargão é esse mesmo... outras linguagens usam vetores e matrizes. Com o numpy é arranjo. Em outra célula do Jupyter, vamos converter 'guarda_linhas' em um arranjo, e para isso vamos criar outra variável. Poderia até ser a mesma, digo, dar o mesmo nome (ex: `x =`

`np.array(x)`), mas assim ficará melhor (Código 12). Não há necessidade de sermos econômicos com variáveis. A menos que seu computador tenha 2 GB de memória RAM com Windows 10.

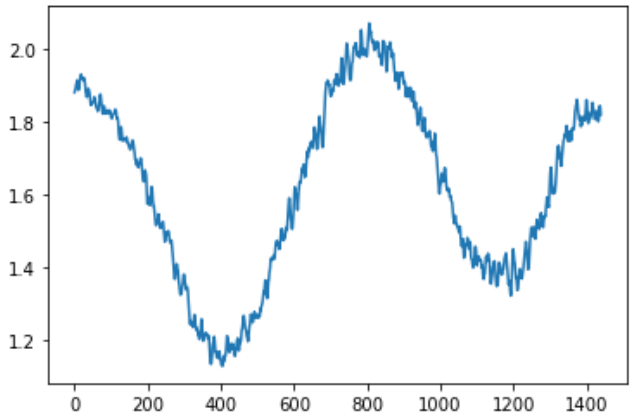
```
[65]: dados = np.array(guarda_linhas)
      print(type(guarda_linhas))
      print(type(dados))
      print(dados.shape)
      print(len(dados))
```

```
<class 'list'>
<class 'numpy.ndarray'>
(1440, 8)
1440
```

Código 12: Convertendo uma lista para um arranjo do numpy.

Aqui vemos que 'guarda_linhas' é uma lista (já sabíamos!), 'dados' é um arranjo do numpy, e que o '`len()`' nativo do python nos dá o número de linhas somente, e o '`shape()`' do numpy fornece o número de linhas e colunas, mas não funciona com listas. Agora que temos uma matriz (que é um arranjo 2D do numpy!), podemos pegar uma das colunas e fazer um gráfico usando o pacote matplotlib (Código 13).

```
[66]: fig, ax = plt.subplots()
      ax.plot(dados[:,7])
      plt.show()
```



Código 13: um gráfico!

Fazer gráficos é essencial! Gráficos bonitos, elegantes e informativos. É como sintetizamos milhares de números em uma informação gráfica que pode ser mais facilmente assimilada, e assim identificar padrões (ou a falta deles!). Por exemplo, a figura acima apresenta a mesma informação contida na última coluna (`dados[:,7]`) do arquivo original. Abra o arquivo e olhe para ele, e olhe para o gráfico. Não preciso dizer mais né. Visite o site do matplotlib e dê uma olhada na galeria de gráficos (https://matplotlib.org/gallery.html)!

Mas nosso gráfico ainda não está completo. O que são os eixos X e Y? O eixo Y é referente aos dados da coluna 8 de ‘dados’ (mas índice 7), e o eixo X são os índices (ou número de linhas, mas começando de 0!). Para o gráfico ficar fisicamente preciso, o X tem que representar o tempo.

O tempo... esse é um tópico meio complicado para qualquer linguagem de programação! O tempo pode ser definido com um vetor que aponta para o futuro, e é contínuo. Mas nós segmentamos o tempo e usamos vários componentes para indicá-lo. E, adicionalmente, esses componentes são fisicamente independentes entre si (ano, mês e dia)! Entendemos por ano o período de translação da Terra. Dia é o período de rotação da Terra. O mês é uma coisa intermediária entre um e outro, mas conceitualmente é o período de revolução da Lua em torno da Terra (em torno do centro de gravidade do sistema Terra-Lua, pra ser mais preciso!). O ano tem 365 dias e uns quebrados (quase 1/4 de dia), e para acertar os quebrados, adicionamos um dia a cada 4 anos, os anos bisextos. Os meses podem ter 28, 29, 30 ou 31 dias, dependendo se o ano é ano bisexto ou não. Quanto ao dia, alguém decidiu dividir em 24 segmentos iguais (egípcios, há muito tempo!), as horas. E estes segmentos foram divididos ainda em 60 partes iguais (babilônios, há muito tempo!), os minutos, e estes em mais 60, os segundos. O segundo não são mais divididos mas podem ter décimos, centésimos, etc. Isso chamamos de calendário Gregoriano, que foi um papa que organizou o tempo, ou pelo menos a cronologia, em 1582! Outra maneira de medirmos o tempo, que é mais coerente, é usando dias julianos (de Júlio César), que contam os dias a partir de uma referência no passado. Façam um tour na Wikipédia!

Para lidar com o tempo no Python há diferentes opções. Eu prefiro usar o pacote **datetime**, embora o numpy tenha o módulo **datetime64** para fazer isso. Questão de gosto. Para usar o tempo que temos no arquivo, que agora está em números na variável ‘dados’, temos que montar um objeto ‘`datetime`’. Há formas diferentes de fazer isso! Aqui vai uma maneira bem explícita! Similar ao que fizemos antes, temos que rodar um looping para montar o elemento ‘`datetime`’ para cada linha (1440) de dados. E, temos que acumular isso. Na montagem do

elemento vamos usar a variável ‘monta_tempo’ com o a função datetime do pacote datetime (confuso?) que converterá os valores que estão contidos em dados. Porém o datetime só funciona com números inteiros. Mesmo que a gente tenha transformado os valores de data e hora em inteiros lá em cima, quando fizemos a conversão para arranjo do numpy, tudo virou float. Isso porque as variáveis var1 e var2 eram do tipo float, e o numpy padroniza tudo deste modo (Código 14).

```
[84]: tempo = []
      for i in range(len(dados)):
          monta_tempo = datetime.datetime(int(dados[i,0]),
                                          int(dados[i,1]),
                                          int(dados[i,2]),
                                          int(dados[i,3]),
                                          int(dados[i,4]),
                                          int(dados[i,5]))

          tempo.append(monta_tempo)

      print(tempo[0])
      print(type(tempo))
      print(len(tempo))
```

```
2020-10-14 00:00:00
<class 'list'>
1440
```

Código 14: Criando objeto datetime.

Como assim? E os ‘-’ e os ‘:’? Depois de tanto trabalho para tirar as ‘/’ e os ‘:’? Pois é... mas antes eram strings, e agora fazem parte de um objeto `datetime` que o Python interpreta como sendo ano, mês, dia, hora, minuto e segundo! Antes era apenas uma sequencia de caracteres. Daria para fazer de outro modo? Sim, inclusive bem mais fácil. Descubra! Eu prefiro deste jeito, porque é o jeito que eu entendo e tenho certeza do que está acontecendo. Há pacotes que tornam o processo mais amigável (o pacote pandas), porém mais obscuro. Repare também de

como o código foi organizado no '`for`'. O Python permite isso para tornar o código mais legível.

Agora com a variável 'tempo' que é um objeto que contém nossos dados de referência temporal, podemos voltar ao gráfico, incluindo alguns refinamentos. No caso, vamos fazer o gráfico com as variáveis 1 e 2, já que não sabemos exatamente o que são. E vamos formatar a legenda do tempo para que apareça tipo hora:minuto, bem como incluir as legendas dos eixos (Código 15).

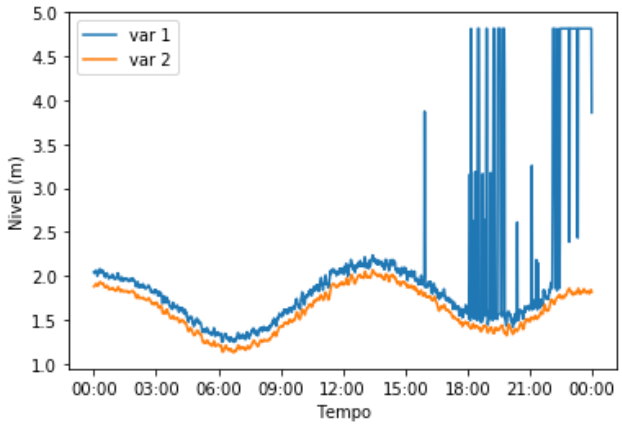
[96]:
```
fig, ax = plt.subplots()
ax.plot(tempo,dados[:,6], label='var 1')
ax.plot(tempo,dados[:,7], label='var 2')

formato = mdates.DateFormatter('%H:%M')
ax.xaxis.set_major_formatter(formato)

ax.set_xlabel('Tempo')
ax.set_ylabel('Nivel (m)')

plt.legend()

plt.show()
```



Código 15: Gráfico referenciado temporalmente.

Deu erro? Certamente. Para formatar as legendas dos ‘ticks’ do eixo X nós precisamos dos módulos ‘dates’ do matplotlib. Então temos que voltar lá no início onde fazemos a importação de pacotes e incluí-lo (Código 16)

```
[91]: import datetime
      import numpy as np
      import matplotlib.pyplot as plt
      import matplotlib.dates as mdates
```

Código 16: importando o matplolib.dates.

Agora sim vai funcionar. Há uma infinidade de opções gráficas no matplotlib. Com o tempo vai aprender e memorizar as mais importantes e rotineiras.

Agora que temos um gráfico com eixos identificados, vem a parte da interpretação. Estou vendo registros maregráficos. Em que os sinais são similares? E em que são diferentes? O que deve ter acontecido com o sinal azul? Qual a altura da maré neste dia em Imbituba? Quantas ondas de maré ocorrem por dia? Etc...

Anexos:

Segue cópia de todo o notebook depurado. Ou seja, só o que funciona! Mas tenham em mente que isto é uma entre tantas maneiras de fazer isto. O Python tem muitos recursos que permitem sintetizar vários processos, o que aumenta a eficiência. Por outro lado, diminui a legibilidade.

Abrindo dados de maré da RMPG/IBGE

2020-Out-31

[1]:
```
import datetime
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
```

[2]:
```
with open('imb201014.txt') as f:

    linhas = f.readlines()
```

[3]:
```
guarda_linhas=[]

for linha in linhas:

    linha_quebrada = linha.split()

    ldata = linha_quebrada[0].split('/')
    lhora = linha_quebrada[1].split(':')
    var1 = linha_quebrada[2].replace(',', '.')
    var2 = linha_quebrada[3].replace(',', '.')

    dia = int(ldata[0])
    mes = int(ldata[1])
    ano = int(ldata[2])
    hora = int(lhora[0])
    minuto = int(lhora[1])
    segundo = int(lhora[2])
    var1 = float(var1)
    var2 = float(var2)

    nova_linha =[ano, mes, dia, hora, minuto, segundo, var1, var2]

    guarda_linhas.append(nova_linha)
```

[4]:
```
dados = np.array(guarda_linhas)
```

[5]:
```
tempo = []
for i in range(len(dados)):
    monta_tempo = datetime.datetime(int(dados[i,0]),
                                    int(dados[i,1]),
                                    int(dados[i,2]),
                                    int(dados[i,3]),
                                    int(dados[i,4]),
                                    int(dados[i,5]))

    tempo.append(monta_tempo)
```

[6]:
```
fig, ax = plt.subplots()
ax.plot(tempo,dados[:,6], label='var 1')
ax.plot(tempo,dados[:,7], label='var 2')

formato = mdates.DateFormatter('%H:%M')
ax.xaxis.set_major_formatter(formato)

ax.set_xlabel('Tempo')
ax.set_ylabel('Nivel (m)')

plt.legend()

plt.show()
```