

Artist

Introdução à programação

Parte II - Python

Este documento serve para ensinar a programar em Python desde as mais simples instruções e algoritmos até à construção de outputs gráficos e interfaces.

Pedro Correia

9 de Setembro de 2010

Índice

Introdução	3
Notas para os leitores	3
Instalar o Python	4
Spyder	5
Consola do Python	7
Objectos	10
Módulos	11
Vectores e Matrizes.....	13
Carregar matrizes em ficheiro para variáveis	18
Strings.....	22
Tipos de dados	27
Tuples, Sets e Dictionaries	28
Funções importantes de se saber	31
Interface de consola.....	35
Instruções em ficheiro.....	37
Matplotlib (output gráfico)	44
Tabela de símbolos para gráficos.....	46
Tabela de cores para gráficos.....	48
Tamanho e transparência dos símbolos	51
Histogramas.....	52
Projecção estereográfica.....	55
Gráficos matplotlib em ficheiro	56
Numpy (tratamento numérico).....	58
Trabalhar directamente no ficheiro	61
Ciclos While	61
Identação.....	64
Lógica booleana (perguntas “if”).....	65
Faz os teus próprios módulos	69
Funções	69
Classes	73
Faz o teu próprio software.....	77
wxPython (interfaces gráficos).....	80

Janela principal.....	83
Inserir Pannel.....	85
Texto estático.....	87
Barra de menu.....	92
Eventos (parte 1).....	101
MessageDialog	103
FileDialog.....	106
AboutBox.....	115
Janelas Secundárias.....	117
Associar janelas secundárias às principais	117
Botões	120
Eventos (parte 2).....	121
TextCtrl.....	123
ComboBox	125
Janelas Secundárias (revisões)	128
StaticBox.....	131
SpinCtrl	133
CheckBox	136
Matplotlib no wxPython (parte 1).....	138
Trabalhar com ficheiros.....	143
Detalhes finais do programa (formatar strings).....	148
Código final.....	150
Py2exe	158
wxPython avançado	160
Matplotlib no wxPython (parte 2).....	160
Notebook.....	165
Matplotlib avançado	169

Introdução

A linguagem de programação Python foi implementada no final da década de 80 por Guido van Rossum na Holanda. No seu princípio a linguagem era pouco célebre entre a comunidade de programação mas com a inserção de novas capacidades começou a ser utilizada por diversos campos, científicos inclusive. O aumento de utilizadores desta linguagem em conjunto com o facto da mesma ser livre e gratuita deu origem a uma explosão de novas funcionalidades e bibliotecas. Qualquer utilizador pode partilhar os seus algoritmos e ideias e assim o fizeram. Em poucos anos o Python atingiu um nível de utilização que parecia somente destinado às linguagens de baixo nível (C, C++, Java, etc...). O facto de ser escrito em C facilitou a partilha de algumas das bibliotecas mais usadas a nível mundial como o OpenGL (pyOpenGL) e Qt (pyQt).

As principais vantagens técnicas do Python são: sintaxe simples e um gigantesco suporte de bibliotecas para as mais diversas áreas. Para além disso dispõe de uma circunstância pouco comum entre linguagens de muito alto nível, que é o facto de ser usada para criar ferramentas que funcionem independente de interpretadores (como é o caso do Octava, MatLab ou Mathematica). A grande desvantagem do Python é a mesma das restantes linguagens de alto nível, é lento. Esta característica é pouco importante para a esmagadora maioria dos programas dado que o volume de dados processado é baixo, mas para software que tenha que lidar com muita informação a melhor alternativa continua a ser os clássicos C, C++, Fortran e Java.

Ainda assim esta é uma linguagem capaz e robusta e nas próximas páginas iremos ver como construir software desde os algoritmos de contas até ao interface e output visual, tudo de maneira simples ou pelo menos da maneira mais simples que encontrei até agora. No fim já deverás ser capaz de fazer os teus próprios programas de janelas e de planear o teu código de maneira a ser reutilizável para futuras aplicações que vás fazer.

Notas para os leitores

Esta publicação é completamente gratuita. Podes utiliza-la e distribui-la da maneira que entenderes desde que não seja para fins comerciais. Peço apenas que a referencies sempre que a utilizares para um projecto. A nossa capacidade de produzir material de ensino advém principalmente de sabermos que as nossas publicações estão a servir para alguma coisa. Os exemplos escrito aqui foram feitos em Windows. Este livro deverá ser tão eficiente quando mais à risca o leitor o seguir.

Instalar o Python

O site do Python é este: www.python.org

Lá poderás encontrar muita informação sobre esta linguagem e, evidentemente, o instalador (Linux, Windows, OSX, etc...). Por descargo de consciência tinha que evidenciar esta página mas se seguires o caminho proposto nesta publicação não irás instalar o Python por aqui.

A minha proposta é que se instale um pacote que já traga uma série de software, bibliotecas e o próprio Python visto que te irá poupar muito trabalho posteriormente e, possivelmente, erros de instalação. O meu pacote de preferência é Python XY, um pacote que trás muitos softwares e bibliotecas para programação com fins científicos (entre outros) e a sua página é:

www.pythonxy.com

Dentro do pacote do Python X Y irás encontrar as principais bibliotecas que vamos encontrar nesta publicação: o numpy e scipy (para calculo numérico e científico), o matplotlib (para visualização de gráficos) e o wxpython (para o interface gráfico de janelas). Também irás encontrar o IDE que tem sido o da minha preferência: o Spyder. Na página do Python X Y basta carregar em downloads na página principal e depois carregar no link do “current release”. De resto é uma instalação bastante corriqueira. No instalador ir-te-á ser feita a pergunta de que bibliotecas é que pretendes instalar (das que estão no pacote, isto é...). Pessoalmente eu costumo meter tudo (dá à volta de um giga de memória no disco rígido) mas se não o quiseres fazer garante que escolheste o wxpython, matplotlib e py2exe. Quando a instalação estiver acabada vai aparecer-te um ícone do Python X Y (python(x,y)). Ao carregares nesse ícone ir-te-á aparecer uma janela de escolha dos softwares que queres abrir. Se pretenderes seguir a mesma linha que eu abro o Spyder.

O spyder é um IDE, melhor dizendo, uma ferramenta que te irá ajudar na tua programação indicando os locais onde a tua sintaxe está mal feita, o número e localização das funções que escreveste, etc... Pessoalmente o Spyder é me apelativo pois consegue introduzir o local onde se escreve código (estilo notepad) e a consola interactiva do python na mesma janela. Isto é espectacular para aprender visto que podemos estar a escrever código e a experimentar comandos ao mesmo tempo na consola. Assim, em vez de compilar o código todo de uma vez, cada vez que pretendemos testar apenas um dos passos que o mesmo tem, podemos inserir o comando a consola e extrapolar se os seus resultados são os mesmos que pretendemos no nosso programa.

Ao princípio poderá tudo parecer-te um pouco confuso mas rapidamente deverás conseguir mexer nesta linguagem com mestria pois o Python tem uma rápida curva de aprendizagem. Iremos começar por trabalhar na consola e posteriormente iremos trabalhar com código directamente no ficheiro. À medida que formos progredindo iremos abordar a maneira de como se criar classes e bibliotecas (módulos) e posteriormente de como fazer visualizações gráficas e interfaces gráficos. Boa sorte...

Spyder

Após carregares o ícone do Python X Y vais deparar-te com o menu da Figura 1 onde podes escolher que softwares queres inicializar. No espaço das aplicações vais encontrar programas como o Eclipse, IDLE, pyQT, MayaVi 2, entre outros. O Python X Y guarda um novo espaço especialmente para o Spyder que encontrar logo após as aplicações. Nas escolhas podes entrar meter vários níveis que ter irão carregar diferentes bibliotecas. O “—basics” serve perfeitamente para as nossa ambições e para meteres o Spyder a funcionar basta que carregas no botão evidenciado com um bola vermelha.

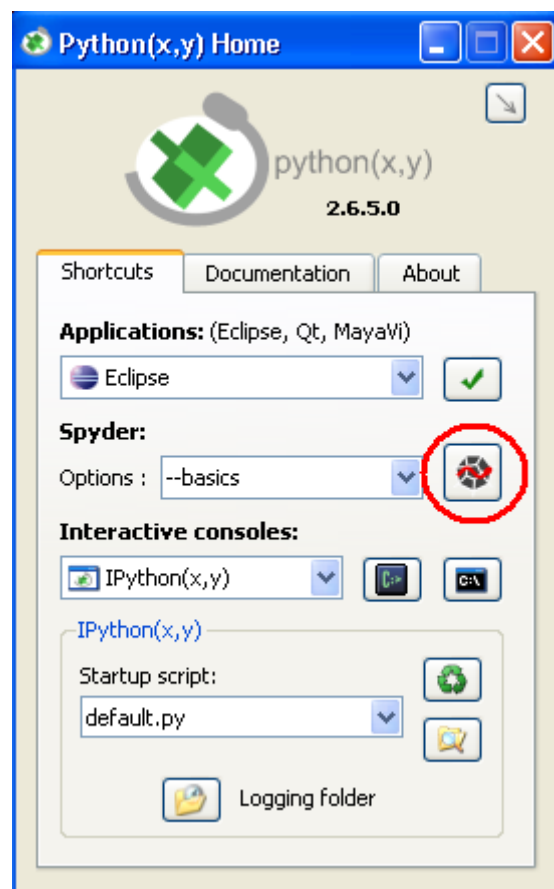


Figura 1 - Menu do Python X Y evidenciando o botão para entrar dentro do Spyder.

De maneira a não deixar a pergunta no ar acerca do que é o espaço que resta (Interactive Consoles) basta apenas dizer que se trata de diferentes consolas interactivas do Python onde podes programar directamente sem ter que fazer necessariamente um script do teu programa. Para cálculos esporádicos isto é ideal. Como o Spyder já trás a sua própria consola, sempre que necessário iremos utilizar essa.

Na Figura 2 podemos ver o interface inicial do Spyder (por vezes a consola interactiva pode aparecer no canto inferior direito em vez do meio, por exemplo, mas a ideia está lá). A verde temos o local por onde iremos começar a aprender esta linguagem, a consola interactiva. A amarelo temos o painel de informações onde podemos, inclusive, ver documentação ou ajuda sobre uma determinada função ou biblioteca. Os restantes painéis, vermelho e lilás, só mais para a frente iremos usar mas para já compreende que são os painéis da programação propriamente dita, directamente no ficheiro.

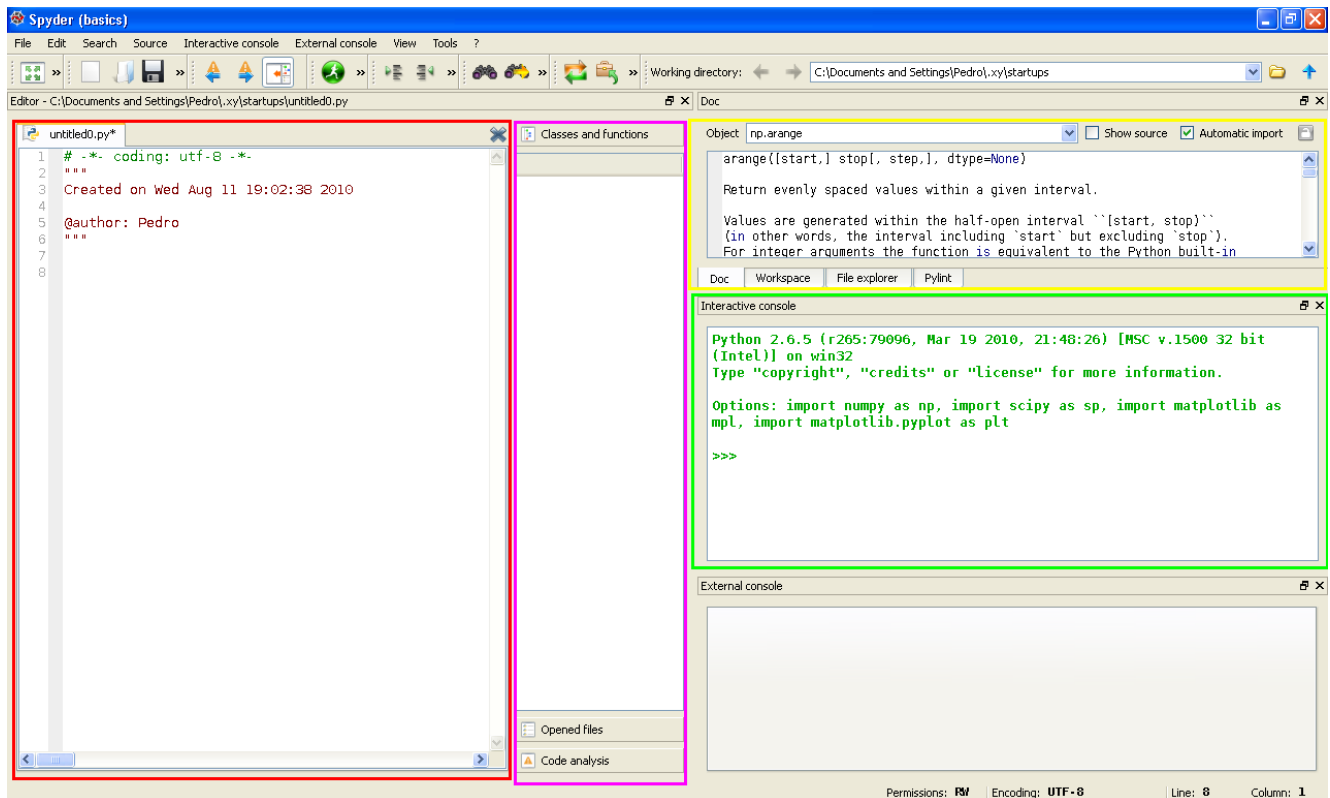


Figura 2 - Interface do Spyder no qual a verde está a consola interactiva, a amarelo o painel de informações, a vermelho a folha para scripting em ficheiros e a lilás as informações sobre o programa que estamos a fazer.

O nosso trabalho inicial vai ser na consola que na Figura 3 mostramos à escala.

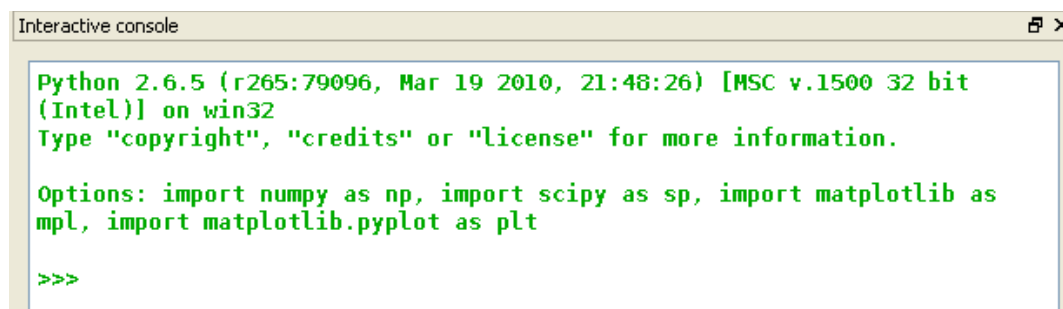


Figura 3 - Consola Python do Spyder (repara que no texto options vemos que as bibliotecas numpy, scipy, matplotlib e matplotlib.pyplot já vêm por defeito ao correr o Spyder).

Consola do Python

A consola onde iremos trabalhar nesta publicação é na do Python. É importante referir isto porque poderão surgir alturas em que poderemos usar bibliotecas que no Spyder já vêm por defeito enquanto noutras consolas poderão ter que ser importadas antes de utilizadas. Tanto quanto possível tentarei contornar estas diferenças para que a experiência seja o mais semelhante possível para outras consolas que possas estar a utilizar para aprender Python.

Ao ser iniciada uma consola (e repara na figura que mostra a do Spyder) vêm descritas uma série de informações acerca da versão que estamos a utilizar (a da figura é Python 2.6.5) e no caso da consola do Spyder os módulos ou bibliotecas que estamos a utilizar (mais uma vez na figura já vem o numpy, scipy, matplotlib e matplotlib.pyplot por defeito). Neste momento não precisas de saber o que é isto dos módulos ou bibliotecas. Em breve compreenderás. Começemos por analisar a consola.

O indicador de comando da consola do Python é “>>>”. Aqui deverás escrever o que pretendes fazer em Python. Se escreves 5+5 a consola dar-te-á a resposta:

```
>>> 5+5
10
>>>
```

A consola funciona como um sistema de perguntas, respostas e declarações. Quando abres a consola não está feita nenhuma declaração por isso se perguntares o que é o “a” ela não saberá responder:

```
>>> a
Traceback (most recent call last):
NameError: name 'a' is not defined
>>>
```

Mas se declarares a tua variável antes de fazeres a pergunta irás obter a resposta:

```
>>> a=5
>>> a
5
>>>
```

Assim funciona a consola do python. Como deves calcular podes fazer operações com as tuas próprias variáveis atribuindo o resultado a outras variáveis ou até mesmo atribuir à mesma variável que está a usar:

```
>>> a=a+2
>>> a
```


7

```
>>> b=a+a+3
```

```
>>> b
```

17

```
>>>
```

O Python tem várias funções embutidas (sem recurso a módulos) que podemos utilizar nos nossos cálculos como é o caso do comando “abs” que transforma valores reais em absolutos (melhor dizendo o módulo passa os negativos a positivos e mantém os positivos positivos).

```
>>> a=-7
```

```
>>> b=abs(a)
```

```
>>> a
```

-7

```
>>> b
```

7

```
>>>
```

Dentro das funções metemos os argumentos. No caso exemplificado acima utilizamos a variável “a” como argumento da função “abs” e atribuímos o resultado da função à variável “b”. Isto é igual para todas as funções excepto se a função em causa não tiver argumentos de entrada ou então se não precisar deles para funcionar. Outras funções não nos dão resultados mas sim pedem-nos como é o caso do input:

```
>>> c=input('mete numero: ')
```

mete numero: 3

```
>>> c
```

3

```
>>>
```

O input na consola para pouco ou nada nos serve mas se tivermos um programa que precise de dados por parte do utilizador poderia utilizar esta função para lhe dar um número com o qual ele precise de funcionar. Este tipo de informações apenas servem para te introduzir ao mundo da programação no qual tu dás a instrução que queres que o computador faça. Ainda não fizemos nenhum programa propriamente dito mas já deverás perceber que a lógica do mesmo é muito estilo calculadora. Uma das grandes diferenças que ganhamos em relação à calculadora é o facto de podermos dizer ao computador para repetir a mesma operação várias vezes, mas lá chegaremos. Antes de abandonar esta fase quero apenas explicar o funcionamento do tipo de dados ou tipo de números com que podemos funcionar. Regra geral, as linguagens de baixo nível precisam de declarações de variáveis, melhor dizendo o programador precisa de dizer se a variável que vamos usar é inteira, real, complexa, etc. O

Python é uma linguagem de alto nível pelo que muitas dessas operações são feitas automaticamente. Não precisamos de declarar a variável “d” para dizer que ela é igual a 3^2 , por exemplo, mas ela precisa de existir para entrar como argumento de uma função:

```
>>> abs(d)
```

```
Traceback (most recent call last):
```

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

```
>>> d=3**2
```

```
>>> abs(d)
```

```
9
```

```
>>>
```

No exemplo acima quando metemos a variável “d” como argumento da função “abs” deu um erro porque “d” não estava definida. No passo a seguir indiquei que “d” era igual a 3^2 e logo a seguir usei “d” novamente como argumento da função “abs”. Desta já funcionou porque “d” já estava declarado ou segundo o Python definido. Embora o Python faça este tipo de declarações automaticamente nem sempre ele percebe como operações da mesma classe de números pode dar uma segunda classe diferente como é o caso da divisão entre inteiros, repara:

```
>>> 1/3
```

```
0
```

```
>>> 2/4
```

```
0
```

```
>>> 4/2
```

```
2
```

```
>>> 3/4
```

```
0
```

```
>>> 4/4
```

```
1
```

```
>>> from __future__ import division
```

```
>>> 1/4
```

```
0.25
```

```
>>
```

Nos exemplos acima dividimos 1 por 3, 2 por 4 e 3 por 4 e ele respondeu com a parte inteira do resultado. $2/4$ é igual a 0.5 e a parte inteira de 0.5 é 0!!! Por esse motivo o Python precisa que lhe indiquem como actuar perante este tipo de situações. Felizmente no desenvolvimento do Python foi feita uma função que torna este tipo de operações absolutamente corriqueira.

Essa função advém do módulo “__future__” e chama-se “division”. Repara que a partir do momento em que chamei a função com o comando “from __future__ import division” eu já consegui obter o resultado certo para uma divisão entre inteiros. Não te preocupes se não perceberes o que o comando para chamar a função de determinado módulo quer dizer, nas próximas páginas veremos isso.

Objectos

Python é o que se chama uma linguagem orientada a objectos. Para elucidar sobre o que isto quer dizer vou dar um exemplo. Imaginemos um carro, que é um objecto, composto pelas suas rodas, volante, bancos, etc. Eu posso dizer que do objecto carro quero apenas os bancos e nesse caso eu peço “quero os bancos do carro”. Python funciona assim só que a maneira de chamar é por pontos, por exemplo, se eu quiser chamar os bancos do carro faço: “carro.bancos”. Se eu quiser chamar a cor dos bancos do carro faço: “carro.bancos.cor”. Percebes? Vai desde o objecto maior até ao menor metendo pontos para indicar que se trata de um objecto dentro de outro. Não existe, de facto, o objecto “carro” em Python mas existem outros objectos que nos vão ser muito úteis que é o caso das bibliotecas que vamos utilizar. Por exemplo, a biblioteca “numpy” é a biblioteca numérica do Python, muito indicada para operações com vectores e matrizes (entre muitas outras coisas). A função “cos” que dá o cosseno de um dado valor argumento está na biblioteca “numpy”. Se utilizarmos apenas o “cos” vai nos dar um erro:

```
>>> f=100
>>> cos(f)

Traceback (most recent call last):

NameError: name 'cos' is not defined

>>>
```

O erro existe porque a função “cos” não existe em Python mas sim na biblioteca de Python “numpy”. Assim precisamos primeiro de importar a biblioteca “numpy” e partir da mesma (é um objecto) chamamos a função “cos” que tem lá dentro (um objecto dentro do objecto numpy):

```
>>> import numpy
>>> numpy.cos(f)

0.86231887228768389

>>> g=numpy.cos(f)

>>> g

0.86231887228768389

>>>
```

Repara que fizemos tal e qual como o exemplo do carro onde primeiro chamamos o objecto grande, depois metemos um “.”, e finalmente chamamos o objecto que pretendemos (numpy.cos(f), no qual “f” é o argumento necessário para o objecto “cos” funcionar). Podemos até atribuir esse resultado a outra variável tal como foi o caso do “g”. O funcionamento do Python orientado a objectos é uma das suas enormes vantagens em relação a outras linguagens de sintaxe mais difícil. Isto torna o teu código mais fácil de se ler e também de se escrever.

Módulos

Módulos ou bibliotecas são objectos, melhor dizendo conjuntos de funções que podemos utilizar para as mais variadas actividades. Nos exemplos acima chamamos um objecto chamado “numpy” para usar uma função que temos lá, “cos”. O método que utilizamos nesse exemplos para chamar a biblioteca numpy é o mais geral mas não o único. De qualquer maneira recorda que a maneira de chamar um módulo é sempre “import ‘nome do modulo’”:

```
>>> import numpy
>>>
```

Agora cada vez que quisermos utilizar uma função do “numpy” apenas temos que a indicar que função é essa:

```
>>> numpy.cos(0)
1.0
>>> numpy.sin(0)
0.0
>>> numpy.mean([1,2,3,4])
2.5
>>>
```

Claro que nem sempre isto é conveniente. Por vezes os nomes dos módulos são tão compridos que estar a escrever o mesmo cada vez que queremos uma função é pura tortura. Assim para minimizar este efeito podemos atribuir outro nome ao módulo:

```
>>> import numpy as np
>>> np.cos(0)
1.0
>>> np.sin(0)
0.0
>>>
```

Atribuímos o nome de “np” ao módulo “numpy” e agora em vez de numpy podemos utilizar a nomenclatura de “np”. Mesmo assim isto pode ser chato se tivermos de utilizar a mesma função repetidas vezes e nesse caso podemos evitar de todo ter que utilizar o nome do módulo fazendo:

```
>>> from numpy import cos  
  
>>> cos(0)  
  
1.0  
  
>>>
```

Esta linha de código recorda-te alguma coisa? Foi assim que importamos a divisão do módulo “__future__” numas páginas para trás. Na altura o que fizemos foi “from __future__ import division”. Agora fizemos “from numpy import cos”, é a mesma coisa para um objecto diferente. Curiosamente podes até tentar encurtar mais a dificuldade em chamar uma função se o teu desespero chegar a esse ponto fazendo:

```
>>> from numpy import cos as c  
  
>>> c(0)  
  
1.0  
  
>>>
```

Basicamente conjuguei todas as instruções que demos até aqui para simplificar ao máximo a função que originalmente era “numpy.cos”. Vou apenas ensinar mais um método para importar todas as funções de um módulo ao mesmo tempo em vez de ser apenas uma:

```
>>> from numpy import *  
  
>>> cos(0)  
  
1.0  
  
>>> sin(0)  
  
0.0  
  
>>>
```

O * quer dizer “tudo” na frase de importação do módulo. A partir do momento em que utilizamos esta instrução podemos usar todas as funções do numpy sem ter que usar o seu nome.

Vectores e Matrizes

Para lidar com vectores e matrizes vamos usar, principalmente, a biblioteca numpy, mas vamos começar por ver como fazer um com Python simples:

```
>>> a=[1,2,3,4]

>>> a

[1, 2, 3, 4]

>>>
```

Basta usar os parêntesis rectos para definir um vector (melhor dizendo lista) em Python com vírgulas entre os seus números. Para aceder a um valor dessa lista basta indicar qual a posição que pretendemos ver:

```
>>> a[0]

1

>>> a[1]

2

>>> a[3]

4

>>>
```

Pode não ser imediatamente intuitivo mas a posição “0” corresponde na realidade à primeira posição e a posição “3” corresponde na realidade à quarta. É importante perceberes isto porque sempre que estiveres a lidar com vectores ou matrizes será este tipo de lógica que terás de aplicar. Agora também vectores podem entrar como argumentos de funções (não em todas as funções, como deves calcular). Já noutro exemplo aqui usamos a função “mean” que nos dá a média de um vector:

```
>>> import numpy

>>> numpy.mean(a)

2.5

>>>
```

Existem outras funções para organizar os valores dentro do vector, ou somar os valores do vector ou mesmo calcular os co-senos dos valores do vector:

```
>>> numpy.cos(a)

array([ 0.54030231, -0.41614684, -0.9899925 , -0.65364362])

>>>
```

Para fazeres um vector a partir da biblioteca numpy podes usar a função “numpy.array()”:

```
>>> b=numpy.array([3,4,5,6])
```

```
>>> b
```

```
array([3,4,5,6])
```

```
>>>
```

Podes interagir com este vector exactamente da mesma maneira que com o anterior, aviso apenas que podem haver funções que apenas lidem com vectores numpy e esse é um dos motivos que me leva a utiliza-los.

```
>>> c=numpy.array([4,3,6,4,6,7])
```

```
>>> c[0]
```

```
4
```

```
>>> c[2]
```

```
6
```

```
>>>
```

Repara que em Python, ao contrário de outras linguagens de alto nível como Octave (a primeira parte desta série de publicações), um número não é um vector de apenas um elemento e por isso ao declarar “a” como sendo um número não nos possibilita aceder a ele como a[0]. Por esse motivo sempre que quisermos que “a” seja um vector ou matriz, mesmo que só com um elemento temos de o declarar como tal:

```
>>> a=999
```

```
>>> a[0]
```

```
Traceback (most recent call last):
```

```
TypeError: 'int' object is unsubscriptable
```

```
>>> a=[999]
```

```
>>> a[0]
```

```
999
```

```
>>> a=numpy.array([999])
```

```
>>> a[0]
```

```
999
```

```
>>> a
```

```
array([999])
```

```
>>>
```

Como podes ver neste exemplo que inseri aqui declarei o “a” como sendo o número 999 mas não consegui aceder à primeira posição dele (posição 0) porque para o Python isto é um número e não um objecto que tenha posições. Assim ao declarar o 999 ao “a” mas com

parêntesis rectos consegui aceder a ele com “a[0]” e ele assim já percebeu que “a” é um objecto com posição 0. Depois exemplifiquei o mesmo com o objecto “numpy.array” que iremos usar mais vezes. Bem se agora quiseses uma matriz em vez de um vector podes faze-lo mas primeiro precisamos de perceber a lógica por trás desta acção. O Python vê uma matriz como um composto de vários vectores, mesmo que vectores de apenas um elemento (como é o caso de uma matriz coluna), assim para definir uma matriz de duas linhas precisamos de definir o vector da primeira linha e o vector da segunda linha e podemos faze-lo no mesmo comando:

```
>>> b=numpy.array([[1,2,3],[4,5,6]])
```

```
>>> b
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>>
```

Tal como fizemos ao inicio declaramos o vector com parêntesis rectos e agora fazemos o mesmo com uma matriz. Como o vector é um elemento da matriz então temos que meter os parêntesis rectos dos vectores (que são separados por vírgulas) e a contê-los os parêntesis rectos da matriz (a negrito estão os vectores e a vermelho os elementos da matriz):

```
[ [ 1 , 2 , 3 ] , [ 4 , 5 , 6 ] ]
```

Evidentemente que podes fazer uma matriz coluna, para isso basta que acrescentes vectores de um só elemento a uma matriz:

```
>>> b=numpy.array([[5],[4],[3],[2],[1],[0]])
```

```
>>> b
```

```
array([[5],
       [4],
       [3],
       [2],
       [1],
       [0]])
```

```
>>>
```

Agora se acederes ao 3º elemento, por exemplo, da matriz “b” o teu resultado irá ser:

```
>>> b[3]
```

```
array([2])
```

```
>>>
```


Parece esquisito mas visto que uma matriz é um vector de vectores o que tu foste buscar foi o vector que está na posição 3. Repara que também não utilizei a posição das linhas e colunas para ir buscar o que está na quarta linha (posição 3) e primeira coluna (posição 0) mas podia tê-lo feito:

```
>>> b[3,0]
```

```
2
```

```
>>>
```

E assim já acedi directamente ao valor que está na matriz e não o vector que estava na quarta linha da matriz (perde uns minutos a fazer experiências com estes objectos se isto for muito confuso para perceber em pouco tempo). Se fizer a mesma experiência com a matriz que fiz à pouco o resultado irá ser igual:

```
>>> b=np.array([[1,2,3],[4,5,6]])
```

```
>>> b
```

```
array([[1, 2, 3],  
       [4, 5, 6]])
```

```
>>> b[0]
```

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

```
>>> b[1]
```

```
array([4, 5, 6])
```

```
>>> b[0,0]
```

```
1
```

```
>>> b[1,2]
```

```
6
```

```
>>>
```

Não te preocupes se a lógica que está associada a estes objectos não te for imediata, o mais provável é que volta e meia venhas à consola fazer uma experiência com vectores e matrizes porque já não te recordas de como funcionam, é normal e não custa nada. Ainda assim se já entendes o que é um vector e uma matriz em Python, especialmente para a biblioteca numpy, podemos seguir em frente. Podemos seleccionar apenas uma das colunas da matriz ou apenas uma das linhas (que nós até já fizemos antes quando escolhemos a posição 0 ou 1 da matriz “b”).

```
>>> b[0]
```

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

```
>>> b[0,:]
```

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

```
>>> b[1]
array([4, 5, 6])
>>> b[1,:]
array([4, 5, 6])
>>>
```

O ":" quer dizer tudo. Comecei por chamar a posição 0 da matriz que é o primeiro vector que inseri lá e a seguir introduzi outra maneira de fazer a mesma coisa mas como se fosse aceder a valores e disse que queria todas as colunas (":") da linha 0 de "b" (b[0,:]). A seguir fiz a mesma coisa mas pedi todas as colunas da linha 1 de "b" (b[1,:]). Todas as colunas da linha 0 é o vector da posição 0 da matriz tal como todas as colunas da linha 1 é o vector 1 da matriz. Podemos fazer o mesmo para seleccionar colunas em vez de linhas dizendo que queremos todas as linhas na coluna que pretendemos:

```
>>> b[:,0]
array([1, 4])
>>> b[:,1]
array([2, 5])
>>> b[:,2]
array([3, 6])
>>> b
array([[1, 2, 3],
       [4, 5, 6]])
>>>
```

Repara que cada vez que sacas uma coluna da matriz ela sai como um vector. O Python faz isto automaticamente. É possível mudar este resultado para ficar como uma coluna (como era originalmente) e é até útil por várias razões, mas por agora não vamos ver isto. Também já deves ter interiorizado que ao chamar um elemento da matriz pela sua coordenada em linhas e colunas a primeira é a da linha e a segunda da coluna (**b[linhas,colunas]**). Claro está se eu chamar todas as linhas ":" e todas as colunas ":" o meu resultado vai ser a própria matriz:

```
>>> b[:,:]
array([[1, 2, 3],
       [4, 5, 6]])
>>>
```

Carregar matrizes em ficheiro para variáveis

Já sabemos como fazer matrizes e vectores a partir da consola do Python mas se queremos fazer programas funcionais o mais provável é que precisemos de uma maneira de carregar dados a partir de ficheiros. Mais para a frente vamos aprender a lidar com ficheiros de uma maneira mais completa mas por agora vamos aprender a carregar dados directamente para uma variável matriz em Python. A função que vamos utilizar é da biblioteca numpy e por isso o objecto de vector ou matriz resultante dessa função vai ser também estilo numpy. Para te ensinar a fazer isto vou antecipar apenas um bocadinho sobre como lidar com strings (no capítulo a seguir vemos isto melhor, por agora será apenas o suficiente para saber carregar um ficheiro para uma variável).

```
>>> b=isto e uma string

      b=isto e uma string
      ^
```

SyntaxError: invalid syntax

Na consola tentei dizer que “b” é igual a uma frase que diz “isto e uma string” mas o Python deu erro. Isto aconteceu porque ele não tem maneira de saber se a frase é mesmo uma string ou uma variável que por acaso se chama “isto e uma string”. Assim para dizermos ao Python que esta frase é uma string precisamos de usar pelicas:

```
>>> b='isto e uma string'

>>> b

'isto e uma string'

>>>
```

Agora sim ele já entendeu e a partir de agora “b” é a string “isto é uma string”. Uma string é um vector, mas ao invés de utilizarmos números, utilizamos caracteres e por isso se chamarmos uma qualquer posição da string vamos obter o caracter correspondente.

```
>>> b[0]

'i'

>>> b[1]

's'

>>> b[0],b[1],b[2],b[3]

('i', 's', 't', 'o')

>>>
```

Chamei a posição 0 e ele devolveu “i” porque esse é o caracter que está na primeira posição da string, depois chamei a posição 1 e ele devolveu o segundo carácter e finalmente chamei as quatro primeiras posições (0,1,2,3) e ele devolveu-me uma lista com os quatro primeiros

caracteres. Agora as strings têm algumas regras entre as quais e esta é particularmente importante é o uso da “\”. Quando tentas atribuir o caracter “\” a uma variável vai dar erro:

```
>>> c='\'  
      c='\'  
      ^
```

SyntaxError: EOL while scanning string literal

Isto é importante porque quando vamos a tentar passar os caminhos para os ficheiros temos que usar barras, por exemplo “c:\minhapasta\teste.txt”. Ele irá reconhecer isto como uma string mas não uma string que consiga computar para encontrar um caminho para um ficheiro então para resolvermos isto podemos repetir a barra em todos os locais onde ela aparece.

```
>>> c='\\'  
>>> c  
'\\'  
>>> c='c:\\minhapasta\\teste.txt'  
>>> c  
'c:\\minhapasta\\teste.txt'  
>>>
```

Agora sim podemos utilizar a função objectivo deste capítulo, o “loadtxt”, melhor dizendo o “numpy.loadtxt”:

```
>>> c  
'c:\\minhapasta\\teste.txt'  
>>> import numpy  
>>> a=numpy.loadtxt('c:\\minhapasta\\teste.txt')
```

Traceback (most recent call last):

```
File "C:\Python26\lib\site-packages\numpy\lib\io.py", line 414, in loadtxt  
    fh = file(fname)
```

IOError: [Errno 22] invalid mode ('r') or filename: 'c:\\minhapasta\\teste.txt'

```
>>> a=numpy.loadtxt('c:\\minhapasta\\teste.txt')  
>>> a=numpy.loadtxt(c)  
>>>
```

Repara que no primeiro comando atribui o caminho do ficheiro com dupla barra à variável “c”. Podia utiliza-la dentro do comando “loadtxt” mas para exemplificar outros casos comecei por usar o “loadtxt(‘c:\minhapasta\teste.txt’)” o que ele recusou (o tal problema das barras), então voltei a fazer o mesmo mas inserindo a dupla barra e desta vez resultou. Finalmente

tentei o mesmo mas em vez de inserir a string dentro do “loadtxt” inseri uma variável que continha essa string e, mais uma vez, correu tudo bem. Se não tiveres paciência para estar sempre a alterar os caminhos para conterem duas barras existe outra alternativa que é acrescentar uma instrução à string que diga ao Python que é uma raw string, melhor dizendo, que é uma string que ele deverá ler sem as regras habituais das strings tais como esta da barra.

```
>>> c=r'c:\minhapasta\teste.txt'

>>> a=numpy.loadtxt(c)

>>> a=numpy.loadtxt(r'c:\minhapasta\teste.txt')
```

Primeiro atribui à variável “c” a string do caminho sem barras duplas mas com um “r” imediatamente antes da pelica que inicia a string. Como podes ver utilizei o comando “numpy.loadtxt” com a variável “c” e ele carregou. A seguir utilizei o caminho directamente na função e utilizei a instrução “r” na mesma e voltou a resultar. Agora tenho estado a escrever comandos que, possivelmente, poderás não conseguir reproduzir porque não tens um ficheiro “teste.txt” dentro de uma pasta “c:\minhapasta\”. Para utilizar este comando eu fui ao C:\ (a localização de origem do disco rígido para que o caminho fosse curto e não uma string muito grande) e criei uma pasta chamada “minhapasta” e dentro dessa pasta criei um ficheiro “teste.txt” com o seguinte lá dentro:

```
4      5      6
8      4      12
312    34      1
0      23     123
1      2      3
```

É uma matriz de três colunas e cinco linhas sem mais nada lá dentro. Assim o comando “numpy.loadtxt” (quando o fores utilizar não te esqueças de importar o numpy: **import numpy**), que serve especialmente para estes casos, consegue ler o ficheiro para dentro de uma matriz numpy e se for esse o teu desejo atribui-lo a uma variável. Vou fazer novamente:

```
>>> a=numpy.loadtxt(r'c:\minhapasta\teste.txt')

>>> a
array([[ 4.,  5.,  6.],
       [ 8.,  4., 12.],
       [312., 34.,  1.],
       [ 0., 23., 123.],
       [ 1.,  2.,  3.]])

>>> a[0]
array([ 4.,  5.,  6.])

>>> a[0,0]
```

4.0

>>>

Como podes ver a partir do momento em que atribui a informação do ficheiro para a variável “a” consigo mexer com ela tal e qual como aprendemos a fazer com as matrizes no capítulo anterior. Da mesma maneira que podemos carregar também podemos salvar com o “savetxt”, melhor dizendo, o “numpy.savetxt”. Para o fazer temos que ter, no mínimo, dois argumentos: o sítio (nome do ficheiro a salvar inclusive) e a variável que vamos salvar:

```
>>> a[0,0]=999
```

```
>>> a
```

```
array([[ 999.,    5.,    6.],
       [   8.,    4.,   12.],
       [ 312.,   34.,    1.],
       [   0.,   23.,  123.],
       [   1.,    2.,    3.]])
```

```
>>> numpy.savetxt(r'c:\minhapasta\teste2.txt',a)
```

```
>>>
```

Comecei por alterar o elemento que está na posição [0,0] da matriz “a” para 999 (só para não ficar igual à que carreguei) e depois salvei com o “numpy.savetxt” onde comecei por meter o caminho (o ficheiro que salvei foi como teste2.txt) e depois um novo argumento que é a variável que vou salvar, “a”. Se ainda não percebeste, repara que quando queremos meter mais de um argumento numa função, temos que separa-los por vírgulas. O ficheiro resultante é este:

```
9.9900000000000000e+02 5.0000000000000000e+00 6.0000000000000000e+00
8.0000000000000000e+00 4.0000000000000000e+00 1.2000000000000000e+01
3.1200000000000000e+02 3.4000000000000000e+01 1.0000000000000000e+00
0.0000000000000000e+00 2.3000000000000000e+01 1.2300000000000000e+02
1.0000000000000000e+00 2.0000000000000000e+00 3.0000000000000000e+00
```

Talvez isto te pareça estranho mas os números estão lá, simplesmente se encontram em notação científica. É possível mete-lo a sair com notação normal mas por agora não vamos abordar essas especificidades até porque há coisas de muito maior importância para perceber por agora, strings inclusive.

Strings

Strings são conjuntos ou vectores de caracteres. Uma string é uma palavra, uma frase ou até um número só que não é para fazer contas mas sim para funcionar como caracter. Como já deves ter percebido pelo capítulo anterior, para definires uma string precisas de a inserir dentro de pelicas:

```
>>> c=palavra

Traceback (most recent call last):

NameError: name 'palavra' is not defined

>>> c='palavra'

>>> c

'palavra'
```

Caso contrário o Python não sabe se estás a atribuir uma string e julga que estás a atribuir à variável “c”, neste caso, o valor da variável “palavra” que não existe! Sempre que quiseres definir uma string insere-a dentro de pelicas. Podes fazer várias operações para poderes manipular strings, entre as quais adicionares:

```
>>> a='isto e'

>>> b='uma'

>>> c='palavra'

>>> d=a+b+c

>>> d

'isto eumapalavra'

>>> d=a+' '+b+' '+c

>>> d

'isto e uma palavra'

>>> f='frase'

>>> e=a+' '+b+' '+f

>>> e

'isto e uma frase'

>>>
```

Repara que comecei por definir três variáveis (a,b e c) com uma string cada e depois somei as três variáveis. Como não havia espaços nas strings as palavras ficaram todas juntas e por esse motivo fiz o somatório novamente mas desta vez somei uma string de espaço (porque o espaço também é um caracter) entre as variáveis e o resultado é uma string grande atribuída à

variável “d”. Dado que podes aplicar este tipo de operação a strings podes imaginar todo o tipo de situações relativamente a interfaces e outputs para os teus futuros programas:

```
>>> a='Ola '
>>> b='Ana'
>>> c=a+b
>>> c
'Ola Ana'
>>> b='Pedro'
>>> c=a+b
>>> c
'Ola Pedro'
>>> d='Ana'
>>> c=a+b+' e '+d
>>> c
'Ola Pedro e Ana'
>>>
```

Ou então coisas mais funcionais como por exemplo:

```
>>> tipo1='.txt'
>>> tipo2='.prn'
>>> nome='teste'
>>> ficheiro=nome+tipo1
>>> ficheiro
'teste.txt'
>>> ficheiro=nome+tipo2
>>> ficheiro
'teste.prn'
>>>
```

Ainda não aprendemos a utilizar comandos para os nossos programas pedirem informações aos seus utilizadores mas quando o fizermos sabermos como conseguir manipular strings pode ser uma espectacular vantagem. A conversão de números para strings e vice-versa é fácil em Python e na maioria dos casos basta indicar o tipo que queremos daquele conjunto de caracteres. Para passar de um número para uma string desse mesmo número basta usar o

comando “str” (faz parte do Python e não é de nenhuma biblioteca exterior). Para fazer o inverso (e se o número for inteiro) podemos usar o comando “int”.

```
>>> c=555
>>> a=str(c)
>>> a
'555'
>>> b=int(a)
>>> b
555
>>>
```

Comecei por atribuir o número 555 à variável “c” e a seguir disse que a variável “a” é a conversão para string de 555, portanto ‘555’. A seguir atribui a “b” a conversão para inteiro da string ‘555’, portanto 555. Por vezes o número que pretendes passar de string para número não é um inteiro e por isso o comando “int” não irá funcionar. Quando o número for decimal temos que utilizar o comando “float”.

```
>>> c=555.5
>>> a=str(c)
>>> a
'555.5'
>>> b=int(a)
Traceback (most recent call last):
ValueError: invalid literal for int() with base 10: '555.5'
>>> b=float(a)
>>> b
555.5
>>>
```

E assim conseguimos fazer conversões bastante simples entre strings e números sem grande esforço. Outra função que podes usar correntemente para transformar um número em uma string é o “repr”:

```
>>> a=repr(123.456)
>>> a
'123.456'
>>>
```

Agora por vezes queremos passar informações do programa para o ecrã para que o utilizador possa ser informado acerca de qualquer coisa do nosso programa. O comando para enviar informações para o ecrã é o “print”. O “print” serve para ambos os tipos de variáveis numéricas ou strings.

```
>>> b=123

>>> print b

123

>>> c='ola'

>>> print c

ola

>>> print 'ola',123

ola 123

>>>
```

Existem várias maneiras de misturar strings e números no mesmo “print” sendo que a mais simples, na minha opinião, é utilizar a que tenho ensinado até agora:

```
>>> a=33

>>> print 'um numero: '+' '+repr(a)

um numero: 33

>>>
```

Isto foi bastante simples. Bastou-me converter o numero que tinha na variável “a” para string para que possa inseri-los no mesmo “print” por meio do somatório dessas duas strings. Se tentasse fazer-lo sem converter o “a” para string iríamos ter um erro porque estarias a juntar na mesma string dois tipos diferentes, string e inteiro.

```
>>> print 'um numero: '+a

Traceback (most recent call last):

TypeError: cannot concatenate 'str' and 'int' objects

>>>
```

Outra maneira de fazer isto e esta deverá ser mais comum para com outras linguagens é usar simbologias dentro de uma string para dizer que no local onde metemos aquele símbolo queremos, na verdade, um número. Então temos “%i” para inteiro, e “%f” para decimal.

```
>>> print 'um numero: %i'%a

um numero: 33

>>> print 'um numero: %f'%a

um numero: 33.000000
```

Repara que nos dois casos ele mandou dois outputs diferentes consoante o tipo de dados que pedimos. Repara também que a seguir à string metemos um “%” a indicar que a variável ou número que vem a seguir é o que é suposto estar dentro da string que vai sair no “print”. Outro símbolo interessante para saber é como mudar de linha numa string (não é só interessante, é mesmo muito necessário e é comum a várias linguagens por isso não te esqueças dele). Para isto utilizamos o “\n” (o sentido da barra é importante).

```
>>> a='primeira linha\nsegunda linha'
```

```
>>> a
```

```
'primeira linha\nsegunda linha'
```

```
>>> print a
```

```
primeira linha
```

```
segunda linha
```

```
>>>
```

Repara que quando chamamos “a” ele devolve-nos a string com o “\n” lá dentro mas quando usamos o “print” ele vai interpretar o “\n” como sinal de que tem de mudar de linha. Futuramente isto poderá ser muito importante porque poderás estar a querer passar muita informação para o utilizador e se não mudares de linha ela vai simplesmente sair do ecrã ficando numa zona não possível de ser visualizada. Outra situação importante é na leitura de ficheiros (noutros comandos que vamos aprender mais para a frente) no qual queremos ler várias linhas de um ficheiro e temos de utilizar o “\n” para dizer ao Python para passar à linha a seguir. Dado o carácter científico desta publicação não me irei prolongar muito no tratamento a strings mas posso deixar mais alguns comando que poderão ser úteis doravante como o “len” para saber o tamanho de uma string (quantos caracteres tem) e o split (da biblioteca string e portanto “string.split”) para transformar uma string de frase num vector de palavras.

```
>>> a='isto e uma frase'
```

```
>>> len(a)
```

```
16
```

```
>>> import string
```

```
>>> b=string.split(a)
```

```
>>> b
```

```
['isto', 'e', 'uma', 'frase']
```

```
>>>
```

Tipos de dados

Existem muitos tipos de dados, alguns como as strings, inteiros e decimais já tratámos aqui. Isto não irá ser informação fundamental para o resto da publicação nem nada que se pareça mas talvez para te iniciar nas considerações que poderás ter que fazer em linguagens que futuramente possas utilizar vou abordar de uma maneira muito geral este tema. Os tipos de dados que já abordamos foram as strings (na qual existem mais tipos semelhantes mas porque para os objectivos desta publicação isso ser pouco importante vou descurar), os inteiros (int) e os decimais (float). Estes tipos de dados existem porque para o computador não existem números, apenas existe verdadeiros ou falsos. Imagina o computador como sendo um interruptor e a única coisa que ele consegue fazer é saber se está ligado ou não. Agora dentro de um computador existem muitos interruptores portanto ele consegue saber quais os que estão ligados (verdadeiros) e quais estão desligados (falsos). Há muito que este tema é estudado, código morse, por exemplo, é uma mistura de batidas e ausência das mesmas entrando num sistema de verdadeiro ou falso. Linguagem binária (a linguagem do computador) também funciona assim e por isso para reproduzir o número 1 o computador precisa de ter três “interruptores” desligados e um ligado: 0001. Para fazer o dois a mesma coisa simplesmente muda o interruptor que estava ligado: 0010. E assim funciona uma máquina. Por esse motivo existe muita diferença entre um número inteiro e um decimal. Somos nós que por cima da lógica binária construímos estes objectos. E tal como construímos o inteiro e o decimal (int e float) construímos também o inteiro grande (long int), o decimal grande (long float, que em Python não sei se existe), os números complexos (complex), etc...

```
>>> a=int(5.5)

>>> a

5

>>> a=float(5.5)

>>> a

5.5

>>> a=complex(5.5)

>>> a

(5.5+0j)

>>> a=complex(5.5,3)

>>> a

(5.5+3j)

>>> a=long(5.5)

>>> a

5L
```

```
>>>
```

E muito resumidamente os tipos de dados é isto. Embora em Python não seja necessário noutras linguagens poderás que ter que fazer declarações do tipo de dados que vais usar ao atribuir uma variável. Nós não vamos fazer nada em Python que exija mexer nos tipos de dados mais do que fizemos até agora (quando convertemos uma string num inteiro ou num decimal), mas é bom teres noção de que esta é uma coisa que o Python faz por ti e que outras linguagens poderão não fazer (por isso se chama ao Python uma linguagem de alto nível, melhor dizendo uma linguagem que está mais longe da maneira do computador falar, outras linguagens aproximam-se da maneira do computador falar e por isso se chamam de linguagens de baixo nível como é o C ou C++ ou mais ainda o Assembly). Não te preocupes com nada disto, no dia em que precisares de o saber, se é que alguma vez irá acontecer, já estarás muito mais à vontade para o perceber.

Tuples, Sets e Dictionaries

Tuples (enuplas), sets e dictionaries (dicionários) são maneiras de organizar informação dentro de um objecto do Python muito ao estilo vector. Na verdade uma tupla é, de facto, um vector com algumas particularidades diferentes. Vou ensinar como mexer com estes objectos porque podem vir a ser úteis nos teus programas ou, no mínimo, dar-te a escolha de qual a tua preferência a programar. Num dos capítulos que ficou para trás nos falamos de strings serem como vectores de caracteres e por isso podes aceder a posições desse vector, no entanto se tentares mudar um elemento dessa posição (como fazemos nos vectores) não irás conseguir:

```
>>> a='palavra'
>>> a[0]
'p'
>>> a[0]='f'
Traceback (most recent call last):
TypeError: 'str' object does not support item assignment
>>>
```

Uma tuple tem a mesma propriedade. Para fazeres uma tuple basta insiras uma lista de elementos separados por vírgulas e enclausurados por parêntesis:

```
>>> a=(1,2,3,'ola',4,5,6)
>>> a[0]
1
>>> a[3]
'ola'
```

```
>>> a[3]='ole'
```

```
Traceback (most recent call last):
```

```
TypeError: 'tuple' object does not support item assignment
```

```
>>>
```

Assim podes criar um objecto estilo vector que cuja atribuição à posição é impossível. Mas existe outro estilo de organização de dados cuja utilidade é maior para os fins a que se destina este livro. Chamam-se sets e servem especialmente para fazer operações de selecção com listas estilos vectores.

```
>>> cesto=['maca','laranja','pessego','maca','banana','uvas']
```

```
>>> cesto
```

```
['maca', 'laranja', 'pessego', 'maca', 'banana', 'uvas']
```

```
>>> frutas=set(cesto)
```

```
>>> frutas
```

```
set(['uvas', 'laranja', 'pessego', 'maca', 'banana'])
```

```
>>>
```

No código acima criei um vector com frutas (repara que não uso caracteres que não são reconhecíveis ou correspondem a outras coisas no Python como “ç” ou “~”, etc...) e depois disse que o mesmo era um set da variável “frutas”. Se imaginares que este set é uma espécie de base de dados de frutas numa qualquer mercearia seria lógico poderes fazer perguntas sobre essa mesma base. É possível:

```
>>> 'laranja' in frutas
```

```
True
```

```
>>> 'tangerina' in frutas
```

```
False
```

```
>>> 'banana' in frutas
```

```
True
```

```
>>>
```

Podemos também fazer outro tipo de operações. Vou fazer um segundo set com frutas:

```
>>> cesto1=['tangerina','amendoa','banana','maca']
```

```
>>> frutas2=set(cesto1)
```

Agora se eu quiser saber as frutas que estão no set “frutas” mas não no set “frutas2” faço isto:

```
>>> frutas-frutas2
```

```
set(['uvas', 'laranja', 'pessego'])
```

Se eu quiser saber os frutos que estão num e noutra set faço:

```
>>> frutas | frutas2
```

```
set(['tangerina', 'laranja', 'banana', 'maca', 'uvas', 'pessego', 'amendoa'])
```

Repara que no set “frutas” existe ‘maca’ duas vezes mas os resultados não inserem ‘maca’ duas vezes. Isto sucede porque o Python percebe que aquele é o mesmo elemento e por isso não precisa de ocorrer novamente para dar essa informação. Continuando, se eu quiser saber as frutas que aparecem em “frutas” e “frutas2” então:

```
>>> frutas & frutas2
```

```
set(['banana', 'maca'])
```

E existem outras combinações possíveis de perguntas nas quais não me vou alongar. Vou apenas deixar esta pequena introdução a estes estilos de objectos e passar para o último deste capítulo, os dicionários. Os dicionários funcionam por correspondência de dados e faze-los é muito simples:

```
>>> alunos={'joao':25732,'joana':37564}
```

```
>>> alunos
```

```
{'joana': 37564, 'joao': 25732}
```

```
>>> 'joana' in alunos
```

```
True
```

```
>>> 'maria' in alunos
```

```
False
```

```
>>>
```

Repara que podemos fazer os sistemas de perguntas também, mas existem ainda outras vantagens. Vou tentar acrescentar mais um aluno:

```
>>> alunos['joano']=32456
```

```
>>> alunos
```

```
{'joano': 32456, 'joana': 37564, 'joao': 25732}
```

```
>>>
```

Vou agora aceder aos números correspondentes dos alunos:

```
>>> alunos['joao']
```

```
25732
```

```
>>> alunos['joana']
```

```
37564
```

```
>>> alunos['joano']
```

```
32456
```

```
>>>
```

Para saber todos os nomes que estão no dicionário de alunos faça:

```
>>> alunos.keys()
```

```
['joano', 'joana', 'joao']
```

```
>>>
```

Para eliminar um aluno do dicionário dos alunos faça:

```
>>> del alunos['joano']
```

```
>>> alunos
```

```
{'joana': 37564, 'joao': 25732}
```

```
>>>
```

Os dicionários são objectos muito úteis quando pretendemos fazer correspondências entre dois géneros de informação como no nosso caso é o nome e o número do aluno. Por agora vamos continuar com outros aspectos da programação mais virada para fins científicos.

Funções importantes de se saber

Existem algumas funções que quero sublinhar por serem muito usadas ao fazerem-se programas de natureza científica. Muitas deles fazem parte de módulos como o numpy por isso também será uma boa oportunidade de poderes consolidar o teu conhecimento de como importar e trabalhar com as bibliotecas. Vamos começar com a função “linspace”. A função “linspace” criar um vector com tantos elementos quantos os que quiseses que estão igualmente espaçados uns dos outros e apenas precisas de inserir o limite inferior, o limite superior, e o número de elementos que há entre eles. A função “**linspace**” está dentro da biblioteca numpy por isso começo por chama-la:

```
>>> import numpy
```

```
>>> a=numpy.linspace(1,10,10)
```

```
>>> a
```

```
array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

Repara que o primeiro elemento que eu meto na função é o limite inferior, o segundo é o limite superior e o terceiro o número de elementos que o vector vai ter. Vou dar outro exemplo:

```
>>> a=numpy.linspace(1,5,10)
```

```
>>> a
```



```
array([ 1.          ,  1.44444444,  1.88888889,  2.33333333,  2.77777778,
        3.22222222,  3.66666667,  4.11111111,  4.55555556,  5.          ])
```

Como eu pedi 10 elementos espaçados entre 1 e 5 a função respondeu com decimais. À partida estarias à espera de receber números de 0.5 em 0.5 mas se assim fosse não estariam 10 elementos dentro do vector mas sim 9:

```
>>> a=np.linspace(1,5,9)
>>> a
array([ 1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ])
```

Outra função que iremos usar aqui muitas vezes para criar dados usáveis para explicar outros comandos é a função “**rand**” que também ela está dentro do numpy só que com mais uma particularidade. A função “rand” está dentro de um módulo chamado random e o random está dentro de um módulo ainda maior chamado numpy. Assim para chamar a função “rand” tenho que fazer “numpy.random.rand”. Exemplificando:

```
>>> numpy.random.rand(10)
array([ 0.46113713,  0.28026996,  0.77137735,  0.17533183,  0.59170129,
        0.10702623,  0.1193613 ,  0.54132303,  0.87954666,  0.93017178])
>>>
```

A função “rand” gera números aleatórios de 0 a 1, neste caso um vector com dez elementos gerados aleatoriamente de 0 a 1. O resultado é o que vez. Esta função não se cinge apenas a vectores mas também a matrizes e nesse caso em vez de metermos o número de elementos metermos o número de linhas e colunas (“rand(linhas,colunas)”).

```
>>> numpy.random.rand(3,2)
array([[ 0.97719407,  0.15743981],
       [ 0.73586763,  0.8089255 ],
       [ 0.07195011,  0.50184256]])
>>>
```

Outra função para gerar números aleatórios é a função “**randint**” que gera inteiros num intervalo que tu determinas.

```
>>> numpy.random.randint(1,100,5)
array([53, 58, 55, 66, 90])
>>>
```

Repara que meti, à semelhança do “linspace”, o limite inferior, o limite superior, e o número de elementos do vector, respectivamente. Existem muitas funções para gerar números aleatórios dentro deste sub-modulo random do numpy que poderás facilmente consultar na

internet como usares se alguma vez precisares. A maioria corresponde a gerar números aleatórios dentro de distribuições conhecidas como a gaussiana, log-normal, etc. Não vamos por ai, vamos sim continuar a ver funções com efeitos mais imediatos sobre os nossos dados. Muitas vezes uma coisa que é importante saber é o número de linhas e colunas que uma matriz tem (isto é especialmente importante para fins de programação mais à frente onde vamos receber dados de ficheiros e não sabemos as suas dimensões). Para o fazer utilizamos a função “shape”. A função “**shape**” (literalmente que dizer “forma”, neste caso forma da matriz, melhor dizendo número de linhas e colunas) não é propriamente uma função mas sim uma característica que está embutida nos nossos vectores e matrizes e por isso chamamos como se fosse a forma da matriz que em Python se escreve “matriz.forma”. Vais perceber pelo exemplo que vou mostrar a seguir onde começo por criar uma matriz e depois pergunto qual a sua “shape”:

```
>>> a=numpy.random.rand(3,2)

>>> a.shape

(3, 2)

>>> a.shape[0]

3

>>> a.shape[1]

2

>>>
```

A matriz que criei tem 3 colunas e 2 linhas e por isso o resultado do “shape” é (3,2). Se eu quiser aceder só ao número de linhas faço “matriz.shape[0]” (porque o primeiro elemento é o número de linhas), caso contrário faço “matriz.shape[1]” (e portanto o número de colunas). Vou agora fazer o mesmo para um vector:

```
>>> a=numpy.linspace(1,10,4)

>>> a

array([ 1.,  4.,  7., 10.])

>>> a.shape

(4,)

>>> a.shape[0]

4

>>> a.shape[1]

Traceback (most recent call last):

IndexError: tuple index out of range

>>>
```

O “shape” consegui retirar, mais uma vez uma lista, neste caso só com um elemento porque o vector não tem duas dimensões, só tem uma. Por esse motivo quando perguntei pelo segundo elemento da lista ele respondeu que estava à procura de um elemento fora das posições da lista. Outra função que vou ensinar é a “zeros” que faz vectores ou matrizes de zeros. Vou começar por fazer um vector:

```
>>> a=numpy.zeros(5)

>>> a

array([ 0.,  0.,  0.,  0.,  0.])

>>>
```

Agora vou fazer uma matriz e repara no que vou ter que escrever para que o comando funcione:

```
>>> b=numpy.zeros(3,2)

Traceback (most recent call last):

TypeError: data type not understood

>>> b=numpy.zeros((3,2))

>>> b

array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]])

>>>
```

Tive que meter o tamanho da matriz dentro de parêntesis e depois esses parêntesis dentro dos parêntesis da função “zeros”. Para esta função (como para outras) isto é necessário porque são funções que podem receber vários argumentos e sem esses parêntesis o Python não saberia dizer se estás a dar argumentos diferentes ou simplesmente o número de linhas e colunas que pretendes. À pouco a função “rand” não precisou disto porque não recebe mais argumentos se não o número de linhas e colunas. Para te exemplificar isto vou meter aqui um exemplo onde acrescento um parâmetro que diz que quero o tipo de zeros como complexos:

```
>>> b=numpy.zeros((3,2),dtype=complex)

>>> b

array([[ 0.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j]])

>>>
```

Já houve várias funções que utilizamos aqui que têm as mesmas particularidades. Tenho ocultado esse facto porque pretendo que a vossa aprendizagem seja o mais continuada

possível sem perder tempo com pormenores que poderão aprender mais para a frente quando o essencial já estiver bem consolidado. Mas vamos continuar com outras funções semelhantes como a “**ones**”:

```
>>> c=numpy.ones((3,2))

>>> c

array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])

>>>
```

Da mesma maneira que puxamos à pouco a forma das matrizes (com o “shape”) podemos também puxar outras características como a **média**, **desvio padrão** e **mediana**:

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

>>> numpy.mean(d)

3.0

>>> numpy.std(d)

1.4142135623730951

>>> numpy.median(d)

3.0

>>>
```

Por agora estas funções vão servir. Mais tarde irão aprender funções para fazer coisas muito mais específicas e embora nós estejamos a utilizar o numpy, existem muitas outras bibliotecas a fazerem coisas semelhantes e cada uma delas tem dezenas e centenas de funções para explorar.

Interface de consola

Interface de consola é a ideia de meter o computador a pedir ou a dar informações ao utilizador. Nós já abordamos anteriormente o comando “print” quando estava a explicar como funcionam as strings e para o que se pode usar. O “print” é uma ótima ideia para passar informação para o utilizador:

```
>>> ano=2010

>>> mes='Agosto'

>>> print 'O programa foi inicializado'

O programa foi inicializado
```

```
>>> print 'Estamos no ano de '+repr(ano)
```

```
Estamos no ano de 2010
```

```
>>> print 'e no mes de '+mes
```

```
e no mes de Agosto
```

```
>>>
```

Ainda te lembrás de como mexer no “print”. Repara que quando quis juntar variáveis numéricas ao resto do texto do “print” fui converte-la para uma string primeiro com o comando “repr” (podia ter sido o “str”). Mas de resto é uma questão de adicionarmos mais informação a uma frase usando o “+”. Claro que o que, por vezes, nós queremos é receber informação do utilizador e não dar informação ao utilizador. Para isso utilizamos o comando “input”:

```
>>> a=input('Insere um numero: ')
```

```
Insere um numero: 33
```

```
>>> a
```

```
33
```

```
>>>
```

O “input” serve apenas para receber números e não ‘strings’ por isso se tentarmos fazê-lo vai dar um erro:

```
>>> a=input('Insere um numero: ')
```

```
Insere um numero: abc
```

```
Traceback (most recent call last):
```

```
File "C:\Python26\lib\site-packages\spyderlib\interpreter.py", line 74, in <lambda>
```

```
self.namespace['input'] = lambda text='': eval(rawinputfunc(text))
```

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

```
>>>
```

Para resolvermos isto temos o comando “raw_input” que recebe strings em vez de números:

```
>>> a=raw_input('Diz o teu nome: ')
```

```
Diz o teu nome: Pedro
```

```
>>> a
```

```
u'Pedro'
```

```
>>>
```

Agora num programa (que ainda não começamos a fazer mas lá chegaremos) de consola podemos comunicar com o utilizador de maneira a dar informação ao mesmo e receber a necessária para funcionar. E o essencial sobre este tema é isto. Se associares esta informação aquela que já sabes sobre strings poderás já poderás fazer muitas coisas. Recordas-te do símbolo '\n', é que é nestes casos que ele é especialmente importante:

```
>>> print 'O ano e '+repr(ano)+'\n'+ 'O mes e '+mes
```

```
O ano e 2010
```

```
O mes e Agosto
```

```
>>>
```

Ah, é verdade. Noutras linguagens as vezes surgem comandos específicos para aguardar apenas que o utilizador pressione uma tecla para continuar. No Python, tanto quanto sei, isto não existe mas podes substituir facilmente por um comando que já aprendemos aqui, o "raw_input".

```
>>> raw_input('Carrega numa tecla para continuares.')
```

```
Carrega numa tecla para continuares.
```

```
u"
```

```
>>>
```

E agora sim terminamos esta fase. A seguir vamos fazer uma série de instruções num ficheiro para te começares a inteirar de como hás-de mexer num para o meteres a funcionar. Ainda não largamos a consola porque quero que aprendam a usar uma ou outra biblioteca que vai ser muito útil especialmente para output gráfico como é o caso do matplotlib. Mas lá chegaremos por agora vamos analisar outra vez o ambiente de trabalho do Spyder.

Instruções em ficheiro

Ainda não chegamos à fase que vamos começar a fazer grandes programas (até porque ainda iremos trabalhar mais na consola), por agora veremos apenas o que fazer se quiséssemos fazer um até porque muitas questões deverão surgir na tua cabeça em relação a isto, entre as quais, onde terá de ficar o ficheiro (e o ficheiro é um ficheiro de texto com código semelhante ao que temos escrito na consola)?, como importo os módulos num ficheiro?, e como correr o ficheiro?. São tudo ótimas perguntas e em poucas páginas saberás a resposta para todas elas mas primeiro vou ensinar-vos a fazer um no Spyder, o IDE que eu estou a usar para explicar o que vos explico nesta publicação. Voltando a visualizar o Spyder podemos ver a área onde escreverás código dentro de um rectângulo vermelho na Figura 4.

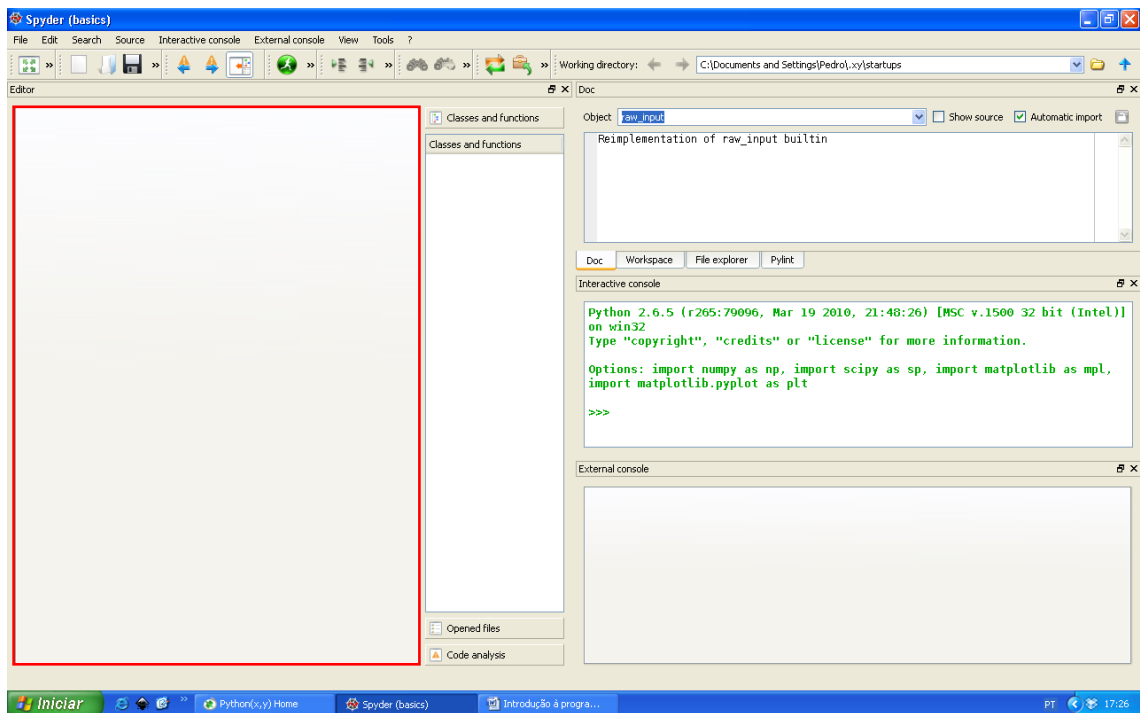


Figura 4 - IDE Spyder com evidênciação da área para escrever código em ficheiro (rectângulo vermelho).

Provavelmente nesta zona já terás um ficheiro aberto, mesmo que temporário mas se não tiveres basta que vás ao “File” e seleccionas “New” (Figura 5).

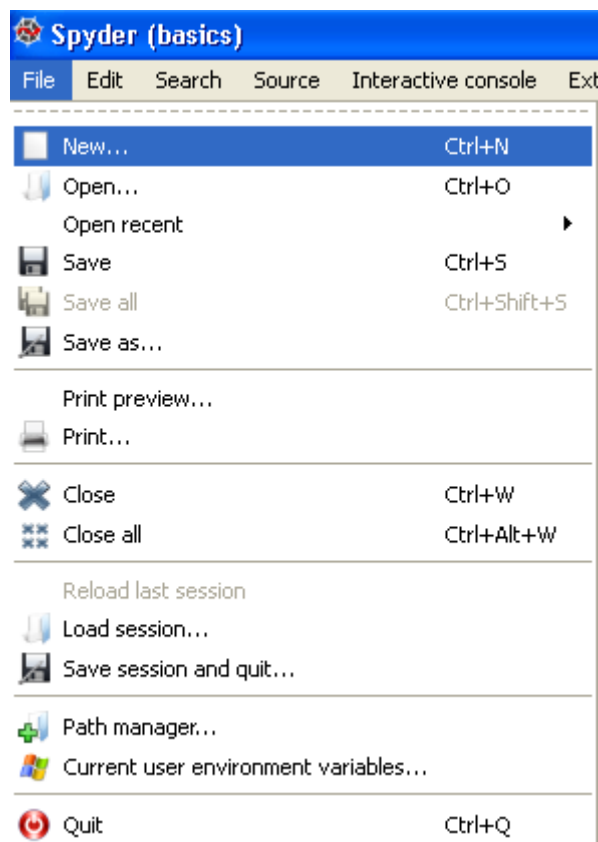
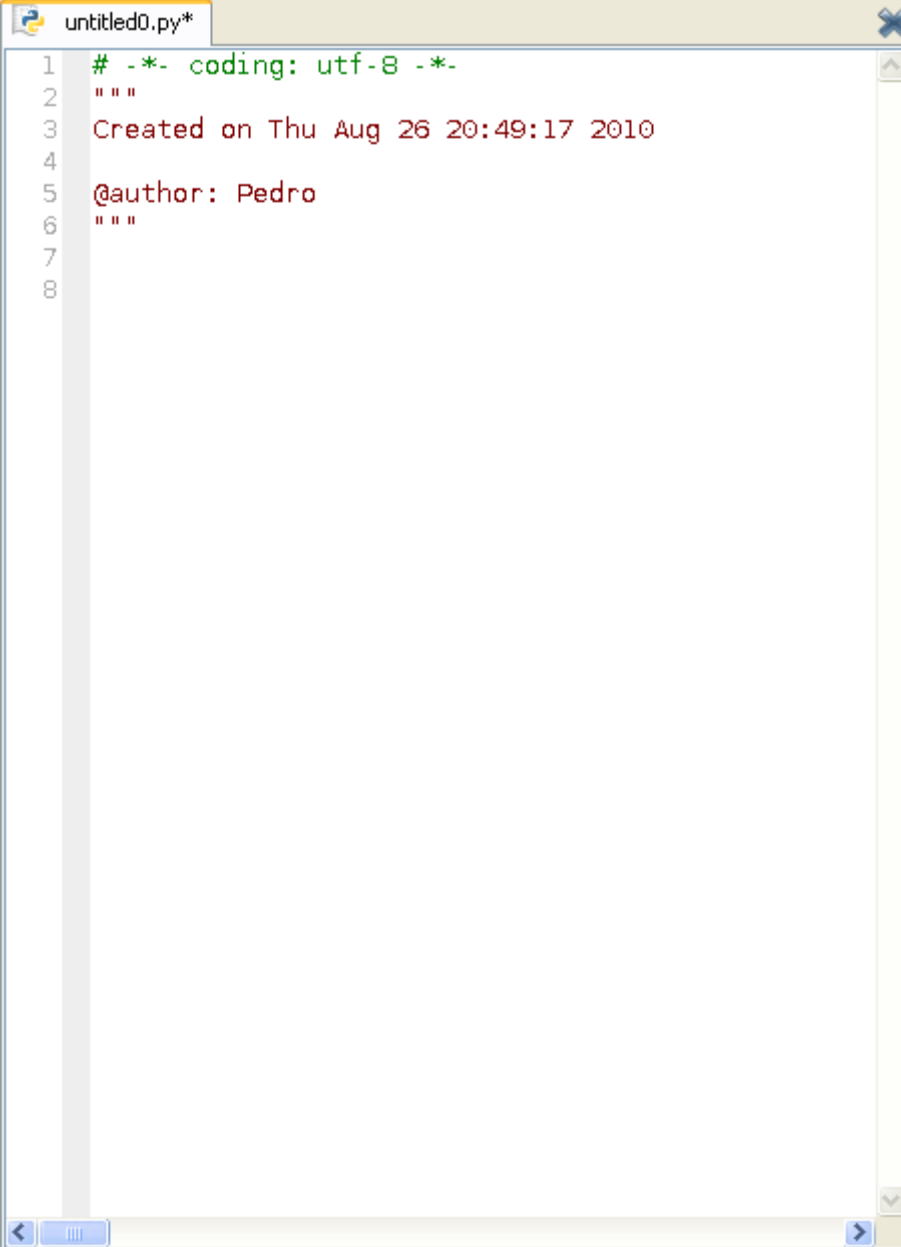


Figura 5 - Novo ficheiro no Spyder.

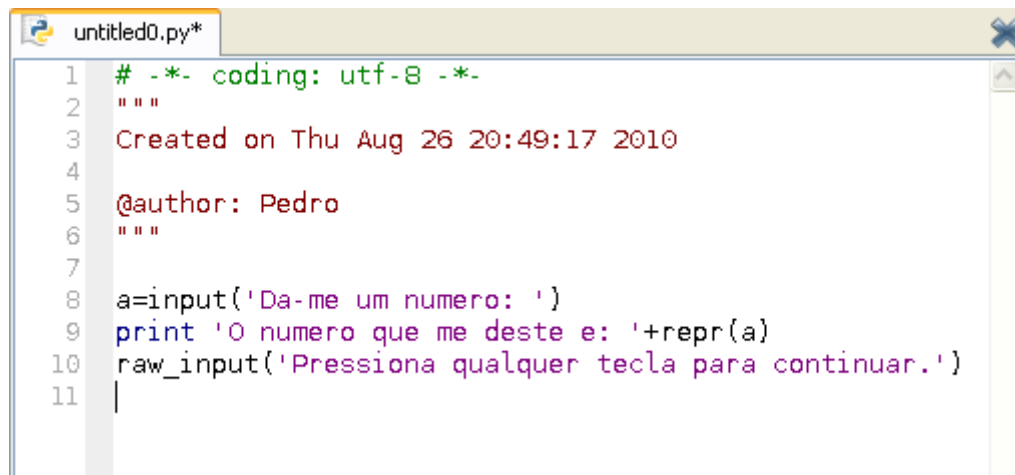
Agora no sítio do rectângulo vermelho deverá aparecer uma folha para escrever (que está embutida no Spyder mas é de facto um ficheiro ao estilo bloco de notas). Na Figura 6 temos um exemplo do que é suposto aparecer no Spyder após seleccionares novo ficheiro se é que já não terias o mesmo activo logo na abertura do Spyder.



```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Thu Aug 26 20:49:17 2010
4
5  @author: Pedro
6  """
7
8
```

Figura 6 - Ficheiro untitled0.py após pedirmos um novo ficheiro para escrever no Spyder.

No início do ficheiro vem uma instrução sobre a maneira como é interpretado o ficheiro (utf-8) entre outras particularidades do mesmo. Faz de conta que não está lá nada e escreve a seguir isso. O que eu vou escrever são três instruções que já aprendemos aqui em capítulos anteriores mas que agora aparecem em ficheiro (Figura 7):



```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Thu Aug 26 20:49:17 2010
4
5 @author: Pedro
6 """
7
8 a=input('Da-me um numero: ')
9 print 'O numero que me deste e: '+repr(a)
10 raw_input('Pressiona qualquer tecla para continuar.')
11 |
```

Figura 7 - O primeiro programa.

Agora para meteres isto a funcionar precisas de ir ao menu “Source” e escolheres “Run in external console” (Figura 8):

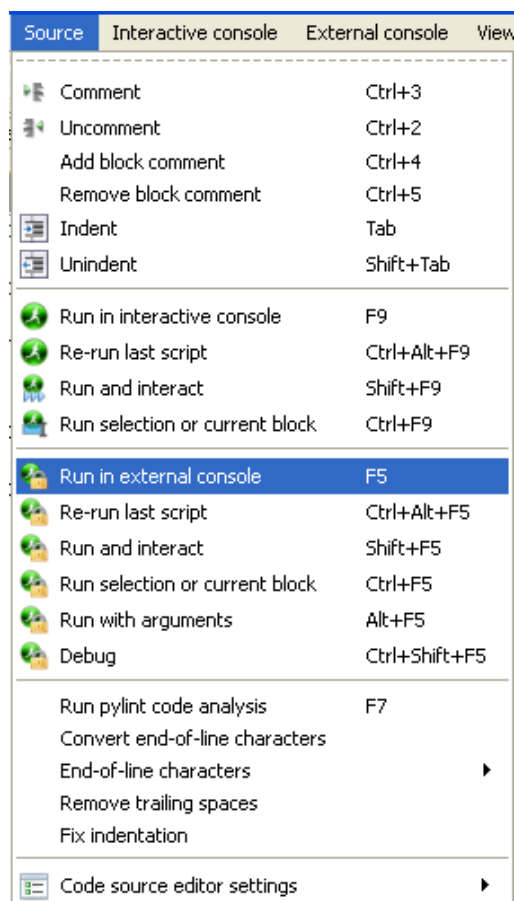


Figura 8 - Menu "Source" do Spyder.

Agora após fazeres isso, como é a primeira vez que vais correr este ficheiro, ele vai ter que o salvar nalgum sítio. Regra geral o que eu faço é salvar dentro da pasta que o Spyder cria

automaticamente aquando a instalação (para que todos os programas que faço no Spyder estejam no mesmo sítio). Salva o teu ficheiro (Figura 9).

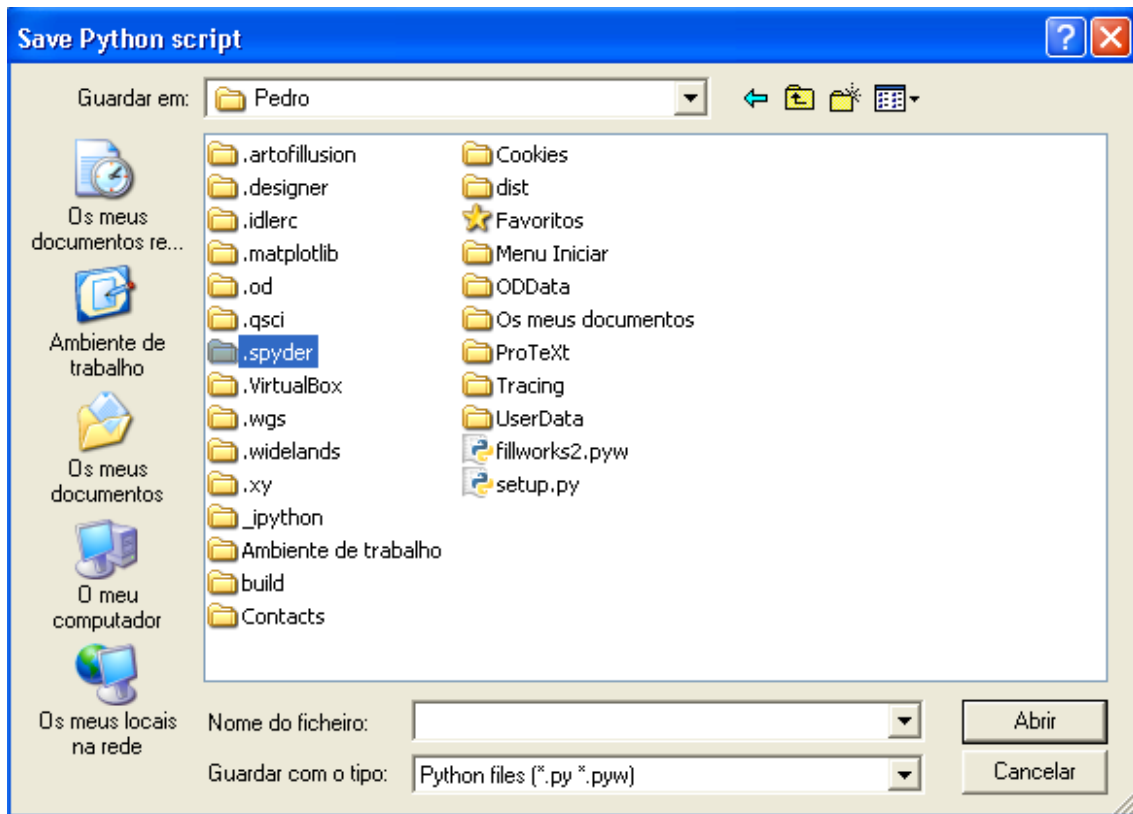


Figura 9 - Salvar ficheiro do Python.

E ele agora vai correr numa consola do interface do Spyder (Figura 10 e Figura 11):

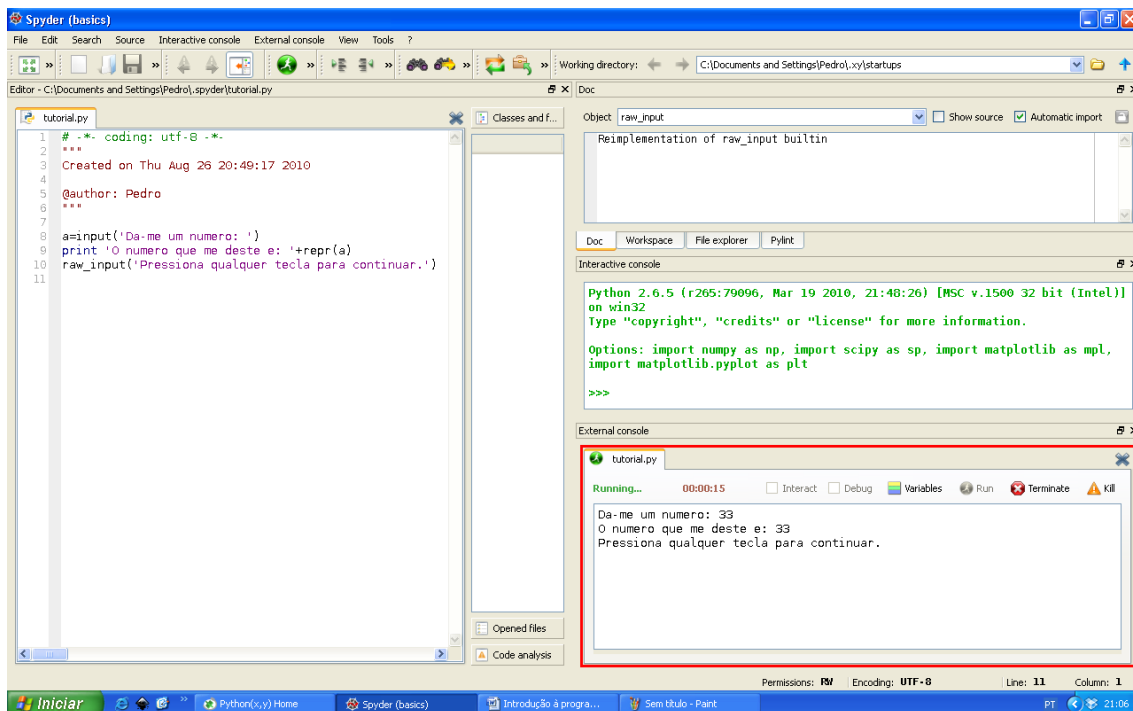


Figura 10 - Programa a funcionar na consola externa do Python.

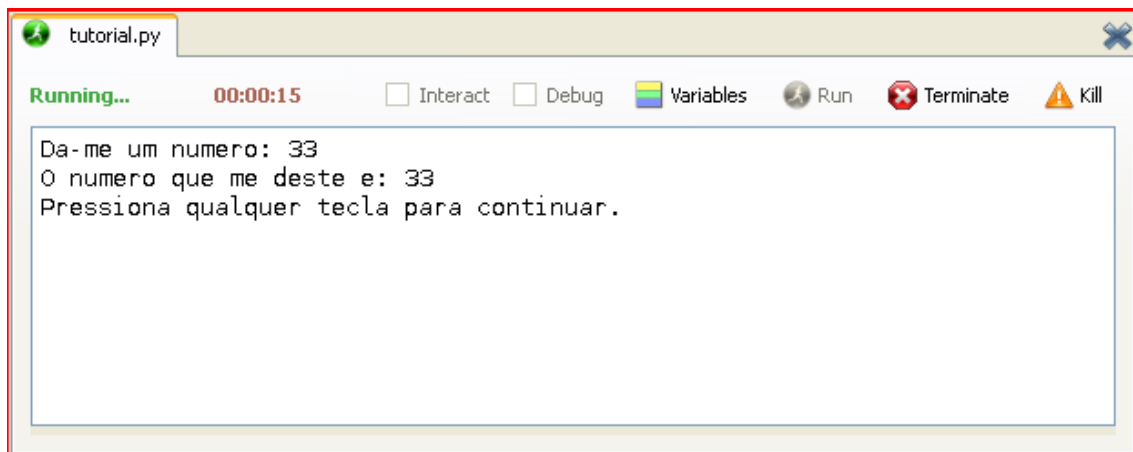


Figura 11 - Aproximação à consola externa do Python.

Percebeste tudo? Com a prática vais lá e vais acabar por fazer isto tudo muito rapidamente. Repara que para meter o teu programa a funcionar não tens de ir ao menu “Source”, podes simplesmente carregar no F5 que é uma tecla atalho para fazer a mesma coisa. Até agora nós só usamos a consola para meter os nossos comandos e se não pretendermos nada de muito definitivo é mais do que suficiente, no entanto eventualmente vais querer que os teus programas não só funcionem para um caso como para todos e assim ao escreveres num ficheiro ele vai ficar sempre guardado e poderás sempre utilizá-lo para novos projectos ou inclusive reutilizar o seu código para novos programas. Eventualmente nos vamos trabalhar quase exclusivamente em ficheiros mas por agora vamos continuar na consola porque ainda estamos num período de experimentação. Antes de voltar às raízes vou apenas indicar como importar módulos num ficheiro. No exemplo da Figura 12 já importei o módulo numpy.

```
tutorial.py*
1  # -*- coding: utf-8 -*-
2  """
3  Created on Thu Aug 26 20:49:17 2010
4
5  @author: Pedro
6  """
7  import numpy
8
9
10 a=input('Da-me um numero: ')
11 print 'O numero que me deste e: '+repr(a)
12
13 c=numpy.array([a,a,a])
14 print c
15
16 raw_input('Pressiona qualquer tecla para continuar.')
17
```

Figura 12 - Pequeno programa onde importei o modulo numpy e criei um vector estilo numpy à parte o que já estava feito do anterior.

A partir do programa que tinha feito anteriormente acrescentei mais algumas linhas de código onde importo o módulo numpy, crio um vector numpy (exactamente como aprendemos na consola) e faço um “print” como esse vector. Fácil, não é?! Como podes ver o resultado na consola é mais que esperado (Figura 13).

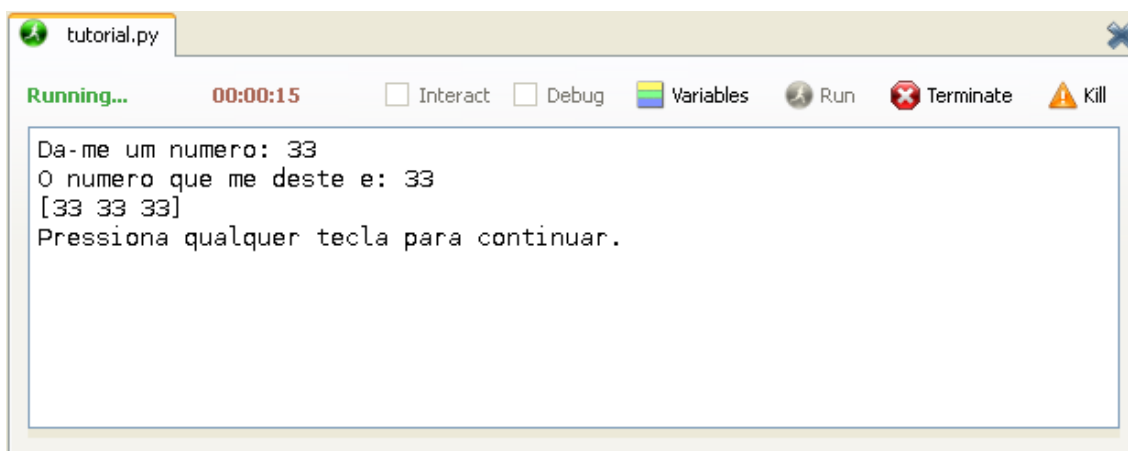


Figura 13 - Pequeno programa a funcionar na consola externa do Spyder.

O resto tu já deves imaginar. Se quiseres importar mais de um módulo basta que o faças em linhas diferentes e até podes usar aquelas formulações que aprendemos perto do inicio da publicação no ficheiro:

```
from __future__ import division
```

```
Import numpy as np
```

```
Import numpy.random as r
```

Apenas deixo um aviso. Se quiseres importar a divisão (*from __future__ import division*) mete sempre essa linha primeiro e só depois é que importas o resto porque poderá dar erro se for feito de outra maneira. Acho que por agora não há nada de muito urgente que preciso de explicar sobre o uso de ficheiros para programar. Podes ir experimentando fazer tu mesmo as tuas instruções em ficheiros e ver o que funciona e não funciona e mesmo como o Spyder te diz quando existem erros e onde eles estão. Isto dar-te-á treino e independência da consola que eventualmente irás precisar. Quando voltarmos a esta fase (de programar em ficheiro) iremos falar de temas como identificação, criação dos teus próprios módulos, funções e classes. Por agora vai experimentando e se não conseguires fazer alguma coisa pela a qual ainda não tenhamos passado com mais pormenor não te preocupes que em breve irás apanhar-lhe o jeito.

Matplotlib (output gráfico)

O matplotlib é uma biblioteca gráfica que serve (imaginem só...) para fazer gráficos (<http://matplotlib.sourceforge.net>). É muito completa e com alguma paciência conseguirás fazer gráficos de muito boa qualidade. Por agora vamos apenas iniciar o nosso conhecimento sobre esta biblioteca mas mais para a frente iremos analisa-la mais de perto para fazer coisas mais ambiciosas. Voltamos à consola mas tenciono ir introduzindo-te no habito de ires trabalhando em ficheiros para que essa progressão seja fácil. Começemos por ver como importar a biblioteca matplotlib.

```
>>> import matplotlib  
>>>
```

E já está, importaste o matplotlib para a consola. Agora o matplotlib é daquelas bibliotecas com muitos submódulos (estilo o random para o numpy, “numpy.random”, como aprendemos em capítulos anteriores), isto porque, como vais ver mais para a frente, dá para fazer montes de coisas. Assim para fazeres um gráfico normal a partir do matplotlib tens de usar o seu submódulo pyplot e dentro do pyplot a função plot (matplotlib.pyplot.plot). Para usar esta função criei dois vectores (um para entrar no eixo dos xx do meu gráfico e outro no eixo dos yy).

```
>>> a=numpy.linspace(1,100,100)  
>>> b=numpy.random.rand(100)  
>>>
```

Agora com estas duas séries de dados já posso fazer um gráfico:

```
>>> matplotlib.pyplot.plot(a,b)  
[matplotlib.lines.Line2D object at 0x0DF21F30]  
>>>
```

Se tudo correu como devia deverá aparecer-te um gráfico no Spyder (Figura 15) e se olhares com atenção vais ver uma série de botões e cada um deles tem uma função (Figura 14). A disquete, por exemplo, serve para salvar a tua imagem (portanto não precisas de usar a consola para fazer isso):



Figura 14 - Menu do gráfico de matplotlib.

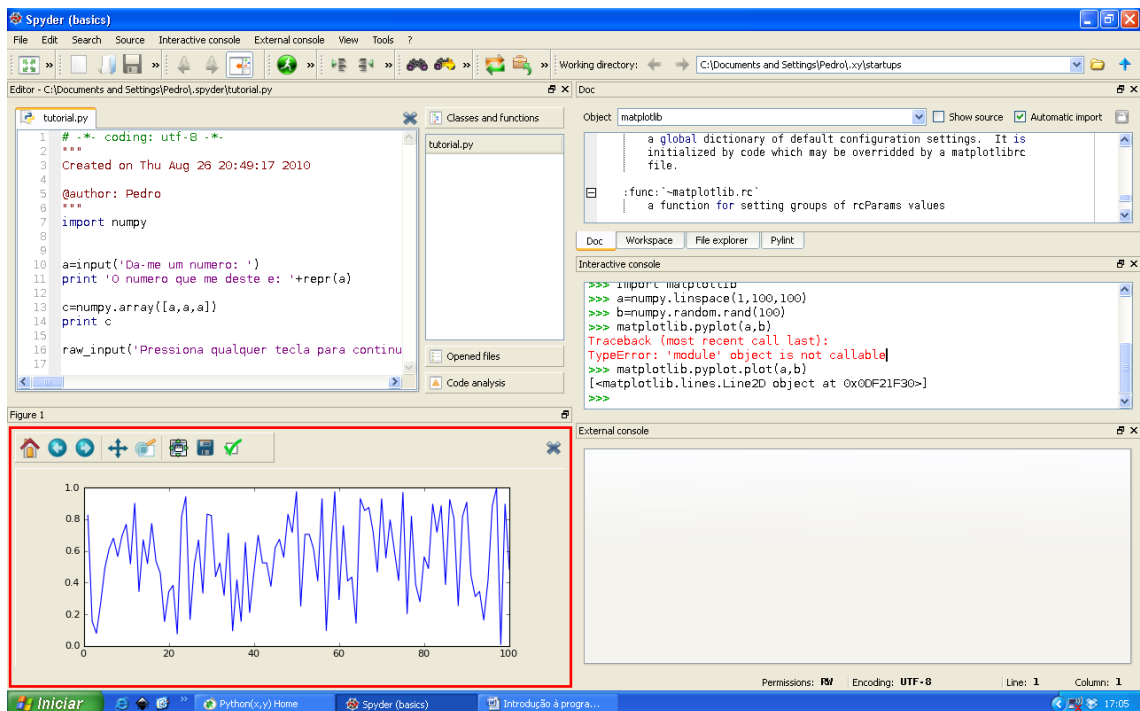


Figura 15 - Gráfico feito com a biblioteca matplotlib no Spyder (evidenciado com rectângulo vermelho).

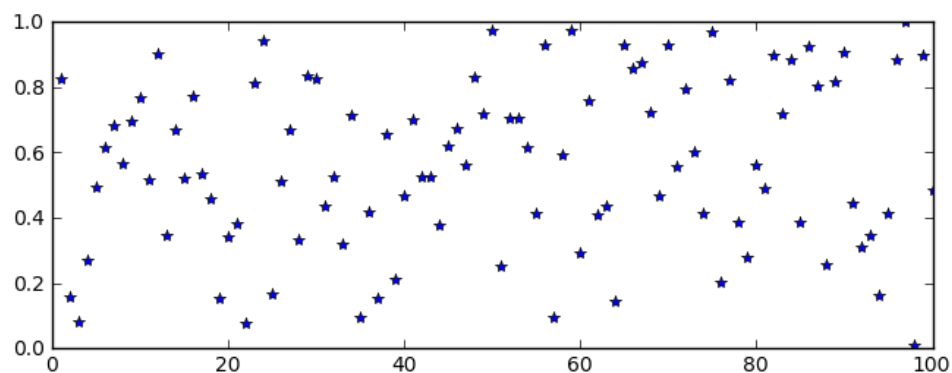
Se carregares no X que está no meu do gráfico saís do mesmo (mais uma vez não precisas de mexer na consola para fazeres isto). Mas nós só fizemos um gráfico de linhas azuis com duas séries de dados. Vamos agora aprender a fazer variações deste estilo de gráficos. Em vez de fazer um gráfico de linhas vou fazer um gráfico de pontos e quero que os meus pontos sejam estrelas (lembra-te de fechar o gráfico que tens agora antes de fazeres um novo):

```
>>> matplotlib.pyplot.plot(a,b,'*')
```

```
[<matplotlib.lines.Line2D object at 0x0E5A6190>]
```

```
>>>
```

O resultado é este:



Se quiser posso meter “+” em vez de estrelas e então faço:

```
>>> matplotlib.pyplot.plot(a,b,'+')
[<matplotlib.lines.Line2D object at 0x0E5A64B0>]
>>>
```

Ou até losangos:

```
>>> matplotlib.pyplot.plot(a,b,'D')
[<matplotlib.lines.Line2D object at 0x0E9588B0>]
>>>
```

Experimenta o que quiseses, existem símbolos para pontos e outros para linhas. Repara que para meteres o símbolo no gráfico basta que metas como primeiro argumento a série que está no eixo do x, segundo argumento, a série que está no eixo do y, e terceiro argumento o símbolo dentro de pelicas (matplotlib.pyplot.plot(xx,yy,'símbolo')). Não conheço todos os símbolos mas sei que estes existem: '.', '+', '*', '<', '>', 'D', 'H', 'p', 's', 'v', '^', 'x', '|', 'o'. E estes são só para os símbolos estilo pontos, depois temos os das linhas que os que conheço são estes: '-', '--', ':', '-.'. Vou fazer o mesmo gráfico com uma linha tracejada a título de exemplo:

```
>>> matplotlib.pyplot.plot(a,b,'--')
[<matplotlib.lines.Line2D object at 0x0EBF3510>]
>>>
```

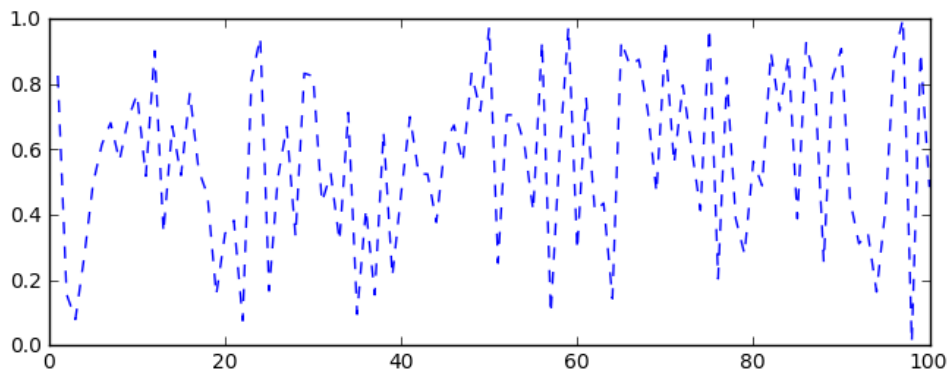


Tabela de símbolos para gráficos

Ainda são uns quantos símbolos e por isso para te facilitar a utilização dos mesmos vou meter aqui uma tabela que contém o resumo daqueles que conheço e o que fazem:

Símbolo (dentro de pelicas)	Descrição	Tipo
'.'	Ponto, losango pequeno	Pontos
'+'	Mais, +	Pontos
'*'	Estrela	Pontos
'<'	Triângulo à esquerda	Pontos
'>'	Triângulo à direita	Pontos
'v'	Triângulo para baixo	Pontos
'^'	Triângulo para cima	Pontos
'D'	Losango	Pontos
'H'	Hexágono	Pontos
'p'	Pentágono	Pontos
's'	Quadrado	Pontos
' '	Semi-rectas verticais	Pontos
'o'	Círculo	Pontos
'x'	Cruz diagonal	Pontos
'-'	Linha contínua	Linhas
'--'	Linha tracejada	Linhas
'.'	Linha por pontos	Linhas
'-.'	Linha – ponto – linha - ponto	Linhas

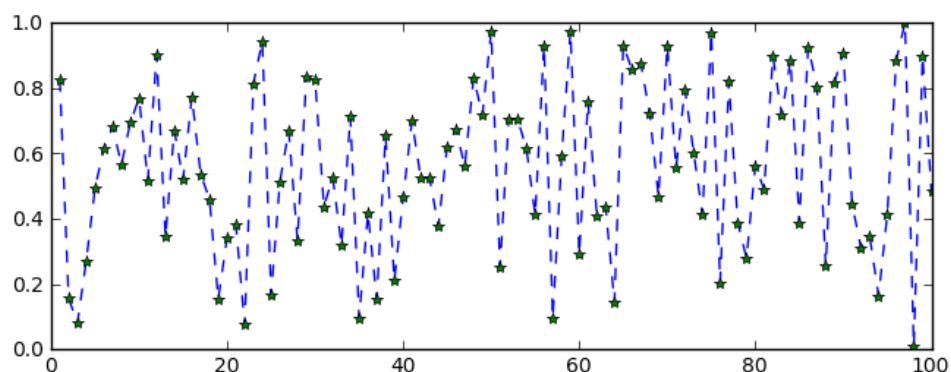
Tabela 1 - Símbolos para gráficos no matplotlib.

Podes meter mais de uma série em cada gráfico ou ainda a mesma série várias vezes. A vantagem disto é que podes fazer combinações de símbolos para teres um gráfico com o aspecto desejado.

```
>>> matplotlib.pyplot.plot(a,b,'--',a,b,'*')
```

```
[<matplotlib.lines.Line2D object at 0x0DEE4BD0>, <matplotlib.lines.Line2D object at 0x0E839A90>]
```

```
>>>
```



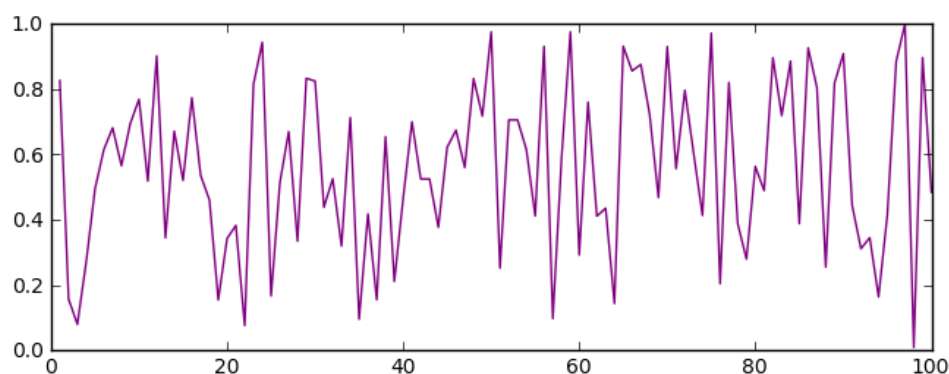
Agora evidentemente que para além do símbolo podemos mudar também a cor (e já deves ter reparado que se houver mais de uma série no mesmo gráfico ele muda a cor automaticamente).

Tabela de cores para gráficos

Existem muitas maneiras de inserir cores. Vou começar pela mais simples que é dizer o nome da cor directamente. Para escolheres uma cor (e se não o fizeres é, por defeito, azul) basta meteres um novo argumento na função “plot”:

```
>>> matplotlib.pyplot.plot(a,b,'-',color='purple')  
[<matplotlib.lines.Line2D object at 0x0EAC6F70>]  
  
>>>
```

Neste caso fiz um gráfico de linhas contínuas e cor púrpura.



Mais uma vez não sei todas as cores que o matplotlib aceita mas vou enunciar algumas que tenho a certeza:

String (dentro de pelicas)	Tradução	Cor
'white'	Branco	
'black'	Preto	
'blue'	Azul	
'red'	Vermelho	
'green'	Verde	
'orange'	Laranja	
'yellow'	Amarelo	
'purple'	Púrpura	
'brown'	Castanho	

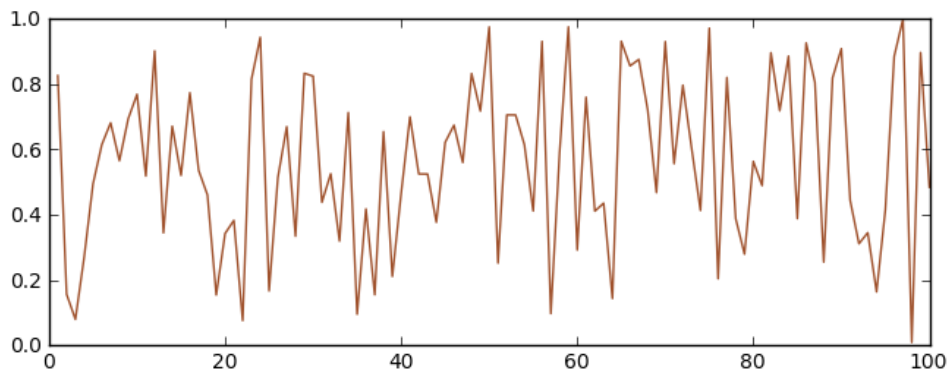
Tabela 2 - Tabela de cores simples.

É bastante provável que existam muitas mais como o ‘grey’ (cinzento) ou ‘turquoise’ (azul bebe) mas é tudo uma questão de se experimentar. De qualquer das maneiras vou ensinar um método de inserir cores que poderá tornar-se bastante mais fácil de usar se quiseses uma cor um pouco mais exótica ou até mesmo se quiseses uma cor banal mas com uma tonalidade um pouco diferente. Isto é especialmente importante porque este é um sistema de códigos internacional e por isso será comum não só ao matplotlib como também a muitos outros softwares.

Experimenta este código:

```
>>> matplotlib.pyplot.plot(a,b,'-',color='#A0522D')  
[<matplotlib.lines.Line2D object at 0x0F336D70>]  
>>>
```

O que eu inseri aqui é um castanho com uma tonalidade um pouco diferente do habitual.



O que está lá no sítio onde devia estar o nome da cor é um código. Esse código corresponde a uma cor que no caso do código que eu utilizei se chama sienna. A este estilo de códigos chama-se “html colors” ou código hexadecimal. Se procurares na internet verás imensos sites relacionados com isto que te dizem inúmeros códigos para inúmeras cores, se estiveres à procura de uma específica é uma questão de procurares (por exemplo, num motor de procura: *html color names*). Um desses sites é, por exemplo (sem querer fazer publicidade) o wikipedia após fazeres a procura por “Web colors” (http://en.wikipedia.org/wiki/Web_colors). Nessa página encontras uma lista enorme de cores para usares e muitas mais haverá. Vou usar por exemplo a cor “indian red” que é a primeira que aparece no título “X11 color names”. Uso o código que está na tabela com o nome “Hex Code – R G B” e acrescento um “#” no início. E então fica (vai tu também a esse site para veres ao que me refiro):

```
>>> matplotlib.pyplot.plot(a,b,'-',color='#CD5C5C')  
[<matplotlib.lines.Line2D object at 0x0E8BAC90>]  
>>>
```

Ou então a cor “DeepPink”:

```
>>> matplotlib.pyplot.plot(a,b,'-',color='#FF1493')  
[<matplotlib.lines.Line2D object at 0x0EA9B3F0>]  
>>>
```

É complicado explicar somente a escrever como usar isto mas conto que estes exemplos sejam o suficiente para tu apanhares o jeito. Fui buscar uma das centenas de tabelas que estão na net para inseri-la aqui como Tabela 3 (e só trás umas poucas cores) para que possas experimentar a partir desta publicação sem teres que estar à procura na net.

000000	000033	000066	000099	0000CC	0000FF
003300	003333	003366	003399	0033CC	0033FF
006600	006633	006666	006699	0066CC	0066FF
009900	009933	009966	009999	0099CC	0099FF
00CC00	00CC33	00CC66	00CC99	00CCCC	00CCFF
00FF00	00FF33	00FF66	00FF99	00FFCC	00FFFF
330000	330033	330066	330099	3300CC	3300FF
333300	333333	333366	333399	3333CC	3333FF
336600	336633	336666	336699	3366CC	3366FF
339900	339933	339966	339999	3399CC	3399FF
33CC00	33CC33	33CC66	33CC99	33CCCC	33CCFF
33FF00	33FF33	33FF66	33FF99	33FFCC	33FFFF
660000	660033	660066	660099	6600CC	6600FF
663300	663333	663366	663399	6633CC	6633FF
666600	666633	666666	666699	6666CC	6666FF
669900	669933	669966	669999	6699CC	6699FF
66CC00	66CC33	66CC66	66CC99	66CCCC	66CCFF
66FF00	66FF33	66FF66	66FF99	66FFCC	66FFFF
990000	990033	990066	990099	9900CC	9900FF
993300	993333	993366	993399	9933CC	9933FF
996600	996633	996666	996699	9966CC	9966FF
999900	999933	999966	999999	9999CC	9999FF
99CC00	99CC33	99CC66	99CC99	99CCCC	99CCFF
99FF00	99FF33	99FF66	99FF99	99FFCC	99FFFF
CC0000	CC0033	CC0066	CC0099	CC00CC	CC00FF
CC3300	CC3333	CC3366	CC3399	CC33CC	CC33FF
CC6600	CC6633	CC6666	CC6699	CC66CC	CC66FF
CC9900	CC9933	CC9966	CC9999	CC99CC	CC99FF
CCCC00	CCCC33	CCCC66	CCCC99	CCCCCC	CCCCFF
CCFF00	CCFF33	CCFF66	CCFF99	CCFFCC	CCFFFF
FF0000	FF0033	FF0066	FF0099	FF00CC	FF00FF
FF3300	FF3333	FF3366	FF3399	FF33CC	FF33FF
FF6600	FF6633	FF6666	FF6699	FF66CC	FF66FF
FF9900	FF9933	FF9966	FF9999	FF99CC	FF99FF
FFCC00	FFCC33	FFCC66	FFCC99	FFCCCC	FFCCFF
FFFF00	FFFF33	FFFF66	FFFF99	FFFFCC	FFFFFF

Tabela 3 - Tabela de cores html (não te esqueças de acrescentar o #).

Se tudo correr bem os códigos que estão nesta e em muitas outras tabelas deverão funcionar bem no matplotlib desde que te lembres de meter o cardinal, "#", antes do código.

Tamanho e transparência dos símbolos

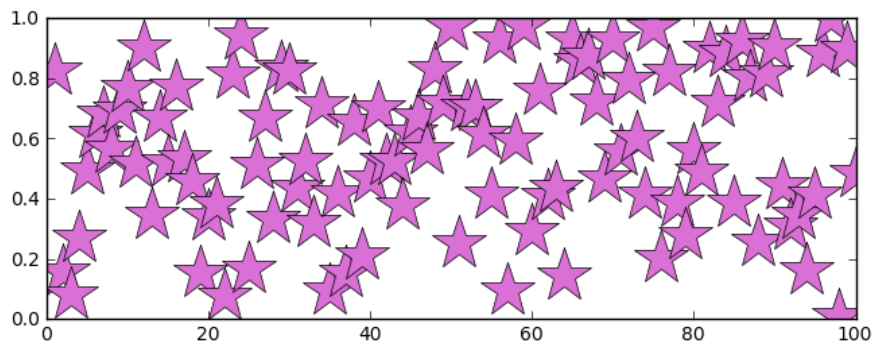
Agora vamos mexer com o tamanho que os nossos símbolos ou marcadores têm nos nossos gráficos. Mais uma vez acrescentamos um argumento (“markersize”) para dizer qual o tamanho. O tamanho vai desde 1 a..., não sei até onde chega mas penso que por defeito o matplotlib faz o tamanho dos seus símbolos como qualquer coisa à volta de 10. Vou então fazer um gráfico com o símbolos estrelas, “*”, cor orquídea, “#DA70D6” e tamanho do marcador de 30.

```
>>> matplotlib.pyplot.plot(a,b,'*',color='#DA70D6',markersize=30)
```

```
[<matplotlib.lines.Line2D object at 0x0E32BBF0>]
```

```
>>>
```

O resultado é este:



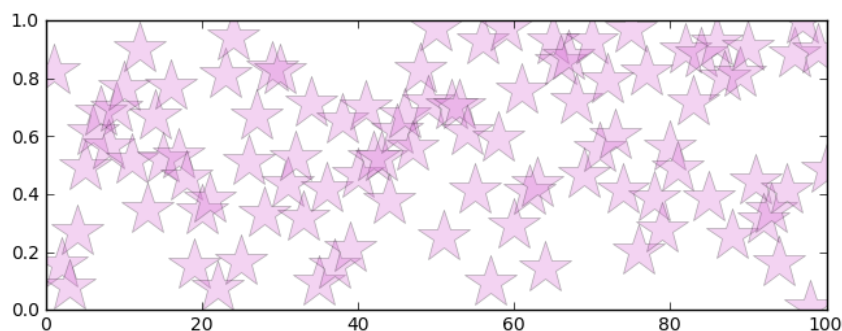
Agora é uma questão de experimentar outros tamanhos (lembra-te que por defeito é aproximadamente 10) para ganhares sensibilidade ao número que deves usar para teres o efeito que pretendes (isto funciona para todos os símbolos menos os de linhas). Em relação à transparência é tudo muito semelhante. Tens que meter mais um argumento, “alpha” e dar um número de 0 a 1 no qual 0 é transparente e 1 completamente opaco. Vou fazer este mesmo gráfico mas com transparência de 0.3.

```
>>> matplotlib.pyplot.plot(a,b,'*',color='#DA70D6',markersize=30,alpha=0.3)
```

```
[<matplotlib.lines.Line2D object at 0x0DF21ED0>]
```

```
>>>
```

O resultado é o seguinte:



Histogramas

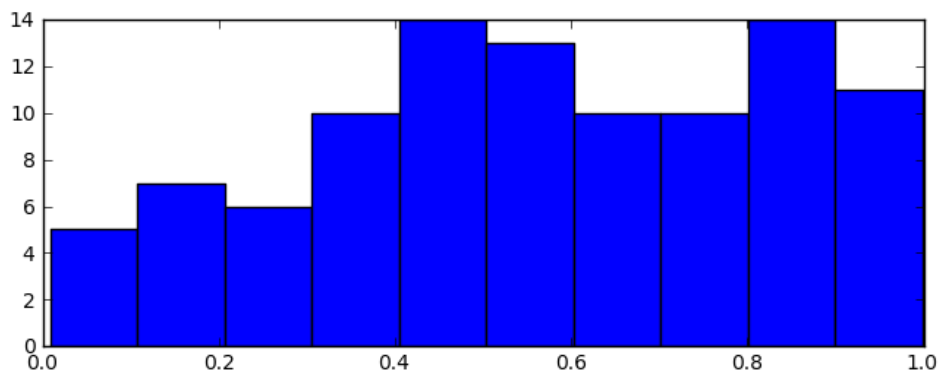
Fazer histogramas com o matplotlib também é muito simples e à semelhança do anterior quanto mais personalizado for mais argumentos vais ter. A função para fazer o histograma é “hist”, melhor dizendo “matplotlib.pyplot.hist”. Ao contrário dos gráficos que tínhamos anteriormente o histograma apenas utiliza uma série para fazer o seu gráfico. Vou dar um exemplo:

```
>>> matplotlib.pyplot.hist(b)

(array([ 5,  7,  6, 10, 14, 13, 10, 10, 14, 11]), array([ 0.0085041 ,  0.10755693,
 0.20660977,  0.3056626 ,  0.40471544,
          0.50376828,  0.60282111,  0.70187395,  0.80092678,  0.89997962,
          0.99903245]), <a list of 10 Patch objects>)

>>>
```

Tem em conta que os dados que me dá a mim serão diferentes dos que te dão a ti porque o vector que eu atribui à variável “b” provem de números aleatórios e portanto diferentes por cada vez que usares a função “rand”. Mas o resultado que deu a mim é:



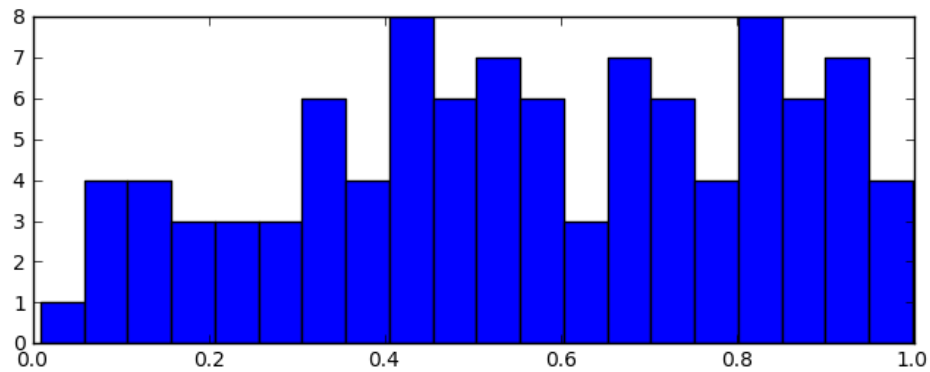
Agora, por defeito ele trás 10 classes mas podemos mudar isso inserindo o argumento “bins” ao nosso código:

```
>>> matplotlib.pyplot.hist(b,bins=20)

(array([1, 4, 4, 3, 3, 3, 6, 4, 8, 6, 7, 6, 3, 7, 6, 4, 8, 6, 7, 4]), array([
 0.0085041 ,  0.05803052,  0.10755693,  0.15708335,  0.20660977,
          0.25613619,  0.3056626 ,  0.35518902,  0.40471544,  0.45424186,
          0.50376828,  0.55329469,  0.60282111,  0.65234753,  0.70187395,
          0.75140036,  0.80092678,  0.8504532 ,  0.89997962,  0.94950603,
          0.99903245]), <a list of 20 Patch objects>)

>>>
```

Do qual resultou:



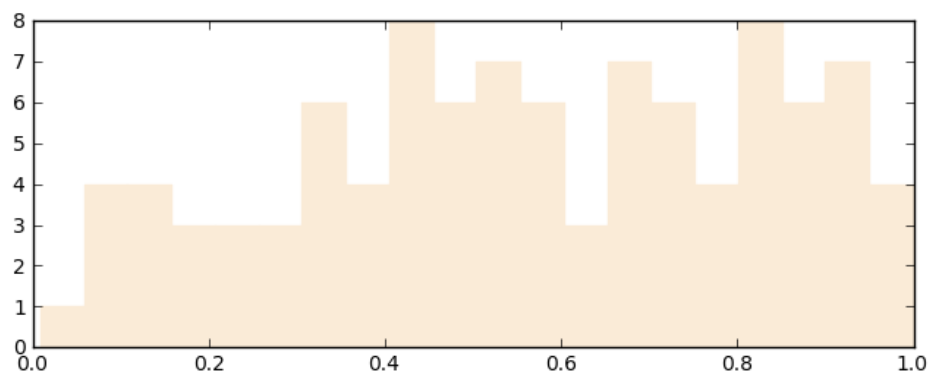
20 classes em vez de 10. Vou agora mudar a cor usando o argumento “color” que nós já conhecemos (escolhi a cor “Antique White”).

```
>>> matplotlib.pyplot.hist(b,bins=20,color='#FAEBD7')

(array([1, 4, 4, 3, 3, 3, 6, 4, 8, 6, 7, 6, 3, 7, 6, 4, 8, 6, 7, 4]), array([
0.0085041 , 0.05803052, 0.10755693, 0.15708335, 0.20660977,
    0.25613619, 0.3056626 , 0.35518902, 0.40471544, 0.45424186,
    0.50376828, 0.55329469, 0.60282111, 0.65234753, 0.70187395,
    0.75140036, 0.80092678, 0.8504532 , 0.89997962, 0.94950603,
    0.99903245]), <a list of 20 Patch objects>)

>>>
```

E o gráfico é...

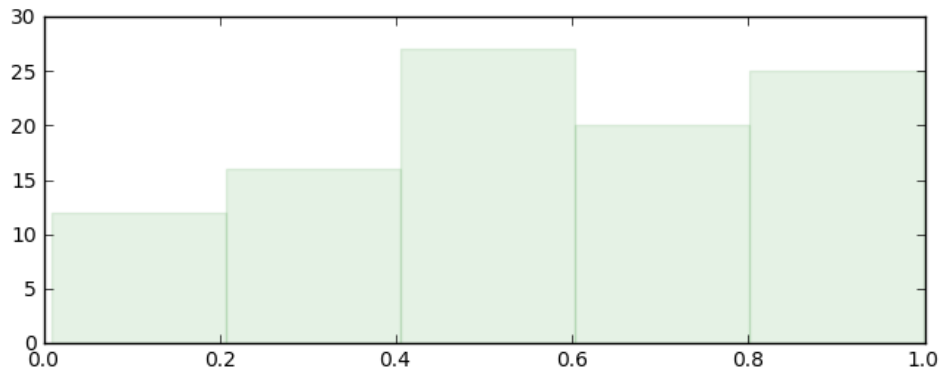


A transparência também funciona da mesma maneira (“alpha”):

```
>>> matplotlib.pyplot.hist(b,bins=5,color='green',alpha=0.1)

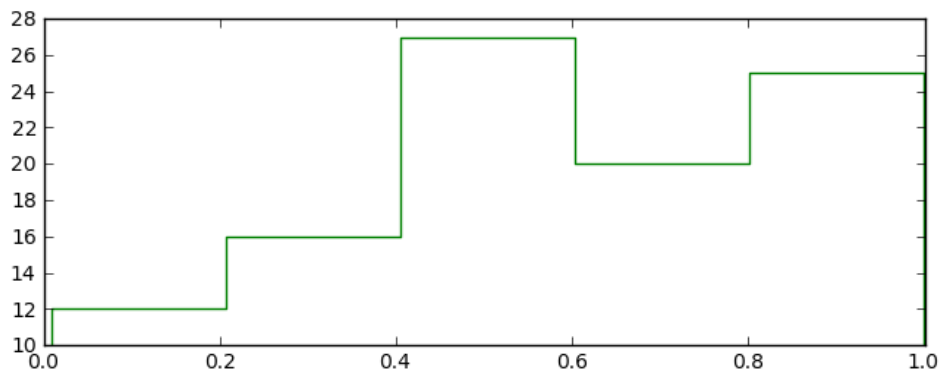
(array([12, 16, 27, 20, 25]), array([ 0.0085041 , 0.20660977, 0.40471544,
0.60282111, 0.80092678,
    0.99903245]), <a list of 5 Patch objects>)

>>>
```



E agora vamos aprender uns argumentos específicos para o caso dos histogramas como é o caso do “histtype” com o qual podes mudar o estilo do teu histograma (os que conheço são o ‘bar’, o ‘step’ e o ‘stepfilled’):

```
>>> matplotlib.pyplot.hist(b,bins=5,color='green',alpha=0.1,histtype='step')
(array([12, 16, 27, 20, 25]), array([ 0.0085041 ,  0.20660977,  0.40471544,
 0.60282111,  0.80092678,
 0.99903245]), <a list of 1 Patch objects>)
>>>
```



Podemos ainda fazer mais coisas como torna-lo cumulativo, mudar o alinhamento ou a orientação mas em favor da brevidade vou evitar esses pormenores por agora. Neste momento é importante que percebas a lógica que está por detrás de utilizar esta biblioteca e não sabe-la utiliza-la na sua plenitude (até porque eu não sei e por isso dificilmente iria conseguir ensinar tal coisa). Vou ainda ensinar a fazer mais um tipo de gráfico que é a projecção estereográfica.

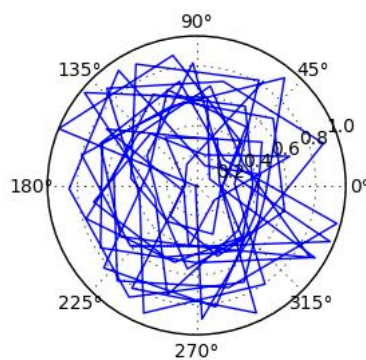
Projecção estereográfica

Regra geral projecções estereográficas funcionam com coordenadas de ângulos e por isso os dados que temos usado são tudo menos adequados para fazer este tipo de gráfico (que acho que foi feito para funcionar com ângulos radianos mas não tenho a certeza) mas como ele aceita tudo vamos fazê-lo na mesma. Para fazer uma projecção estereográfica basta utilizar o comando “polar” e inserir duas séries de dados:

```
>>> matplotlib.pyplot.polar(a,b)

[<matplotlib.lines.Line2D object at 0x0E2D9610>]

>>>
```



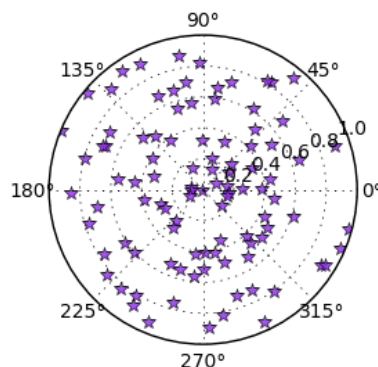
Não é muito bonito mas percebe-se a intenção. De qualquer maneira vamos fazer umas mudanças que por esta altura já deverás reconhecer.

```
>>> matplotlib.pyplot.polar(a,b,'*',color='#8A2BE2',markersize=7,alpha=0.8)

[<matplotlib.lines.Line2D object at 0x0E8B4470>]

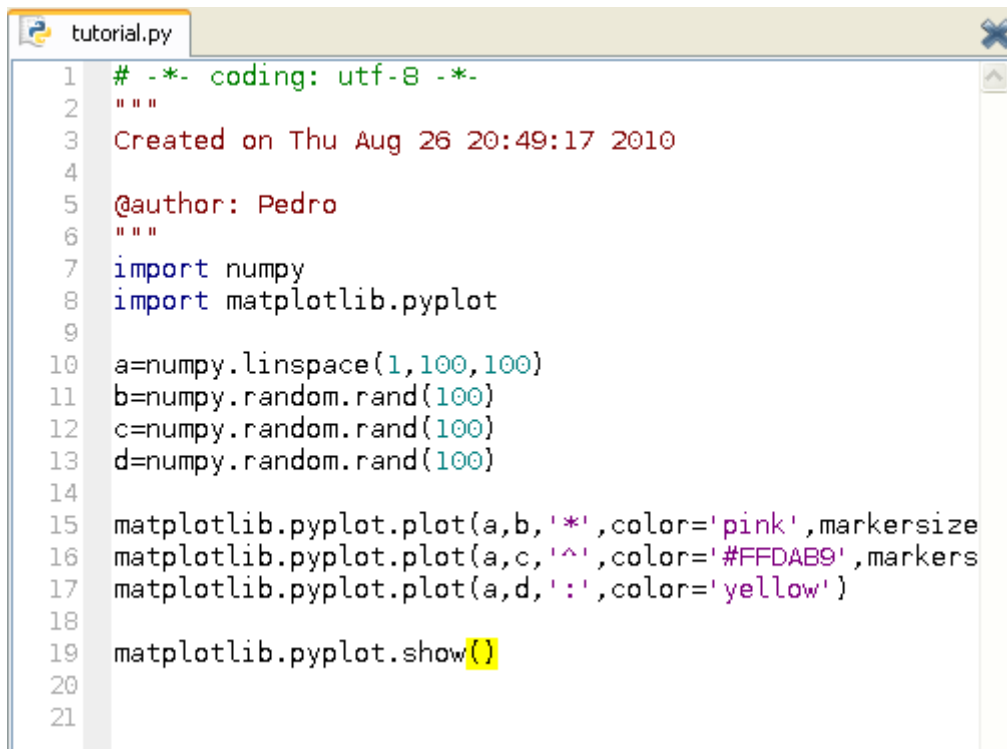
>>>
```

Reconheces todos os argumentos que inseri? Primeiro foram as séries, “a” e “b”, depois o símbolo, “*”, depois a cor, que é o Blue Violet, e finalmente o tamanho do marcador, “markersize” e transparência, “alpha”.



Gráficos matplotlib em ficheiro

Antes de te mostrar um gráfico num ficheiro vou apenas fazer-te notar que, em vez de estares a meter várias séries de dados no mesmo comando “plot”, por exemplo, podes ir actualizando aquele que já tens. Talvez já tenhas reparado que se não fores apagando os gráficos no Python sempre que fores metendo um novo gráfico ele vai actualizando o que lá está. É assim também que vou fazer no ficheiro. Vou criar quatro séries de dados em que uma, “a”, é a base, e as outras diferentes gráficos que no fim vão finir no mesmo com características diferentes (“b”, “c” e “d”). Então no meu ficheiro escrevi o seguinte:

A screenshot of a text editor window titled "tutorial.py". The code is as follows:

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Thu Aug 26 20:49:17 2010
4
5  @author: Pedro
6  """
7  import numpy
8  import matplotlib.pyplot
9
10 a=numpy.linspace(1,100,100)
11 b=numpy.random.rand(100)
12 c=numpy.random.rand(100)
13 d=numpy.random.rand(100)
14
15 matplotlib.pyplot.plot(a,b,'*',color='pink',markersize
16 matplotlib.pyplot.plot(a,c,'^',color='#FFDAB9',markers
17 matplotlib.pyplot.plot(a,d,':',color='yellow')
18
19 matplotlib.pyplot.show()
```

Deves notar duas coisas: primeiro para importar o matplotlib não utilizei o “import matplotlib” mas sim importei o submódulo pyplot directamente (“import matplotlib.pyplot”), segundo, ao contrário da consola que está sempre a actualizar um gráfico, no ficheiro precisas de dizer quando é que ele faz o gráfico com todas as instruções que meteste e fazes isso com o comando “show”, melhor dizendo “matplotlib.pyplot.show()”. A seguir mostro o código escrito directamente aqui porque na figura aparece cortado (a negrito estão as linhas que eu comentei no texto).

```
import numpy
```

```
import matplotlib.pyplot
```

```
a=numpy.linspace(1,100,100)
```

```
b=numpy.random.rand(100)
```

```
c=numpy.random.rand(100)
```

```
d=numpy.random.rand(100)
```

```
matplotlib.pyplot.plot(a,b,'*',color='pink',markersize=20)
```

```
matplotlib.pyplot.plot(a,c,'^',color='#FFDAB9',markersize=10)
```

```
matplotlib.pyplot.plot(a,d,':',color='yellow')
```

```
matplotlib.pyplot.show()
```

O resultado vai ser uma janela de Windows totalmente manipulável e com os mesmos botões que vias no Spyder mas sem estar preso ao mesmo (Figura 16).

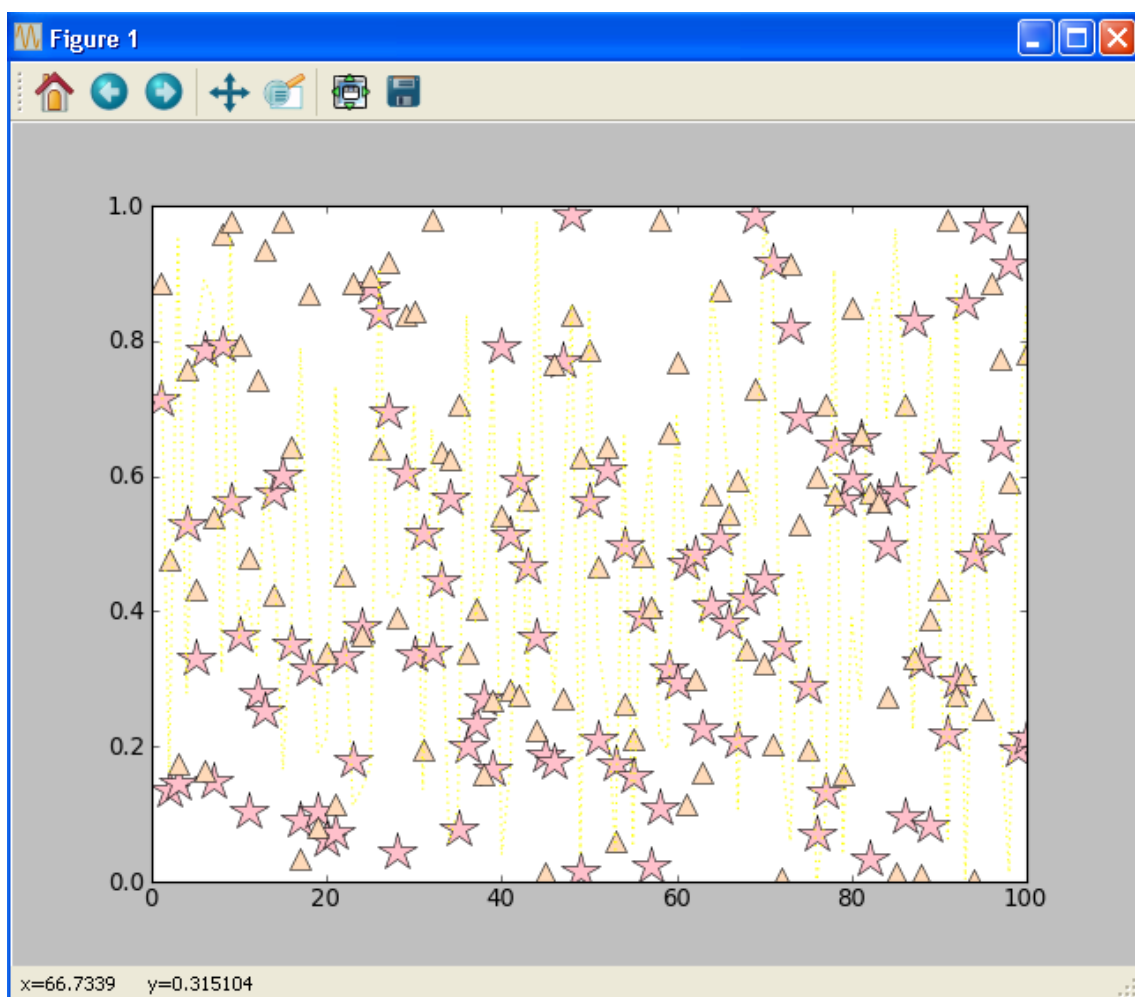


Figura 16 - Saída de uma janela do matplotlib após escrevermos o gráfico num ficheiro.

Se te deres bem a fazer os teus programas em ficheiros isto é uma vantagem espectacular visto que o utilizador recebe uma janela já totalmente feita que pode maximizar e minimizar e gravar a figura a partir dela. Mais para a frente voltarei a abordar esta biblioteca para fazermos outro tipo de gráficos (iremos ver como fazer gráficos 3D, superfícies, topográficos, etc...) mas já deverás ter ficado com a maioria das noções necessárias para trabalhares com ela.

Numpy (tratamento numérico)

Nós já falámos aqui sobre o numpy e por várias vezes o temos vindo a utilizar mas vamos agora conhecer um pouco melhor esta biblioteca. O numpy tal como o matplotlib tem submódulos entre os quais o “random” (numpy.random). Outro submódulo é o “linalg” (numpy.linalg) que se refere a funções para álgebra linear e onde podes encontrar funções para calcular o determinante de uma matriz, o produto interno, o produto externo, etc. Outro ainda é o “fft” (numpy.fft) para transformadas de Fourier. Não vou abordar estes módulos por várias razões, entre as quais, eu nunca mexi com eles, vamos sim ver algumas funções do numpy que podem simplificar os nossos programas por fazerem acções que são muito utilizadas como saber o máximo ou o mínimo de um vector, a média, etc. Comecei por criar dois vectores em recurso ao numpy, tal como tenho feito até aqui.

```
>>> import numpy
>>> a=numpy.random.rand(100)
>>> b=numpy.random.rand(100)
```

Agora se pretender saber o máximo ou o mínimo do vector da variável “a” basta-me usar os comandos “amax” ou “amin”.

```
>>> numpy.amax(a)
0.97438154255072784
>>> numpy.amin(a)
0.013499772854860814
>>>
```

Para saber o a “distância” entre máximo e mínimo (máximo – mínimo) podemos utilizar o comando “ptp”.

```
>>> numpy.ptp(a)
0.96088176969586703
>>>
```

Para saber a média e mediana basta-me usar os comandos correspondentes “mean” e “median”.

```
>>> numpy.mean(a)
0.49921198850610099
>>> numpy.median(a)
0.46556814379226164
>>>
```

Para o desvio padrão e variância usamos o “std” e “var”.

```
>>> numpy.std(a)
0.27447455627907247
>>> numpy.var(a)
0.075336282044593722
>>>
```

O numpy também tem várias funções matemáticas e constantes embutidas tais como:

```
>>> c=50
>>> numpy.cos(c) # co-seno
0.96496602849211333
>>> numpy.sin(c) # seno
-0.26237485370392877
>>> numpy.tan(c) # tangente
-0.27190061199763077
>>> numpy.exp(c) # exponencial
5.184705528587072e+21
>>> numpy.log(c) # logaritmo, numpy.log10() para logaritmo de base 10
3.912023005428146
>>> numpy.sqrt(c) # raiz quadrada
7.0710678118654755
>>> numpy.pi # constante pi
3.1415926535897931
>>> numpy.e # constante e
2.7182818284590451
>>>
```

Como já debes ter visto antes também existem várias funções para criar vectores ou matrizes de um determinado tipo com, por exemplo, os comandos “identity” para criar matrizes identidade, “zeros” para vectores ou matrizes de zeros, “ones” para vectores ou matrizes de uns, “random.rand” para vectores ou matrizes de números aleatórios de 0 a 1, “linspace” para vectores com números igualmente espaçados e “arange” semelhante ao linspace mas em vez de dizermos quantos números queremos simplesmente indicamos qual o passo que tem de existir entre eles (e ele calcula automaticamente quantos têm de haver).

```
>>> numpy.identity(3)
array([[ 1.,  0.,  0.]
```

```

        [ 0.,  1.,  0.],
        [ 0.,  0.,  1.]])
>>> numpy.zeros((3,3))
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> numpy.ones((3,3))
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> numpy.random.rand(3,3)
array([[ 0.58005453,  0.59124533,  0.42926947],
       [ 0.1440495 ,  0.7115918 ,  0.20709425],
       [ 0.67985668,  0.36375812,  0.7988591 ]])
>>> numpy.linspace(1,5,5)
array([ 1.,  2.,  3.,  4.,  5.])
>>> numpy.arange(1,10,3)
array([1, 4, 7])
>>>

```

Fiz esta pequena segunda apresentação a esta biblioteca para que tenhas noção da quantidade de funções que pode haver em cada um destes conjuntos de funções e que por vezes, vale a pena procurar por uma função que resolva o nosso problema ao invés de estarmos nós a programa-la e com isso perder tempo possivelmente importante. Isto não só é importante para cada biblioteca como para o número de bibliotecas que estão disponíveis na net para Python.

Trabalhar directamente no ficheiro

A partir daqui só voltaremos à consola para pequenos testes porque pretendo o ideal é que ganhes prática em trabalhar directamente em ficheiros. Felizmente como no Spyder a consola está mesmo ao lado do sítio onde escrevemos os nossos ficheiros (e um não influencia o outro) podes estar a escrever no ficheiro e ir experimentar à consola se tiveres alguma dúvida de como funciona um comando ou outro. Vamos começar por escrever uns pequenos programas no ficheiro e depois passaremos para outros objectivos mais ambiciosos, mas antes precisamos de perceber algumas das instruções mais fundamentais da programação, ciclos “while” e perguntas “if”, e no caso especial do Python indentação.

Ciclos While

Ciclos while são muito comuns a quase todas as linguagens de programação e é o que te vai permitir fazer tarefas repetitivas que não usando programação te iriam demorar muito mais tempo. Um ciclo While funciona como uma pergunta permanente e enquanto a resposta a essa pergunta for verdadeira então as instruções que estão dentro do ciclo são aplicadas. Vou meter um pequeno exemplo em código (não o escrevas em lado nenhum, presta apenas atenção):

```
i=0  
  
while i<2:  
    i=i+1
```

Um ciclo while é isto vou fazer à mão o que ele faz automaticamente:

“i” é igual a 0

-> (entra o ciclo while) “i” (0) é menor que 2? A resposta é sim por isso $i=0+1=1$

-- “i” passou a ser 1.

-> (segunda volta do while) “i” (1) é menor que 2? A resposta é sim por isso $i=1+1=2$

-- “i” passou a ser 2.

-> (terceira volta do while) “i” (2) é menor que 2? A resposta é não por isso o ciclo while acaba.

-- “i” continua a ser 2.

A única instrução que inserimos dentro do ciclo while é o “i” a somar 1 a si próprio até igualar o 2. Não parece muito útil mas imagina que queres fazer o que a função “zeros” ou “ones” faz (vectores de zeros e uns) mas com 2 (podia ser a função “twos”). Podias escrever assim:

```
import numpy

a=numpy.zeros(3)

i=0

while i<3:

    a[i]=2

    i=i+1
```

Consegues perceber o que este código faria? Estamos a percorrer um vector inteiro começando na posição $i=0$ (lembra-te que zero é a primeira posição em Python) e em todas as posições até chegar ao final estamos a dizer que a posição $a[i]$ é igual a 2. Repara que o contador continua a estar lá porque se não estivesse o “i” nunca avançava e o ciclo iria estar a funcionar permanentemente. Se por exemplo eu quisesse percorrer uma matriz em vez de um vector posso usar dois ciclos while, um para as linhas outro para as colunas:

```
import numpy

a=numpy.zeros((3,2))

j=0

while j<2:

    i=0

    while i<3:

        a[i,j]=2

        i=i+1

    j=j+1
```

Com isto percorri a matriz inteira porque repara, primeiro entrei no ciclo em que faço a contagem das colunas:

```
-> (entra o primeiro ciclo while) “j” (0) é menor que 2? Sim, então i=0 e

    -> (entra o segundo while) “i” (0) é menor que 3? Sim, então a[i,j] (a[0,0]) é
    igual a 2 e i=0+1=1. “i” é agora igual a 1.

    -> (continua o segundo ciclo) “i” (1) é menor que 3? Sim, então a[i,j]
    (a[1,0]) é igual a 2 e i=1+1=2. “i” é agora igual a 2.

    -> (continua o segundo ciclo) “i” (2) é menor que 3? Sim, então a[i,j]
    (a[2,0]) é igual a 2 e i=2+1=3. “i” é agora igual a 3.

    -> (continua o segundo ciclo) “i” (3) é menor que 3? Não, então sai do ciclo.

    -> j=0+1=1

-> (reinicia o primeiro ciclo agora com j=1) “j” (1) é menor que 2? Sim então i=0 e
```

-> (entra no segundo ciclo) "i" (0) é menor que 3? Sim então a[i,j] (a[0,1]) é igual a 2 e i=0+1=1. "i" é agora igual a 1.

-> (continua o segundo ciclo) "i" (1) é menor que 3? Sim, então a[i,j] (a[1,1]) é igual a 2 e i=1+1=2. "i" é agora igual a 2.

-> (continua o segundo ciclo) "i" (2) é menor que 3? Sim, então a[i,j] (a[2,1]) é igual a 2 e i=2+1=3. "i" é agora igual a 3.

-> (continua o segundo ciclo) "i" (3) é menor que 3? Não, então sai do ciclo.

-> j=1+1=2

-> (reinicia o primeiro ciclo agora com j=2) "j" (2) é menor que 2? Não, então sai fora do ciclo.

A tua matriz está dispersa da seguinte maneira (a[i,j]):

a[0,0]	a[0,1]
a[1,0]	a[1,1]
a[2,0]	a[2,1]

Originalmente começou por ser isto:

0	0
0	0
0	0

E depois posição a posição foi transformando-se nisto:

2	0	2	0	2	0	2	2	2	2	2	2
0	0	→ 2	0	→ 2	0	→ 2	0	→ 2	2	→ 2	2
0	0	0	0	2	0	2	0	2	0	2	2

Se tiveres dificuldade em perceber tenta perceber este esquema em matriz com o texto que escrevi acima a descrever os ciclos while do princípio ao fim. Os ciclos while poderão ser um pouco ser um pouco difíceis de perceber ao princípio mas logo que entendas a lógica sair-te-ão naturalmente. Em termos de código o que está alguns dos trechos de código que escrevi acima funcionam perfeitamente. Experimenta meter isto num ficheiro e vê o que resulta quando o correres:

```
import numpy
```

```
a=numpy.zeros(3)
```

```
i=0
```

```
while i<3:
```

```
    a[i]=2
```

```
    i=i+1
```

```
print a
```

```
raw_input('Pressiona qualquer tecla para continuar...')
```

Agora quero ensinar sobre indentação visto que já a utilizamos mas não disse porquê.

Indentação

Antes de explicar o que é tenho que dizer que ao contrário de outras linguagens (a esmagadora maioria) no Python a indentação é obrigatória. A razão de o ser é muito simples. Python foi uma linguagem feita especialmente para ser fácil de ler e escrever. Indentação é algo que a grande maioria dos programadores fazem por isso as pessoas que criaram o Python decidiram tirar proveito disso de maneira a tornar a linguagem ainda mais legível. A indentação é uma arrumação do código. Vou recuperar um trecho de código que escrevi mais acima:

```
a=numpy.zeros(3)

i=0

while i<3:

    a[i]=2

    i=i+1
```

Na primeira e segunda a linha as instruções estão encostadas à esquerda. Depois chega um ciclo while e a partir daí as instruções estão um pouco mais à direita. O facto de essas instruções estarem mais à direita dizem que só são para ser feitas dentro do while. Quando nós metemos um segundo while dentro do primeiro temos de indenta-lo também para ficar dentro do primeiro e as instruções que ficam dentro do primeiro ficam no primeiro espaço (geralmente usa-se o “tab” para meter os espaços), as instruções do segundo ficam no segundo espaço.

```
import numpy

a=numpy.zeros((3,2))

j=0

while j<2:

    i=0

    while i<3:

        a[i,j]=2

        i=i+1

    j=j+1
```

espaço1▲espaço2▲

O segundo while é uma instrução do primeiro mas o `a[i,j]=2` é uma instrução do segundo e por isso estão no segundo espaço. A indentação é obrigatório porque se não existisse o Python não

saberia dizer onde é que acabam as instruções dos ciclos while e outros semelhantes. Existem linguagens (como o C++) onde se utilizam caracteres estilo chavetas para determinar o início e o fim das instruções dos while (em C++ até metes o fim da instrução com “;”).

```
while (i<3)

{

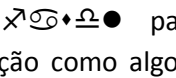
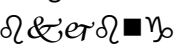
    a[i]=2;

    i=i+1;

}
```

E assim poderias escrever o código assim:

```
while (i<3) {a[i]=2; i=i+1;}
```

Tudo numa só linha mas se imaginares que vais ter dezenas, centenas ou milhares de linhas de código num programa, perceber o que está lá escrito assim é  pa !!!(perdoem-me o francês). Assim o Python utiliza a indentação como algo necessário e passa a ser escusado usar chavetas ou “;” ou qualquer outra simbologia. A indentação em pouco tempo ser-te-á bastante intuitiva se é que ainda não o é.

Lógica booleana (perguntas “if”)

Já falamos sobre ciclos while (que basicamente é uma pergunta que se repete quantas vezes tu quiseses) mas as vezes só pretendemos fazer a pergunta uma vez e noutras vezes temos que fazer várias vezes mas apenas com o ciclo while não é possível. Assim existem as perguntas “if”. Enquanto o while significa “fazer algo enquanto qualquer coisa for verdadeira”, o if significa “fazer algo se (if) qualquer coisa for verdadeira”. Então posso escrever assim (podes meter no ficheiro para experimentar):

```
a=input('Da-e um numero (1 a 10): ')

if a<5:

    a=5

elif a>5:

    a=4

else:

    a=999
```

```
print 'O que resultou foi '+repr(a)
```

O que resultou deste pequeno programa foi:

Da-e um numero (1 a 10): 3

O que resultou foi 5

Então é assim, primeiro pedi um número de 1 a 10 ao utilizador. Inseri o 3. Depois o programa entre no sistema de perguntas e a primeira é se “a” (que agora é 3) é menor que 5. Como é ele fez a instrução que está lá dentro que é atribuir o número 5 à variável “a”. Como foi verdade na primeira já não precisou de ir para a segunda ou terceira (elif e else). Vou agora experimentar outro caso:

Da-e um numero (1 a 10): 9

O que resultou foi 4

Desta inseri o número 9 e o programa passou ao sistema de perguntas como “a” (igual a 9) não é menor que 5 passou a primeira pergunta à frente e passou para a segunda. Agora como “a” é maior que 5 o programa seguiu a instrução que tinha que foi igualar o “a” a 4 e daí a sua resposta final. Finalmente o último caso:

Da-e um numero (1 a 10): 5

O que resultou foi 999

“a” não é maior nem menor que 5 por isso foi à terceira pergunta (else) e por isso ficou com o número 999 atribuído. Se eu quisesse escrever estas instruções para uma pessoa diria:

(if) Se o “a” for menor que 5 então...

“a” passa a ser igual a 5.

(elif) então e se o “a” for maior que 5 então...

“a” passa a ser igual a 4.

(else) caso contrário, então...

“a” passa a ser igual a 999.

Quando queremos utilizar um sistema de perguntas começamos pelo “if” e todas as outras prerrogativas que inserirmos no mesmo sistema a partir daí passam a ser “elif” (que provem de else if). Se pretendermos meter uma pergunta final que se aplique se todos os casos acima falharem metemos “else”. Vou meter aqui alguns exemplos de sistemas de perguntas para perceberes como funciona isto.

```
a=input('Da-e um numero (1 a 10): ')
```

```
if a==5:
```

```
    a=5*5
```

```
elif a==4:
```

```
    a=4*4
```

```

elif a==6:

    a=6*6

elif a<4:

    a=4

elif a>6:

    a=6

```

```

print 'O numero que resultou e '+repr(a)

```

Repara que aqui já não metemos a instrução “else” porque nenhuma delas é obrigatória excepto o próprio “if” o qual tem que ser usado para iniciar o sistema de perguntas mesmo que seja só uma:

```

a=input('Da-e um numero (1 a 10): ')

if a==5:

    a=5*5

print 'O numero que resultou e '+repr(a)

```

Não sei se já comentei isto antes mas deves ter reparado que meti dois iguais (==) na pergunta “if”. Isto foi deliberado e não uma gafe. Quando inseres um igual (=) está a fazer uma atribuição, quando metes dois (==) está a fazer uma pergunta (a==b significa: a é igual a b?) Dai teres que utilizar os dois iguais dentro dos sistemas “if” ou mesmo dos whiles ou qualquer outro tipo de ciclo. Se quiseses experimentar isto na consola força nisso:

```

>>> 5==5

True

>>> 5==4

False

>>>

```

Como podes ver o Python respondeu True (verdadeiro) porque 5 é, de facto, igual a 5 (eu sei, isto foi inesperado) e False (falso) quando eu perguntei se 5 é igual a 4. Agora as tuas perguntas não precisam de ser apenas como uma instrução, eu poderia perguntar se o número está entre o número 3 e 7 perguntando se é maior que 3 e menor que 6:

```

a=input('Da-e um numero (1 a 10): ')

if a>3 & a<7:

    a=999

```

```
print 'O que resultou foi '+repr(a)
```

O “&” funciona como o acrescento de mais perguntas à mesma. Se “a” for maior que 3 e (&) “a” for menor que 7, então “a” é igual a 999. Agora isto é tudo especialmente útil quando começas a misturar os ciclos while com as perguntas “if”. Um exemplo é:

```
import numpy
```

```
b=numpy.random.rand(5)
```

```
print b
```

```
i=0
```

```
while i<5:
```

```
    if b[i]>0.5:
```

```
        b[i]=2
```

```
    else:
```

```
        b[i]=1
```

```
    i=i+1
```

```
print b
```

Experimentei o resultado disto e repara (a negrito) que meti dois prints para saber o vector inicial e outro para saber o final. Resultou (no meu caso, porque o vector é de números aleatórios) nisto:

```
[ 0.13052822  0.88246145  0.65724695  0.53141686  0.37728761]
```

```
[ 1.  2.  2.  2.  1.]
```

Utilizei o sistema de perguntas dentro de um ciclo while para que as mesmas possam ser feitas a todas os elementos dos vectores. A conjugação destas duas técnicas ou a utilização singular das mesmas é praticamente tudo o que é necessário para fazeres a esmagadora maioria dos programas que envolvam cálculo. Existem outro tipo de sistemas semelhantes mas são tão parecido que penso não valer a pena explicar mostro apenas um exemplo de um ciclo for (semelhante ao while) por ser, também ele, bastante comum em programação.

```
import numpy
```

```
b=numpy.random.rand(5)
```

```
print b
```

```
for i in b:
```

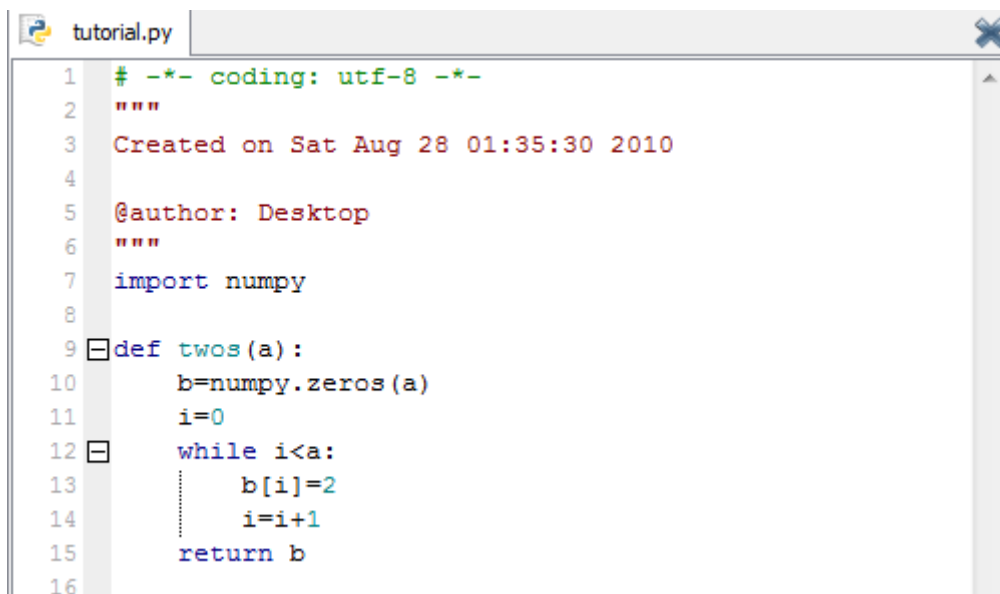
```
    print i
```

Faz os teus próprios módulos

Até ao momento utilizamos uma série de módulos como o numpy ou o matplotlib mas tu também podes fazer os teus próprios módulos. Para o saberes como fazer precisas de saber como fazer funções e também classes. Vamos começar por ver o que é uma função mas antes vou apenas dizer-te que os módulos regra geral, utilizam outros módulos para funcionar, por exemplo o matplotlib utiliza o numpy para algumas tarefas e por isso se não tiveres o numpy instalado o matplotlib poderá não funcionar, ou pelo menos em algumas funções. Outro caso é o scipy, uma biblioteca que ainda não utilizamos que utiliza o numpy e o matplotlib. É bastante normal os módulos estarem interligados uns com os outros porque não faz sentido estarem a repetir código em funções que já estão mais que testadas noutras bibliotecas. Vamos então ver o que é uma função.

Funções

No Spyder abri um ficheiro que salvei como tutorial.py e escrevi o seguinte:



```
1 # -*- coding: utf-8 -*-
2 '''
3 Created on Sat Aug 28 01:35:30 2010
4
5 @author: Desktop
6 '''
7 import numpy
8
9 def twos(a):
10     b=numpy.zeros(a)
11     i=0
12     while i<a:
13         b[i]=2
14         i=i+1
15     return b
16
```

A única coisa que este ficheiro tem é uma função. Se eu tentar meter este ficheiro a funcionar o Python vai lê-lo mas não fazer nada (visível pelo menos), no entanto a função foi criada e tem um nome, “twos”. A maneira de definir uma função é utilizando a nomenclatura “def” e à sua frente tem de estar o nome da função e, dentro dos parêntesis, os argumentos que tem (neste caso apenas um que é o tamanho do vector de dois que pretendemos. Para além disso repara que no fim aparece um “return”, e à frente do “return” metes o resultado da tua função. Com esta função em fiz algo semelhante às funções “numpy.zeros” e “numpy.ones” (semelhante porque está apenas preparada para vectores e não matrizes) mas desta vez a fazer vectores com 2. Repara que todas as instruções dentro da função estão, no mínimo, após o primeiro espaço de indentação. Ao passares a linha no Spyder ele vai fazer a indentação automaticamente por ti por isso deverá ser fácil apanhar-lhe o jeito. Agora parecendo que não

isto que temos aqui é um módulo, só com uma função, é verdade, mas não deixa de ser um módulo tal como o numpy ou matplotlib. Tal como tinha dito eu salvei este ficheiro como tutorial.py numa pasta que é atribuída ao Spyder (no meu caso C:\Users\Desktop\spyder). Agora se isto é um módulo então posso chegar à consola e importa-lo (e tens que importa-lo com o nome do ficheiro).

```
>>> import tutorial
```

```
Traceback (most recent call last):
```

```
ImportError: No module named tutorial
```

```
>>>
```

Deu um erro. O problema não é do módulo, ele existe mesmo, mas é do sítio onde ele está. O Python vai à procura dos seus módulos numa pasta específica que tem para bibliotecas. Eu fui à procura dessa pasta que para fica em C:\Python26\Lib. Copiei para lá o meu ficheiro e tentei novamente.

```
>>> import tutorial
```

```
>>>
```

Agora resultou. O módulo foi importado e posso usa-lo. A minha função é a “twos” por isso vou fazer um vector com ela:

```
>>> a=tutorial.twos(5)
```

```
>>> a
```

```
array([ 2.,  2.,  2.,  2.,  2.])
```

```
>>> b=tutorial.twos(3)
```

```
>>> b
```

```
array([ 2.,  2.,  2.])
```

```
>>>
```

E já está o meu módulo está a funcionar na perfeição e agora tenho acesso a uma função que faz vectores de 2. Agora à medida que vais construir o teu código por vezes vale a pena comentá-lo de maneira a que pessoas que olha para ele o consigam compreender facilmente. Se escreveres o símbolo “#” numa linha o Python não irá ler o que está à frente dele e passará para a linha seguinte. No código a seguir podes ver que comentei a função “twos” para que quem chegar lá possa saber exactamente o que a função faz e como faz.

```
tutorial.py
1  # -*- coding: utf-8 -*-
2  """
3  Created on Sat Aug 28 01:35:30 2010
4
5  @author: Desktop
6  """
7  import numpy
8
9  def twos(a):
10     # Esta função faz vectores de 2.
11     b=numpy.zeros(a)    # Criei um vector de zeros.
12     i=0
13     while i<a:          # E transformo esse vector em 2.
14         b[i]=2
15         i=i+1
16     return b            # No fim utilizo o return para o resultado.
17
```

Agora um módulo com apenas uma função não é algo muito útil por isso vou acrescentar as funções “threes”, “fours” e “fives”.

```
tutorial.py
1  # -*- coding: utf-8 -*-
2  """
3  Created on Sat Aug 28 01:35:30 2010
4
5  @author: Desktop
6  """
7  import numpy
8
9  def twos(a):
10     # Esta função faz vectores de 2.
11     b=numpy.zeros(a)    # Criei um vector de zeros.
12     i=0
13     while i<a:          # E transformo esse vector em 2.
14         b[i]=2
15         i=i+1
16     return b            # No fim utilizo o return para o resultado.
17
18  def threes(a):
19     # Esta função faz vectores de 3.
20     b=numpy.zeros(a)    # Criei um vector de zeros.
21     i=0
22     while i<a:          # E transformo esse vector em 3.
23         b[i]=3
24         i=i+1
25     return b            # No fim utilizo o return para o resultado.
26
27  def fours(a):
28     # Esta função faz vectores de 4.
29     b=numpy.zeros(a)    # Criei um vector de zeros.
30     i=0
31     while i<a:          # E transformo esse vector em 4.
32         b[i]=4
33         i=i+1
34     return b            # No fim utilizo o return para o resultado.
35
36  def fives(a):
37     # Esta função faz vectores de 5.
38     b=numpy.zeros(a)    # Criei um vector de zeros.
39     i=0
40     while i<a:          # E transformo esse vector em 5.
41         b[i]=5
42         i=i+1
43     return b            # No fim utilizo o return para o resultado.
44
```

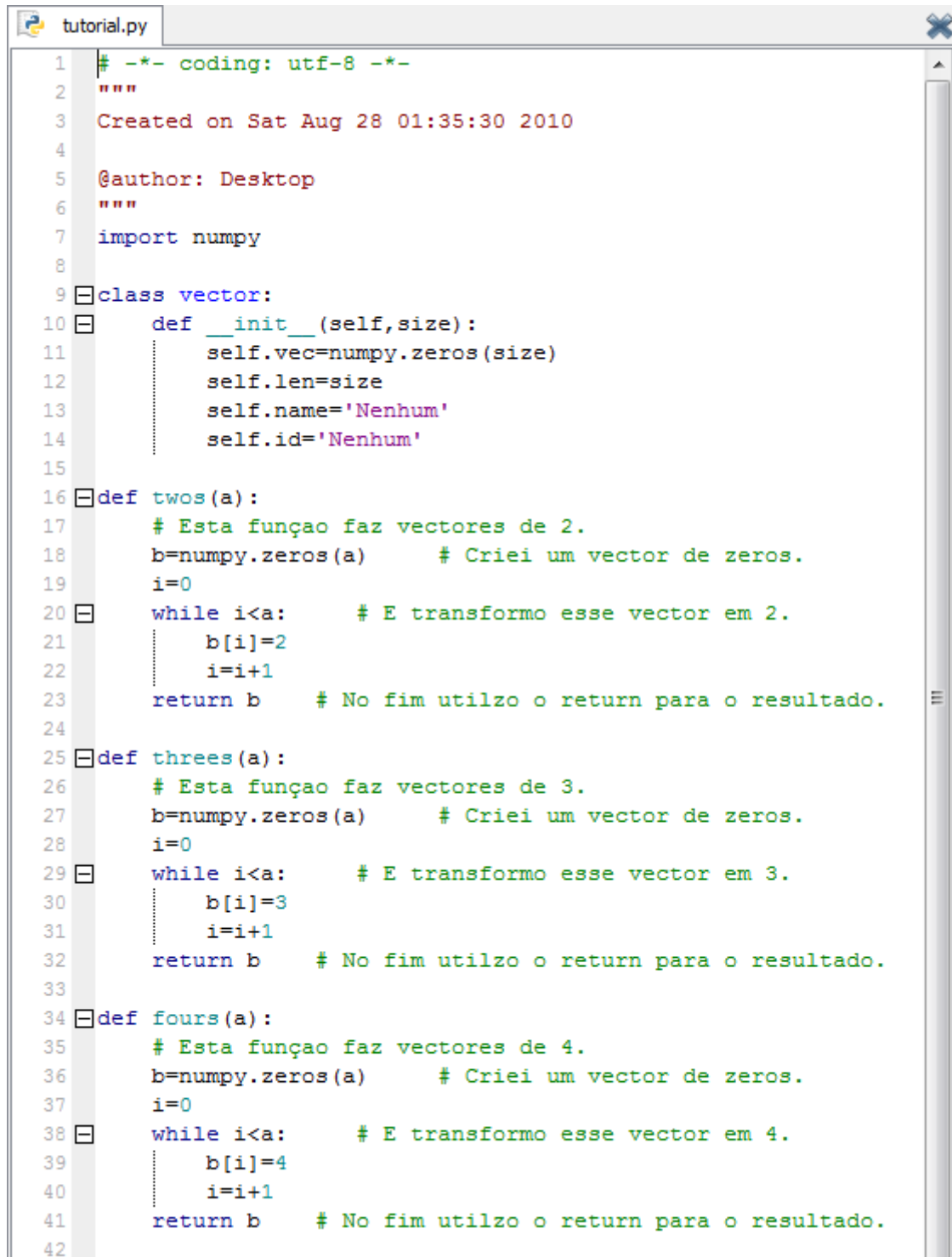

Agora o módulo tutorial mudou por isso é necessário importá-lo novamente numa nova sessão da consola caso contrário é possível que não consiga ler as novas funções. A maneira que eu uso de reiniciar a consola é desligando o Spyder e voltando-o a ligar (mas é mais uma questão de preguiça do que outra coisa porque provavelmente existe algo no Spyder que permite reiniciar a consola sem ter desligar o programa inteiro). De qualquer das maneira após reiniciar a consola e estando seguro que gravei o meu módulo (a versão mais recente) na pasta das bibliotecas do Python posso importá-lo e usá-lo.

```
>>> import tutorial
>>> tutorial.twos(3)
array([ 2.,  2.,  2.])
>>> tutorial.threes(3)
array([ 3.,  3.,  3.])
>>> tutorial.fours(3)
array([ 4.,  4.,  4.])
>>> tutorial.fives(3)
array([ 5.,  5.,  5.])
>>>
```

Catita. Já temos várias funções dentro do meu módulo e agora sempre que quiser posso utiliza-lo seja na consola, seja nos ficheiros. Existem muitos programadores mundo fora, muitos nas mesmas circunstâncias que tu, apenas a aprender que acabam por fazer bibliotecas especificas para uma dada área e esse é um dos motivos por o Python ser uma linguagem tão completa. Havendo gente suficiente a trabalhar com ele também há gente bibliotecas suficiente para fazer quase tudo. Como fazer uma biblioteca é algo tão simples como acabaste de ver aqui a velocidade a que todo este processo se propaga é verdadeiramente vertiginosa. Antes de acabar de explicar como fazeres os teu próprios módulos quero também que percebas o que é uma classe.

Classes

Classes são objectos tais como os módulos, só que mais pequeninos. Lembras-te dos vectores próprios do numpy, são semelhantes aos vectores normais do Python mas são um tipo de objecto um pouco diferente. Imagino que isto seja assim porque são também uma classe. Então para criarmos uma classe fazemos o seguinte.



```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Sat Aug 28 01:35:30 2010
4
5  @author: Desktop
6  """
7  import numpy
8
9  class vector:
10     def __init__(self, size):
11         self.vec=numpy.zeros(size)
12         self.len=size
13         self.name='Nenhum'
14         self.id='Nenhum'
15
16     def twos(a):
17         # Esta função faz vectores de 2.
18         b=numpy.zeros(a)      # Criei um vector de zeros.
19         i=0
20         while i<a:           # E transformo esse vector em 2.
21             b[i]=2
22             i=i+1
23         return b             # No fim utilizo o return para o resultado.
24
25     def threes(a):
26         # Esta função faz vectores de 3.
27         b=numpy.zeros(a)      # Criei um vector de zeros.
28         i=0
29         while i<a:           # E transformo esse vector em 3.
30             b[i]=3
31             i=i+1
32         return b             # No fim utilizo o return para o resultado.
33
34     def fours(a):
35         # Esta função faz vectores de 4.
36         b=numpy.zeros(a)      # Criei um vector de zeros.
37         i=0
38         while i<a:           # E transformo esse vector em 4.
39             b[i]=4
40             i=i+1
41         return b             # No fim utilizo o return para o resultado.
42
```

Criamos uma classe com a nomenclatura “class” com o nome respectivo a seguir. Como queria que esta classe tivesse argumentos tive que criar uma função lá dentro cujo o nome é `__init__` (eu não te sei explicar exactamente porquê porque é uma coisa que eu faço porque já vi outros antes fazerem mas penso que tem a ver com o ser a primeira função a funcionar para o caso de haver mais ou então força-la decididamente a funcionar sem ter que a chamar). Dentro da função `__init__` há dois argumentos, o “self” e o “size”. O “size” é um número que vai dizer qual o tamanho do vector que a classe vector tem lá dentro e que tem de ser introduzido pelo utilizador. O “self” é um argumento que vais ver muitas vezes mais para a frente que na maioria dos casos significa a variável a quem estás a atribuir a tua classe. Por exemplo assumindo que o “`numpy.array`” é uma classe neste código:

```
>>> a=numpy.array([1,2,3])
>>>
```

O “a” é o “self”. Então se chamares a forma do “a” ele vai-te responder:

```
>>> a=numpy.array([1,2,3])
>>> a.shape
(3,)
```

Isto porque na sua classe existia um argumento que dizia “`self.shape`” como nós no código que escrevi temos alguns como “`self.name`” e “`self.id`”. Salvei o meu ficheiro na pasta respectiva das bibliotecas do Python e reiniciei o Spyder e comencei a experimentar com a classe que tinha no meu módulo:

```
>>> import tutorial
>>> a=tutorial.vector(3)
>>> a.vec
array([ 0.,  0.,  0.])
>>> a.len
3
>>> a.name
'Nenhum'
>>> a.name='Algum'
>>> a.name
'Algum'
>>> a.id='01'
>>> a.id
'01'
```

```
>>> a.vec=tutorial.fives(3)

>>> a.vec

array([ 5.,  5.,  5.])

>>> a

<tutorial.vector instance at 0x07509990>

>>>
```

Repara que comecei por chamar a minha classe como se fosse uma função e só meti um argumento porque a classe precisa dele (existem classes sem argumentos). A partir daí cada vez que chamava um dos objectos que existiam dentro da classe o Python respondia e podia até fazer novas atribuições a esses objectos. As classes são uma invenção espectacular pois permitem guardar muita informação e ter um funcionamento especial lá dentro. Imagina que quando fazes um vector ou uma matriz a classe que utilizas para o fazer guarda imediatamente o número de linhas e número de colunas, máximo, mínimo, média, variância, etc., e apenas precisas de chamar esses objectos para teres acesso a eles sem teres que o calcular por fora. O código final do módulo tutorial ficou como:

```
import numpy

class vector:

    def __init__(self,size):

        self.vec=numpy.zeros(size)

        self.len=size

        self.name='Nenhum'

        self.id='Nenhum'

def twos(a):

    # Esta função faz vectores de 2.

    b=numpy.zeros(a)      # Criei um vector de zeros.

    i=0

    while i<a:            # E transformo esse vector em 2.

        b[i]=2

        i=i+1

    return b              # No fim utilizo o return para o resultado.

def threes(a):
```

```

# Esta função faz vectores de 3.

b=numpy.zeros(a)    # Criei um vector de zeros.

i=0

while i<a:    # E transformo esse vector em 3.

    b[i]=3

    i=i+1

return b    # No fim utilizo o return para o resultado.


def fours(a):

    # Esta função faz vectores de 4.

    b=numpy.zeros(a)    # Criei um vector de zeros.

    i=0

    while i<a:    # E transformo esse vector em 4.

        b[i]=4

        i=i+1

    return b    # No fim utilizo o return para o resultado.


def fives(a):

    # Esta função faz vectores de 5.

    b=numpy.zeros(a)    # Criei um vector de zeros.

    i=0

    while i<a:    # E transformo esse vector em 5.

        b[i]=5

        i=i+1

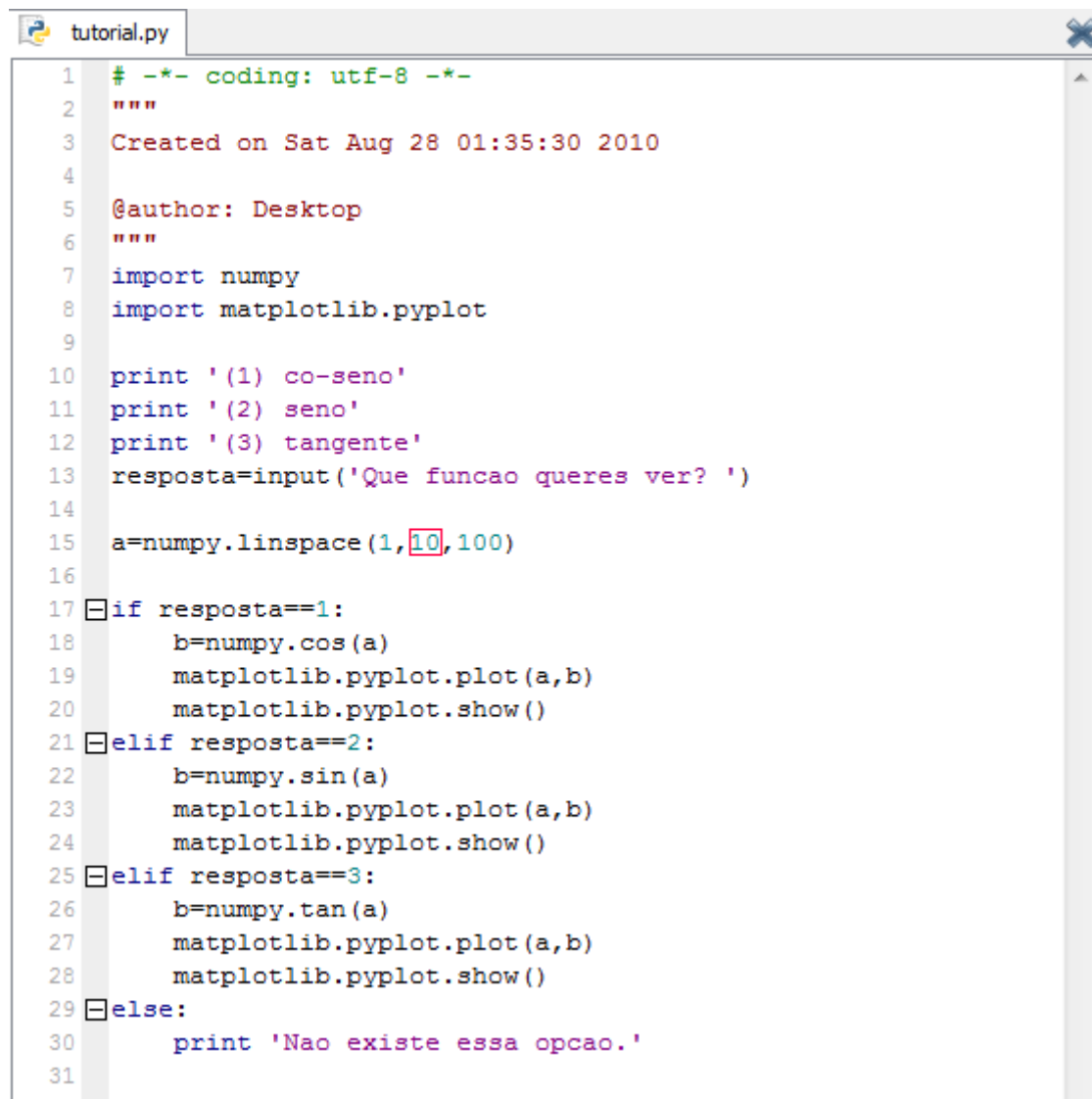
    return b    # No fim utilizo o return para o resultado.

```

Analisa, experimenta e garante que percebeste tudo o que é necessário para fazeres e usares um módulo. É isto que te vai permitir construir programas verdadeiramente úteis para o utilizador mais comum além de te poder vir a poupar muito trabalho com reutilização de código.

Faz o teu próprio software

Agora já sabes o que fazer para criar um módulo inclusive as funções e classes que poderão existir dentro do mesmo. Está preparado para fazer o teu próprio software. Vou fazer um ou outro exemplo de programas com interface de consola para que tenhas algo com que comparar nas tuas experiências mas em breve iremos começar a criar interfaces de janelas. Por agora repara no seguinte código.



```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Sat Aug 28 01:35:30 2010
4
5  @author: Desktop
6  """
7  import numpy
8  import matplotlib.pyplot
9
10 print '(1) co-seno'
11 print '(2) seno'
12 print '(3) tangente'
13 resposta=input('Que funcao queres ver? ')
14
15 a=numpy.linspace(1,10,100)
16
17 if resposta==1:
18     b=numpy.cos(a)
19     matplotlib.pyplot.plot(a,b)
20     matplotlib.pyplot.show()
21 elif resposta==2:
22     b=numpy.sin(a)
23     matplotlib.pyplot.plot(a,b)
24     matplotlib.pyplot.show()
25 elif resposta==3:
26     b=numpy.tan(a)
27     matplotlib.pyplot.plot(a,b)
28     matplotlib.pyplot.show()
29 else:
30     print 'Nao existe essa opcao.'
31
```

A ideia é criar um pequeno programa que dê a oportunidade de o utilizador visualizar as funções co-seno, seno, e tangente (opções 1, 2 e 3). Existe um vector criado com o “numpy.linspace” e consoante a resposta do utilizador vai ser calculado outro a partir desse com as funções “numpy.cos”, “numpy.sin” e “numpy.tan”. Utilizei o matplotlib para fazer a visualização dos gráficos e inseri um comentário, “Não existe essa opção”, para o caso do utilizador escolher algum número que não seja o 1, 2 ou 3. Ao correr o programa isto acontece:

(1) co-seno

(2) seno

(3) tangente

Que funcao queres ver? 4

Nao existe essa opcao.

Neste caso escolhi a opção 4 que não existe e por isso ele deu-me o comentário que eu programei para ele fazer nestes casos. Vou agora meter um funcional.

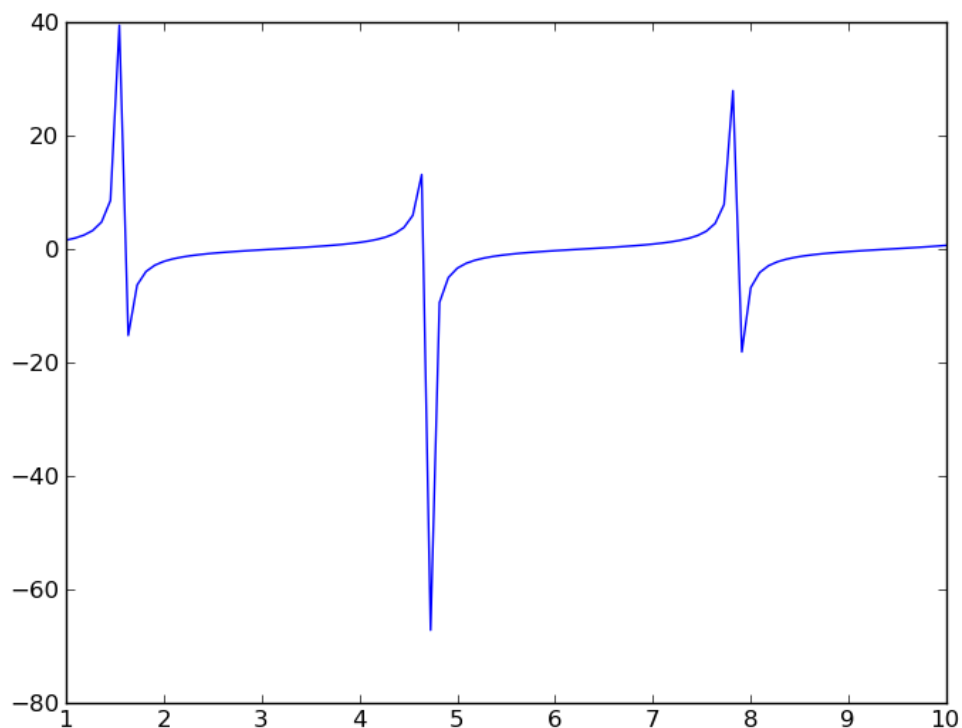
(1) co-seno

(2) seno

(3) tangente

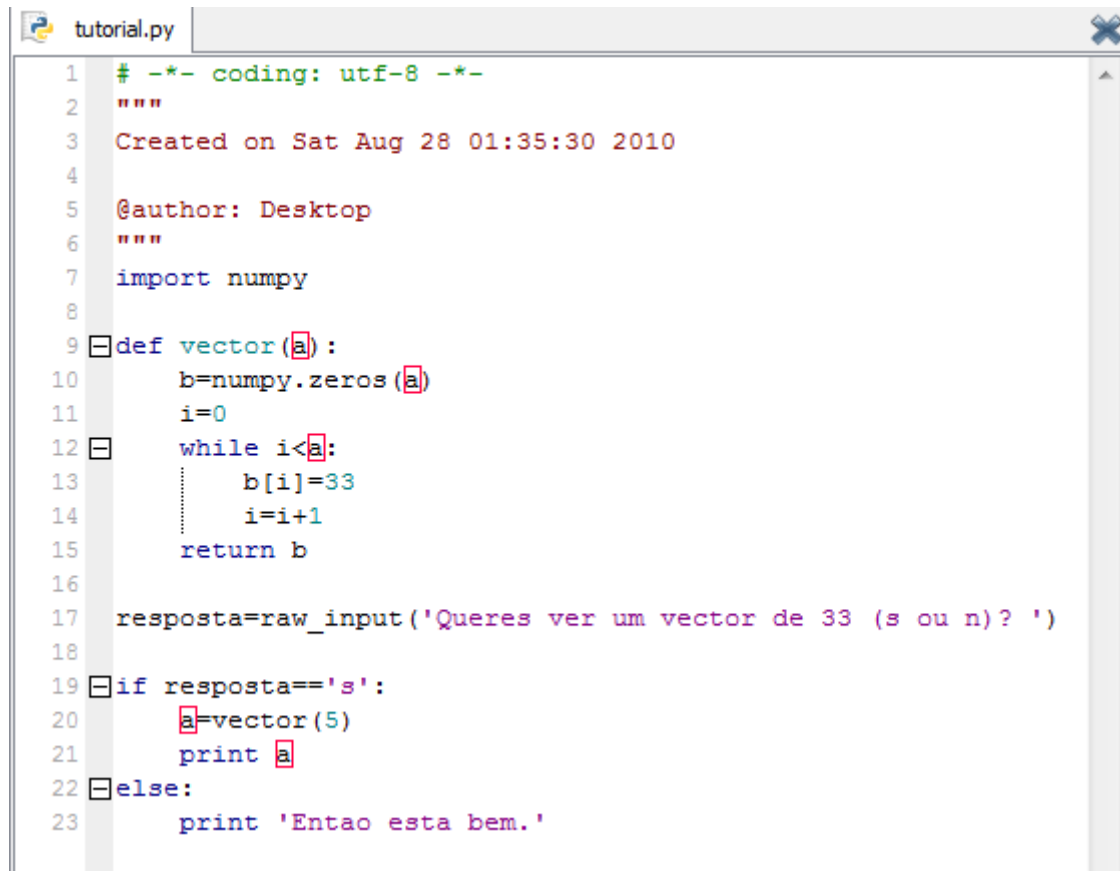
Que funcao queres ver? 3

Escolhi ver a tangente e o gráfico apareceu (é possível que tenhas de minimizar o Spyder para veres a janela do matplotlib, mas está lá).



E assim um programa interactivo com recurso aos módulos numpy e matplotlib. Não é nada de especial, muito pelo contrário, mas apresenta bem o que poderia ser um software feito com alguma paciência. Da mesma maneira que importamos os módulos numpy e matplotlib também podíamos ter importado o que já fizemos nesta publicação. Só não o fiz porque não tinha nenhuma utilidade para ele.

Agora o teu programa não precisa de ser uma série de instruções seguidas, pode ter funções e outras coisas lá para o meio.



```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Sat Aug 28 01:35:30 2010
4
5  @author: Desktop
6  """
7  import numpy
8
9  def vector(a):
10     b=numpy.zeros(a)
11     i=0
12     while i<a:
13         b[i]=33
14         i=i+1
15     return b
16
17  resposta=raw_input('Queres ver um vector de 33 (s ou n)? ')
18
19  if resposta=='s':
20     a=vector(5)
21     print a
22  else:
23     print 'Entao esta bem.'
```

Neste programa comecei por criar uma função chamada vector que ao correres o ficheiro o Python não vai ler. Vai ler sim as instruções que estão a seguir na qual a primeira faz-te uma pergunta de sim ou não (s ou n). Se a resposta for não então:

Queres ver um vector de 33 (s ou n)? n

Entao esta bem.

Se a resposta for sim então a instrução que está na linha 20 (olha para a figura) vai buscar a função vector e o resultado é este:

Queres ver um vector de 33 (s ou n)? s

[33. 33. 33. 33. 33.]

E para fazeres programas simples as regras são estas. A prática pode levar-te a fazer coisas cada vez mais complexas e difíceis de gerir e se estiveres mesmo virado para a computação científica então o mais provável é que venhas a despende muito tempo a criar algoritmos com whiles e if complexos de se perceber e no fim criar um pequeno interface por cima para que esteja minimamente disponível para o mais casual dos utilizadores. Ainda assim o que vamos aprender a seguir é, a meu ver, muito importante. Vamos começar a produzir programas com interface de janelas que é até ao momento a maneira (das comuns pelo menos) mais fácil de se mexer num computador.

wxPython (interfaces gráficos)

Existe uma sigla muito conhecida no mundo dos computadores que é GUI (graphical user interface). Um GUI é o que permite ao utilizador interagir com o computador. Por exemplo o Windows ou o Ubuntu têm um GUI por onde podes escolher as tuas pastas, criar ficheiros, correr programas, etc. Outros programas como o teu browser de internet, o teu Messenger, o que quer que esteja no teu telemóvel são interfaces para evitar que tu tenhas de mexer no código directamente. Se não houvesse interfaces toda a gente tinha que ser uma especialista em programação para fazer seja o que for. Assim fazemos a vida mais fácil a muita gente se adornarmos os nossos programas com o interface mais simpático que conseguirmos. O wxPython é um biblioteca tal como nós já vimos o numpy (para tratamento numérico) e o matplotlib (gráficos), mas esta serve para interfaces gráficos. Ao contrário do numpy e matplotlib que mexemos na consola, com o wxPython vamos fazer tudo no ficheiro. Num ficheiro escrevi isto:



```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Sat Aug 28 01:35:30 2010
4
5  @author: Desktop
6  """
7  import wx
8
9  class MinhaFrame(wx.Frame):
10     def __init__(self, parent, id):
11         wx.Frame.__init__(self, parent, -1, 'Ola mundo', size = (200, 100))
12
13 if __name__ == '__main__':
14     app=wx.App()
15     frame=MinhaFrame(parent=None,id=999)
16     frame.Centre()
17     frame.Show()
18     app.MainLoop()
```

Não consegues ver o código todo porque a linha 11 é muito longa mas para o que tenho que explicar agora serve. Lembra-te sempre que as maiúsculas e minúsculas são importantes e por isso tens de meter os comandos tal e qual como eles aparecem. Existem aqui duas coisas que têm (ou convêm) estar sempre nos teus programas com interface gráfico do wxPython. O primeiro é a importação da biblioteca do wxPython (import wx), a segunda é o que acontece numa pergunta “if” no fim e que agora vou passar a explicar (o melhor que conseguir pelo menos):

```
if __name__ == '__main__':
    app=wx.App()

    frame=MinhaFrame(parent=None,id=999)

    frame.Centre()
```

```
frame.Show()

app.MainLoop()
```

Não sei porque razão temos (ou devemos) que fazer a instrução “if __name__=='__main__':” mas julgo ser para que este ficheiro saiba responder no caso de ser chamado como modulo. É que como deves calcular o Python não tem maneira de saber se um ficheiro é um módulo ou não, simplesmente responda às instruções que estão lá dentro. Assim julgo que esta linha de código faz essa pergunta “este ficheiro está a ser corrido ou está a ser importado”. De qualquer das maneiras mal não faz. A segunda linha de código neste conjunto:

```
app=wx.App()
```

Criar algo fundamental para o wxPython que é uma aplicação. Tu não vez essa aplicação, mas ela está criada, não tem janelas, não tem nada mas está lá. A seguir crio a minha primeira janela (que depois a tenho de definir como fiz mais acima). As últimas instruções são:

```
frame.Centre()
```

Para centrar a janela que criei ao ecrã se não aparecia-me numa ponta qualquer. E:

```
frame.Show()
```

Para mostrar a dita janela visto que embora tu a tenhas criado ela ainda não está visível. Agora analisando melhor a instrução:

```
frame=MinhaFrame(parent=None,id=999)
```

Notas que tem dois argumentos. Aqui ainda não chamamos uma janela do wxPython mas sim uma classe que eu criei que tem uma janela do wxPython lá dentro. Essa janela do wxPython vai precisar de dois argumentos e por isso a minha classe (chamada MinhaFrame) também tem de ter dois argumentos. O primeiro argumento é o parentesco da janela que eu respondo como não tendo (None, sem pelicas). O segundo argumento é id da janela. O id é tipo o bilhete de identidade da janela, como podem existir muitas janelas no teu programa convém que não as confundas e assim podes atribui um id diferente. O parentesco da janela por outro lado diz-te se a tua janela depende de outra ou não. Neste caso esta é a minha primeira e única janela do programa por isso não tem parentesco nenhum. Vamos agora analisar o código em que eu defino a classe.

```
class MinhaFrame(wx.Frame):
```

```
    def __init__(self, parent, id):

        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (300, 200))
```

Esta da classe segue um mesmo estilo que eu já tinha ensinado mais para trás quando ensinei sobre classes. Começa com uma função __init__ e nessa função tem os dois argumentos que eu inseri quando chamei a classe. Agora dentro da função está esta instrução:

```
        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (300, 200))
```

Esta instrução é que é a janela do wxPython e então chamamos essa janela com “wx.Frame.__init__” e inserimos os argumentos lá dentro. Juntamente com o “parent” e

“id”temos uma string e um lista de dois elementos para o size (tamanho). Essa string é o que vai dar o título que aparece na barra superior da janela. O “size” é o tamanho da janela e eu disse que tinha 300 pontos na horizontal e 200 na vertical. Corri o meu programa e isto aconteceu (Figura 17).

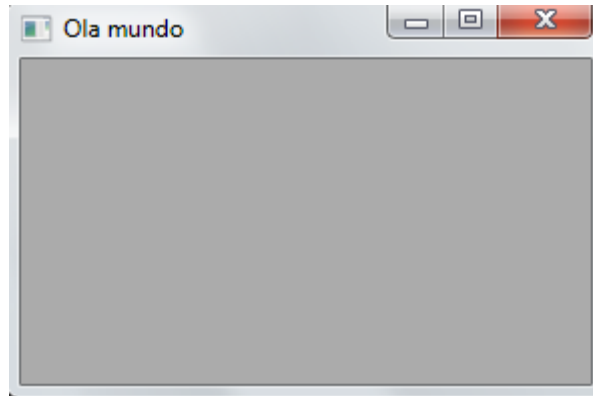


Figura 17 - Janela do wxPython.

O estilo da janela vai depender o sistema operativo. Por exemplo penso que esta que está na figura é do Windows 7, no XP, Ubuntu ou Mac OSX já vai parecer diferente. De qualquer das maneiras repara que a string que meti no código como argumento aparece lá na barra do topo e o tamanho da janela, se experimentaste, reproduz algo que bem pode ser 300 na horizontal e 200 na vertical. Vou agora meter aqui o código todo para que possa analisar com mais calma:

```
import wx

class MinhaFrame(wx.Frame):

    def __init__(self, parent, id):

        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (300, 200))

if __name__=='__main__':

    app=wx.App()

    frame=MinhaFrame(parent=None,id=999)

    frame.Centre()

    frame.Show()

    app.MainLoop()
```

Comecei por importar o wxPython com “import wx” depois criei a classe da minha janela e finalmente meti as tais instruções para inicializar o programa. Nem todos os objectos precisam de classes para se fazerem comecei por ensina-lo porque é o que vamos ver a seguir.

Janela principal

O que eu estou aqui a chamar de janela principal é o tipo de janela que já fizemos no nosso primeiro programa em wxPython mas agora vamo ver tudo o que se pode fazer com ela, primeiro com os argumentos que ela pode ter e a seguir com os objectos que podemos inserir nela. Então começemos por ver o programa que tínhamos ainda agora mas vamos acrescentar um argumento que não tinha usado que é o “style” (estilo). Podemos inserir mais de um estilo ao mesmo tempo mas primeiro precisamos de saber quais existem. Os estilos dizem-te o que a tua janela tem nomeadamente na barra de topo. Por defeito a janela vem com uma série de particularidades (Figura 18).

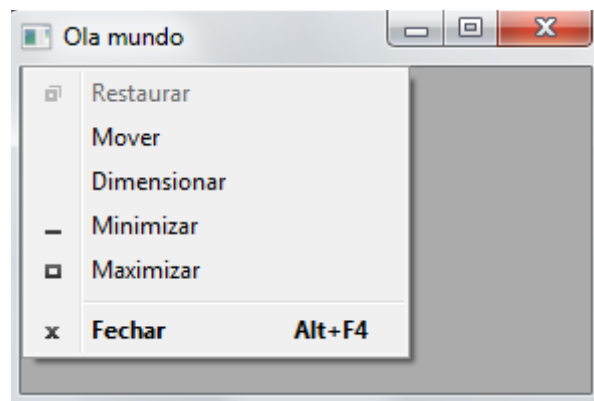


Figura 18 – Objectos na janela do wxPython.

Repara que no programa que fizemos a nossa janela já trazia um botão no topo esquerdo onde podemos fazer as opções que aparecem na figura e do lado direito os botões para minimizar, maximizar e fechar. Agora por vezes apenas queremos alguns destes objectos na nossa janela por isso temos que inserir o que queremos no argumento “style”. Existem dois estilos que aconselho vivamente a utilizar sempre, o wx.CAPTION e o wx.SYSTEM_MENU, isto porque são os estilos que te dão a caixa que está à volta da tua janela e que contém os botões. No meu exemplo a seguir vou fazer um caixa que em vez de ter tudo , tem apenas um botão para fechar:

```
import wx

class MinhaFrame(wx.Frame):

    def __init__(self, parent, id):

        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (300, 200), style=wx.CAPTION|wx.CLOSE_BOX|wx.SYSTEM_MENU)

if __name__=='__main__':

    app=wx.App()

    frame=MinhaFrame(parent=None,id=999)
```

```

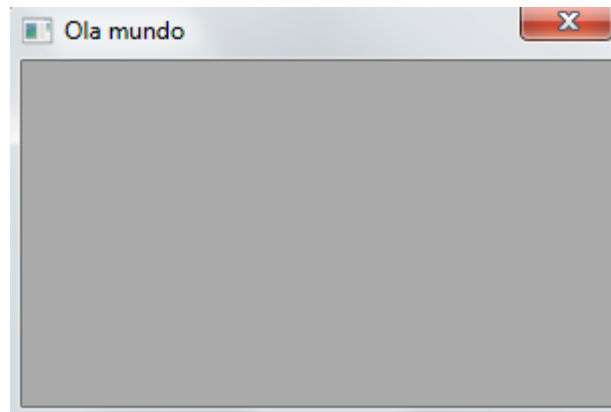
frame.Centre()

frame.Show()

app.MainLoop()

```

Reparas-te como meti os estilos (meti 3). Fiz “style=estilo1|estilo2|estilo3”, isto porque o “|” é o símbolo que separa os estilos se quiseses meter mais de um. Se experimentares o código acima deverá aparecer-te isto:



Repara que já não tem os botões todos mas sim apenas o que permite fechar a caixa. A lista de estilos é esta (não está completa mas serve por agora):

Estilo	Descrição
wx.DEFAULT_FRAME_STYLE	Vem por defeito e é equivalente a fazer: style= wx.MINIMIZE_BOX wx.MAXIMIZE_BOX wx.RESIZE_BORDER wx.SYSTEM_MENU wx.CAPTION wx.CLOSE_BOX wx.CLIP_CHILDREN
wx.CAPTION	Cria uma captação na janela.
wx.SYSTEM_MENU	Cria um sistema para os botões na janela (mas não os activa)
wx.MINIMIZE	Abre a janela minimizada
wx.MINIMIZE_BOX	Mete o botão para minimizar
wx.MAXIMIZE	Abre a janela maximizada
wx.MAXIMIZE_BOX	Mete o botão para maximizar
wx.CLOSE_BOX	Mete o botão para fechar a janela
wx.RESIZE_BORDER	Mete uma borda que pode ser utilizada para redimensionar a janela com o rato
wx.STAY_ON_TOP	Força a janela a ficar à frente das outras janelas

Tabela 4 - Tabela de estilos para a janela do wxPython.

Com estes dados já consegues editar a tua janela consoante a finalidade a que se destina. Vamos agora ver que tipo de objectos podemos meter numa janela que no nosso exemplo seria a janela principal do programa.

Inserir Painel

As janelas da maneira como vêm são desprovidas de conteúdo. Ao acrescentares um painel está a acrescentar um objecto que pode conter outros objectos. Acrescentei a seguinte linha de código ao nosso programa (ah, e retirei os estilos que tínhamos no último exemplo para ficar mais legível).

```
import wx

class MinhaFrame(wx.Frame):

    def __init__(self, parent, id):

        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (300, 200))

        self.panel=wx.Panel(self)

if __name__=='__main__':

    app=wx.App()

    frame=MinhaFrame(parent=None,id=999)

    frame.Centre()

    frame.Show()

    app.MainLoop()
```

Lembra-te que a “MinhaFrame” é um classe e por isso podes inserir objectos lá dentro de maneira a que possam ser acedidos fora da classe. Por isso atribui um painel ao “self” (portanto a própria classe) com o comando “wx.Panel(self)” à variável “self.panel”. O resultado disto é:

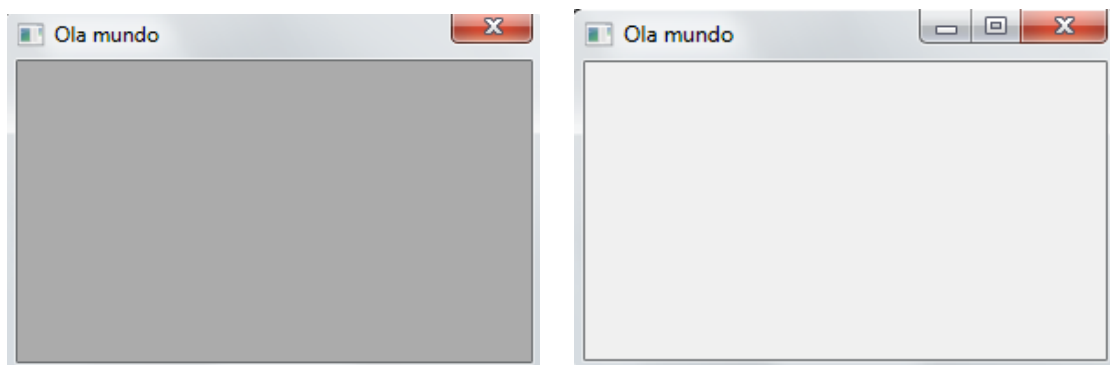


Figura 19 – À esquerda sem painel, à direita com painel.

Parece que apenas mudou a cor mas na verdade aconteceu mais do que isso. A partir de agora podemos dar instruções dentro da classe para inserir objectos no painel, mas primeiro vamos ver como editar o nosso painel, nomeadamente a cor.

```
import wx

class MinhaFrame(wx.Frame):

    def __init__(self, parent, id):

        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (300, 200))

        self.panel=wx.Panel(self)

        self.panel.SetBackgroundColour('green')

if __name__=='__main__':

    app=wx.App()

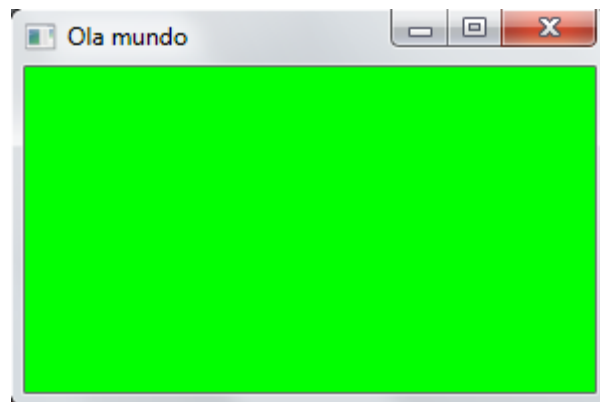
    frame=MinhaFrame(parent=None,id=999)

    frame.Centre()

    frame.Show()

    app.MainLoop()
```

Com o comando “SetBackgroundColour” (e mais uma vez alerta que as maiúsculas e minúsculas são importantes) dei uma cor nova ao painel.



E não está limitado às cores simples, podia muito bem ser com código hexadecimal como já aprendemos para os gráficos:

```
self.panel.SetBackgroundColour('#FF00CC')
```

Ao meter a cor como uma string ‘#FF00CC’ mudei a cor do painel para um violeta. Experimenta. Agora por defeito a cor do painel é ‘Ligth_Grey’ e é essa que vou usar doravante. Vamos agora ver como adicionar objectos ao painel e que tipos de objectos existem.

Texto estático

O texto estático é exactamente o que o nome diz. Texto que aparece nas nossas janelas e com o qual não se pode interagir. Se quiseres mesmo que ele fique imutável ao longo do programa então nem precisas de atribuí-lo a uma variável.

```
import wx

class MinhaFrame(wx.Frame):

    def __init__(self, parent, id):

        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (300, 200))

        self.panel=wx.Panel(self)

        self.panel.SetBackgroundColour('Ligth_Grey')

        wx.StaticText(self.panel,-1,"Texto estatico:",(10,10))

        wx.StaticText(self.panel,-1,"Texto estatico dois:",(100,100))

if __name__=='__main__':

    app=wx.App()

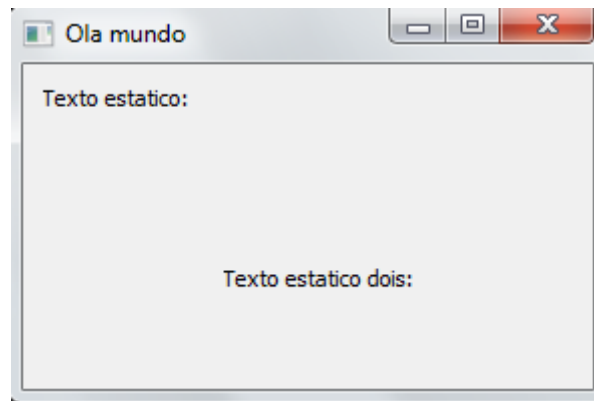
    frame=MinhaFrame(parent=None,id=999)

    frame.Centre()

    frame.Show()

    app.MainLoop()
```

Para fazeres um texto estático tens de usar o comando “wx.StaticText”. O primeiro argumento é o sítio (objecto) onde vai estar, o segundo argumento é o id (identificador que já tínhamos visto quando chamamos a “MinhaFrame” e lhe demos id=999, quando metes id=-1 o Python atribui um qualquer identificador automaticamente), o terceiro argumento é o texto em si, o quarto argumento é a localização exacta do texto no objecto onde ele está. O primeiro texto que inserir está a 10 pontos na horizontal e 10 na vertical, o segundo texto está a 100 pontos na horizontal e 100 na vertical. O resultado é este:



Não estava propriamente a pensar numa organização quando inseri os dois objectos de texto no painel mas mexer com a localização do texto é mesmo uma questão de prática pelo que vais ganhando sensibilidade ao mesmo. Além disso podes sempre ir correndo o programa para ver se queres meter o texto mais para a esquerda, mais para a direita, cima, baixo, etc. Vou ajeitar um bocadinho o meu texto no painel e adicionar outros texto mas desta vez vou atribuí-lo a variáveis:

```
import wx

class MinhaFrame(wx.Frame):

    def __init__(self, parent, id):

        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (300, 200))

        self.panel=wx.Panel(self)

        self.panel.SetBackgroundColour('Ligth_Grey')

        wx.StaticText(self.panel,-1,"Nome do aluno:",(10,10))
        wx.StaticText(self.panel,-1,"Curso do aluno:",(10,30))

        self.text1=wx.StaticText(self.panel,-1,"Nenhum",(130,10))
        self.text2=wx.StaticText(self.panel,-1,"Nenhum",(130,30))

if __name__=='__main__':

    app=wx.App()

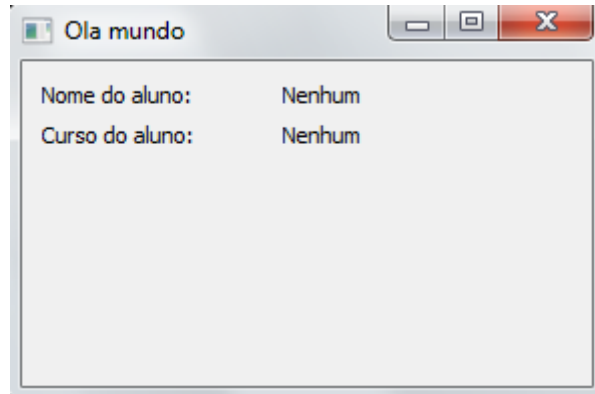
    frame=MinhaFrame(parent=None,id=999)

    frame.Centre()

    frame.Show()

    app.MainLoop()
```

A negrito estão as linhas que acrescentei e onde fiz modificações. Repara que os textos estáticos que criei estão atribuídos às variáveis “self.text1” e “self.text2” porque tenciono mudar o texto que têm (mas só mais para frente) e neste momento quero mudar-lhes a cor para chamar a atenção do utilizador.



Para mudar a cor do texto estático tenho de utilizar o comando “SetForegroundColor” como podes ver no código a seguir (repara no que está a negrito):

```
import wx

class MinhaFrame(wx.Frame):

    def __init__(self, parent, id):

        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (300, 200))

        self.panel=wx.Panel(self)

        self.panel.SetBackgroundColour('Ligth_Grey')

        wx.StaticText(self.panel,-1,"Nome do aluno:",(10,10))

        wx.StaticText(self.panel,-1,"Curso do aluno:",(10,30))

        self.text1=wx.StaticText(self.panel,-1,"Nenhum",(130,10))
        self.text1.SetForegroundColor('red')

        self.text2=wx.StaticText(self.panel,-1,"Nenhum",(130,30))
        self.text2.SetForegroundColor('red')

if __name__=='__main__':

    app=wx.App()

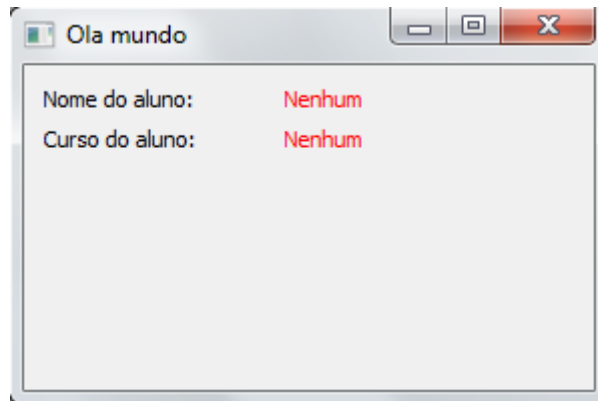
    frame=MinhaFrame(parent=None,id=999)
```

```
frame.Centre()

frame.Show()

app.MainLoop()
```

Dei a cor vermelha aos textos estáticos que têm “Nenhum” para evidenciar ao utilizador que ainda não inseriu informação nenhuma (se bem que ainda não fizemos nada para possibilitar isso mas tudo a seu tempo).



Ainda não vamos utilizar isto (porque por agora não faz sentido) mas se quisesse mudar o que está escrito no texto estático basta utilizar o comando “SetLabel” do qual vou exemplificar no código abaixo:

```
import wx
```

```
class MinhaFrame(wx.Frame):

    def __init__(self, parent, id):

        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (300, 200))

        self.panel=wx.Panel(self)

        self.panel.SetBackgroundColour('Ligth_Grey')


        wx.StaticText(self.panel,-1,"Nome do aluno:",(10,10))
        wx.StaticText(self.panel,-1,"Curso do aluno:",(10,30))


        self.text1=wx.StaticText(self.panel,-1,"Nenhum",(130,10))
        self.text1.SetForegroundColour('red')

        self.text2=wx.StaticText(self.panel,-1,"Nenhum",(130,30))
        self.text2.SetForegroundColour('red')
```

```
self.text1.SetLabel('Algum')

self.text1.SetForegroundColour('green')

if __name__=='__main__':

    app=wx.App()

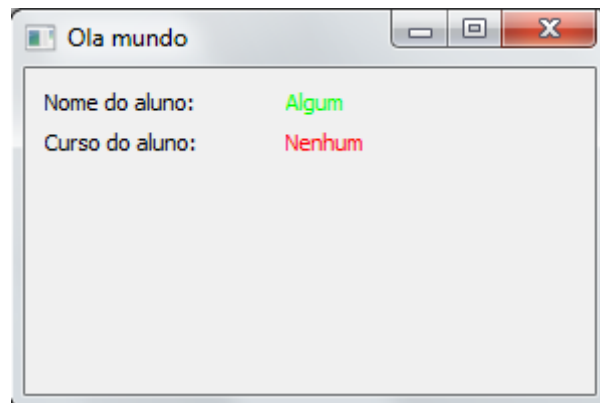
    frame=MinhaFrame(parent=None,id=999)

    frame.Centre()

    frame.Show()

    app.MainLoop()
```

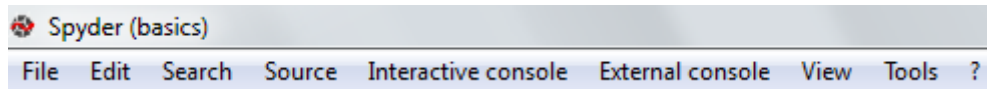
O resultado disto é:



Mas vamos utilizar o código anterior para continuar o nosso tutorial visto que não faz grande sentido estarmos a fazer um texto estático e logo a seguir mudar o que está lá escrito, não é? Vamos ver como criar menus dentro de uma janela.

Barra de menu

A barra de menu é aquelas divisórias que geralmente aparecem nos programas e que trazem uma série de acções que o utilizador pode escolher. A do Spyder é esta:



Agora precisas de ter noção de uma coisa. Uma coisa é a barra de menu, outra coisa são os menus em si. Por exemplo, o Spyder tem uma barra de menu com 9 menus lá dentro (File, Edit, Search, etc...). Vou começar por adicionar então a barra de menu:

```
import wx
```

```
class MinhaFrame(wx.Frame):
```

```
    def __init__(self, parent, id):
```

```
        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (300, 200))
```

```
        self.panel=wx.Panel(self)
```

```
        self.panel.SetBackgroundColour('Ligth_Grey')
```

```
        wx.StaticText(self.panel,-1,"Nome do aluno:",(10,10))
```

```
        wx.StaticText(self.panel,-1,"Curso do aluno:",(10,30))
```

```
        self.text1=wx.StaticText(self.panel,-1,"Nenhum",(130,10))
```

```
        self.text1.SetForegroundColour('red')
```

```
        self.text2=wx.StaticText(self.panel,-1,"Nenhum",(130,30))
```

```
        self.text2.SetForegroundColour('red')
```

```
        menu=wx.MenuBar()
```

```
        self.SetMenuBar(menu)
```

```
if __name__=='__main__':
```

```
    app=wx.App()
```

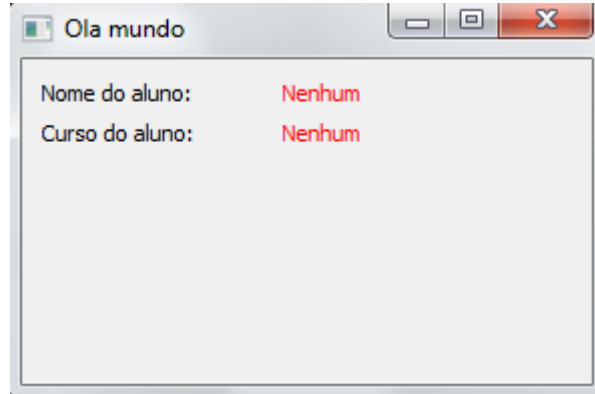
```
    frame=MinhaFrame(parent=None,id=999)
```

```
    frame.Centre()
```

```
frame.Show()

app.MainLoop()
```

Se fores ver o resultado, vais notar que não houve diferença nenhuma em relação ao que tinhas antes.



Na realidade existe uma diferença pois a barra de menu existe e está lá simplesmente como não tem nada lá dentro não a consegues ver. Em relação ao código que fizemos:

```
menu=wx.MenuBar()
```

Com isto estou a atribuir à variável “menu” um objecto barra de menu.

```
self.SetMenuBar(menu)
```

E com isto estou a dizer que ela fique activa na minha janela (na classe que eu estou a criar, daí o “self”). Vou agora meter lá um menu “File” para que possas visualizar a barra.

```
import wx
```

```
class MinhaFrame(wx.Frame):
```

```
    def __init__(self, parent, id):
```

```
        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (300, 200))
```

```
        self.panel=wx.Panel(self)
```

```
        self.panel.SetBackgroundColour('Ligth_Grey')
```

```
        wx.StaticText(self.panel,-1,"Nome do aluno:",(10,10))
```

```
        wx.StaticText(self.panel,-1,"Curso do aluno:",(10,30))
```

```
        self.text1=wx.StaticText(self.panel,-1,"Nenhum",(130,10))
```

```
        self.text1.SetForegroundColour('red')
```

```
        self.text2=wx.StaticText(self.panel,-1,"Nenhum",(130,30))
```

```

self.text2.SetForegroundColour('red')

menu=wx.MenuBar()

self.SetMenuBar(menu)

menu_file=wx.Menu()

menu.Append(menu_file,"File")

if __name__=='__main__':
    app=wx.App()

    frame=MinhaFrame(parent=None,id=999)

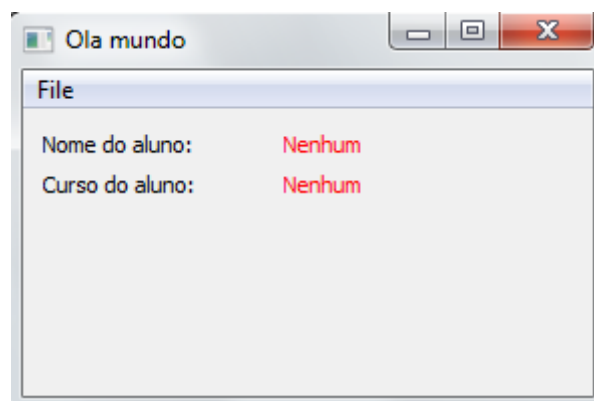
    frame.Centre()

    frame.Show()

    app.MainLoop()

```

Agora se experimentares o código já vais ver qualquer coisa.



Ainda não dá para fazer nada mas já sabemos como inserir menus. A primeira linha de código que acrescentei foi para atribuir a uma variável “menu_file” um objecto do estilo “wx.Menu” (que mais uma vez alerto para o facto de ser diferente do wx.MenuBar).

```
menu_file=wx.Menu()
```

Na outra linha a seguir adicionei este objecto de menu à barra de menu (cuja variável é “menu”). Nota também que no segundo argumento meti uma string com o nome do menu.

```
menu.Append(menu_file,"File")
```

E agora já aparece a barra de menu com um menu lá encaixado. Agora temos de saber como acrescentar tópicos aos nossos menus que estão na barra de menu. Antes de o ensinar e nota que isto é importante é que tens de ter atenção a onde colocas o teu código porque o Python vai ler o código da primeira para a última linha e nesse caso precisas de definir primeiro a barra

de menu, depois o menu, depois os tópicos de menu e por aí adiante. Mas vamos lá agora acrescentar tópicos ao nosso menu “File”.

```
import wx

class MinhaFrame(wx.Frame):

    def __init__(self, parent, id):

        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (300, 200))

        self.panel=wx.Panel(self)

        self.panel.SetBackgroundColour('Ligth_Grey')


        wx.StaticText(self.panel,-1,"Nome do aluno:",(10,10))

        wx.StaticText(self.panel,-1,"Curso do aluno:",(10,30))


        self.text1=wx.StaticText(self.panel,-1,"Nenhum",(130,10))

        self.text1.SetForegroundColour('red')

        self.text2=wx.StaticText(self.panel,-1,"Nenhum",(130,30))

        self.text2.SetForegroundColour('red')


        menu=wx.MenuBar()

        self.SetMenuBar(menu)


        menu_file=wx.Menu()

        menu.Append(menu_file,"File")

        menu_file.Append(1,"Exit")


if __name__=='__main__':

    app=wx.App()

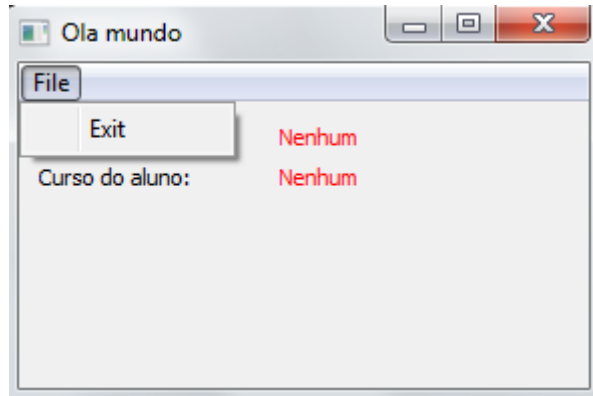
    frame=MinhaFrame(parent=None,id=999)

    frame.Centre()

    frame.Show()

    app.MainLoop()
```


Seguiu a mesma lógica dos menus anteriores excepto por não haver necessidade de criar um objecto antes de adiciona-lo ao menu. O primeiro argumento é o id que já vimos em tantos outros objectos nesta aprendizagem do wxPython, o segundo é a string que contém o nome que o tópico vai ter. O resultado é este:



Por agora estes menus e tópicos não vão servir para nada porque só mais tarde é que vamos aprender a dar-lhes funções. Vou inserir mais um tópico antes do "Exit" e acrescentar um separador entre eles (repara no sítio onde meto o código, como que que ele esteja antes do "Exit" tenho que meter o seu código antes de o do "Exit" também).

```
import wx
```

```
class MinhaFrame(wx.Frame):
```

```
    def __init__(self, parent, id):
```

```
        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (300, 200))
```

```
        self.panel=wx.Panel(self)
```

```
        self.panel.SetBackgroundColour('Ligth_Grey')
```

```
        wx.StaticText(self.panel,-1,"Nome do aluno:",(10,10))
```

```
        wx.StaticText(self.panel,-1,"Curso do aluno:",(10,30))
```

```
        self.text1=wx.StaticText(self.panel,-1,"Nenhum",(130,10))
```

```
        self.text1.SetForegroundColour('red')
```

```
        self.text2=wx.StaticText(self.panel,-1,"Nenhum",(130,30))
```

```
        self.text2.SetForegroundColour('red')
```

```
        menu=wx.MenuBar()
```

```
        self.SetMenuBar(menu)
```

```

        menu_file=wx.Menu()

        menu.Append(menu_file,"File")

        menu_file.Append(2,"Load")

        menu_file.AppendSeparator()

        menu_file.Append(1,"Exit")

if __name__=='__main__':

    app=wx.App()

    frame=MinhaFrame(parent=None,id=999)

    frame.Centre()

    frame.Show()

    app.MainLoop()

```

O resultado disto é este:

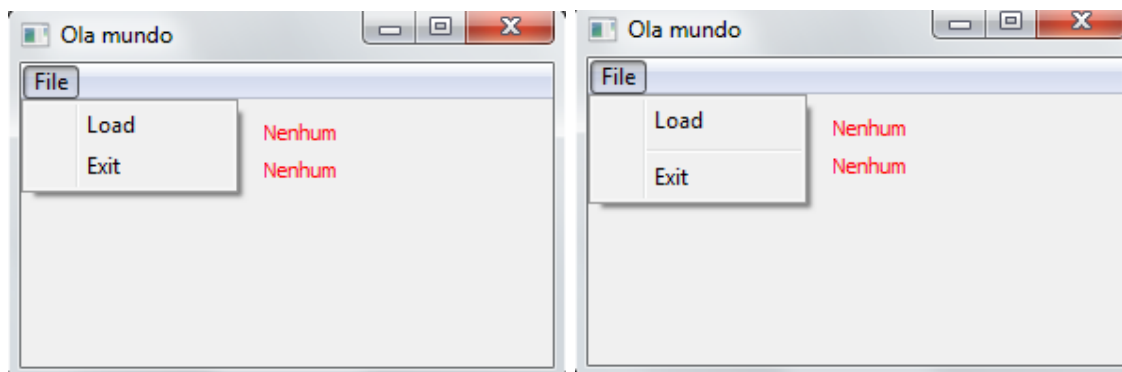


Figura 20- À esquerda estão os dois tópicos sem separador entre eles, à direita já acrescentei o separador.

Estas a ver o separador no meio deles (dependendo do sistema operativo o separador vai ser diferente). Na Figura 20 podes comparar a diferença entre não ter (esquerda) ou ter (direita) separador (neste caso no Windows 7). Vou agora acrescentar mais código ao nosso programa para que tenha mais um menu na barra de menus e um tópico lá dentro. Quero ter um menu chamado “Edit” e lá dentro o tópico “Definitions”.

```

import wx

class MinhaFrame(wx.Frame):

    def __init__(self, parent, id):

        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (300, 200))

```

```

self.panel=wx.Panel(self)

self.panel.SetBackgroundColour('Ligth_Grey')


wx.StaticText(self.panel,-1,"Nome do aluno:",(10,10))
wx.StaticText(self.panel,-1,"Curso do aluno:",(10,30))


self.text1=wx.StaticText(self.panel,-1,"Nenhum",(130,10))
self.text1.SetForegroundColour('red')
self.text2=wx.StaticText(self.panel,-1,"Nenhum",(130,30))
self.text2.SetForegroundColour('red')


menu=wx.MenuBar()

self.SetMenuBar(menu)

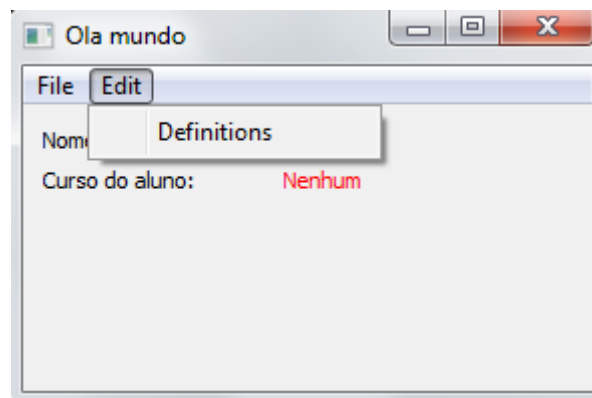

menu_file=wx.Menu()
menu.Append(menu_file,"File")
menu_file.Append(2,"Load")
menu_file.AppendSeparator()
menu_file.Append(1,"Exit")


menu_edit=wx.Menu()
menu.Append(menu_edit,"Edit")
menu_edit.Append(3,"Definitions")


if __name__=='__main__':
    app=wx.App()
    frame=MinhaFrame(parent=None,id=999)
    frame.Centre()
    frame.Show()
    app.MainLoop()

```

O resultado é este. Tenta perceber o que fiz agora à semelhança do que fiz anteriormente e nota os ids que tenho sempre a vir metendo diferente para não confundir os tópicos uns com os outros mais para a frente (e isto vai ser importante).



Para terminar a explicação sobre a barra de menu vou ensinar outra que está directamente associada que é a barra de status. Uma barra de status é uma barra que te dá informação sobre o tópico sobre o qual tens o ponteiro do rato em cima. Agora para existir essa informação tens de a escrever, e para a escreveres tens que fazer como terceiro argumento dos tópicos dos menus.

```
import wx
```

```
class MinhaFrame(wx.Frame):
```

```
    def __init__(self, parent, id):
```

```
        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (300, 200))
```

```
        self.panel=wx.Panel(self)
```

```
        self.panel.SetBackgroundColour('Ligth_Grey')
```

```
        wx.StaticText(self.panel,-1,"Nome do aluno:",(10,10))
```

```
        wx.StaticText(self.panel,-1,"Curso do aluno:",(10,30))
```

```
        self.text1=wx.StaticText(self.panel,-1,"Nenhum",(130,10))
```

```
        self.text1.SetForegroundColour('red')
```

```
        self.text2=wx.StaticText(self.panel,-1,"Nenhum",(130,30))
```

```
        self.text2.SetForegroundColour('red')
```

```
        menu=wx.MenuBar()
```

```
        self.SetMenuBar(menu)
```

```

status=self.CreateStatusBar()

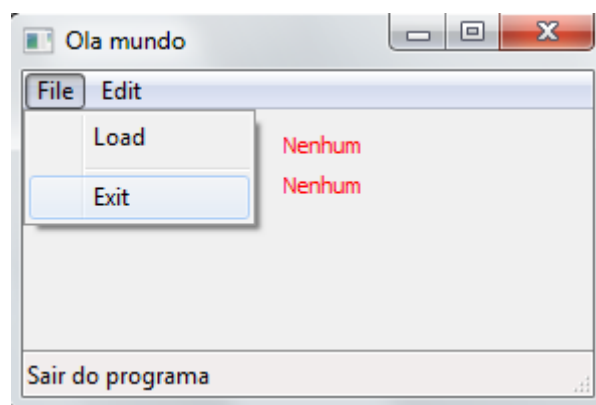
menu_file=wx.Menu()
menu.Append(menu_file,"File")
menu_file.Append(2,"Load", "Carregar dados de ficheiros.")
menu_file.AppendSeparator()
menu_file.Append(1,"Exit","Sair do programa")

menu_edit=wx.Menu()
menu.Append(menu_edit,"Edit")
menu_edit.Append(3,"Definitions","Definicoes do aluno.")

if __name__=='__main__':
    app=wx.App()
    frame=MinhaFrame(parent=None,id=999)
    frame.Centre()
    frame.Show()
    app.MainLoop()

```

O resultado é este (meti a barra de status logo após o código de criar a barra de menu com o comando “self.CreatStatusBar”).



Repara na explicação do “Exit” quando meti o ponteiro do rato lá em cima (compara com o código). Em relação à barra de menu e associados já terminamos, agora vamos ver como dar funcionalidades aos tópicos que estão no menu.

Eventos (parte 1)

Quando tu carregas no t pico “Exit”, por exemplo, ele est  a receber um sinal, simplesmente n o sabe o que h -de fazer ap s o receber e   precisamente isso que vamos fazer agora, dizer-lhe o que ele tem de fazer.   ac  o de carregar num objecto “carreg vel” e isso ter uma consequ ncia chama-se evento. Existem objectos como o painel, por exemplo, que n o t m eventos por muito que carregues neles, h  outros que a raz o de existir   precisamente de gerar um evento como   o caso dos nossos t picos nos menus da barra de menu. Vamos ver como atribuir uma ac  o ao “Exit”.

```
tutorial.py
1  # -*- coding: utf-8 -*-
2  """
3  Created on Sat Aug 28 01:35:30 2010
4
5  @author: Desktop
6  """
7  import wx
8
9  class MinhaFrame(wx.Frame):
10     def __init__(self, parent, id):
11         wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (300, 200))
12         self.panel=wx.Panel(self)
13         self.panel.SetBackgroundColour('Ligth_Grey')
14
15         wx.StaticText(self.panel,-1,"Nome do aluno:",(10,10))
16         wx.StaticText(self.panel,-1,"Curso do aluno:",(10,30))
17
18         self.text1=wx.StaticText(self.panel,-1,"Nenhum", (130,10))
19         self.text1.SetForegroundColour('red')
20         self.text2=wx.StaticText(self.panel,-1,"Nenhum", (130,30))
21         self.text2.SetForegroundColour('red')
22
23         menu=wx.MenuBar()
24         self.SetMenuBar(menu)
25         status=self.CreateStatusBar()
26
27         menu_file=wx.Menu()
28         menu.Append(menu_file,"File")
29         menu_file.Append(2,"Load", "Carregar dados de ficheiros.")
30         menu_file.AppendSeparator()
31         menu_file.Append(1,"Exit", "Sair do programa")
32         self.Bind(wx.EVT_MENU,self.sair,id=1)
33
34         menu_edit=wx.Menu()
35         menu.Append(menu_edit,"Edit")
36         menu_edit.Append(3,"Definitions", "Definicoes do aluno.")
37
38     def sair(self,event):
39         self.Close()
40
41 if __name__ == '__main__':
42     app=wx.App()
43     frame=MinhaFrame(parent=None,id=999)
44     frame.Centre()
45     frame.Show()
46     app.MainLoop()
```

Este é o código do programa como ele está agora com tudo o que fizemos. Existem duas coisas novas: o código que aparece na linha 32 e uma função nova que aparece na linha 38. Primeiro de tudo repara que a função está dentro da classe MinhaFrame. Isto é importante porque se queres criar um evento num objecto que está na classe MinhaFrame então a função que faz esse evento tem também de estar na MinhaFrame. Outra coisa importante é tu perceberes que um evento é uma função e por isso sempre que quiseses fazer um tens que fazer uma função correspondente. Existem (que eu conheça pelo menos) duas maneiras de chamar eventos. Agora vou ensinar uma que costumo utilizar nos menus (porque com esta maneira não preciso de ter variáveis atribuídas a esses menus), melhor dizendo aos tópicos dos menus. Vou transcrever a linha 32:

```
self.Bind(wx.EVT_MENU,self.sair,id=1)
```

Este comando “self.Bind” está a associar uma função a um objecto da nossa classe. No primeiro argumento tens que dizer o tipo de evento que é. Como se trata de um evento de menu (existem outros para botões, caixas de texto, etc., que vamos aprender mais para a frente) tens que utilizar o “wx.EVT_MENU”. No segundo argumento tens que dizer que função é que vai fazer esse evento (neste caso self.sair porque a função que criei se chama “sair”), no terceiro argumento tens que dar o id do objecto para o qual estás a associar o evento. O id do tópico “Exit” é 1 e por isso o id do terceiro argumento também é 1. Vamos agora ver a função:

```
def sair(self,event):  
    self.Close()
```

Fiz a função da mesma maneira que aprendemos a fazer anteriormente com o comando “def”, dando-lhe um nome (“sair”) e os respectivos argumentos que neste caso é “self”, porque vamos fazer uma acção sobre a própria janela onde estamos, e “event” porque temos que dizer à função que isto se trata de um evento e por isso tem de responder quando receber esse sinal. O único comando que inseri dentro da função é:

```
self.Close()
```

O “Close” serve para fechar a janela. Como aplicamos o “Close” à classe MinhaFrame (dai self.Close()) isto irá dar origem à saída da janela (e consequentemente do programa). Se experimentares agora o programa com este código ao carregares no “Exit” o programa irá sair. Agora nos próximos capítulos vou ensinar a mexer com mais objectos mas fica atento porque tenciono associar isso aos eventos para o “Load” e “Definitions”.

MessageDialog

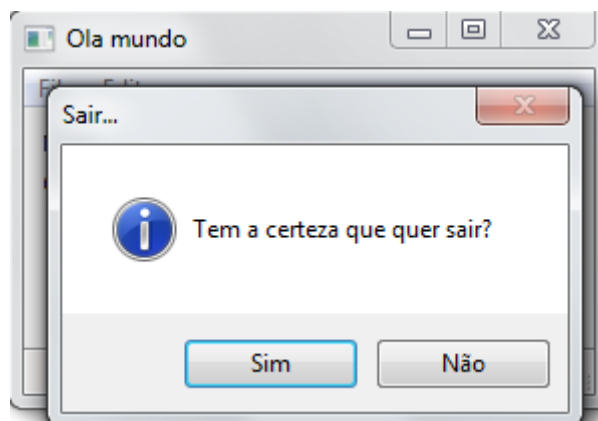
O wxPython já tem janelas feitas que não precisam que se lhe acrescente nada porque simplesmente servem um propósito muito específico e então apenas precisam de ter as funcionalidades que permitam esse propósito. Uma dessas janelas é o MessageDialog. No capítulo anterior associamos o tópico de menu “Exit” ao evento sair da janela e consequentemente do programa, mas regra geral o que se faz primeiro é fazer uma pergunta ao utilizador de se deseja mesmo sair do programa. Se sim então sai mesmo, se não permanece com o programa activo. Uma MessageDialog é ideal para este tipo de coisas. Vamos melhorar a função que utilizamos para o evento de fechar o programa de maneira a que possa ter esta funcionalidade. A função que temos para fechar o programa é:

```
def sair(self,event):  
  
    self.Close()
```

Vamos agora substituir isto por este código:

```
def sair(self,event):  
  
    a=wx.MessageDialog(self,"Tem a certeza que quer sair?","Sair...",wx.YES_NO)  
  
    if a.ShowModal()==wx.ID_YES:  
  
        self.Close()
```

Agora em vez de fazer o comando sair de imediato (“self.Close”) vai criar uma caixa com o comando “wx.MessageDialog” que fica atribuída à variável “a”. Esta linha vai instantaneamente criar uma caixa mal carregues no “Exit”. Os argumentos deste objecto são: o primeiro o sítio que cria a caixa (que é o “self” porque é a nossa classe MinhaFrame”), o segundo é a pergunta que aparece na caixa, o terceiro o nome que aparece no topo da caixa e o quarto o estilo da caixa. Neste caso escolhi o estilo wx.YES_NO porque preciso que o utilizador tenhas os botões de sim e não para poder responder à pergunta. Se experimentares e carregares no “Exit” com o código acima isto vai aparecer:



Agora consoante a resposta o programa tem de saber reagir e por isso fazemos essa pergunta com um “if”.

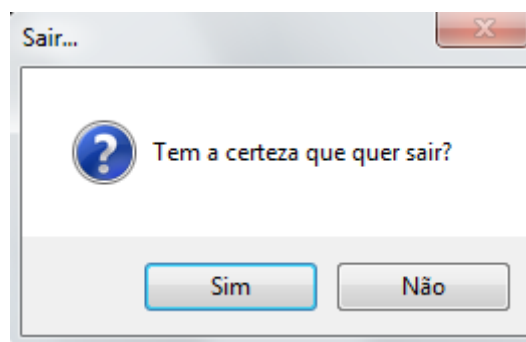

```
if a.ShowModal()==wx.ID_YES:

    self.Close()
```

O “ShowModal()” é o equivalente à resposta que que dás na caixa. Se o “a.ShowModal()” for igual a uma resposta de sim (“wx.ID_YES”) então faz o comando “self.Close()”, se for não então não faz nada porque nós não o programamos para fazer seja o que for. O ícone que aparece por defeito na caixa é o ícone de informação, mas podemos pedir para ter outros e para o fazer basta que digamos isso no estilo da caixa (lembra-te que o “|” separa os estilos).

```
a=wx.MessageDialog(self,"Tem a certeza que quer sair?" ,"Sair..." ,wx.YES_NO|wx.ICON_QUESTION)
```

Se estiver assim o ícone vai mudar:



Os estilos que podemos utilizar para este tipo de caixas são:

Estilo	Descrição
wx.OK	Mostra um botão de ok.
wx.CANCEL	Mostra um botão de cancelar.
wx.YES_NO	Mostra um botão de sim e outro de não.
wx.ICON_EXCLAMATION	Mostra um ícone de ponto de exclamação.
wx.ICON_HAND	Mostra um ícone de erro.
wx.ICON_ERROR	Mostra um ícone de erro.
wx.ICON_QUESTION	Mostra um ícone de ponto de interrogação.
wx.ICON_INFORMATION	Mostra um ícone de “i” de informação.
wx.STAY_ON_TOP	Força a janela a ficar por cima das outras.

Tabela 5- Tabela de estilos para uma MessageDialog.

Os ícones vão depender do sistema operativo em que estás a programar. No Windows vai aparecer uma coisa, no Ubuntu outra e por ai adiante, por isso não te preocupes se as tuas imagens não forem sempre iguais às minhas, pode muito bem ter a ver com esse facto. Para veres o estado em que está o código agora vou transcrevê-lo:

```
import wx

class MinhaFrame(wx.Frame):
    def __init__(self, parent, id):
        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (300, 200))
        self.panel=wx.Panel(self)
```

```

self.panel.SetBackgroundColour('Ligth_Grey')

wx.StaticText(self.panel,-1,"Nome do aluno:",(10,10))
wx.StaticText(self.panel,-1,"Curso do aluno:",(10,30))

self.text1=wx.StaticText(self.panel,-1,"Nenhum",(130,10))
self.text1.SetForegroundColour('red')
self.text2=wx.StaticText(self.panel,-1,"Nenhum",(130,30))
self.text2.SetForegroundColour('red')

menu=wx.MenuBar()
self.SetMenuBar(menu)
status=self.CreateStatusBar()

menu_file=wx.Menu()
menu.Append(menu_file,"File")
menu_file.Append(2,"Load", "Carregar dados de ficheiros.")
menu_file.AppendSeparator()
menu_file.Append(1,"Exit", "Sair do programa")
self.Bind(wx.EVT_MENU,self.sair,id=1)

menu_edit=wx.Menu()
menu.Append(menu_edit,"Edit")
menu_edit.Append(3,"Definitions", "Definicoes do aluno.")

def sair(self,event):
    a=wx.MessageDialog(self,"Tem a certeza que quer sair?","Sair...",wx.YES_NO|wx.ICON_QUESTION)
    if a.ShowModal()==wx.ID_YES:
        self.Close()

if __name__=='__main__':
    app=wx.App()
    frame=MinhaFrame(parent=None,id=999)
    frame.Centre()
    frame.Show()
    app.MainLoop()

```

Temos umas poucas dezenas de linhas de código mas o nosso programa já começa a ter forma. Vamos agora ver outra caixa que nos vai acompanhar daqui para a frente, a `FileDialog` para ir buscar ou salvar ficheiros.

FileDialog

O nosso código está a crescer e é cada vez mais difícil metê-lo aqui na totalidade. Tenho-o feito para que possas acompanhar e comparar directamente mas agora vou ser um pouco mais moderado (embora quando for caso disso volto a transcrevê-lo na totalidade novamente). Agora vamos fazer uma daquelas caixas de quando queremos abrir um ficheiro no computador. Chama-se FileDialog mas primeiro temos de criar o evento respectivo para o tópico de menu “Load”. Na porção de código correspondente a esta parte escrevi:

```
...

menu_file=wx.Menu()

menu.Append(menu_file,"File")

menu_file.Append(2,"Load", "Carregar dados de ficheiros.")

self.Bind(wx.EVT_MENU,self.abrir,id=2)

menu_file.AppendSeparator()

menu_file.Append(1,"Exit","Sair do programa")

self.Bind(wx.EVT_MENU,self.sair,id=1)

...
```

Repara que mais uma vez utilizei o comando “self.Bind” para gerar um evento para uma função que ainda não disse o que faz mas que se chama “abrir”. O id é, naturalmente, correspondente ao tópico “Load”. A função “abrir” por seu lado ficou escrita da seguinte maneira:

```
...

menu_edit=wx.Menu()

menu.Append(menu_edit,"Edit")

menu_edit.Append(3,"Definitions","Definicoes do aluno.")


def abrir(self,event):

    b=wx.FileDialog(self,"Abrir ficheiro de dados...",style=wx.OPEN,wildcard="*.*.")

    if b.ShowModal() == wx.ID_OK:

        b.Close()


def sair(self,event):

    a=wx.MessageDialog(self,"Tem a certeza que quer sair?","Sair...",wx.YES_NO|wx.ICON_QUESTION)

    ...
```

O que precisamos de observar atentamente é o que está a negrito e por agora já debes ir reconhecendo mais ou menos a maneira como a maioria dos objectos em wxPython são construídos (o código que está acima não faz nada excepto abrir uma janela de ficheiros mas em breve voltaremos a usa-la para abrir mesmo os ficheiros em si).

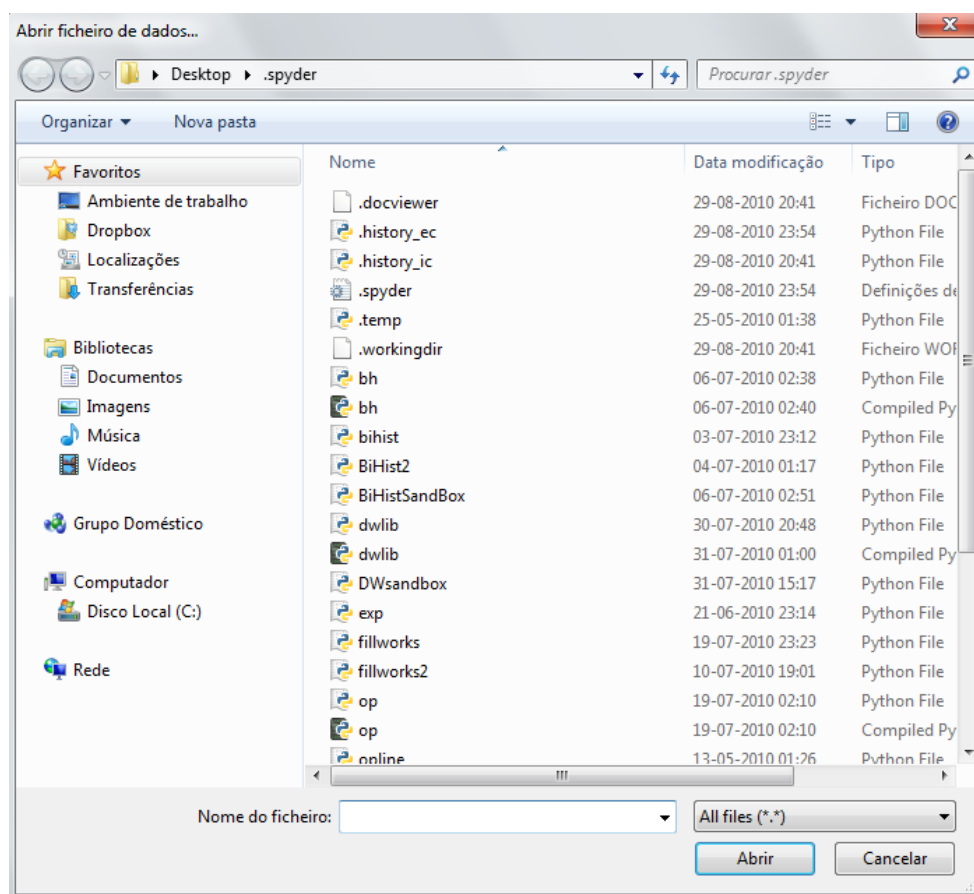
```
b=wx.FileDialog(self,"Abrir ficheiro de dados...",style=wx.OPEN,wildcard="*..*")
```

Atribui à variável “b” uma caixa de ficheiro com o comando “wx.FileDialog”. O primeiro argumento é a classe em que a caixa está alocada (que, mais uma vez, é “self” porque se trata da nossa classe). O segundo argumento é o título que aparece no topo da caixa, o terceiro argumento o estilo (podia ser wx.SAVE para salvar ficheiros em vez de carrega-los) e o quarto argumento é o wildcard ao qual eu meti uma string “*..*”. Esta string quer dizer que o ficheiro pode ter qualquer extensão (.txt, .prn, .py, etc...). Se eu quisesse que na caixa apenas aparecessem ficheiros de determinada extensão era aqui que eu tinha de definir isso. O resto das linhas de código que inseri:

```
if b.ShowModal() == wx.ID_OK:
```

```
    b.Close()
```

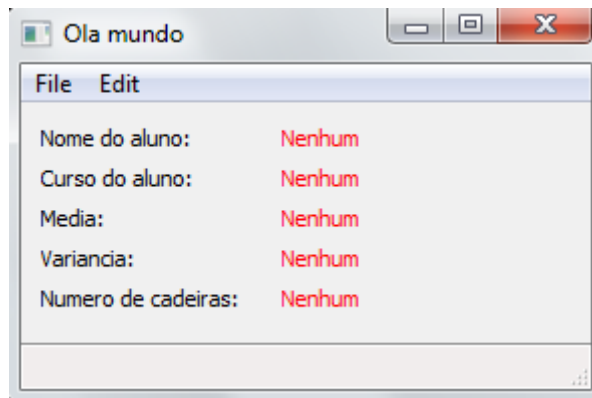
Não estão a fazer nenhuma acção em particular (excepto potencialmente mandar o programa abaixo quando tentas carregar um ficheiro). Meti lá porque por razões que desconheço a caixa não parecia funcionar se eu não fizesse isso. De qualquer das maneiras o resultado deverá ser algo semelhante a isto:



Agora para darmos uma real utilidade a esta caixa precisamos de criar um real utilidade para o programa no que toca a carregar dados de ficheiros. Decidi que o ficheiro que vamos receber teria no seu conteúdo as notas das cadeiras do curso e assim o programa irá informar da média, variância e número de cadeiras. Vou fazer-lo com textos estáticos na janela principal e por isso alterei o código nesse sentido.

```
...  
  
wx.StaticText(self.panel, -1, "Nome do aluno:", (10, 10))  
  
wx.StaticText(self.panel, -1, "Curso do aluno:", (10, 30))  
  
wx.StaticText(self.panel, -1, "Media:", (10, 50))  
  
wx.StaticText(self.panel, -1, "Variancia:", (10, 70))  
  
wx.StaticText(self.panel, -1, "Numero de cadeiras:", (10, 90))  
  
  
self.text1=wx.StaticText(self.panel, -1, "Nenhum", (130, 10))  
self.text1.SetForegroundColour('red')  
self.text2=wx.StaticText(self.panel, -1, "Nenhum", (130, 30))  
self.text2.SetForegroundColour('red')  
self.text3=wx.StaticText(self.panel, -1, "Nenhum", (130, 50))  
self.text3.SetForegroundColour('red')  
self.text4=wx.StaticText(self.panel, -1, "Nenhum", (130, 70))  
self.text4.SetForegroundColour('red')  
self.text5=wx.StaticText(self.panel, -1, "Nenhum", (130, 90))  
self.text5.SetForegroundColour('red')  
  
  
menu=wx.MenuBar()  
self.SetMenuBar(menu)  
status=self.CreateStatusBar()  
  
...
```

Mais uma vez, a negrito está o código acrescentado. O resultado na janela principal é mais uns textos estáticos acrescentados. Agora a minha ideia é que mal se carregue o ficheiro o programa calcule a média, variância e número de cadeiras e actualize os textos estáticos (convertendo os números em strings) na janela principal.



Agora vamos ter de mudar o código da função “abrir” para que execute todas essas funções. A primeira coisa que precisamos de ter noção é que a caixa “FileDialog” não carrega o ficheiro apenas dá o caminho ou localização do sítio onde o ficheiro está num string. Com isso podemos usar, por exemplo, a função “numpy.loadtxt” que já aprendemos num capítulo anterior. Vamos então ver que modificações fiz ao código.

```
import wx

import numpy

class MinhaFrame(wx.Frame):

    def __init__(self, parent, id):

        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (300, 200))

        ...

        menu_edit=wx.Menu()

        menu.Append(menu_edit,"Edit")

        menu_edit.Append(3,"Definitions","Definicoes do aluno.")

    def abrir(self,event):

        b=wx.FileDialog(self,"Abrir ficheiro de dados...",style=wx.OPEN,wildcard="*.*.")

        if b.ShowModal() == wx.ID_OK:

            self.filename=b.GetPath()

            self.file=numpy.loadtxt(self.filename)

            numero1=repr(numpy.mean(self.file))

            self.text3.SetLabel(numero1)

            numero2=repr(numpy.var(self.file))

            self.text4.SetLabel(numero2)

            numero3=repr(self.file.shape[0])

            self.text5.SetLabel(numero3)
```

```

        b.Close()

def sair(self,event):

    a=wx.MessageDialog(self,"Tem a certeza que quer sair?", "Sair...",wx.YES_NO|wx.ICON_QUESTION)

    if a.ShowModal()==wx.ID_YES:

        self.Close()

    ...

```

Como vou utilizar funções da biblioteca numpy comecei por importar o numpy com “import numpy”. Depois passei para a função “abrir” em si. A maneira de ir buscar a string do caminho do ficheiro que escolhemos no FileDialog é com o comando “GetPath()”, neste caso “b.GetPath()” porque atribui o FileDialog à variável “b”. Esta string ficou atribuída à variável “self.filename” que a seguir foi utilizada no comando “numpy.loadtxt” para atribuir os dados do ficheiro a uma matriz na variável “self.file”. Na parte de código que vem a seguir:

```

numero1=repr(numpy.mean(self.file))

self.text3.SetLabel(numero1)

numero2=repr(numpy.var(self.file))

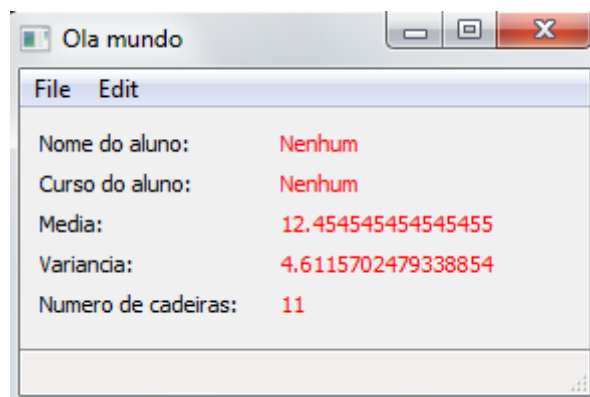
self.text4.SetLabel(numero2)

numero3=repr(self.file.shape[0])

self.text5.SetLabel(numero3)

```

Estou a fazer algumas das operações que já utilizamos antes. Primeiro calculo a média da matriz na variável “self.file” e logo de pois converti-a numa string com o comando “repr” (está tudo na mesma linha de código portanto tenta não confundir-te). Agora que já tenho uma string posso mudar o que está na minha variável de texto estático que se encontra na janela principal (“self.text3”, se tiveres dúvidas vai ver que variável é esta no código). As outras quatro linhas de código que restam são a mesma coisa. O resultado quando insiro um ficheiro lá dentro (eu já mostro o que contém) é este:



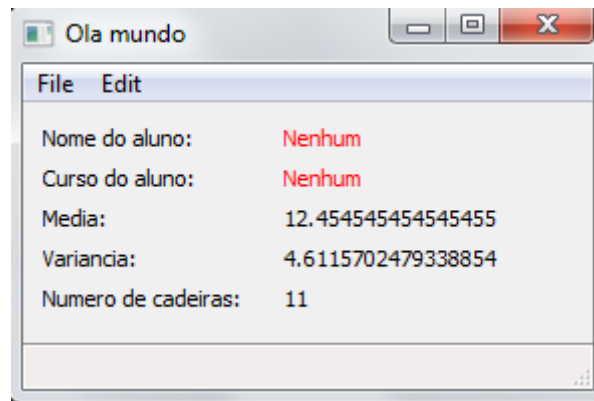
Criei um ficheiro teste com uns números em formato coluna (teste.txt) que contém isto:

10
11
15
12
16
13
10
10
11
15
14

E foram estes dados que deram o resultado que viste na figura. E prontos, já demos uma utilidade à caixa. Os números ficam com uma precisão muito acima do necessário e é possível corrigir esse problema mas prefiro não me desviar muito do essencial porque este é um exemplo de pequenas situações que vão sempre aparecer para resolver (mas no fim quando terminarmos de fazer eu vou aproveitar algumas coisas que quero ensinar para resolver este problema). Vou sim fazer com que os textos estáticos deixem de ser vermelhos quando são calculados os números e se percebeste o código acima isto então vai ser muito fácil de entender. Acrescentei o seguinte ao código:

```
...  
  
    numero2=repr(numpy.var(self.file))  
  
    self.text4.SetLabel(numero2)  
  
    numero3=repr(self.file.shape[0])  
  
    self.text5.SetLabel(numero3)  
  
    self.text3.SetForegroundColour('black')  
  
    self.text4.SetForegroundColour('black')  
  
    self.text5.SetForegroundColour('black')  
  
    b.Close()  
  
...
```

O resultado é este:



Vou agora transcrever o código tal como ele está neste momento para que possas ir comparando com o teu:

```
import wx

import numpy

class MinhaFrame(wx.Frame):

    def __init__(self, parent, id):

        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (300, 200))

        self.panel=wx.Panel(self)

        self.panel.SetBackgroundColour('Ligth_Grey')


        wx.StaticText(self.panel,-1,"Nome do aluno:",(10,10))

        wx.StaticText(self.panel,-1,"Curso do aluno:",(10,30))

        wx.StaticText(self.panel,-1,"Media:",(10,50))

        wx.StaticText(self.panel,-1,"Variancia:",(10,70))

        wx.StaticText(self.panel,-1,"Numero de cadeiras:",(10,90))


        self.text1=wx.StaticText(self.panel,-1,"Nenhum",(130,10))
        self.text1.SetForegroundColour('red')

        self.text2=wx.StaticText(self.panel,-1,"Nenhum",(130,30))
        self.text2.SetForegroundColour('red')

        self.text3=wx.StaticText(self.panel,-1,"Nenhum",(130,50))
        self.text3.SetForegroundColour('red')

        self.text4=wx.StaticText(self.panel,-1,"Nenhum",(130,70))
        self.text4.SetForegroundColour('red')
```

```

self.text5=wx.StaticText(self.panel,-1,"Nenhum",(130,90))

self.text5.SetForegroundColour('red')


menu=wx.MenuBar()

self.SetMenuBar(menu)

status=self.CreateStatusBar()


menu_file=wx.Menu()

menu.Append(menu_file,"File")

menu_file.Append(2,"Load", "Carregar dados de ficheiros.")

self.Bind(wx.EVT_MENU,self.abrir,id=2)

menu_file.AppendSeparator()

menu_file.Append(1,"Exit","Sair do programa")

self.Bind(wx.EVT_MENU,self.sair,id=1)


menu_edit=wx.Menu()

menu.Append(menu_edit,"Edit")

menu_edit.Append(3,"Definitions","Definicoes do aluno.")


def abrir(self,event):

    b=wx.FileDialog(self,"Abrir ficheiro de dados...",style=wx.OPEN,wildcard="*.*.")

    if b.ShowModal() == wx.ID_OK:

        self.filename=b.GetPath()

        self.file=numpy.loadtxt(self.filename)

        numero1=repr(numpy.mean(self.file))

        self.text3.SetLabel(numero1)

        numero2=repr(numpy.var(self.file))

        self.text4.SetLabel(numero2)

        numero3=repr(self.file.shape[0])

        self.text5.SetLabel(numero3)

        self.text3.SetForegroundColour('black')

        self.text4.SetForegroundColour('black')

```

```

        self.text5.SetForegroundColour('black')

        b.Close()

def sair(self,event):

    a=wx.MessageDialog(self,"Tem a certeza que quer sair?","Sair...",wx.YES_NO|wx.ICON_QUESTION)

    if a.ShowModal()==wx.ID_YES:

        self.Close()

if __name__=='__main__':

    app=wx.App()

    frame=MinhaFrame(parent=None,id=999)

    frame.Centre()

    frame.Show()

    app.MainLoop()

```

AboutBox

Uma “AboutBox” serve para dar descrições do programa como licença, copyright, programadores, documentação, etc. É uma caixa já feita e apenas a precisamos de a chamar mas primeiro que tudo tenho que criar o sitio onde a vou buscar. Inseri as seguintes linhas de código no programa:

```
...

menu_edit=wx.Menu()

menu.Append(menu_edit,"Edit")

menu_edit.Append(3,"Definitions","Definicoes do aluno.")


menu_about=wx.Menu()

menu.Append(menu_about,"About")

menu_about.Append(10,"Acerca..","Acerca do programas tutorial...")

self.Bind(wx.EVT_MENU,self.acerca,id=10)

...
```

Nada que nunca tenhamos visto. Limitei-me a criar um novo menu na barra de menus e dentro dele criei um tópico chamado “Acerca...”. A seguir relacionei o seu evento à função “acerca” que a seguir vemos escrita:

```
...

def acerca(self,event):

    info = wx.AboutDialogInfo()

    info.SetName('Tutorial')

    info.SetVersion('0.1')

    info.SetDescription('Tutorial')

    info.SetCopyright('Versao teste sem copyrigh')

    info.SetWebSite('http://numist.ist.utl.pt')

    info.SetLicence('Nao tem.')

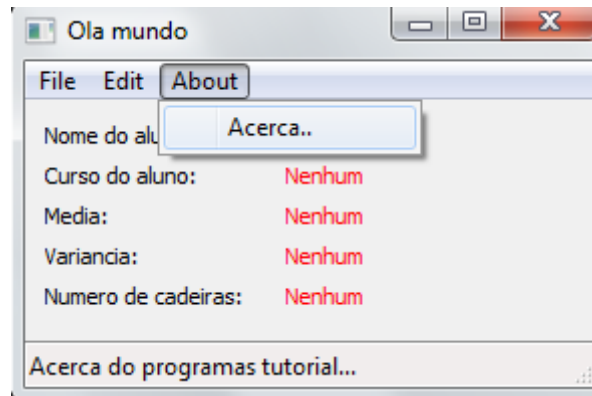
    info.AddDeveloper('Pedro Correia')

    info.AddDocWriter('Pedro Correia')

    wx.AboutBox(info)

...
```

É bastante simples na verdade. Criei a função da maneira que tenho criado até agora e atribui à variável `info` um objecto chamado `"wx.AboutDialogInfo"`. Este objecto diz respeito à informação da caixa e é isso que começo a definir nas instruções a seguir, sempre com strings (`"SetName"`, `"SetVersion"`, `"SetDescription"`, etc...). A última linha define a caixa em si e apenas precisamos de adicionar a informação `"info"` à mesma com o comando `"wx.AboutBox"`. O resultado é este:



Para o menu que criamos, e este para a caixa em si:

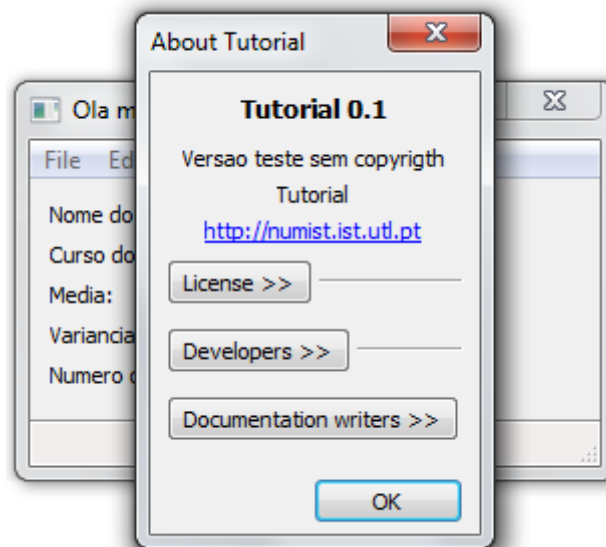


Figura 21 - Exemplo de AboutBox.

Como podes reparar tem lá os campos todos e basta carregar nos botões para os visualizar. Ainda existem outros campos que podes adicionar (sim, porque se não quiseses meter algum basta que não o adiciones ao `"info"`) que podes procurar na net.

Janelas Secundárias

Janelas secundárias são em tudo iguais às principais, simplesmente as editamos para servir outros propósitos que não sejam ter barra de menus e outro tipo de coisas que aparecem numa janela principal. Os objectos que vão aparecer aqui podem ser metidos também na principal, simplesmente escolhi mostra-los aqui porque, regra geral, é onde aparecem. O que é importante para começar este capítulo é saber como as fazer (que até já sabemos porque já construímos uma) e como as associar à principal. Já deves ter reparado que existe um menu de “Edit” no programa que contém um tópico “Definitions”. Neste momento não faz nada mas vou gerar um evento nele para que chame uma janela própria onde podes definir o nome do aluno e o curso (que aparecem na janela principal).

Associar janelas secundárias às principais

Pouco deverás ir inteirando-te melhor do que é uma classe. Se olhares para o que é a classe da janela principal vais reparar que tens não só múltiplos objectos lá dentro como também funções. O que nós vamos fazer a seguir é em tudo igual simplesmente declaramos uma janela como sendo pai (a principal) e as restantes como sendo filhas (a janela secundária). Já veremos como fazer isto. Vou começar por criar uma nova classe que no código significa isto:

```
import wx

import numpy

class DefFrame(wx.Frame):

    def __init__(self, parent, id):

        wx.Frame.__init__(self, parent, id, 'definicoes', size = (300, 150))

class MinhaFrame(wx.Frame):

    def __init__(self, parent, id):

        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (300, 200))

        self.panel=wx.Panel(self)

        self.panel.SetBackgroundColour('Ligth_Grey')

    ...
```

Repara, a minha nova classe chama-se DefFrame e se comparares com o início da classe referente à janela principal vai ver que é exactamente igual em termos estruturais, limitei-me a mete-la mais pequena verticalmente e mudar-lhe o título que aparece no topo. Não podemos usa-la agora porque ainda não criamos um evento que faça isso. Assim fui ao sítio

onde está o tópico “Definitions” e fiz a ligação (“Bind”) com uma função que chamei “definicoes” (e que à semelhança das anteriores está dentro da classe “MinhaFrame” porque o evento é na “MinhaFrame”).

```
...

menu_file.AppendSeparator()

menu_file.Append(1,"Exit","Sair do programa")

self.Bind(wx.EVT_MENU,self.sair,id=1)


menu_edit=wx.Menu()

menu.Append(menu_edit,"Edit")

menu_edit.Append(3,"Definitions","Definicoes do aluno.")

self.Bind(wx.EVT_MENU,self.definicoes,id=3)


menu_about=wx.Menu()

menu.Append(menu_about,"About")

menu_about.Append(10,"Acerca..","Acerca do programas tutorial...")

...
```

Agora precisamos de fazer a função “definicoes”. Para isso vou-te recordar como chamamos nós a classe principal (está no fim do código):

```
...

if __name__=='__main__':

    app=wx.App()

    frame=MinhaFrame(parent=None,id=999)

    frame.Centre()

    frame.Show()

    app.MainLoop()
```

Repara no que está a negrito. Quando começamos o programa atribuímos a classe “MinhaFrame” à variável “frame” e depois centramos a mesma e pedimos para que fosse visível (“Centre” e “Show”). Vamos fazer exactamente a mesma coisa para a nova classe só que é chamada dentro da classe “MinhaFrame” e o argumento parente em vez de ser nenhum (None) será a nossa janela principal que está atribuída à variável “frame”.

```

...

menu_about.Append(10,"Acerca..","Acerca do programas tutorial...")

self.Bind(wx.EVT_MENU,self.acerca,id=10)


def definicoes(self,event):

    dframe=DefFrame(parent=frame,id=998)

    dframe.Centre()

    dframe.Show()


def acerca(self,event):

    info = wx.AboutDialogInfo()

    info.SetName('Tutorial')

    info.SetVersion('0.1')

    ...

```

E ai está. O mesmo processo a ocorrer novamente mas com os argumentos diferentes. Agora já temos o início da janela de definições (Figura 22).

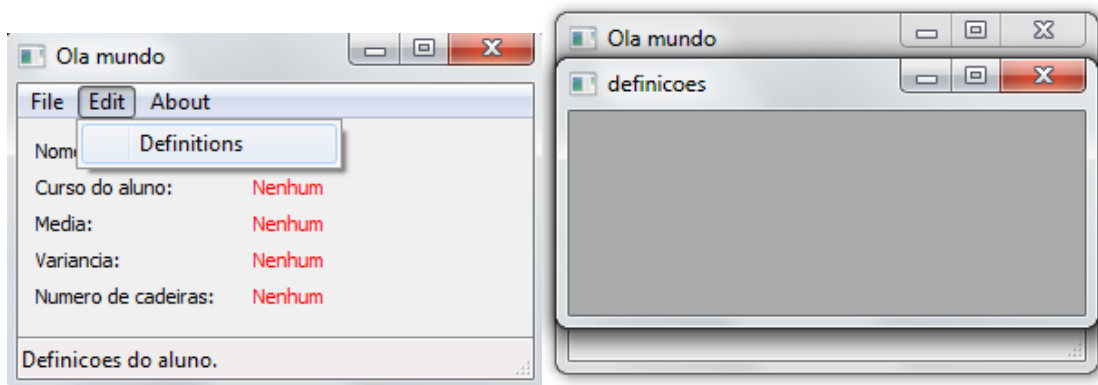


Figura 22 - Janela secundária associada a uma principal.

Vamos agora começar a adicionar objectos à nossa janela das definições.

Botões

Os botões são, geralmente, ferramentas úteis quando se pretende iniciar uma acção qualquer, confirma-la ou até cancela-la. Quando mexemos com o `MessageDialog` vimos botões mas já estavam feitos, agora vamos nós fazê-los. Mas antes de meter os botões tenho inserir o sítio onde eles estão que, caso estejas esquecido, é um painel.

```
import wx

import numpy

class DefFrame(wx.Frame):

    def __init__(self, parent, id):

        wx.Frame.__init__(self, parent, id, 'definicoes', size = (300, 150))

        panel=wx.Panel(self)

        panel.SetBackgroundColour('Ligth_Grey')

        ...
```

O painel está inserido e agora pode conter objectos sendo que vamos construir dois botões, “Ok” e “Cancelar”.

```
class DefFrame(wx.Frame):

    def __init__(self, parent, id):

        wx.Frame.__init__(self, parent, id, 'definicoes', size = (300, 150))

        panel=wx.Panel(self)

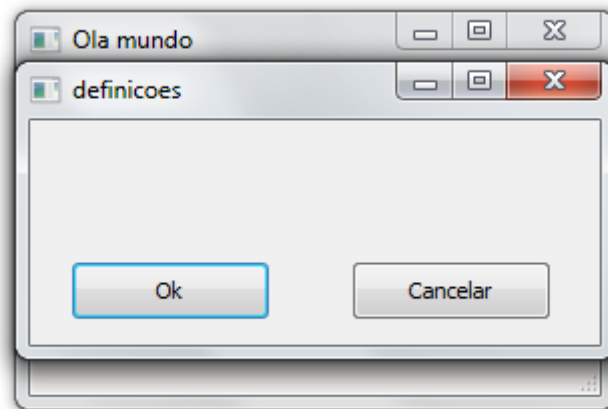
        panel.SetBackgroundColour('Ligth_Grey')

        self.btn_ok=wx.Button(panel, -1, 'Ok', (20,70), (100,30))

        self.btn_cancel=wx.Button(panel, -1, 'Cancelar', (160,70), (100,30))

        ...
```

Atribui os botões com o comando “`wx.Button`” a variáveis porque daqui a pouco vou fazer eventos relacionados com eles (da tal maneira diferente do que já aprendemos). O primeiro argumento é o sítio onde se encontra (painel que atribui à variável “panel”), o segundo argumento o id (que ao fazer -1 estou a dizer ao wxPython para por lá um por mim), o terceiro argumento a string que vai aparecer no botão, o quarto argumento a posição onde está e por fim o quinto argumento o tamanho do botão. O resultado deste acréscimo de código é este:



Eventos (parte 2)

Agora vamos fazer o evento para o botão de cancelar. O que eu quero que aconteça quando o utilizador carregar cancelar é que a janela desapareça sem fazer nada. Para isso basta dar um comando para a janela fechar mas vamos ver como fazer o evento para o botão:

```
...
class DefFrame(wx.Frame):
    def __init__(self, parent, id):
        wx.Frame.__init__(self, parent, id, 'definicoes', size = (300, 150))
        panel=wx.Panel(self)
        panel.SetBackgroundColour('Ligth_Grey')

        self.btn_ok=wx.Button(panel,-1,'Ok',(20,70),(100,30))
        self.btn_cancel=wx.Button(panel,-1,'Cancelar',(160,70),(100,30))
        wx.EVT_BUTTON(self,self.btn_cancel.GetId(),self.cancel)
    ...
```

Foi simples. Chamei o comando “wx.EVT_BUTTON” e meti como primeiro argumento a própria classe (“self”), como segundo argumento o id do botão de cancelar (com o comando “GetId()” nem precisamos de saber o número ele vai buscar automaticamente, por isso eu costumo inserir o -1 nos sítios dos ids), como terceiro argumento a função a que este evento está ligado (que ainda não escrevi). Agora escrevendo a função teremos dado uma funcionalidade ao botão cancelar.

```

import wx

import numpy

class DefFrame(wx.Frame):

    def __init__(self, parent, id):

        wx.Frame.__init__(self, parent, id, 'definicoes', size = (300, 150))

        panel=wx.Panel(self)

        panel.SetBackgroundColour('Ligth_Grey')


        self.btn_ok=wx.Button(panel,-1,'Ok',(20,70),(100,30))

        self.btn_cancel=wx.Button(panel,-1,'Cancelar',(160,70),(100,30))

        wx.EVT_BUTTON(self,self.btn_cancel.GetId(),self.cancel)


    def cancel(self,event):

        self.Destroy()


class MinhaFrame(wx.Frame):

    def __init__(self, parent, id):

        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (300, 200))

        self.panel=wx.Panel(self)

        ...

```

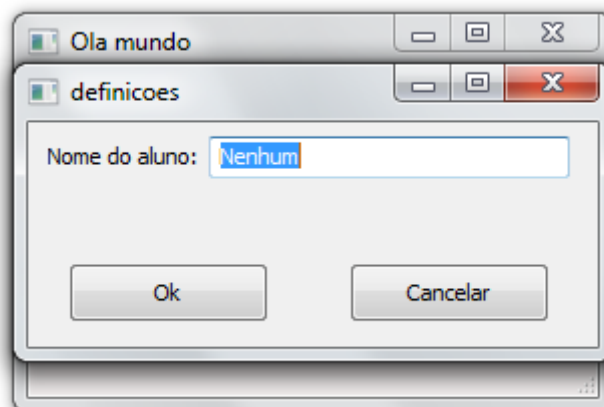
Utilizei o comando “Destroy” que é semelhante ao “Close” que já utilizamos anteriormente (não sei bem qual é a diferença entre os dois, se é que a há), de resto temos uma função que já vimos antes. A partir de agora o botão cancelar já está a funcionar e mal carregues nele irás sair da janela de definições. Mais tarde vamos criar um evento para o botão de ok mas por agora vamos meter mais alguns objectos na janela para que ele possa fazer qualquer coisa. Se bem te lembras existem dois campos na janela principal (nome do aluno e curso) que ainda não mexemos. É com esta caixa que o vamos fazer inserindo um TextCtrl (um sítio onde poderás escrever o nome do aluno) e uma ComboBox (um sítio onde poderás escolher um curso a partir de uma lista). Vamos começar pelo TextCtrl.

TextCtrl

Um TextCtrl é um caixa que serve para dar informações ao computador a partir do teclado. O que pretendo fazer é que o utilizador do nosso programa possa usar essa caixa para inserir o nome do aluno o qual está a analisar. Para o fazer terá que ir ao tópico “Definitions” e assim entrar na janela secundária que definimos para trás. Acrescentei o código seguinte à classe correspondente à janela das definições:

```
...  
  
panel=wx.Panel(self)  
  
panel.SetBackgroundColour('Ligth_Grey')  
  
  
wx.StaticText(panel,-1,"Nome do aluno:",(10,10))  
  
self.nome_aluno=wx.TextCtrl(panel,-1,'Nenhum',(90,7),(180,-1))  
  
  
self.btn_ok=wx.Button(panel,-1,'Ok',(20,70),(100,30))  
  
self.btn_cancel=wx.Button(panel,-1,'Cancelar',(160,70),(100,30))  
  
...
```

Não queria deixar a caixa sozinha sem dar a indicação que seria o nome do aluno teria que ser inserido naquele sítio. Assim acrescentei também um texto estático para identificar esse sítio. Atribui também o objecto “wx.TextCtrl” à variável “self.nome_aluno”. Em relação aos argumentos muitos já deverão parecer familiares. O primeiro é o sítio onde a caixa se encontra (no painel), o segundo o id deste objecto (-1 para ser o wxPython a escolher um), o terceiro argumento para ser a string que já se encontra na caixa por defeito, o quarto a posição da caixa, e quinto o tamanho (o -1 que se encontra no tamanho vertical é uma medida standard que basicamente diz para o tamanho vertical da caixa se adaptar ao tamanho normal do texto que lá é inserido). O resultado é este:



Agora pretendo que quando o utilizador carregar no botão “Ok” seja actualizada a informação de nome do aluno na janela principal. Para isso tenho que criar o evento correspondente e a função que ele vai fazer funcionar. Acrescentei o seguinte código:

```
...

self.btn_ok=wx.Button(panel,-1,'Ok',(20,70),(100,30))

wx.EVT_BUTTON(self,self.btn_ok.GetId(),self.ok)

self.btn_cancel=wx.Button(panel,-1,'Cancelar',(160,70),(100,30))

wx.EVT_BUTTON(self,self.btn_cancel.GetId(),self.cancel)


def ok(self,event):

    a=self.nome_aluno.GetValue()

    frame.text1.SetLabel(a)

    frame.text1.SetForegroundColour('black')

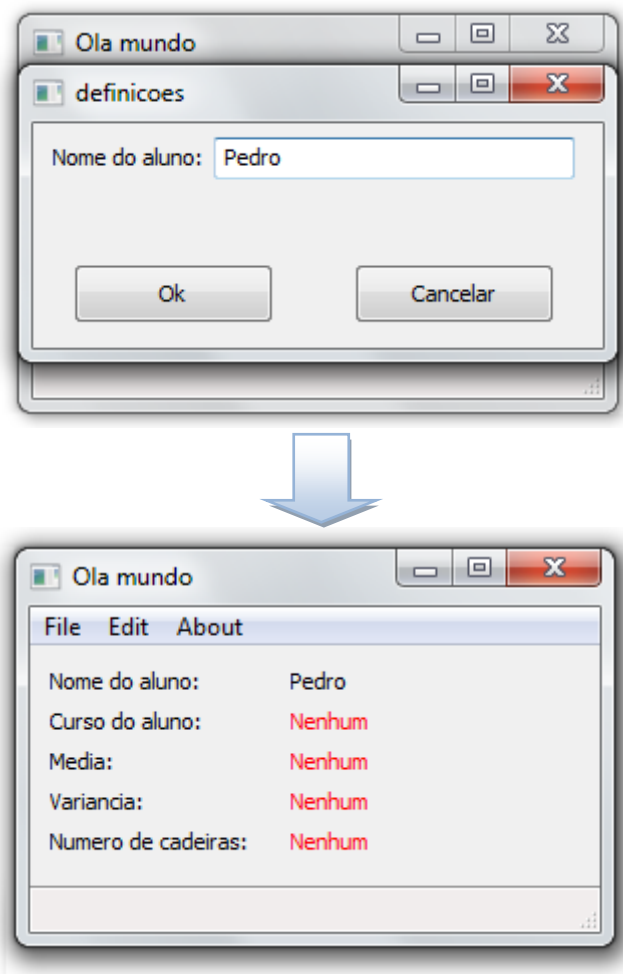
    self.Destroy()


def cancel(self,event):

    self.Destroy()

...
```

A ligação do evento mais uma vez foi feita com o comando “wx.EVT_BUTTON” neste caso para o botão de “Ok” e ficou ligado à função de nome “ok”. A primeira coisa que fiz dentro dessa função foi receber o valor que está dentro da caixa TextCtrl e fiz isso com o comando “GetValue()” a partir da variável a qual estava atribuída esse objecto. A seguir modifiquei o texto que aparece na janela principal e aqui chamo a atenção para uma coisa. Repara que em vez de utilizar o “self” utilizei o “frame”, isto porque a variável “frame” é a variável à qual está atribuída a classe da janela principal (olha para o código para compreenderes isto). Se eu utilizasse o “self” aqui estaria a referir-me à classe onde estou a escrever esta função que é a classe da janela de definições. Para que isto não acontecesse tive que chamar a variável correspondente. Depois mudei a cor desse mesmo texto estático para preto (estava vermelho antes). No final dei a instrução para fechar a janela porque não faria sentido ficar aberta se as informações já estavam feitas.



Agora só nos falta arranjar uma maneira de mudar o curso do aluno também. Para fazer isso vou usar uma ComboBox como verás no capítulo a seguir.

ComboBox

Uma ComboBox é um objecto que contém uma lista já feita do qual o utilizador pode escolher uma das opções. Em relação ao curso é assim que vamos fazer. Fazer uma lista que contenha alguns cursos e a partir daí dar ao utilizador a hipótese de escolher uma delas actualizando a janela principal com essa hipótese. Acrescentei o seguinte código:

...

```
wx.StaticText(panel,-1,"Nome do aluno:",(10,10))

self.nome_aluno=wx.TextCtrl(panel,-1,'Nenhum',(90,7),(180,-1))

wx.StaticText(panel,-1,"Curso do aluno:",(10,40))

cursos=['Ambiente','Biomedica','Civil','Fisica','Minas','Mecanica']
```

```
self.curso=wx.ComboBox(panel,-1,value=cursos[0],pos=(90,37),size=(180,-1),
choices=cursos,style=wx.CB_READONLY)
```

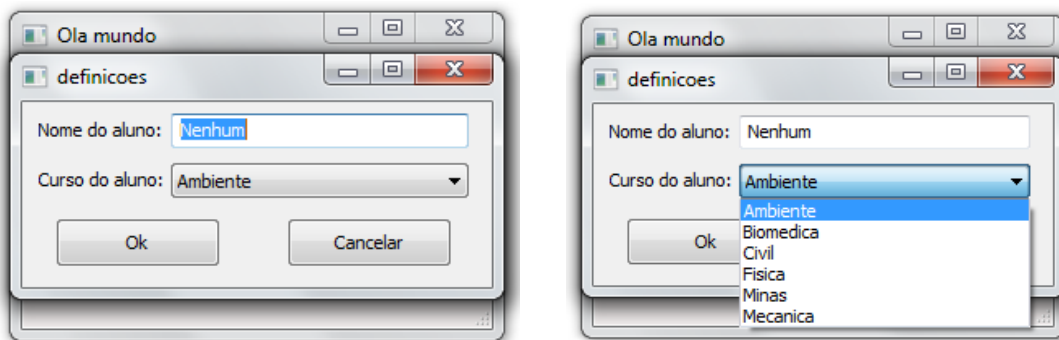
```
self.btn_ok=wx.Button(panel,-1,'Ok',(20,70),(100,30))
```

```
wx.EVT_BUTTON(self,self.btn_ok.GetId(),self.ok)
```

```
self.btn_cancel=wx.Button(panel,-1,'Cancelar',(160,70),(100,30))
```

...

Mais uma vez para que a ComboBox não ficasse sem informação acerca do que se tratava, meti um texto estático a indicar que ali era o sítio para dizer qual o curso do aluno. O resultado é este:



Primeiro criei uma lista de strings que ficou atribuída à variável “cursos” e continha os nomes de uma série de cursos superiores. Depois construí o objecto “wx.ComboBox” atribuindo-o à variável “self.curso” (o texto está em duas linhas porque não há espaço na folha para meter só numa linha). O primeiro argumento é o sítio onde se encontra a ComboBox, o segundo o id, o terceiro o valor que aparecer por defeito (que eu indiquei que seria o primeiro da lista “cursos”), o quarto a posição da ComboBox, o quinto o tamanho e finalmente o sexto o estilo da ComboBox. Em relação aos estilos não os conheço de todo embora sei que existem mais. A seguir uma tabela com os mais usados.

Estilo	Descrição
wx.CB_SIMPLE	Lista de visualização permanente.
wx.CB_READONLY	Lista de queda ao carregar.
wx.CB_SORT	Organizar as entradas alfabeticamente.

Tabela 6 - Tabela de estilos de uma ComboBox.

Bem agora para que tudo isto tenha algum efeito prático precisamos de actualizar o evento do botão “Ok” para que quando carregado actualize a informação na janela principal. Assim escrevi o seguinte código:

```

...

self.btn_cancel=wx.Button(panel,-1,'Cancelar',(160,70),(100,30))

wx.EVT_BUTTON(self,self.btn_cancel.GetId(),self.cancel)

def ok(self,event):

    a=self.nome_aluno.GetValue()

    frame.text1.SetLabel(a)

    frame.text1.SetForegroundColour('black')

    b=self.curso.GetValue()

    frame.text2.SetLabel(b)

    frame.text2.SetForegroundColour('black')

    self.Destroy()

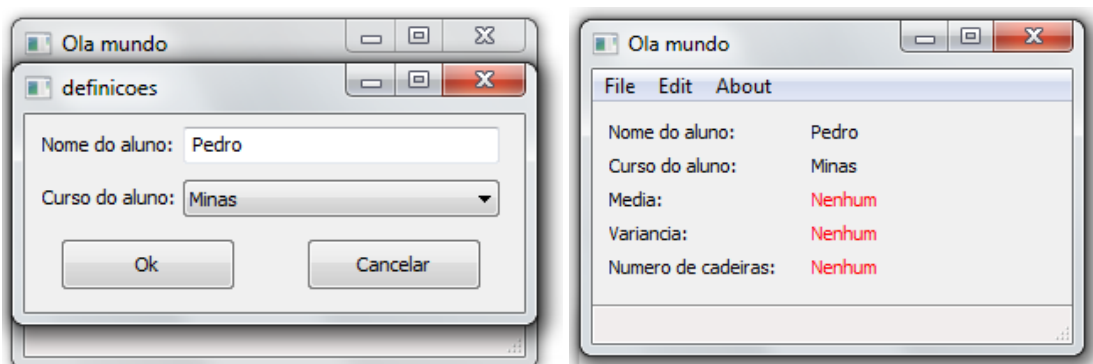
def cancel(self,event):

    self.Destroy()

...

```

À semelhança do TextCtrl também para a ComboBox utilizamos o comando “GetValue()” para obter o valor que está na mesma. O resto já deverás perceber o que é, primeiro actualizo a string que está na janela principal com relação ao nome do curso, depois dou a cor de preto a esse texto estático.



Agora vamos criar mais uma janela secundária para inserir mais algumas informações que ainda não estão disponíveis na janela principal.

Janelas Secundárias (revisões)

É um pouco esquisito eu estar a dar um título para fazer algo que já expliquei antes mas assim aproveito para rever a criação de uma janela secundária bem como criar objectos na primeira (um deles que ainda não usamos), e mais dois objectos inéditos na nova janela secundária. Primeiro que tudo vou criar um novo tópico no menu “Edit” e um evento associado ao mesmo. Para isso tenho que voltar a fazer código na classe da minha janela principal, a “MinhaFrame”.

```
...

menu.Append(menu_edit, "Edit")

menu_edit.Append(3, "Definitions", "Definicoes do aluno.")

self.Bind(wx.EVT_MENU, self.definicoes, id=3)

menu_edit.Append(4, "Atribuicoes", "Atribuicoes feitas pelo utilizador.")

self.Bind(wx.EVT_MENU, self.atrib, id=4)


menu_about=wx.Menu()

menu.Append(menu_about, "About")

menu_about.Append(10, "Acerca..", "Acerca do programas tutorial...")

...
```

Criei um tópico chamado “Atribuicoes” dentro do menu “Edit” e fiz um evento para uma função chamada “atrib” que a seguir mostro o código:

```
...

menu_about.Append(10, "Acerca..", "Acerca do programas tutorial...")

self.Bind(wx.EVT_MENU, self.acerca, id=10)


def atrib(self, event):

    aframe=AtribFrame(parent=frame, id=997)

    aframe.Centre()

    aframe.Show()


def definicoes(self, event):

    dframe=DefFrame(parent=frame, id=998)

    dframe.Centre()

    dframe.Show()
```

...

Mais uma vez já vimos isto antes. A função começa por chamar uma nova frame (AtribFrame, da qual já mostro o código), centra-la e depois dar a sua visualização ao utilizador. Vamos ver agora como ficou a nova classe escrita:

```
import wx
```

```
import numpy
```

```
class AtribFrame(wx.Frame):
```

```
    def __init__(self, parent, id):
```

```
        wx.Frame.__init__(self, parent, id, 'Atribuicoes', size = (300, 150))
```

```
        panel=wx.Panel(self)
```

```
        panel.SetBackgroundColour('Ligth_Grey')
```

```
class DefFrame(wx.Frame):
```

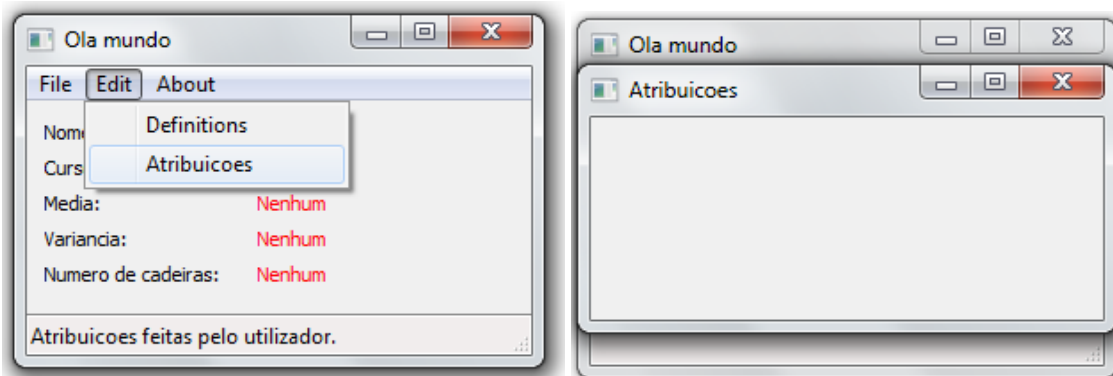
```
    def __init__(self, parent, id):
```

```
        wx.Frame.__init__(self, parent, id, 'definicoes', size = (300, 150))
```

```
        panel=wx.Panel(self)
```

...

É igual à janela da classe DefFrame, simplesmente mudei o título de “definicoes” para “Atribuicoes”. O resultado deverá ser óbvio:



Agora na minha nova janela vou acrescentar os botões de “Ok” e “Cancel”.

```

tutorial.py*
1  # -*- coding: utf-8 -*-
2  """
3  Created on Sat Aug 28 01:35:30 2010
4
5  @author: Desktop
6  """
7  import wx
8  import numpy
9
10 class AtribFrame(wx.Frame):
11     def __init__(self, parent, id):
12         wx.Frame.__init__(self, parent, id, 'Atribuicoes', size = (300, 150))
13         panel=wx.Panel(self)
14         panel.SetBackgroundColour('Ligth_Grey')
15
16         self.btn_ok=wx.Button(panel,-1,'Ok',(20,70),(100,30))
17         #wx.EVT_BUTTON(self,self.btn_ok.GetId(),self.ok)
18         self.btn_cancel=wx.Button(panel,-1,'Cancelar',(160,70),(100,30))
19         #wx.EVT_BUTTON(self,self.btn_cancel.GetId(),self.cancel)
20
21
22 class DefFrame(wx.Frame):
23     def __init__(self, parent, id):
24         wx.Frame.__init__(self, parent, id, 'definicoes', size = (300, 150))
25         panel=wx.Panel(self)
26         panel.SetBackgroundColour('Ligth_Grey')
27

```

Mostro esta figura também para que vejas como está escrito no código em Spyder porque inseri também os eventos mas utilizei o símbolo “#” para os comentar (e comentando o Python já não vai ler essas linhas de código), caso contrário o programa não funcionaria porque não podes criar eventos sem as funções correspondentes. Transcrevendo para aqui:

```

...

class AtribFrame(wx.Frame):

    def __init__(self, parent, id):

        wx.Frame.__init__(self, parent, id, 'Atribuicoes', size = (300, 150))

        panel=wx.Panel(self)

        panel.SetBackgroundColour('Ligth_Grey')

        self.btn_ok=wx.Button(panel,-1,'Ok',(20,70),(100,30))

        #wx.EVT_BUTTON(self,self.btn_ok.GetId(),self.ok)

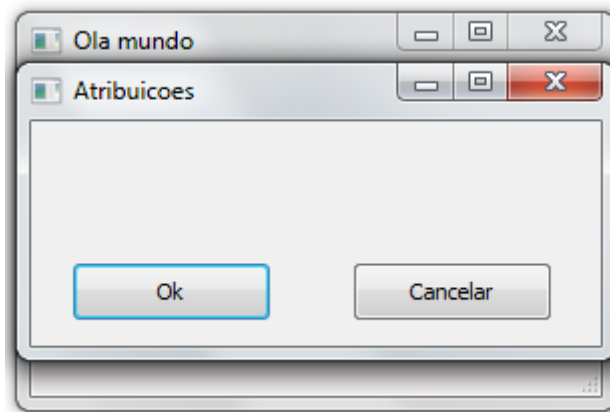
        self.btn_cancel=wx.Button(panel,-1,'Cancelar',(160,70),(100,30))

        #wx.EVT_BUTTON(self,self.btn_cancel.GetId(),self.cancel)

...

```

O aspecto da janela como deves calcular é este:



Agora só temos que preencher o conteúdo desta janela mas primeiro quero preparar a janela principal para receber a informação que vem desta e com isso aproveito para introduzir um objecto chamado StaticBox.

StaticBox

Uma StaticBox é um objecto que serve na esmagadora maioria dos casos para arrumar informação visual para o utilizador. Inseri o seguinte código na classe da janela principal:

```
...

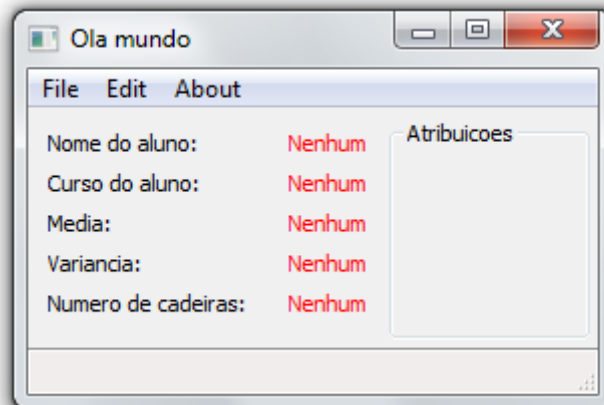
self.text4=wx.StaticText(self.panel,-1,"Nenhum",(130,70))
self.text4.SetForegroundColour('red')
self.text5=wx.StaticText(self.panel,-1,"Nenhum",(130,90))
self.text5.SetForegroundColour('red')

wx.StaticBox(self.panel,-1,'Atribuicoes',(180,5),(100,110))

menu=wx.MenuBar()
self.SetMenuBar(menu)
status=self.CreateStatusBar()

...
```

O comando é o “wx.StaticBox” com primeiro argumento o sítio onde se encontra (o painel self.panel), segundo o id (-1), terceiro o título que deve aparecer na StaticBox, quarto a posição e quinto o tamanho. O aspecto é este:



No nosso caso esta caixa vai servir para facilitar ao utilizador os dados que vêm de fora e os que o próprio utilizador tem de decidir. Vou inserir uns textos estáticos que dizem respeito ao número do processo e se o aluno já acabou o curso ou não.

...

```
self.text4.SetForegroundColour('red')

self.text5=wx.StaticText(self.panel,-1,"Nenhum",(130,90))

self.text5.SetForegroundColour('red')

wx.StaticBox(self.panel,-1,'Atribuicoes',(180,5),(100,110))

wx.StaticText(self.panel,-1,"Processo:",(200,30))

wx.StaticText(self.panel,-1,"Terminado:",(200,70))

self.text6=wx.StaticText(self.panel,-1,"Nenhum",(200,45))

self.text7=wx.StaticText(self.panel,-1,"Desconhecido",(200,85))

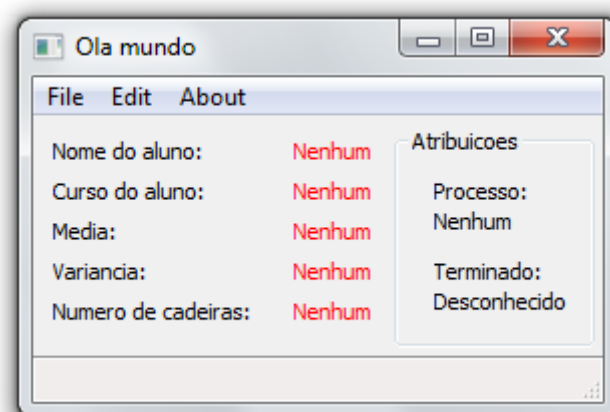
menu=wx.MenuBar()

self.SetMenuBar(menu)

status=self.CreateStatusBar()
```

...

O resultado disto é o seguinte:



O processo é relativo a um número por isso vou introduzir o objecto SpinCtrl que vamos inserir na nossa janela de atribuições e por isso o código será na classe AtribFrame.

SpinCtrl

O SpinCtrl é uma espécie de ComboBox mas de onde podes escolher números. Na classe AtribFrame introduzi o seguinte código:

```
...

panel=wx.Panel(self)

panel.SetBackgroundColour('Ligth_Grey')

wx.StaticText(panel, -1, "Processo:", (20,10))

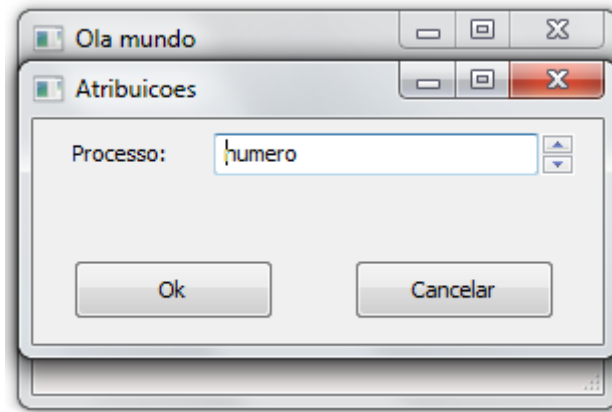
self.spin_num=wx.SpinCtrl(panel, -1, "numero", (90,7), (180, -1), min=1, max=500, initial=1)

self.btn_ok=wx.Button(panel, -1, 'Ok', (20,70), (100,30))

#wx.EVT_BUTTON(self, self.btn_ok.GetId(), self.ok)

...
```

Inseri um texto estático para indicar que ali é preciso inserir o número do processo. O comando para fazer um SpinCtrl é “wx.SpinCtrl” onde o primeiro argumento é o local onde se encontra, o segundo o id, o terceiro uma string que vai aparecer no SpinCtrl enquanto o utilizador não o utilizar, o quarto a posição, quinto o tamanho, sexto o número mínimo que pode ser escolhido, sétimo o número máximo, e oitavo o inicial. Se fores à janela o aspecto inicial é este:



Agora já podes escolher o número do processo, temos é que fazer o evento correspondente ao botão Ok (e já agora fazemos também o evento para o botão cancelar também). Acrescentei o seguinte código:

```
...

wx.StaticText(panel,-1,"Processo:",(20,10))

self.spin_num=wx.SpinCtrl(panel,-1,"numero",(90,7),(180,-1),min=1,max=500,initial=1)


self.btn_ok=wx.Button(panel,-1,'Ok',(20,70),(100,30))
wx.EVT_BUTTON(self,self.btn_ok.GetId(),self.ok)
self.btn_cancel=wx.Button(panel,-1,'Cancelar',(160,70),(100,30))
wx.EVT_BUTTON(self,self.btn_cancel.GetId(),self.cancel)


def ok(self,event):
    a=self.spin_num.GetValue()
    frame.text6.SetLabel(repr(a))
    self.Destroy()


def cancel(self,event):
    self.Destroy()

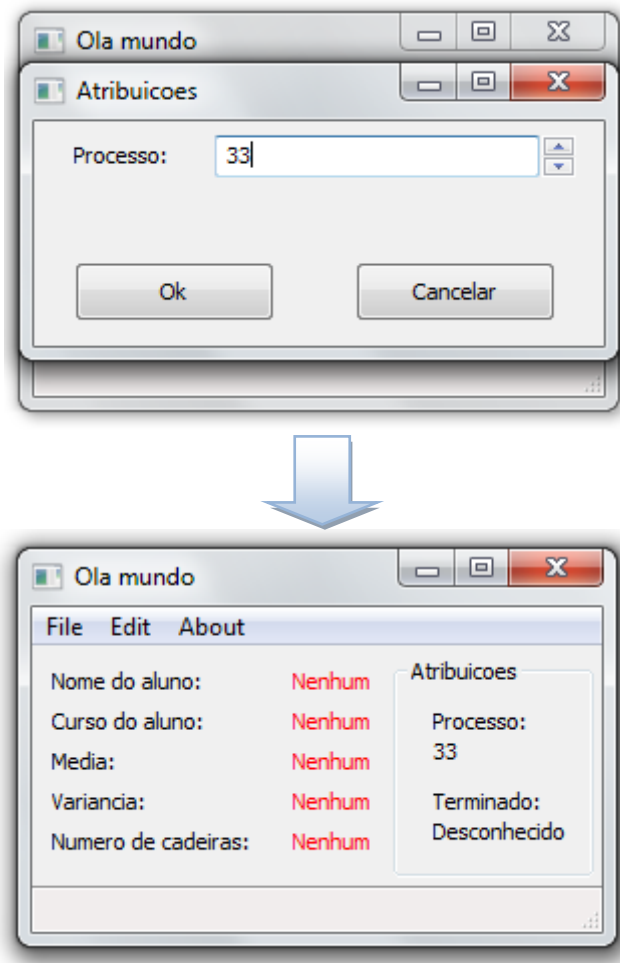

class DefFrame(wx.Frame):

    def __init__(self, parent, id):
        wx.Frame.__init__(self, parent, id, 'definicoes', size = (300, 150))
```

```
panel=wx.Panel(self)
```

...

Agora se carregares no botão “Ok” o texto estático da janela principal vai ser alterado para dar o número do processo. À semelhança da ComboBox e TextCtrl o comando para receber o valor do SpinCtrl é “GetValue()” e como isto é um número quando defini o novo texto para o texto da janela principal tive que utilizar a função “repr” para passar de número para string. O resultado é o seguinte:



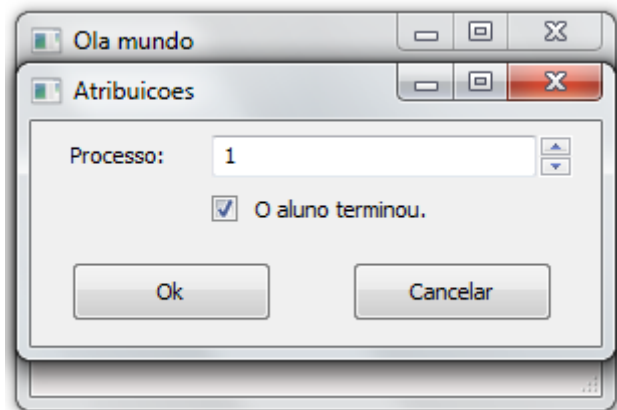
Agora para o status de terminado ou não quero que apareça lá sim ou não. Podíamos fazer isto com uma ComboBox mas quero introduzir um novo objecto que é o CheckBox.

CheckBox

Uma CheckBox é uma espécie de interruptor, ou está ligado ou está desligado. Para inserir a CheckBox na minha janela de atribuições inseri o seguinte código:

```
...  
  
wx.StaticText(panel,-1,"Processo:",(20,10))  
  
self.spin_num=wx.SpinCtrl(panel,-1,"numero",(90,7),(180,-1),min=1,max=500,initial=1)  
  
self.check_sim=wx.CheckBox(panel,-1,' O aluno terminou.',(90,37))  
  
  
self.btn_ok=wx.Button(panel,-1,'Ok',(20,70),(100,30))  
  
wx.EVT_BUTTON(self,self.btn_ok.GetId(),self.ok)  
  
...
```

O aspecto visual é este:



Vamos agora ver o que fazer quando o botão “Ok” é carregado. Inseri o seguinte código na função do evento de carregar no botão de “Ok”:

```
...  
  
self.btn_cancel=wx.Button(panel,-1,'Cancelar',(160,70),(100,30))  
  
wx.EVT_BUTTON(self,self.btn_cancel.GetId(),self.cancel)  
  
  
def ok(self,event):  
  
    a=self.spin_num.GetValue()  
  
    frame.text6.SetLabel(repr(a))
```

```

if self.check_sim.GetValue():

    frame.text7.SetLabel('Sim')

else:

    frame.text7.SetLabel('Nao')

self.Destroy()

def cancel(self,event):

    self.Destroy()

...

```

O comando para retirar o valor do CheckBox continua a ser o “GetValue()” mas há uma diferença. O TextCtrl e a ComboBox recebem uma string, o SpinCtrl recebe um número, o CheckCtrl recebe uma resposta de verdadeiro ou falso, melhor dizendo ligado ou desligado para o Python estes valores são True e False e chama-se a isto um valor booleano porque só pode ser uma de duas alternativas). Assim ao fazer:

```

if self.check_sim.GetValue():

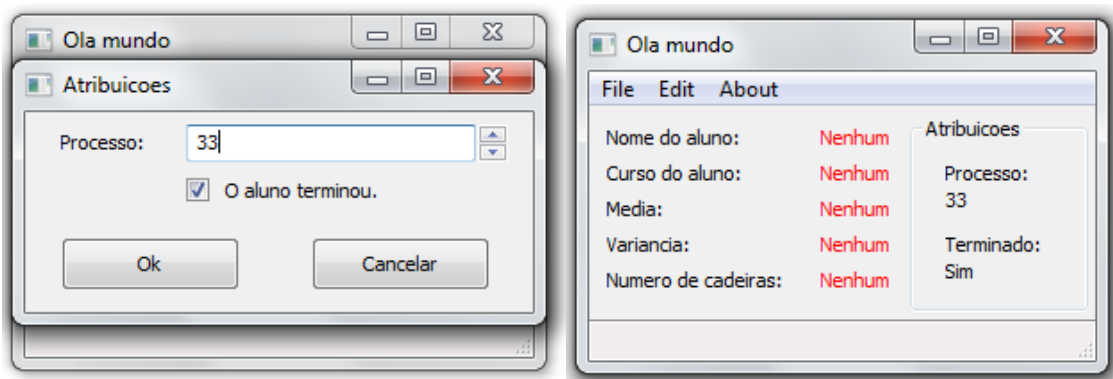
    frame.text7.SetLabel('Sim')

else:

    frame.text7.SetLabel('Nao')

```

Estamos a perguntar se a CheckBox está ligada. Se estiver então altera o texto estático da janela principal para sim, caso contrário altera para não. O resultado é este:



Matplotlib no wxPython (parte 1)

Já vimos como mexer no matplotlib e já vimos como mexer no wxPython vamos agora utilizar os dois pela primeira vez. O que eu quero fazer é dar a possibilidade do utilizador ver um histograma das notas das cadeiras do aluno (aquelas que é suposto receber a partir do ficheiro). Com isto vamos fazer uma primeira aproximação à gestão de erros porque só podes fazer o gráfico se o utilizador tiver carregado as notas dos alunos, caso contrário o programa tem de saber reagir para não dar erro. Para fazer isto vou fazer mais um menu na barra de menus da janela principal chamado “Graficos” com um tópico lá dentro chamado “Histograma” que vai chamar uma nova janela onde podemos escolher o número de classes que queremos no nosso histograma. Para isso vamos ter que criar uma nova classe onde possamos escolher o número de classes. Para além disto tudo lembra-te que é preciso importar o matplotlib agora que o vamos utilizar. Então começando exactamente por aqui:

```
import wx

import numpy

import matplotlib.pyplot
```

...

Importei o matplotlib.pyplot para além do wx e numpy que já estávamos a utilizar anteriormente. Agora preciso que te recordes do que fizemos quando criamos a função para carregar o ficheiro:

...

```
def abrir(self,event):

    b=wx.FileDialog(self,"Abrir ficheiro de dados...",style=wx.OPEN,wildcard="*.*.")

    if b.ShowModal() == wx.ID_OK:

        self.filename=b.GetPath()

        self.file=numpy.loadtxt(self.filename)          # Está aqui a self.file

        numero1=repr(numpy.mean(self.file))

        self.text3.SetLabel(numero1)
```

...

Nós, ao carregarmos o ficheiro estamos a criar uma nova variável self.file na classe MinhaFrame que ante não existia. Para fazer isto (pelo método que vou ensinar pelo menos) temos de criar a variável self.file antes. Para isso fui ao código da MinhaFrame e inseri:

```

...
class MinhaFrame(wx.Frame):
    def __init__(self, parent, id):
        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (300, 200))
        self.panel=wx.Panel(self)
        self.panel.SetBackgroundColour('Ligth_Grey')

        self.file='None'

        wx.StaticText(self.panel,-1,"Nome do aluno:",(10,10))
        wx.StaticText(self.panel,-1,"Curso do aluno:",(10,30))
        ...

```

A única coisa para que isto serve é para criarmos a nossa variável quando abrirmos o programa e não quando carregamos o ficheiro. Já vais perceber para quê tudo isto. Por agora vou criar o meu novo menu e respectivo tópico e dele fazer um evento para a função “hist” a qual a seguir dou o código:

```

...
menu_edit.Append(4,"Atribuicoes","Atribuicoes feitas pelo utilizador.")
self.Bind(wx.EVT_MENU,self.atrib,id=4)

menu_graf=wx.Menu()
menu.Append(menu_graf,"Graficos")
menu_graf.Append(5,"Histograma","Faz o histograma das notas do aluno.")
self.Bind(wx.EVT_MENU,self.hist,id=5)

menu_about=wx.Menu()
menu.Append(menu_about,"About")
...

```

Agora que já temos o menu vamos lá analisar a função “hist”:

```

...

menu_about.Append(10,"Acerca..","Acerca do programas tutorial...")

self.Bind(wx.EVT_MENU,self.acerca,id=10)


def hist(self,event):

    if self.file=='None':

        wx.MessageBox("Nao foram carregadas notas.,"Erro")

    else:

        hframe=HistFrame(parent=frame,id=996)

        hframe.Centre()

        hframe.Show()


def atrib(self,event):

    aframe=AtribFrame(parent=frame,id=997)

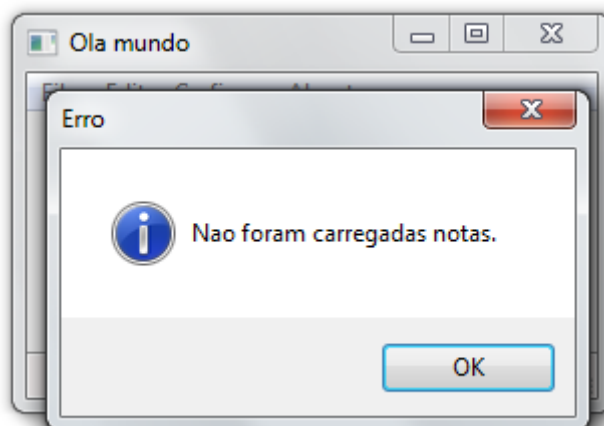
    aframe.Centre()

    aframe.Show()

...

```

Repara que a primeira coisa que a função faz é perguntar se a variável `self.file` é igual a 'None'. Se for então nenhum ficheiro foi carregado e por isso não pode ser feito o histograma (se não existisse a variável `self.file` então esta pergunta não poderia ser feita e assim conseguimos contornar este erro). Não havendo ficheiro o programa vai lançar uma `MessageBox` (esta ao contrário da que já fizemos para a saída do programa só tem botão de Ok):



Se, por outro lado, já tiver sido carregado um ficheiro então a variável `self.file` vai ser diferente de 'None' (será a matriz do próprio ficheiro) e nesse caso as instruções dizem para chamar a classe `HistFrame` que agora mostro o código:

```

...

import matplotlib.pyplot

class HistFrame(wx.Frame):
    def __init__(self, parent, id):
        wx.Frame.__init__(self, parent, id, 'Histograma', size = (300, 150))
        panel=wx.Panel(self)
        panel.SetBackgroundColour('Ligth_Grey')

        wx.StaticText(panel,-1,"Classes:",(20,30))
        self.spin_num=wx.SpinCtrl(panel,-1,"numero",(90,27),(180,-1),min=1,max=20,initial=1)

        self.btn_ok=wx.Button(panel,-1,'Ok',(20,70),(100,30))
        wx.EVT_BUTTON(self,self.btn_ok.GetId(),self.ok)
        self.btn_cancel=wx.Button(panel,-1,'Cancelar',(160,70),(100,30))
        wx.EVT_BUTTON(self,self.btn_cancel.GetId(),self.cancel)

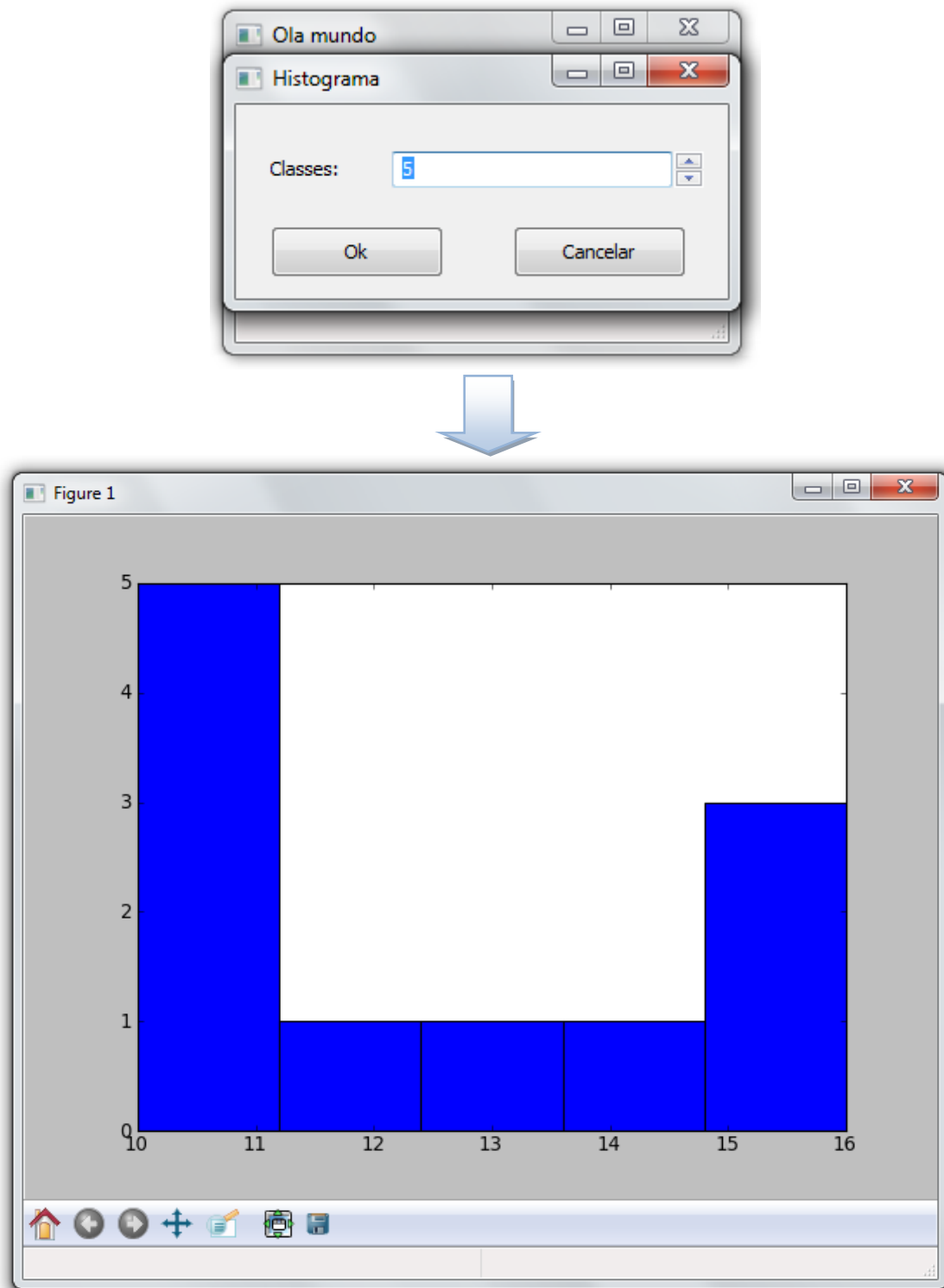
    def ok(self,event):
        a=self.spin_num.GetValue()
        matplotlib.pyplot.hist(frame.file,a)
        matplotlib.pyplot.show()
        self.Destroy()

    def cancel(self,event):
        self.Destroy()

class AtribFrame(wx.Frame):
    def __init__(self, parent, id):
        wx.Frame.__init__(self, parent, id, 'Atribuicoes', size = (300, 150))
        panel=wx.Panel(self)
        ...

```

À semelhança das janelas secundárias anteriores inseri o botão de cancelar (que vai fechar janela). O botão de Ok vai buscar o valor inserido no SpinCtrl e depois vai plotar um histograma no qual o primeiro argumento é a série de dados atribuída à variável `frame.file` (agora já não pode ser `self.file` porque estamos a escrever código numa classe que não é a `MinhaFrame`) e o segundo argumento o número de classes. De resto apenas tenho que dizer para o gráfico ser visível (**repara que o “show” do matplotlib começa com letra pequena ao contrário do do wxPython que começa com letra grande**). O resultado disto é:



Trabalhar com ficheiros

Agora quero que seja possível ao utilizador salvar o processo contendo no ficheiro um cabeçalho que contenha o número do processo, depois se terminou ou não, o nome do aluno e do curso e finalmente as suas notas. Para isto vamos utilizar o `FileDialog` novamente mas vamos precisar de aprender a mexer com ficheiros de maneira mais completa visto que a função que aprendemos até agora para salvar é o `"numpy.savetxt"` que apenas serve para salvar matrizes e vectores e por isso não conseguiríamos fazer um cabeçalho (nós vamos, mesmo assim, usar esta função mas misturado com outro método do Python para mexer em ficheiros). Primeiro que tudo é preciso criar o tópico para salvar o processo e o evento associado:

```
...

menu_file=wx.Menu()

menu.Append(menu_file,"File")

menu_file.Append(2,"Load", "Carregar dados de ficheiros.")

self.Bind(wx.EVT_MENU,self.abrir,id=2)

menu_file.Append(6,"Save","Salvar o processo do aluno.")

self.Bind(wx.EVT_MENU,self.salvar,id=6)

menu_file.AppendSeparator()

menu_file.Append(1,"Exit","Sair do programa")

self.Bind(wx.EVT_MENU,self.sair,id=1)

...
```

Criei um tópico da mesma maneira que já tinha feito anteriormente portanto aqui não deverá haver surpresas. Repara que o evento associado chama a função `"salvar"` que ainda não transcrevi para aqui. Segue agora:

```
...

def salvar(self,event):

    if self.file=='None':

        wx.MessageBox("Nao foram carregadas notas.,"Erro")

    else:

        b=wx.FileDialog(self,"Salvar ficheiro de dados...",style=wx.SAVE,wildcard="*.txt")

        if b.ShowModal() == wx.ID_OK:

            self.filename2=b.GetPath()

            fid = open(self.filename2,'w')
```



```

processo=self.text6.GetLabel()

terminou=self.text7.GetLabel()

nome=self.text1.GetLabel()

curso=self.text2.GetLabel()

media=self.text3.GetLabel()

variancia=self.text4.GetLabel()

cadeiras=self.text5.GetLabel()

fid.write('Processo numero: '+processo+'\n')

fid.write('Terminou: '+terminou+'\n')

fid.write('*****\n')

fid.write('Nome do aluno: '+nome+'\n')

fid.write('Curso do aluno: '+curso+'\n')

fid.write('*****\n')

fid.write('Media do curso: '+media+'\n')

fid.write('Variancia do curso: '+variancia+'\n')

fid.write('Numero de cadeiras: '+cadeiras+'\n')

fid.write('*****\n')

numpy.savetxt(fid,self.file,fmt='%5.1f')

b.Close()

```

...

Este é o código total que inseri para fazer a função salvar. Algumas operações já deverás saber para que servem, outras estás a ver primeira vez. Para o primeiro e segundo caso vamos analisar o código parte a parte.

```
def salvar(self,event):
```

```
    if self.file=='None':
```

```
        wx.MessageBox("Nao foram carregadas notas.", "Erro")
```

```
    else:
```

```
        b=wx.FileDialog(self,"Salvar ficheiro de dados...",style=wx.SAVE,wildcard="*.txt")

```

Fiz a função salvar e inseri uma pergunta “if” por questões de gestão de erros porque eu quero que o utilizador apenas possa salvar o processo se tiver carregado as notas do aluno para dentro do programa. Se não tiver a variável “self.file” vai ser ‘None’ e por isso o programa manda uma MessageBox ao utilizador a avisar que não foram carregados dados, caso contrário chama uma FileDialog igual à primeira que fizemos para carregar os dados mas mudei o título de “Abrir ficheiro de dados” para “Salvar ficheiro de dados” e mudei o estilo de “wx.OPEN”

para “wx.SAVE” (porque a primeira serve para escolher um ficheiro e a segunda para criar um completamente novo ou para gravar por cima de um que já exista). Deixei o wildcard como “*.*” para que o utilizador possa escolher a extensão que bem entender. A seguir vem:

```
if b.ShowModal() == wx.ID_OK:

    self.filename2=b.GetPath()

    fid = open(self.filename2,'w')
```

Mais uma vez utilizei o comando “ShowModal()” para fazer a validação do ficheiro que foi feito ou escolhido pelo utilizador no FileDialog. Se tudo correu bem é feita a atribuição do caminho do ficheiro para a variável “self.filename2” utilizando o comando “GetPath()”. E é agora que temos uma novidade, o comando “open”. As funções que já vimos para mexer em ficheiros como o “loadtxt” e “savetxt” são óptimas funções para o fim a que se destinam que é gravar e carregar matrizes mas são pouco flexíveis porque se limitam a executar essa função. Por esse motivo quando pretendemos “adornar” o nosso ficheiro quer na leitura quer na escrita precisamos de algo mais abrangente. É o caso da função “open”. O primeiro argumento é o caminho (dado pela variável “self.filename2”), o segundo é o modo que nosso caso é ‘w’ de ‘write’ ou escrita. Se estivéssemos a ler o ficheiro teria que estar lá um ‘r’ de ‘read’ ou leitura. Vamos ver o que acontece a seguir.

```
processo=self.text6.GetLabel()

terminou=self.text7.GetLabel()

nome=self.text1.GetLabel()

curso=self.text2.GetLabel()

media=self.text3.GetLabel()

variancia=self.text4.GetLabel()

cadeiras=self.text5.GetLabel()
```

Eu pretendo salvar as informações de nome de aluno, curso, média, etc., para um ficheiro e o único sítio onde essas informações estão gravadas (ao contrário das notas das cadeiras que estão numa variável específica, “frame.file”) é nos textos estáticos da janela principal. Nós já vimos como mudar o seu conteúdo (“SetLabel”) mas para saber o que está lá precisamos de usar o comando “GetLabel()”. Assim as várias variáveis a que eu estou a atribuir nesta função têm nos seus conteúdos strings da informação que está na janela principal. Agora já posso usar estas variáveis para escrever no ficheiro. Para escrever vem este código a seguir:

```
fid.write('Processo numero: '+processo+'\n')

fid.write('Terminou: '+terminou+'\n')

fid.write('*****\n')
```

Com a variável “fid” (à qual está atribuída a abertura do ficheiro) executo o comando “write” para escrever uma string numa dada linha. Neste código comecei por escrever ‘Processo numero: ‘ e logo a seguir vem a string atribuída à variável “processo”. Repara que logo a seguir

a isso inseri também a string '\n' que se te recordares serve para mudar de linha (se não o fizesse estaríamos sempre a escrever na mesma linha). Depois faço o mesmo para o estado do curso do aluno (terminado ou não) e depois insiro uma string de "*" para criar uma divisória no ficheiro para ser mais fácil a sua leitura. O resto é muito semelhante (nota que estou sempre a meter o '\n' para ir sempre mudando de linha):

```
fid.write('*****\n')

fid.write('Nome do aluno: '+nome+'\n')

fid.write('Curso do aluno: '+curso+'\n')

fid.write('*****\n')

fid.write('Media do curso: '+media+'\n')

fid.write('Variancia do curso: '+variancia+'\n')

fid.write('Numero de cadeiras: '+cadeiras+'\n')

fid.write('*****\n')
```

Agora o que vem a seguir é que é uma mistura entre o que aprendemos antes e depois.

```
numpy.savetxt(fid,self.file,fmt='%5.1f')

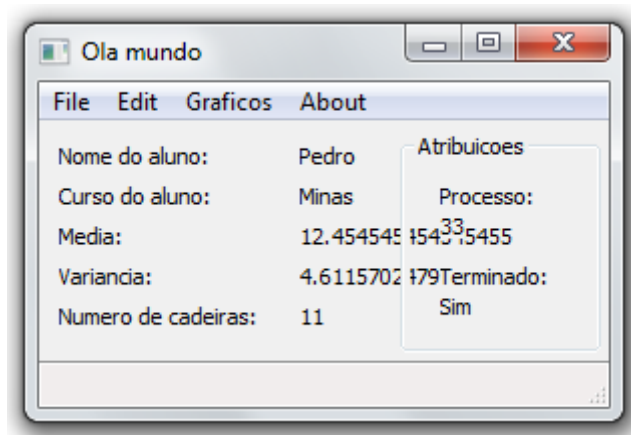
b.Close()
```

Utilizei o "numpy.savetxt" para salvar as notas das cadeiras e repara que como primeiro argumento onde deveria esta o caminho do ficheiro ("self.filename2") está a variável "fid" à qual foi atribuída a abertura do ficheiro. Isto é importante porque desta maneira estou a dizer ao "numpy.savetxt" para gravar a partir da última linha que fiz com o "fid":

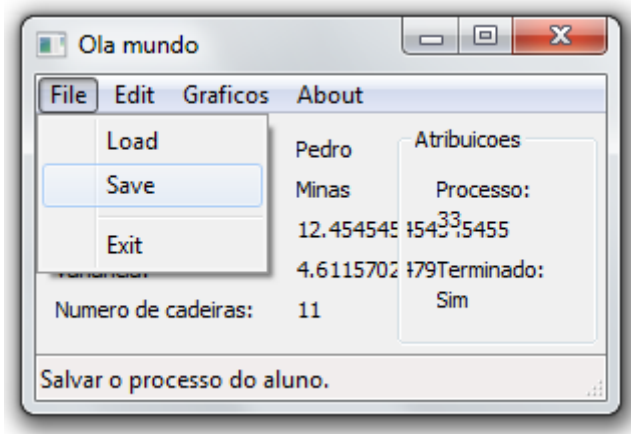
```
fid.write('*****\n')
```

Se tivesse inserido o "numpy.savetxt" com a variável "self.filename2" como argumento do caminho estaria a reescrever tudo por cima e ou ficaria com o cabeçalho em cima dos primeiros elementos da matriz ou apagaria o cabeçalho com os primeiros elementos da matriz. O segundo argumento é, naturalmente, a variável de dados que vamos salvar ("self.file") e o terceiro é novo. Se bem te lembras a função "numpy.savetxt" grava com notação científica e eu quero gravar com notação..., bem a normal que nem sei se tem um nome específico. Para isso metemos um terceiro argumento que é o argumento de formatação "fmt" que como podes ver é uma string, '%5.1f'. O "%" metes sempre, o "5" é o número de caracteres que o número que estás a gravar vai ter, depois separado pelo ponto está o número de casas decimais que pretendes e finalmente o tipo de número que queres gravar que neste caso escolhi ser "f" de "float" ou decimais (podia tê-lo feito para inteiros, complexos, etc...). Como sabia que as notas só iriam de 0 a 20 e no máximo teriam uma casa decimal escolhi ter 5 caracteres para haver um espaço no início de cada número do ficheiro ('12345' = ' 13.0' porque o "." também é um carácter, se tivesse escolhido 10 para o número de caracteres teria algo assim ' 13.0') e um número decimal. O comando final é para fechar o FileDialog:

```
b.Close()
```



Fui ao programa inseri os dados todos inclusive carregar as notas a partir de um ficheiro (sim eu sei, os número que estão a aparecer estão muita foleiros mas uma coisa de cada vez) e agora quero salvar o processo para um ficheiro. Vou ao tópico para salvar:



Escolho o nome do ficheiro, salvo-o e depois fui ver o aspecto dele:

Processo numero: 33

Terminou: Sim

Nome do aluno: Pedro

Curso do aluno: Minas

Media do curso: 12.454545454545455

Variância do curso: 4.6115702479338854

Numero de cadeiras: 11

10.0

11.0

```
15.0
12.0
16.0
13.0
10.0
10.0
11.0
15.0
14.0
```

Catita. Está feito com sucesso o código para salvar o processo para ficheiro.

Detalhes finais do programa (formatar strings)

Parece que temos um problema de visualização porque quando a média e variância aparecem na janela principal têm um precisão tão grande que a sua string vai para cima da informação que está ao lado. Em teoria vocês já têm conhecimentos para resolver isto considerando o último capítulo no qual mexemos com o formato de uma string e o capítulo Strings onde aprendemos a mexer com as strings em si. Lembras-te disto:

```
>>> print 'um numero: %f'%a
um numero: 33.000000
```

O que nos estamos a fazer aqui é usar o “%f” para introduzir um número decimal na string mas com o que já aprendemos com formatação de strings conseguimos melhor que isto. Experimenta a consola:

```
>>> a=33
>>> print 'Um numero: %0.1f'%a
Um numero: 33.0
>>>
```

Comecei por atribuir o número 33 a “a” e depois inseri-o numa string com precisão decimal de 1 (meti zero no número de caracteres porque ele irá inserir o número na mesma, só se meteres a mais é que ele começa a meter espaços). Se o nosso número for decimal então:

```
>>> a=33.33333333333333
>>> print 'Um numero: %0.1f'%a
Um numero: 33.3
>>>
```

A ideia é esta. No nosso programa, em vez de estarmos a utilizar o comando “repr” vamos fazer uma string com formatação. Assim no código da função “abrir” da MinhaFrame, onde começa todo este problema passei disto:

```
...
def abrir(self,event):

    b=wx.FileDialog(self,"Abrir ficheiro de dados...",style=wx.OPEN,wildcard="*.*.")

    if b.ShowModal() == wx.ID_OK:

        self.filename=b.GetPath()

        self.file=numpy.loadtxt(self.filename)

        numero1=repr(numpy.mean(self.file))

        self.text3.SetLabel(numero1)

        numero2=repr(numpy.var(self.file))

        self.text4.SetLabel(numero2)

    ...
```

Para isto:

```
...
def abrir(self,event):

    b=wx.FileDialog(self,"Abrir ficheiro de dados...",style=wx.OPEN,wildcard="*.*.")

    if b.ShowModal() == wx.ID_OK:

        self.filename=b.GetPath()

        self.file=numpy.loadtxt(self.filename)

        numero1="%1f"%numpy.mean(self.file)

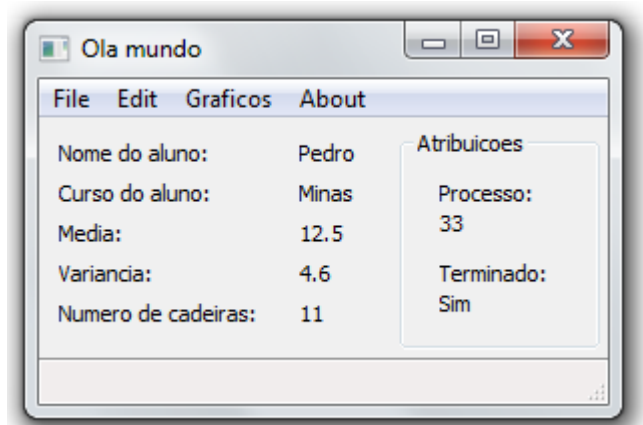
        self.text3.SetLabel(numero1)

        numero2="%1f"%numpy.var(self.file)

        self.text4.SetLabel(numero2)

    ...
```

E o problema ficou resolvido:



Código final

Parece-me boa altura para te mostrar o código final do programa. Ficou com cerca de duzentas e tais linhas de código e é perfeitamente funcional (se bem que não propriamente útil para ninguém mas tinha que arranjar algo que servisse para mostrar o essencial e ao mesmo tempo exemplificar como se faz um programa normalmente, se quiseses culpa a minha falta de originalidade). Aqui vai:

```
import wx

import numpy

import matplotlib.pyplot

class HistFrame(wx.Frame):

    def __init__(self, parent, id):

        wx.Frame.__init__(self, parent, id, 'Histograma', size = (300, 150))

        panel=wx.Panel(self)

        panel.SetBackgroundColour('Ligth_Grey')

        wx.StaticText(panel, -1, "Classes:", (20,30))

        self.spin_num=wx.SpinCtrl(panel, -1, "numero", (90,27), (180, -1), min=1, max=20, initial=1)

        self.btn_ok=wx.Button(panel, -1, 'Ok', (20,70), (100,30))

        wx.EVT_BUTTON(self, self.btn_ok.GetId(), self.ok)

        self.btn_cancel=wx.Button(panel, -1, 'Cancelar', (160,70), (100,30))

        wx.EVT_BUTTON(self, self.btn_cancel.GetId(), self.cancel)

    def ok(self, event):

        a=self.spin_num.GetValue()

        matplotlib.pyplot.hist(frame.file, a)

        matplotlib.pyplot.show()

        self.Destroy()

    def cancel(self, event):

        self.Destroy()
```

```

class AtribFrame(wx.Frame):
    def __init__(self, parent, id):
        wx.Frame.__init__(self, parent, id, 'Atribuicoes', size = (300, 150))
        panel=wx.Panel(self)
        panel.SetBackgroundColour('Ligth_Grey')

        wx.StaticText(panel, -1, "Processo:", (20,10))
        self.spin_num=wx.SpinCtrl(panel, -1, "numero", (90,7), (180,-1), min=1, max=500, initial=1)
        self.check_sim=wx.CheckBox(panel, -1, ' 0 aluno terminou.', (90,37))

        self.btn_ok=wx.Button(panel, -1, 'Ok', (20,70), (100,30))
        wx.EVT_BUTTON(self, self.btn_ok.GetId(), self.ok)
        self.btn_cancel=wx.Button(panel, -1, 'Cancelar', (160,70), (100,30))
        wx.EVT_BUTTON(self, self.btn_cancel.GetId(), self.cancel)

    def ok(self, event):
        a=self.spin_num.GetValue()
        frame.text6.SetLabel(repr(a))
        if self.check_sim.GetValue():
            frame.text7.SetLabel('Sim')
        else:
            frame.text7.SetLabel('Nao')
        self.Destroy()

    def cancel(self, event):
        self.Destroy()

class DefFrame(wx.Frame):
    def __init__(self, parent, id):

```



```

wx.Frame.__init__(self, parent, id, 'definicoes', size = (300, 150))

panel=wx.Panel(self)

panel.SetBackgroundColour('Ligth_Grey')


wx.StaticText(panel,-1,"Nome do aluno:",(10,10))

self.nome_aluno=wx.TextCtrl(panel,-1,'Nenhum',(90,7),(180,-1))


wx.StaticText(panel,-1,"Curso do aluno:",(10,40))

cursos=['Ambiente','Biomedica','Civil','Fisica','Minas','Mecanica']

self.curso=wx.ComboBox(panel,-1,value=cursos[0],pos=(90,37),size=(180,-1),choices=cursos,style=wx.CB_READONLY)


self.btn_ok=wx.Button(panel,-1,'Ok',(20,70),(100,30))

wx.EVT_BUTTON(self,self.btn_ok.GetId(),self.ok)

self.btn_cancel=wx.Button(panel,-1,'Cancelar',(160,70),(100,30))

wx.EVT_BUTTON(self,self.btn_cancel.GetId(),self.cancel)


def ok(self,event):

    a=self.nome_aluno.GetValue()

    frame.text1.SetLabel(a)

    frame.text1.SetForegroundColour('black')

    b=self.curso.GetValue()

    frame.text2.SetLabel(b)

    frame.text2.SetForegroundColour('black')

    self.Destroy()


def cancel(self,event):

    self.Destroy()


class MinhaFrame(wx.Frame):

    def __init__(self, parent, id):

        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (300, 200))

```

```

self.panel=wx.Panel(self)

self.panel.SetBackgroundColour('Ligth_Grey')


self.file='None'


wx.StaticText(self.panel,-1,"Nome do aluno:",(10,10))
wx.StaticText(self.panel,-1,"Curso do aluno:",(10,30))
wx.StaticText(self.panel,-1,"Media:",(10,50))
wx.StaticText(self.panel,-1,"Variancia:",(10,70))
wx.StaticText(self.panel,-1,"Numero de cadeiras:",(10,90))


self.text1=wx.StaticText(self.panel,-1,"Nenhum",(130,10))
self.text1.SetForegroundColour('red')
self.text2=wx.StaticText(self.panel,-1,"Nenhum",(130,30))
self.text2.SetForegroundColour('red')
self.text3=wx.StaticText(self.panel,-1,"Nenhum",(130,50))
self.text3.SetForegroundColour('red')
self.text4=wx.StaticText(self.panel,-1,"Nenhum",(130,70))
self.text4.SetForegroundColour('red')
self.text5=wx.StaticText(self.panel,-1,"Nenhum",(130,90))
self.text5.SetForegroundColour('red')


wx.StaticBox(self.panel,-1,'Atribuicoes',(180,5),(100,110))
wx.StaticText(self.panel,-1,"Processo:",(200,30))
wx.StaticText(self.panel,-1,"Terminado:",(200,70))
self.text6=wx.StaticText(self.panel,-1,"Nenhum",(200,45))
self.text7=wx.StaticText(self.panel,-1,"Desconhecido",(200,85))


menu=wx.MenuBar()

self.SetMenuBar(menu)

status=self.CreateStatusBar()

```

```

menu_file=wx.Menu()
menu.Append(menu_file,"File")
menu_file.Append(2,"Load", "Carregar dados de ficheiros.")
self.Bind(wx.EVT_MENU,self.abrir,id=2)
menu_file.Append(6,"Save","Salvar o processo do aluno.")
self.Bind(wx.EVT_MENU,self.salvar,id=6)
menu_file.AppendSeparator()
menu_file.Append(1,"Exit","Sair do programa")
self.Bind(wx.EVT_MENU,self.sair,id=1)

menu_edit=wx.Menu()
menu.Append(menu_edit,"Edit")
menu_edit.Append(3,"Definitions","Definicoes do aluno.")
self.Bind(wx.EVT_MENU,self.definicoes,id=3)
menu_edit.Append(4,"Atribuicoes","Atribuicoes feitas pelo utilizador.")
self.Bind(wx.EVT_MENU,self.atrib,id=4)

menu_graf=wx.Menu()
menu.Append(menu_graf,"Graficos")
menu_graf.Append(5,"Histograma","Faz o histograma das notas do aluno.")
self.Bind(wx.EVT_MENU,self.hist,id=5)

menu_about=wx.Menu()
menu.Append(menu_about,"About")
menu_about.Append(10,"Acerca..","Acerca do programas tutorial...")
self.Bind(wx.EVT_MENU,self.acerca,id=10)

def salvar(self,event):
    if self.file=='None':
        wx.MessageBox("Nao foram carregadas notas.,"Erro")
    else:
        b=wx.FileDialog(self,"Abrir ficheiro de dados...",style=wx.SAVE,wildcard="*.*.")

```

```

if b.ShowDialog() == wx.ID_OK:

    self.filename2=b.GetPath()

    fid = open(self.filename2,'w')

    processo=self.text6.GetLabel()

    terminou=self.text7.GetLabel()

    nome=self.text1.GetLabel()

    curso=self.text2.GetLabel()

    media=self.text3.GetLabel()

    variancia=self.text4.GetLabel()

    cadeiras=self.text5.GetLabel()

    fid.write('Processo numero: '+processo+'\n')

    fid.write('Terminou: '+terminou+'\n')

    fid.write('*****\n')

    fid.write('Nome do aluno: '+nome+'\n')

    fid.write('Curso do aluno: '+curso+'\n')

    fid.write('*****\n')

    fid.write('Media do curso: '+media+'\n')

    fid.write('Variancia do curso: '+variancia+'\n')

    fid.write('Numero de cadeiras: '+cadeiras+'\n')

    fid.write('*****\n')

    numpy.savetxt(fid,self.file,fmt='%5.1f')

    b.Close()

```

```

def hist(self,event):

    if self.file=='None':

        wx.MessageBox("Nao foram carregadas notas.", "Erro")

    else:

        hframe=HistFrame(parent=frame,id=996)

        hframe.Centre()

        hframe.Show()

```

```

def atrib(self,event):

    aframe=AtribFrame(parent=frame,id=997)

    aframe.Centre()

    aframe.Show()


def definicoes(self,event):

    dframe=DefFrame(parent=frame,id=998)

    dframe.Centre()

    dframe.Show()


def acerca(self,event):

    info = wx.AboutDialogInfo()

    info.SetName('Tutorial')

    info.SetVersion('0.1')

    info.SetDescription('Tutorial')

    info.SetCopyright('Versao teste sem copyrigh')

    info.SetWebSite('http://numist.ist.utl.pt')

    info.SetLicence('Nao tem.')

    info.AddDeveloper('Pedro Correia')

    info.AddDocWriter('Pedro Correia')

    wx.AboutBox(info)


def abrir(self,event):

    b=wx.FileDialog(self,"Abrir ficheiro de dados...",style=wx.OPEN,wildcard="*.*.")

    if b.ShowModal() == wx.ID_OK:

        self.filename=b.GetPath()

        self.file=numpy.loadtxt(self.filename)

        numero1="%1f"%numpy.mean(self.file)

        self.text3.SetLabel(numero1)

        numero2="%1f"%numpy.var(self.file)

        self.text4.SetLabel(numero2)

        numero3=repr(self.file.shape[0])

```

```

        self.text5.SetLabel(numero3)

        self.text3.SetForegroundColour('black')

        self.text4.SetForegroundColour('black')

        self.text5.SetForegroundColour('black')

        b.Close()

def sair(self,event):

    a=wx.MessageDialog(self,"Tem a certeza que quer sair?", "Sair...",wx.YES_NO|wx.ICON_QUESTION)

    if a.ShowModal()==wx.ID_YES:

        self.Close()

if __name__=='__main__':

    app=wx.App()

    frame=MinhaFrame(parent=None,id=999)

    frame.Centre()

    frame.Show()

    app.MainLoop()

```

E é isto. Repara como as funções correspondetes a cada classe estão organizadas dentro da mesma e também como fomos construindo o código de algo extremamente simples para um software com um aspecto robusto.

Py2exe

Até agora temos feito programas que só funcionam porque o Python está instalado no computador. Isto não tem de ser assim, é possível compilar os programas de maneira a funcionar sem precisar de qualquer instalação do Python. Dado que isto é um passo que podes querer dar eventualmente vou introduzir a utilização do Py2exe. É muito simples de utilizar para a maioria dos casos, noutros é preciso muito mais trabalho porque o Py2exe consegue funcionar bem com muitas bibliotecas mas não com todas (o matplotlib costuma dar problemas). De qualquer maneira por agora não é problema e o que eu quero é passar um programa que vamos fazer para uma instalação que não precisa de Python para funcionar. Escrevi o seguinte programa num ficheiro chamado tutorial2.py:

```
import wx

class MinhaFrame(wx.Frame):

    def __init__(self, parent, id):

        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (300, 200))

        self.panel=wx.Panel(self)

        self.panel.SetBackgroundColour('Ligth_Grey')

if __name__=='__main__':

    app=wx.App()

    frame=MinhaFrame(parent=None,id=999)

    frame.Centre()

    frame.Show()

    app.MainLoop()
```

Não tem nada de complicado apenas faz uma pequena janela igual a tantas que já fizemos mas vai servir para testar o py2exe. Na mesma pasta onde está o tutorial2.py fiz um ficheiro chamado script.py que lá dentro tem escrito:

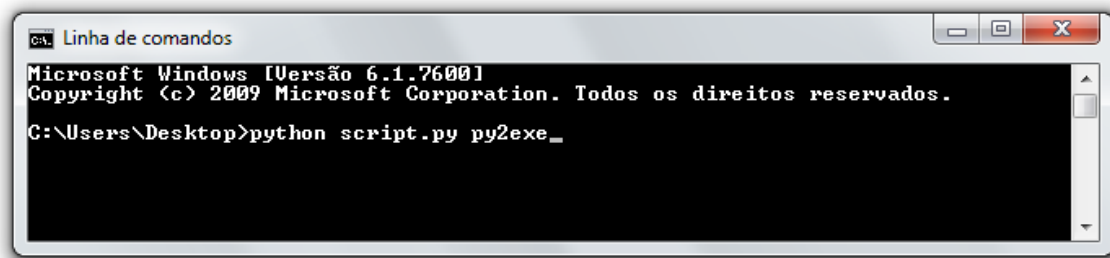
```
from distutils.core import setup

import sets

import py2exe

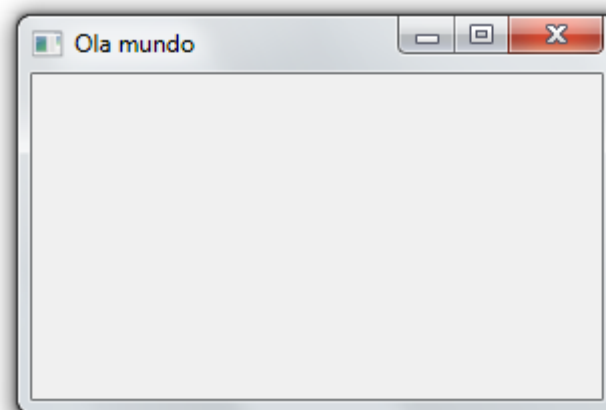
setup(windows=['tutorial2.py'])
```

Está a importar alguns módulos necessários para poder executar o py2exe e a seguir dou as opções que pretendo para o meu programa dentro do comando “setup”. Primeiro digo que se trata de um programa de janelas (windows), depois insiro o nome do ficheiro que tem o programa que vou utilizar. E está quase. Agora vou à linha de comandos do Windows e faço isto:



```
C:\Users\Desktop>python script.py py2exe_
```

Repara que na linha de comandos tenho que ir para o sítio onde estão os meus ficheiros tutorial2.py e script.py. Ao pressionar enter vais ter uma série de instruções no ecrã e no fim se fores à pasta onde estão os ficheiros vais ver duas pastas novas “build” e “dist”. A pasta “dist” tem o teu programa já compilado do Python que podes correr mesmo em computadores sem Python. O executável tem um nome igual ao do ficheiro que lhe deu origem: tutorial2.exe e ao corrê-lo:



Aparece o programa que escrevi. Claro que se podem fazer muitas mais coisas que não ensinei aqui como inserir ícones para o programa ou incluir ou excluir alguns submódulos das bibliotecas (para que o tamanho do programa fique mais pequeno). Não vamos ver todas essas hipóteses, na verdade não passei o programa que fizemos na explicação do wxPython porque esse programa tem o matplotlib que é muito problemático (pelo menos na versão que tenho do py2exe e matplotlib) para ser compilado. Esperemos que seja um problema que irão resolver em breve se é que já não o fizeram. A página do py2exe tem um pequeno tutorial e algumas explicações sobre este e outros problemas que te podem auxiliar nesta tarefa (<http://www.py2exe.org/>). Para além disso podes sempre experimentar outras alternativas ao py2exe (existem várias para várias plataformas).

wxPython avançado

Já sabemos o essencial sobre wxPython mas agora vamos aprender a fazer algumas coisas que podem melhorar o interface dos nossos programas. Não iremos fazer outro program estilo tutorial, vou sim dar exemplos de ferramentas ou alternativas a algumas que já utilizamos. Vou começar por ensinar como inserir um gráfico do matplotlib (não interactivo) embutido na janela do wxPython em vez de chamar uma janela nova cada vez que queremos um gráfico.

Matplotlib no wxPython (parte 2)

Comecei por escrever um programa no Spyder com uma janela um pouco maior do que a que utilizamos antes:

```
import wx

from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as FigureCanvas

from matplotlib.figure import Figure

import numpy.random


class MinhaFrame(wx.Frame):

    def __init__(self, parent, id):

        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (600, 400))

        self.panel=wx.Panel(self)

        self.panel.SetBackgroundColour('Ligth_Grey')


if __name__=='__main__':

    app=wx.App()

    frame=MinhaFrame(parent=None,id=999)

    frame.Centre()

    frame.Show()

    app.MainLoop()
```

Por agora debes perceber tudo o que vai aqui except as importações iniciais que nesse código ainda não são necessárias mas mais para a frente vão ser fundamentais. Vamos analisar as importações melhor:

```
import wx
```

```

from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as FigureCanvas

from matplotlib.figure import Figure

import numpy.random

```

A primeira e última já usamos sendo que o “wx” é a biblioteca que estamos a usar para fazer interfaces e a “numpy.random” serve para criar vectores de números aleatório (é com isto que vou gerar as séries para fazer o gráfico embutido na janela). Agora com isto:

```

from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as FigureCanvas

```

Estou a importar a ligação que existe entre o matplotlib e o wxPython. E com isto:

```

from matplotlib.figure import Figure

```

Estou a importar o equivalente ao “matplotlib.pyplot” que já usamos anteriormente só que este vai fazer o gráfico no sítio onde dissermos para ele o fazer enquanto o “matplotlib.pyplot” tem a sua própria janela. Vou começar por arranjar uma maneira do utilizador poder gerar um gráfico:

```

import wx

from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as FigureCanvas

from matplotlib.figure import Figure

import numpy.random

```

```

class MinhaFrame(wx.Frame):

    def __init__(self, parent, id):

        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (600, 400))

        self.panel=wx.Panel(self)

        self.panel.SetBackgroundColour('Ligth_Grey')


        menu=wx.MenuBar()

        self.SetMenuBar(menu)

        status=self.CreateStatusBar()


        menu_file=wx.Menu()

        menu.Append(menu_file,"File")

        menu_file.Append(1,"Grafico", "Cada vez que carregas tens um grafico novo.")

```

```

if __name__=='__main__':

    app=wx.App()

    frame=MinhaFrame(parent=None,id=999)

    frame.Centre()

    frame.Show()

    app.MainLoop()

```

Agora já vai aparecer um tópico para escolher na janela ao qual precisamos de associar um evento e a respectiva função:

```

import wx

from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as FigureCanvas

from matplotlib.figure import Figure

import numpy.random


class MinhaFrame(wx.Frame):

    def __init__(self, parent, id):

        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (600, 400))

        self.panel=wx.Panel(self)

        self.panel.SetBackgroundColour('Ligth_Grey')


        menu=wx.MenuBar()

        self.SetMenuBar(menu)

        status=self.CreateStatusBar()


        menu_file=wx.Menu()

        menu.Append(menu_file,"File")

        menu_file.Append(1,"Grafico", "Cada vez que carregas tens um grafico novo.")

        self.Bind(wx.EVT_MENU,self.grafico,id=1)


    def grafico(self,event):

        self.figure = Figure(figsize=(12, 7), dpi=49)

        self.axes=self.figure.add_subplot(111)

        a=numpy.random.rand(100)

```

```

        b=numpy.random.rand(100)

        self.axes.plot(a,b,'*',color='#00FF33',markersize=30)

        self.canvas = FigureCanvas(self.panel, wx.ID_ANY, self.figure)

if __name__=='__main__':

    app=wx.App()

    frame=MinhaFrame(parent=None,id=999)

    frame.Centre()

    frame.Show()

    app.MainLoop()

```

Liguei um evento ao tópico “Gráfico” e agora sempre que carregares no tópico vai surgir um novo gráfico no painel. Repara:

```

        self.figure = Figure(figsize=(12, 7), dpi=49)

```

Com isto estou a criar a imagem do gráfico (que ainda não existe) e a dizer-lhe o tamanho, “figsize”, e penso que a resolução, “dpi”. A seguir:

```

        self.axes=self.figure.add_subplot(111)

```

Estou a dizer que o gráfico que inserimos vai ter um dado rácio entre horizontal e vertical (estou bastante a falar de cor e não tenho, de todo, a certeza do que estou a dizer mas vai acompanhando) e a criar o sítio onde ele vai estar (penso que pode ter vários na mesma figura). Depois:

```

        a=numpy.random.rand(100)

        b=numpy.random.rand(100)

```

Estou a criar a minha série de dados (“a” e “b”). Agora precisamos de chamar o gráfico a partir do sítio onde ele vai estar:

```

        self.axes.plot(a,b,'*',color='#00FF33',markersize=30)

```

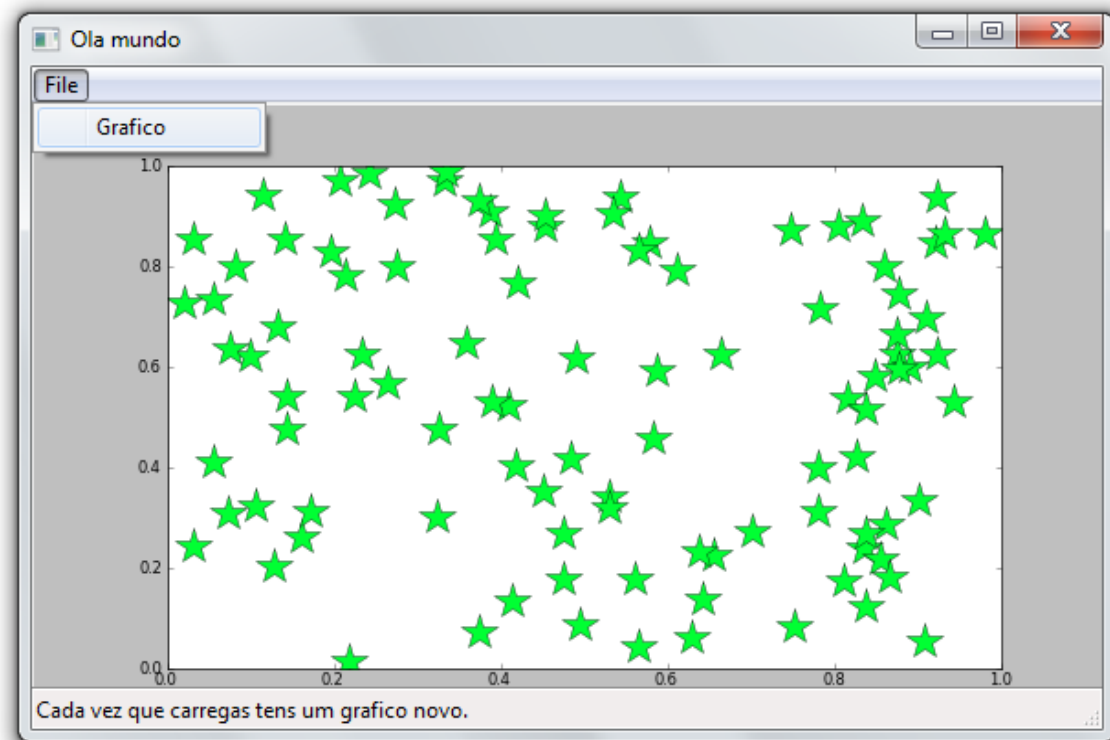
Assim o fiz usando as séries “a” e “b” com estrela como símbolo, cor verde e tamanho do marcador de 30 (repara que chamei a partir do “self.axes” que supostamente é o gráfico em si). Agora só precisamos de o colocar no painel que criamos na janela, “self.panel”. O segundo argumento é o id (“wx.ID_ANY” serve para meter um id qualquer) e terceiro o que vamos colocar no painel que no nosso caso foi a figura que criamos em primeiro lugar.

```

        self.canvas = FigureCanvas(self.panel, wx.ID_ANY, self.figure)

```

Resumindo, começamos por criar uma figura, depois criamos o sítio do gráfico na figura, fizemos o gráfico e finalmente associamos a figura ao painel da nossa janela. O resultado é isto:



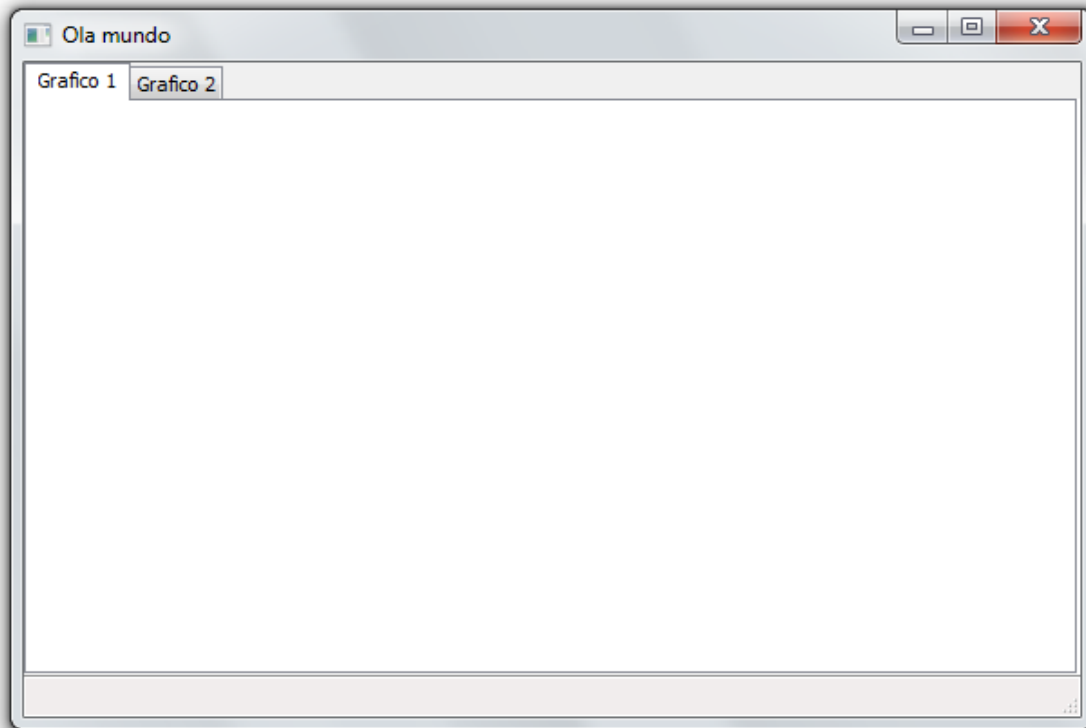
Tudo isto foi um pouco esquisito, sou o primeiro a concordar, mas é o método que o matplotlib tem de se inserir no wxPython. Embora não perceba totalmente tudo o que acontece nestes comandos tenha utilizado várias vezes esta receita nos meus programas. É uma questão de ir experimentando e procurares o que pretendes.

Notebook

O Notebook é uma maneira de fazer com que uma janela tenha mais do que uma página e para o que nos interessa, mais do que um painel. Fiz novamente um janela como a do capítulo anterior e inseri isto em vez do normal painel (neste capítulo vou omitir partes do código que por agora já deverão ser intuitivas):

```
...  
  
class MinhaFrame(wx.Frame):  
    def __init__(self, parent, id):  
        wx.Frame.__init__(self, parent, id, 'Ola mundo', size = (600, 400))  
  
        #self.panel=wx.Panel(self)  
  
        #self.panel.SetBackgroundColour('Ligth_Grey')  
  
        menu=wx.MenuBar()  
        self.SetMenuBar(menu)  
        status=self.CreateStatusBar()  
  
        self.tabbed=wx.Notebook(self, -1, style=(wx.NB_TOP))  
  
        self.panel1=wx.NotebookPage(self.tabbed, 101)  
        panel1=wx.Panel(self.panel1)  
        page1=self.tabbed.AddPage(self.panel1, "Grafico 1")  
        self.panel2=wx.NotebookPage(self.tabbed, 102)  
        panel2=wx.Panel(self.panel2)  
        page2=self.tabbed.AddPage(self.panel2, "Grafico 2")  
  
        ...
```

Se fores a experimentar correr esta classe o aspecto deverá ser este:



Repara que podes escolher entre visualizar uma página ou outra podendo duplicar, triplicar ou eneplicar quantas vezes tu quiseres a área das tuas páginas. Analisando o que fizemos:

```
self.tabbed=wx.Notebook(self, -1, style=(wx.NB_TOP))
```

Isto irá criar um ambiente preparado para receber páginas.

```
self.panel1=wx.NotebookPage(self.tabbed,101)
```

```
panel1=wx.Panel(self.panel1)
```

```
page1=self.tabbed.AddPage(self.panel1,"Grafico 1")
```

Com isto começamos por criar uma página do Notebook no qual o primeiro argumento é o objecto do Notebook em si que está atribuído à variável “self.tabbed”. A seguir insiro um painel dentro dessa página e depois dou um nome à página, “Grafico 1”. Depois fiz exactamente o mesmo para fazer uma segunda página.

```
self.panel2=wx.NotebookPage(self.tabbed,102)
```

```
panel2=wx.Panel(self.panel2)
```

```
page2=self.tabbed.AddPage(self.panel2,"Grafico 2")
```

A partir daqui podemos usar isto como se fosse um qualquer painel normal dos que temos utilizado até ao momento. Vou fazer o programa do capítulo anterior mas desta vez temos dois tópicos para fazer dois gráficos, um em cada página.

...

```
self.panel1=wx.NotebookPage(self.tabbed,101)
```

```

panel1=wx.Panel(self.panel1)

page1=self.tabbed.AddPage(self.panel1,"Grafico 1")

self.panel2=wx.NotebookPage(self.tabbed,102)

panel2=wx.Panel(self.panel2)

page2=self.tabbed.AddPage(self.panel2,"Grafico 2")


menu_file=wx.Menu()

menu.Append(menu_file,"File")

menu_file.Append(1,"Grafico", "Cada vez que carregas tens um grafico novo.")

self.Bind(wx.EVT_MENU,self.grafico,id=1)

menu_file.Append(2,"Grafico2", "Cada vez que carregas tens um grafico novo.")

self.Bind(wx.EVT_MENU,self.grafico2,id=2)


def grafico2(self,event):

    self.figure = Figure(figsize=(12, 6), dpi=49)

    self.axes=self.figure.add_subplot(111)

    a=numpy.random.rand(100)

    b=numpy.random.rand(100)

    self.axes.plot(a,b,'--',color='pink',markersize=30)

    self.canvas = FigureCanvas(self.panel2, wx.ID_ANY, self.figure)


def grafico(self,event):

    self.figure = Figure(figsize=(12, 6), dpi=49)

    self.axes=self.figure.add_subplot(111)

    a=numpy.random.rand(100)

    b=numpy.random.rand(100)

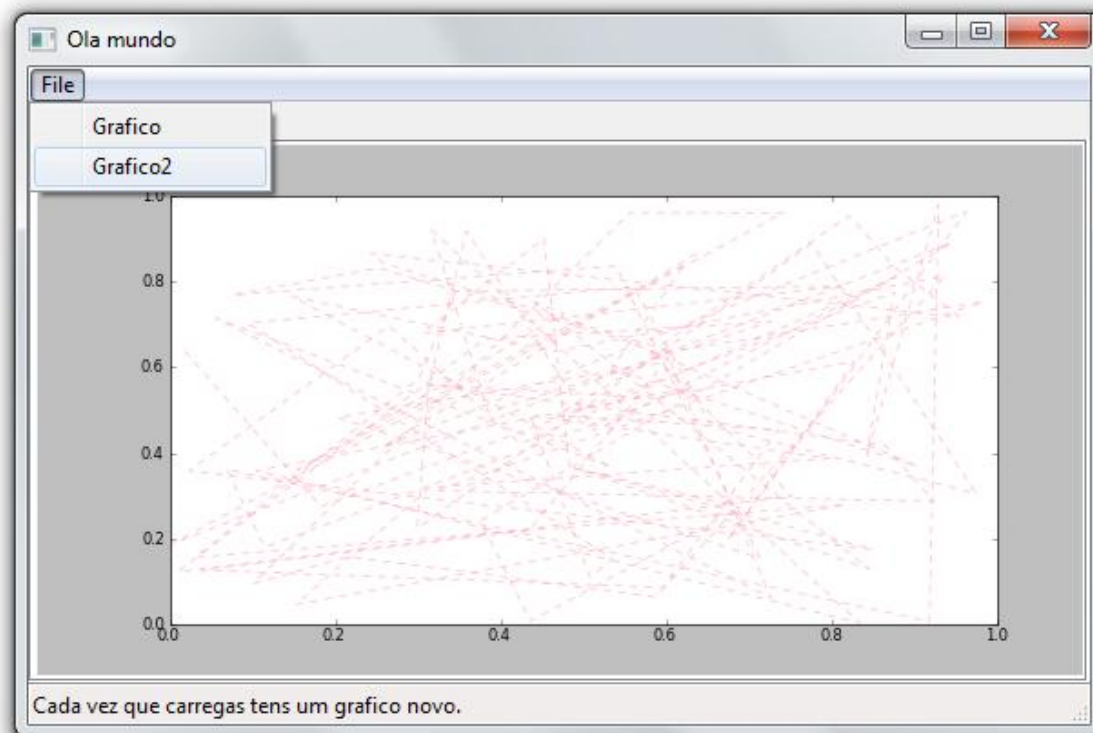
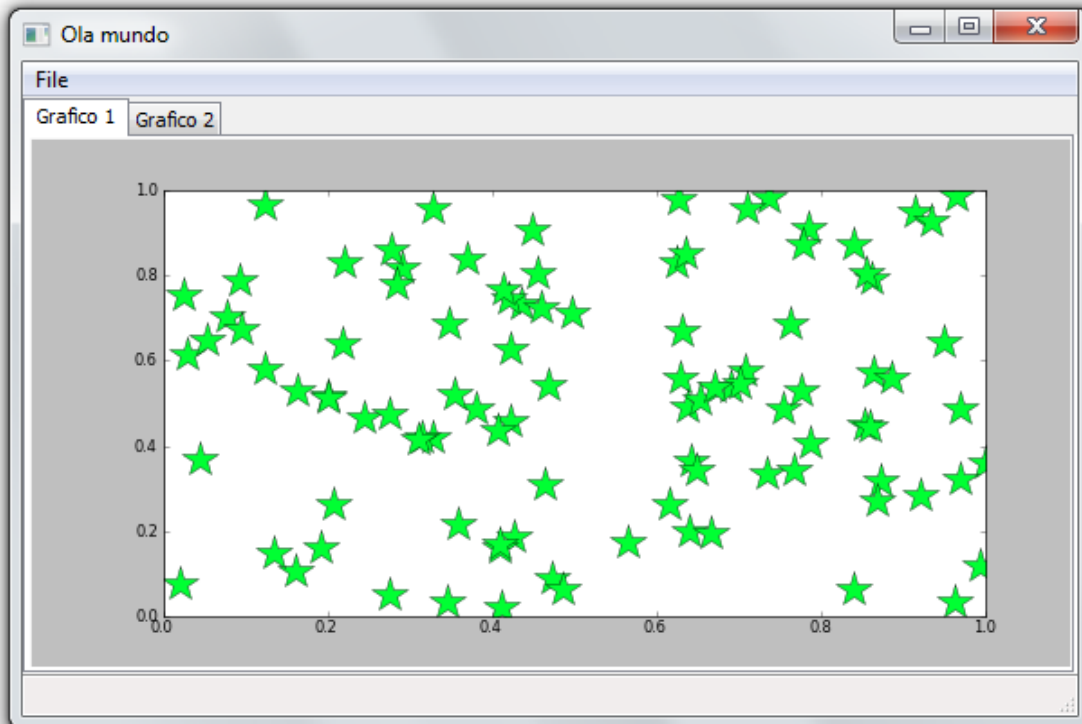
    self.axes.plot(a,b,'*',color='#00FF33',markersize=30)

    self.canvas = FigureCanvas(self.panel1, wx.ID_ANY, self.figure)

...

```

Agora se tentares correr o programa vais ver que tudo isto é possível:



Matplotlib avançado

O matplotlib tem muitos tipos de gráficos alguns deles complexos de se fazer. Vou tentar passar por uma boa parte deles nesta secção para que possas ter um leque variado por onde trabalhar.

Este documento está ainda em desenvolvimento e é apresentada esta primeira versão para análise do leitor. Para dúvidas ou sugestões sobre esta publicação por favor envie um mail para numist@ist.utl.pt sobre o tópico “Sugestão para livro do Python”. Logo que possível esta publicação será actualizada para conter instruções sobre:

- a) Outros gráficos em matplotlib.*
- b) Pequena introdução ao Fortran.*
- c) Associação entre Python e Fortran (isto é importante devido à baixa performance do Python em relação a outras linguagens, usando esta mistura conseguimos o melhor dos dois mundos)*
- d) e talvez mais algumas bibliotecas de interesse*

Boa sorte e grande abraço,

Pedro Correia