

Manual de Python

Francisco Silva, Tiago Dinis

27 de Agosto de 2009

Capítulo 1

Introdução

Este manual foi escrito no intuito de apoiar os projectos por nós dirigidos na Escola de Verão de Física, visto que ambos passam por aprender o básico sobre Python - uma linguagem de programação simples e fácil de aprender.

Mas, porquê escrever mais um manual de Python? É verdade que na internet se encontram vários manuais de Python, mas o problema é que esses manuais ora estão em inglês ora são demasiado avançados ora estão centrados em aspectos que não nos interessam... E é por isso que nós nos juntamos para escrever um pequeno manual onde explicamos aquilo que necessitamos da forma como queremos. No fim da escola os alunos poderão guardar este manual como referência caso queiram brincar um pouco mais com o Python.

Este manual não pretende ser completo. Caso se interessem por aprender mais podem começar por [\[link\]](#).

1.1 O que é simulação

Suponhamos que estamos interessados em construir um carro que seja seguro. Precisamos então de testar o comportamento do carro em vários tipos de acidentes. O modo mais directo de testar a sua segurança seria realizar uma colisão real, ou seja, metemo-nos no carro aceleramos e chocamos contra um obstáculo. Este método é incómodo pois só podemos fazer a experiência uma vez e nem podemos analisar os resultados.

E se no lugar de pessoas usássemos bonecos, os quais simulariam o condutor e os passageiros? É certamente melhor para a nossa integridade, e é uma das técnicas que usam os fabricantes de automóveis.

Mesmo assim temos um problema, para cada simulação precisamos dum carro, o que fica extremamente caro. Outra opção seria simular a situação dentro dum computador, poupando carros e bonecos. A estratégia passa por calcular todas as forças, velocidades e posições do carro e dos passageiros ao longo do acidente usando um computador, estimando no fim o dano causado nos

vários passageiros. Podemos assim simular acidentes de forma muito mais barata e rápida e em inúmeras situações diferentes.

Claro que a simulação computacional, em geral, é menos fidedigna que o embate real dum carro num obstáculo. Por isso, usa-se as duas técnicas complementarmente - começa-se por se simular no computador e numa fase mais avançada usa-se um protótipo.

Note-se que há imensas situações as quais só se podem estudar por simulação computacional - por exemplo a simulação da origem do universo ou nano-engrenagens, até agora impossíveis de fabricar.

1.2 Linguagens de programação

Claro que para dar as ordens ao computador precisamos de saber comunicar com ele, numa linguagem que ele compreenda. Desde os primórdios da informática (>1940s) que se tem vindo a criar inúmeras destas linguagens, chamadas linguagens de programação, cada uma com o seu propósito e sua estrutura. Dado isto, a escolha de uma linguagem de programação para uma dada tarefa não deve ser aleatória - imaginem que têm à vossa disposição um martelo e uma chave de fendas - é possível colocar um parafuso à martelada, mas é muito mais fácil com a chave.

Estas linguagens podem ser classificadas segundo vários critérios. Por exemplo, dependendo da forma como o programador dá as ordens ao computador, existem linguagens imperativas ou declarativas.

Nas linguagens imperativas o programador escreve explicitamente todas as acções que o computador deve fazer, à semelhança de uma receita.

Nas declarativas, de um modo geral, o programador escreve o que quer que o programa faça, sem especificar exactamente como ou em que ordem. Note-se que isto não quer dizer que o programador possa escrever 'Computador, calcula-me o sentido da vida' - o computador não saberia o que fazer. O programador necessita de utilizar os elementos básicos da linguagem e relacioná-los de maneira a resolver o problema. A diferença é que nas linguagens imperativas o programador dá ordens de manipulação destes elementos, nas declarativas o programador estabelece relações entre eles.

Embora fosse interessante, não vamos falar aqui sobre as várias linguagens de programação diferentes, está fora dos nossos objectivos. No entanto, os mais curiosos podem espreitar no apêndice [NUMERODOAPENDICE].

1.3 Porquê Python

Vejamos os nossos requerimentos para as nossas simulações:

- Necessitamos de uma linguagem que nos permita fazer cálculos e manipular os resultados facilmente;

- Esta deve permitir-nos expôr os resultados de forma conveniente - números, gráficos, animações, etc;
- Deve ser de acesso fácil e gratuito, pômos assim de parte linguagens que se pagam ou que necessitam de um curso só para as instalar; [Ainda não é perfeito.]
- Necessitamos de uma linguagem com uma sintaxe simples cujos elementos básicos possam ser ensinados em menos de 1 dia.

Embora existam várias linguagens que de certo modo preencham os 4 requerimentos, existe uma que achamos especialmente adequada: Python. Esta distingue-se especialmente por ter uma sintaxe clara e ter elementos básicos simples. Ser relativamente fácil de usar. Esta tem ainda a vantagem de possuir vastas bibliotecas igualmente simples de usar.

Dado isto, passemos à acção.

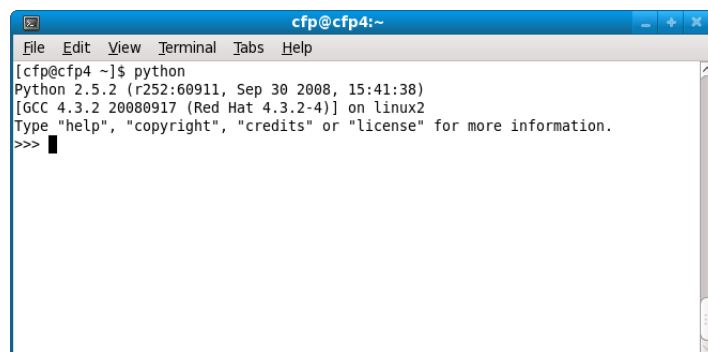
Capítulo 2

Rudimentos de Python

Neste capítulo vamos aprender a usar o básico do Python, com ele o aluno será capaz de fazer vários programas, simples mas poderosos. Supomos que o aluno usa Linux, as diferenças para outros sistemas operativos (Windows, MacOSX...) não são significativas. [pensar num apêndice]

2.1 Interpretador e Operações Básicas

O método mais simples de trabalhar com Python é através do interpretador. Este pode ser usado por vários caminhos: na linha de comandos, pelo internet [], por interfaces gráficas próprias [] ou até dentro doutros programas. Na Escola de Física será usado na linha de comandos. Para abrir o interpretador, introduz-se *python* na linha de comandos.

A screenshot of a terminal window titled 'cfp@cfp4:~'. The window has a menu bar with 'File', 'Edit', 'View', 'Terminal', 'Tabs', and 'Help'. The terminal shows the command '[cfp@cfp4 ~]\$ python' being executed. The output is 'Python 2.5.2 (r252:60911, Sep 30 2008, 15:41:38)' followed by '[GCC 4.3.2 20080917 (Red Hat 4.3.2-4)] on linux2'. Below this, it says 'Type "help", "copyright", "credits" or "license" for more information.' and then '>>>' with a cursor on the next line.

O conceito do interpretador é simples, dá-se uma ordem e ele executa-a. Para sair do interpretador escreve-se *exit()* ou prime-se Control-d.

Começemos por usar o interpretador como uma calculadora. Temos as operações elementares da álgebra:

Operação	Símbolo
Adição	+
Subtração	-
Multiplicação	*
Divisão	/
Exponenciação	**
Resto	%

Exercício 2.1. Calcule $4.5 \times 3.14 + 3.4^{2.1}$. Reparem que o Python usa um ponto como separador decimal.

Exercício 2.2. Quanto vale $3/2$? E $3.01/2$?

Exercício 2.3. Quanto falta a 2354785 para ser um múltiplo de 7?

2.2 Variáveis

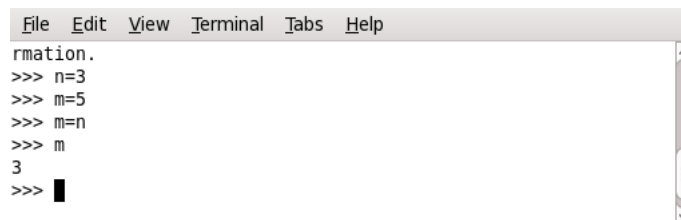
2.2.1 Variáveis numéricas

Em Python existem três tipos principais de variáveis numéricas:

- Os números inteiros, por exemplo $n = 3$;
- Os números reais, por exemplo $x = -3.14$;
- E os números complexos, como por exemplo $z = 3.2 + 1j$. Notem que o Python usa j , e não i como é habitual.

Caso não haja separador decimal o Python assume que o número é inteiro. Isto pode trazer problemas nas divisões, por exemplo $1/2 = 0$, mas $1./2. = 0.5$.

Quando se escreve $n = 3$ está-se a guardar o valor 3 em n . Vejamos o exemplo seguinte:



```

File Edit View Terminal Tabs Help
rmation.
>>> n=3
>>> m=5
>>> m=n
>>> m
3
>>>

```

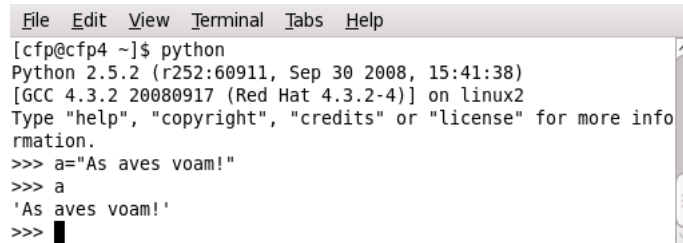
O que acontece:

1. Na primeira linha guardamos o valor 3 na variável n ;
2. Na segunda guardamos o valor 5 em m ;
3. Na terceira linha estamos a copiar o valor da variável n (que neste caso é 3) para a variável m ;
4. Por fim perguntamos o valor de m ao computador.

Exercício 2.4. Se num quinto passo atribuirmos o valor 7 a n , quanto vale agora m ?

2.2.2 Cadeias de caracteres

Mas nem todas as variáveis são números, podemos também definir cadeias de caracteres. As cadeias de caracteres são delimitadas por `""` ou por `' '`, vejamos um exemplo:



```
File Edit View Terminal Tabs Help
[cfp@cfp4 ~]$ python
Python 2.5.2 (r252:60911, Sep 30 2008, 15:41:38)
[GCC 4.3.2 20080917 (Red Hat 4.3.2-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> a="As aves voam!"
>>> a
'As aves voam!'
```

Infelizmente, nem todos os caracteres se podem usar em cadeias de caracteres, por vezes é preciso usar certos atalhos, por exemplo uma nova linha escreve-se `"\n"`, as aspas `"\""`...

As cadeias de caracteres são estruturas rígidas, ou seja, não se podem alterar. Mas em contrapartida pode-se aceder a apenas parte da cadeia:

- `a[i]` acede ao *i*-ésimo carácter, por exemplo `a[4]` é igual a `"v"`. Note-se que a numeração dos caracteres começa pelo 0.
- `a[:j]` acede a todos os caracteres até à posição *j*, *j* não incluído, por exemplo `a[:8]` é igual `"As aves"`.
- `a[i:]` é igual ao anterior mas acedendo a todos os números após *i*, *i* incluído.
- `a[i:j]` acede aos caracteres entre as posições *i* e *j*, por exemplo `a[3:6]` é igual a `"ave"`.

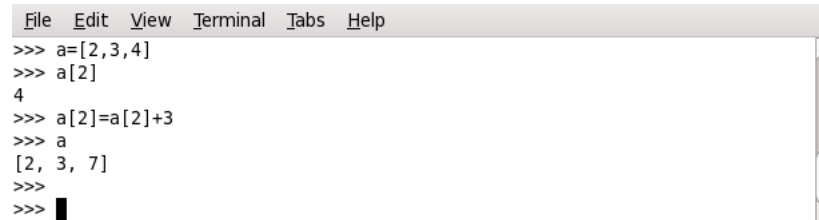
Pode-se também juntar várias cadeias, usando um sinal de `+`, por exemplo `"Eu tenho uma"+" mesa"` é igual a `"Eu tenho uma mesa"`. Ou multiplicar uma cadeia por um número natural: `2*"po"` é igual a `"popo"`.

Exercício 2.5. *Crie uma variável com o valor "Das uvas se faz o vinho!".*

Exercício 2.6. *Modifique a variável para "Das macas se faz a cidra!". É possível pôr cedilhas e acentos nas palavras, mas nós vamos evitá-los.*

2.2.3 Listas

Uma lista é um conjunto ordenado de objectos, estes podem ser números, cadeias de caracteres ou mesmo outras listas. Em Python uma lista é delimitada por parentesis rectos, por exemplo:

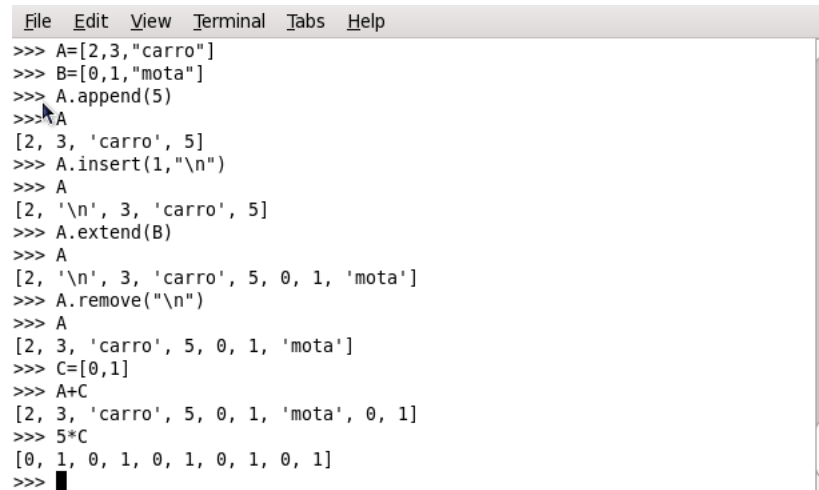


```
File Edit View Terminal Tabs Help
>>> a=[2,3,4]
>>> a[2]
4
>>> a[2]=a[2]+3
>>> a
[2, 3, 7]
>>>
>>> █
```

Como podem ver as listas são mais flexíveis que as cadeias. Vejamos algumas das operações que podemos realizar com elas:

- `A.append(X)` acrescenta o elemento `X` ao fim da lista `A`;
- `A.insert(I,X)` acrescenta o elemento `X` na posição `I` da lista `A`;
- `A.extend(B)` une a lista `A` com a lista `B`. Com a `B` após a `A`;
- `A.index(X)` devolve a posição da primeira aparição do elemento `X` na lista `A`;
- `A.remove(X)` apaga o primeiro elemento `X` da lista `A`;
- `len(A)` calcula o comprimento da lista `A`, ou seja, o número de elementos que a lista `A` contém;
- `A+B` devolve a união das listas `A` e `B`;
- `n*A` devolve a lista `A` repetida `n` vezes.

Vejamos um exemplo:



```
File Edit View Terminal Tabs Help
>>> A=[2,3,"carro"]
>>> B=[0,1,"mota"]
>>> A.append(5)
>>> A
[2, 3, 'carro', 5]
>>> A.insert(1,"\n")
>>> A
[2, '\n', 3, 'carro', 5]
>>> A.extend(B)
>>> A
[2, '\n', 3, 'carro', 5, 0, 1, 'mota']
>>> A.remove("\n")
>>> A
[2, 3, 'carro', 5, 0, 1, 'mota']
>>> C=[0,1]
>>> A+C
[2, 3, 'carro', 5, 0, 1, 'mota', 0, 1]
>>> 5*C
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
>>> █
```

Exercício 2.7. Crie duas listas, digamos `A` e `B`, e calcule `A.append(B)`, o que é que acontece?

Exercício 2.8. Crie uma lista com todos os números inteiros ordenados entre 1 e 50. Dica experimente o que acontece se fizer `range(10)`.

2.3 Ficheiros Python

Imaginem que estamos a fazer um cálculo e ao final de 30 min reparamos que nos enganamos. Ora temos que repetir quase tudo. Mas existe uma solução que é escrever num ficheiro as ordens, a que nós chamamos normalmente de código, que nós queremos que o computador siga.

Por isso, começamos por abrir um editor de texto qualquer, e escrevemos:

```
#coding=utf8

frase="Este é o meu primeiro programa!" #Isto é um comentário
print frase
```

Gravamos o ficheiro, por exemplo com o nome primeiro.py (o .py serve para identificar os ficheiros de Python), e por fim corremos o Python com o ficheiro usando o comando *python primeiro.py*.

Vamos analisar o programa:

- `#coding=utf8` permite usar caracteres acentuados e o `ç`;
- `frase="Este é o meu primeiro programa."` cria uma cadeia de caracteres;
- `#` indica um comentário, o Python ignora a partir deste símbolo até à mudança de linha;
- `print frase` é o comando que escreve (ou imprime) o valor da variável `frase` na linha de comandos.

O código deve ser limpo, bem organizado e muito bem comentado. Senão, ninguém será capaz de perceber o que vocês estão a fazer nem mesmo vocês daqui a uma semana.

No caso especial do Python, o código tem que ser devidamente indentado (que consiste na adição de tabulações no início de cada linha), explicaremos este conceito à medida que precisarmos dele.

2.4 Funções

Imaginemos que queremos calcular o seno de 0.3, imprimir¹ uma frase ou colocar uma lista em ordem alfabética. Para efectuar estas tarefas vamos precisar de usar funções.

2.4.1 Print

De nada serve fazer um programa se ele não comunicar com o utilizador. A função `print` imprime aquilo que nós quisermos para a linha de comandos, como viram no exemplo anterior. Mas podemos fazer coisas mais complicadas:

¹Imprimir significa que o nosso programa nos devolve algo, por exemplo na linha de comandos, num ficheiro pdf ou em papel.

```
#coding=utf8
n=5
print "Este é o meu primeiro cálculo:", "\n", n, "ao quadrado é", n**2
```

O resultado é simples de interpretar:

- Começamos por imprimir a frase: “Este é o meu primeiro cálculo:”;
- Criamos uma nova linha com o “\n”;
- Imprimimos o valor de n , lembrem-se que n vale 5;
- Imprimimos “ao quadrado é”;
- E por fim, o programa devolve o valor de n^2 .

Exercício 2.9. Crie um ficheiro que imprima a frase “Se $n = 3$ e $m = 2$, então $n \times m = 6$ ”, e onde seja fácil mudar os valores de n e m .

2.4.2 Input

Vamos ser audaciosos e construir um programa que peça valores ao utilizador:

```
#coding=utf8

n=input("Escreve um número:")
print "O número que escreveu foi", n
```

A função `input` imprime um texto e fica à espera que o utilizador devolva um número, uma lista, uma cadeia de caracteres... escritos como se fosse no ficheiro, ou seja, no caso duma cadeia de caracteres é preciso colocar as aspas.

Existe uma outra versão que é

```
#coding=utf8
s=raw_input("Como é que te chamas?")
print "O teu nome é", s
```

Esta função faz exactamente a mesma coisa que `input` mas grava tudo o que utilizador escrever em forma de cadeia de caracteres.²

Exercício 2.10. Escreve um programa que dê uma lista com todos os números inteiros entre dois valores definidos pelo utilizador.

²Para inserir uma cadeia de caracteres não é preciso colocar aspas.

2.4.3 Transformar cadeias de caracteres em números e vice-versa

Para o Python “45” e 45 são coisas diferentes, sendo, respectivamente, uma cadeia de caracteres e um número. Mas podemos convertê-los facilmente:

- A função `str()` transforma um número ou uma lista numa cadeia de caracteres;
- Enquanto que a função `eval()` faz a transformação inversa.

Exercício 2.11. *Explique, passo por passo, o que acontece ao escrevermos:*

```
#coding=utf8
a=56
b=str(a)
c=2*b
d=eval(c)
print 2*d
```

2.4.4 Definir funções

Por vezes queremos usar funções que ainda não estão definidas, nesse caso temos que ser nós a defini-las. Vejamos um exemplo duma definição:

```
#coding=utf8
def soma(x,y):
    u=x+y
    return 2*u

g=3
h=2
print soma(g,h)
```

Neste exemplo existem vários promenores importantes:

- Uma função precisa dum nome, por exemplo, “soma”, os nomes devem ser simples e explícitos;
- Para definir uma função usa-se “def função(variáveis):”;
- Pode haver quantas variáveis nós queiramos, de notar que os nomes que usamos aqui só servem para a definição;
- O Python é uma linguagem de indentação, ou seja, ele sabe que determinadas ordens pertencem à função porque estão indentadas;
- Dentro da função podemos fazer o que quisermos;
- “return” devolve-nos o resultado da função, note-se que não é obrigatório haver um resultado, a função pode imprimir algo na linha de comandos;

- A função acaba quando o texto deixar de ser indentado.

Exercício 2.12. Escreva um programa que calcule n^2 .

Exercício 2.13. Escreva um programa que nos devolva $n!$. Use o facto de se $n = 1$, $n! = 1$, escrito na linha: `"If n==1: return 1"`³

2.5 Controlo de fluxo

Às vezes queremos que uma certa ordem seja realizada apenas se uma certa condição é respeitada ou então queremos repetir uma acção umas tantas vezes. A isso chamamos de controlo de fluxo.

2.5.1 Condições

Em linguagem corrente uma condição exprime-se “Se chover eu não sairei de casa.”, ora em inglês “se” escreve-se “if”, vejamos um exemplo:

```
#coding=utf8
n=input("Escreva um número: ")
if n==0:
    print "O número é nem positivo nem negativo"
    print "Ou seja, é o zero."
elif n>0:
    print "O número é positivo"
else:
    print "O número é negativo"
```

Existe três comandos a ter em conta:

- “If” que significa “se”, neste caso “se $n=0$ ” ele cumpre as ordens que aparecem em baixo indentadas;
- “elif” significa “senão se”, ou seja, se as condições que a precedem não forem respeitadas ela aplica mais uma condição;
- “else” significa “senão” e significa que se nenhuma das condições anteriores for respeitada ela cumpre as ordens que aparecem indentadas em baixo.

As condições podem ser escritas usando os símbolos apresentados na tabela:

símbolo	significado
<code>==</code>	igual
<code>!=</code>	diferente
<code>></code>	maior
<code><</code>	menor
<code>>=</code>	maior ou igual
<code><=</code>	menor ou igual

³Se n for igual a 1 devolve 1.

Podemos ainda combinar várias condições usando “and”, “or” e “not”⁴, por exemplo:

```
#coding=utf8
n=input("Escreva um número: ")
m=input("E um segundo número: ")

if n>0 and m>0:
    print "São ambos positivos."

if (n>0 or m>0) and not (n>0 and m>0):
    print "Um e só um é positivo."
```

No primeiro exemplo temos que “ $n > 0$ e $m > 0$ ”, enquanto que o segundo exemplo lê-se “um ou outro” e não “um e outro”.

Exercício 2.14. Crie um programa que verifique se certo número é par ou ímpar.

Exercício 2.15. Dado um determinado par de números queremos que um e apenas um seja positivo. Já damos um exemplo de como fazer isso, dê outros exemplos.

2.5.2 Repetições: para ... em ...

Temos uma operação e queremos-la repetir mil vezes. Ou escrevemos a ordem mil vezes, ou usamos um ciclo for (para em inglês):

```
#coding=utf8
n=input("Até que limite? ")

for i in range(n):
    print 2*i
```

Este programa imprime o dobro de todos os inteiros entre 0 e $n-1$. Vejamos passo por passo:

- “for i in range(n)”, significa que para todo o elemento, ao qual damos o nome de “i”, que está na lista “range(n)” efectuamos as ordens seguintes;
- As ordens aparecem nas linhas em baixo e aparecem identadas, neste caso imprimimos o dobro de cada elemento.

Por vezes, queremos parar um ciclo “for” abruptamente, para isso usamos o comando “break”:

```
#coding=utf8
n=input("Até que número? ")

for i in range(n):
    print 2*i
```

⁴Em português “e”, “ou” e “não”.

```

if i>=100:
    print "Atingiu o limite"
    break

```

Note que:

- As operações dentro do ciclo “if” estão indentadas em relação ao “if”;
- O “break” quebra o ciclo “for” imediatamente antes do “if”.

Exercício 2.16. *Crie uma lista de naipes (paus, ouros, copas e espadas) e uma lista de valores de cartas (Ás, 2, 3, 4, 5, 6, 7, Valete, Dama e Rei) e imprima todas as cartas do baralho português.*

Exercício 2.17. *Verifique se determinado número é primo. Use a função “break”.*

Exercício 2.18. *Faça um programa que gere uma lista de números primos.*

2.5.3 Repetições: enquanto ...

Podemos também repetir uma ordem enquanto (while em inglês) algo aconteça, por exemplo:

```

#coding=utf8
n=0
while n<=0:
    n=input("Escreva um número positivo: ")
print n

```

O ciclo vai ser repetido enquanto n for negativo.

Opialmente o ciclo pode ser acompanhado por um “else” que será realizado assim que a condição deixe de ser cumprida.

À imagem do ciclo “for” podemos quebrar um ciclo usando o comando “break”. E nesse caso o “else” não vai ser realizado.

Exercício 2.19. *Queremos calcular a raiz quadrada de n . Um método rudimentar é de pedir estimativas ao utilizador até que ele se aproxime o suficiente.*

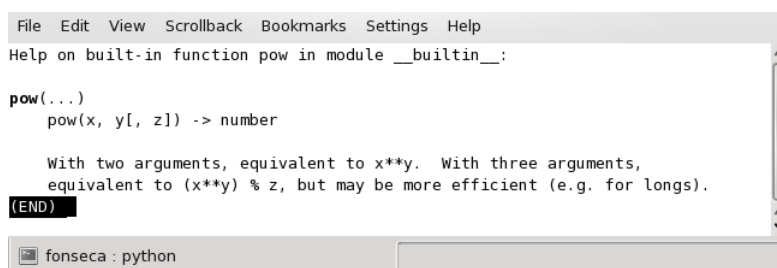
Exercício 2.20. *Um outro método é usando o algoritmo: seja r uma estimativa da raiz quadrada, sabemos que $\frac{1}{2}(r + \frac{n}{r})$ vai estar mais próximo da raiz. Calcule a raiz quadrada com uma boa precisão.*

Capítulo 3

Python mais avançado

3.1 Ajuda

Mesmo depois de usar Python durante anos, não é suposto saber tudo sobre ele. Para isso existem manuais, a internet e a ajuda do Python. É sobre a última que nos debruçaremos a seguir. Num interpretador de Python escreva “help(pow)”:



```
File Edit View Scrollback Bookmarks Settings Help
Help on built-in function pow in module __builtin__:

pow(...)
pow(x, y[, z]) -> number

    With two arguments, equivalent to x**y. With three arguments,
    equivalent to (x**y) % z, but may be more efficient (e.g. for longs).
(END)
```

fonseca : python

Exercício 3.1. Escreva um programa que nos devolva $n!$. Use o facto de se $n = 1$, $n! = 1$, escrito na linha: “If $n==1$: return 1”

Como podem ver, o Python explica-nos como funciona a função “pow”. Ela pega em dois ou três argumentos e devolve-nos um número.

3.2 Bibliotecas

Experimente escrever “cos(2.1)” no interpretador. Em princípio o Python não reconhece a função. Mas o Python pode ser extendido graças a bibliotecas que contém a definição de algumas funções:

- A biblioteca padrão adiciona várias funcionalidades ditas padrão ao Python. Dentro desta biblioteca existem vários módulos:

- O módulo `math` onde estão definidas várias funções como o seno, a exponencial, o logaritmo, a raiz quadrada entre outras (ver a ajuda do `math`) e também duas constantes o π e a constante de euler e . Veremos na secção 3.2.1 como utilizar este módulo. Só para números reais;
- O módulo `cmath` permite funcionalidade parecidas com o módulo `math` mas definidas para números complexos;
- O módulo `random` permite-nos usar números pseudo-aleatórios;
- O módulo `time` permite várias operações com o tempo;
- A biblioteca `matplotlib` que contém o módulo `pylab` que define várias rotinas para desenhar gráficos, ver secção 3.2.2;
- `numpy` estende as potencialidades do Python, definindo vectores, matrizes multi-dimensionais bem como funções que operam nesses objectos. Existe ainda o `scipy` que o complementa;

Para aceder a um módulo usamos o comando:

```
#Primeiro importamos o módulo  
import modulo  
  
#Para usar uma função presente no módulo, escrevemos:  
modulo.funcao(x)
```

ou então

```
#Importamos uma função presente no módulo  
from modulo import funcao  
  
#Para usar essa função presente no módulo, escrevemos:  
funcao(x)
```

A primeira importa todo o módulo ao início enquanto que a segunda apenas importa uma função presente no módulo. A escolha de um dos métodos depende da nossa intenção.

3.2.1 O módulo `math`

Um dos módulos que nós mais vamos usar é o `math`, dele fazem parte as seguintes funções:

Função	Comando
Coseno	cos()
Seno	sin()
Tangente	tan()
Arco-coseno	acos()
Arco-seno	asin()
Arco-tangente	atan()
Exponencial	exp()
Logaritmo	log()
Raiz Quadrada	sqrt()
Potência (x^y)	pow(x,y)
Valor absoluto	fabs()
Arredondamento por cima	ceil()
Arredondamento por baixo	floor()
Constante de Euler	e
π	pi

Duas notas. O seno, coseno e tangente usam radianos. No caso do logaritmo não neperiano usa-se log(base)

Vamos ver alguns exemplos de utilização:

```
#coding=utf8
import math

print "O coseno de pi sobre 3 é:",math.cos(math.pi/3)
```

ou então

```
#coding=utf8
from math import pow, pi

print "A raiz cúbica de pi é:",pow(pi,1./3.)
```

Exercício 3.2. *Imprima uma tabela com o coseno entre 0 e 1.*

3.2.2 O módulo pylab

O nosso próximo passo vai ser aprender a fazer gráficos.

Vejamos um exemplo de utilização:

```
#coding=utf8
import pylab

#Criamos duas listas com os valores de x e de y
lista_X=[1,2,3,4,5,6,7,8,9,10]
lista_Y=[1,3,7,10,5,7,2,0,-2,0]

#Criamos um gráfico com as duas listas
```

```
pylab.plot(lista_X, lista_Y, "--g")

#Damos-lhe o título de "O meu primeiro grafico!"
pylab.title("O meu primeiro grafico!")

#Gravamos o gráfico no ficheiro grafico.png
pylab.savefig("grafico.png")

#Desenhamos o gráfico no ecrã
pylab.show()
```

O módulo pylab contém muitas funções já pré-definidas, dentro das quais encontramos:

- `plot(X,Y,opções)`, desenha um gráfico com as abcissas X, as ordenadas Y. Nas opções podemos definir o estilo, a cor, a espessura, o nome entre outras propriedades da linha (ver a ajuda de `pylab.plot`). De notar que ele desenha o gráfico na memória mas não o mostra nem grava nenhum ficheiro. Podemos desenhar várias linhas.;
- `show()`, comando que usaremos quase sempre no fim do programa. Ele mostra os gráficos que nós desenhamos;
- `savefig("figura")`, grava o ficheiro em formato png com o nome `figura.png`;
- `title("Titulo giro")`, dá ao gráfico o título: "Titulo giro";
- `xlabel("x_nome")`, dá um nome ao eixo das abcissas;
- `ylabel("y_name")`, o equivalente para o eixo das ordenadas;
- `axis([x_min,x_max,y_min,y_max])`, define o tamanho dos eixos, permite outras opções como "auto" que permite ao Python de escolher os intervalos;
- `xlim()`, o mesmo que `axis()` mas só para o eixo de x;
- `ylim()`, o mesmo que `axis()` mas só para o eixo de y;
- `legend()`, cria uma legenda. Atenção que o nome que aparece é definido no `plot()` (ou no `subplot()`);
- `subplot()`, cria uma grelha de gráficos, útil para quando se quer desenhar vários gráficos lado a lado;
- `hist()`, faz um histograma;
- `figure()`, cria um outro gráfico. Útil quando se quer fazer vários gráficos, mas em janelas diferentes. Quando se faz `show()` as várias figuras vão aparecer. Pode ser escrito na forma `fig=figure()`, agora `fig` representa esta figura;

- `close()`, serve para recomençar um gráfico. Pode ser usado em conjugação com o `figure()`, por exemplo, depois de fazer 5 figuras (`fig1`, `fig2`, `fig3`, `fig4` e `fig5`) e de as gravar todas em ficheiros, queremos ver todas menos a 4ª. Nesse caso fazemos `close(fig4)`, e todas ficam activas excepto a 4ª.

Exercício 3.3. *Desenhe o gráfico do seno de 0 a π . Faça com 5, 100 e 1000 pontos.*

Exercício 3.4. *Embeleze o gráfico com título, legenda, cores...*

3.3 Trabalhar com ficheiros

Frequentemente, necessitamos de guardar os resultados dos nossos programas em ficheiros de texto. Ou então ir buscar dados a um ficheiro... Para abrir um ficheiro (quer ele exista ou não) usa-se o comando:

```
#coding=utf8
# Abrimos um ficheiro chamado dados.txt
fic=open("dados.txt", "w")

# Escrevemos uma frase no ficheiro
fic.write("Galileu Galilei morreu no ano:\n")

# Para escrever um número temos primeiro que convertê-lo em cadeia de caracteres
s=str(1642)

# E finalmente escrevê-lo no ficheiro
fic.write(s)

# Fechamos o ficheiro
fic.close()
```

Agora sempre que quisermos chamar o ficheiro usamos “fic”. Devem ter reparado no “w” que vem do inglês write, isto quer dizer que abrimos um ficheiro para escrever nele (caso já exista um ficheiro com esse nome o Python apaga todo o seu conteúdo). No lugar do “w”, poderíamos usar um “r” do inglês read e nesse caso só poderíamos ler o ficheiro. Ou “a” de append, logo qualquer coisa que escrevamos vai imediatamente para o fim do ficheiro. E “r+” que nos permite ler e escrever.

Para ler o ficheiro podemos usar:

```
#coding=utf8
# Abrimos o ficheiro dados.txt em modo de leitura
fic=open("dados.txt", "r")

# Lemos a primeira e depois a segunda linha
print fic.readline()
print fic.readline()
```

```
# Fechamos o ficheiro  
fic.close()
```

O comando `fic.readline()` lê uma linha e passa para a seguinte. Já o comando `fic.read()` lê o ficheiro completo.

Exercício 3.5. *Criar uma tabela com os pontos do gráfico do seno entre 0 e π com mil pontos. Imprimi-la num ficheiro.*

Capítulo 4

Soluções

4.1 Soluções do capítulo 2

1. Basta escrever no interpretador $4.5*3.14+3.4**2.1$ o que resulta em 27.194889045094623.
2. O primeiro vale 1 e o segundo vale 1.505. O que acontece é que no primeiro caso ele está a dividir dois números inteiros e então o resultado também vai ser um inteiro.
3. A resposta é $7-2354785\%7$ que é igual a 1.
4. m não muda, logo obtém-se 3.
5. frase="Das uvas se faz vinho!".
6. frase=frase[:4]+"macas"+frase[8:16]+"cidra!".
7. Se criarmos a lista A=[1,2,3] e B=[4,5,6], A.append(B) vai-nos dar A=[1,2,3,[4,5,6]], ou seja, o quarto elemento da nova lista A vai ser [4,5,6].
8. Por exemplo X=range(1,51).
9. Código:

```
#coding=utf8
n=3
m=2

print "Se n=",n,"e m=",m," , então nxm=",n*m
```

10. ...
11. Código:

```
#coding=utf8
# Começamos por pedir os dois valores ao utilizador
x_min=input("Escreve o primeiro número: ")
x_max=input("Escreve o último número: ")

# Imprimimos o resultado
print range(x_min,x_max+1)
```

12. Passo por passo:

- (a) Criamos uma variável numérica com valor 56;
- (b) Transformamo-la numa cadeia que vale "56";
- (c) Duplicamos a cadeia: "5656";
- (d) Transformamos a cadeia no número 5656;
- (e) Duplicamos o número para 11312 e imprimimos.

13. Código:

```
#coding=utf8
# Criamos a função factorial
def fac(n)
    if n==1:
        return 1
    return n*fac(n-1)

# Pedimos um número ao utilizador
m=input("Escreva um número: ")

# Devolvemos o seu factorial
print m, "factorial vale", fac(m)
```

14. Código:

```
#coding=utf8
# Pedimos um número
n=input("Escreva um número inteiro: ")

# Verificamos se ele é par ou impar
if n%2==0:
    print n, "é par."
else:
    print n, "é impar."
```

15. Código:

```
#coding=utf8
# Pedimos ambos os números
n=input("Escreva um número: ")
m=input("E um segundo número: ")

if (n>0 or m>0) and n*m<=0:
    print "Um e só um é positivo."
```

16. Código:

```
#coding=utf8
# Criamos os naipes e os valores das cartas:
naipe=["paus","ouros","copas","espadas"]
valor=["Ás","Duque","Terno","Quadra","Quina","Sena","Bisca","Valete","Dama","Rei"]

# Imprimimos as cartas todas
for r in valor:
    for s in naipe:
        print r,"de",s
```

17. Código:

```
#coding=utf8
# Pedimos um número
n=input("Escreva o número natural supostamente primo: ")

# p é uma variável auxiliar que valerá um se n não for primo.
p=0

# Verificamos se n é maior que 1 e inteiro
if n>1 and n%1==0:

# Para todos os i entre 2 e n/2 testamos se é divisível
    for i in range(2,n/2+1):
        if n%i==0:
            print "O número",n,"é divisível por",i
            p=1
            break

# Se p continuar nulo é porque n é primo
    if p==0:
        print "O número",n,"é primo."

# Tratamos o caso da unidade à parte
    elif n==1:
        print "A unidade não é um número primo."

# Só nos resta os número não inteiros positivos
```

```

else:
    print "O número não é inteiro positivo."

```

Como podem verificar este método é lento para números muito grandes, pois ele tem que construir uma lista do tamanho de n mesmo que o número seja par. Um ciclo while (que aparece no capítulo a seguir) seria mais adequado:

```

#coding=utf8
# Pedimos um número
n=input("Escreva o número natural supostamente primo: ")

def primo(n):
    i=2
    while(i<=n/2):
        if n%i==0:
            print n,"é divisível por",i
            return 0
        i=i+1
    return 1

# Verificamos se n é maior que 1 e inteiro
if n>1 and n%1==0:
    if primo(n)==1:
        print n,"é primo."
    elif n==1:
        print "A unidade não é um número primo."
    else:
        print "O número não é inteiro positivo."

```

18. Código:

```

#coding=utf8
# Vamos procurar primos até a N_max
N_max=2000

# Verifica se n é primo. Devolve 1 se n for primo e 0 senão
def primo(n):
    for i in range(2,n/2+1):
        if n%i==0:
            return 0
    return 1

# O principal do programa
for n in range(2,N_max+1):
    if primo(n)==1:
        print n

```

19. Código:

```
#coding=utf8
# n é o número a calcular a raiz quadrada, e é uma espécie de precisão
n=5
e=0.0005
m=1
print "Queremos calcular a raiz quadrada de",n

# Enquanto a aproximação for má, ele continua
while m**2>e**2:
    p=input("Escreva um número: ")
    m=p**2-n
    print "A sua previsão falhou por:"m

# Imprime o resultado
print "A raiz quadrada é",p
```

20. Código:

```
#coding=utf8
# n é o número a calcular a raiz quadrada, e é uma espécie de precisão
n=5
e=0.000005
r=1
print "Queremos calcular a raiz quadrada de",n

# Enquanto a aproximação for má, ele continua
while (r**2-n)**2>e**2:
    r=0.5*(r+n/r)

# Imprime o resultado
print "A raiz quadrada é",r
```

4.2 Soluções do capítulo 3

1. Código:

```
#coding=utf8
# Vamos buscar o coseno e o pi
from math import cos, pi

# Definimos o número de pontos
p=20

# Imprimimos o resultado
for i in range(p):
    print cos(i*pi/p)
```

2. Código:

```
#coding=utf8
# Vamos buscar o coseno e o pi e carregamos o módulo pylab
from math import cos, pi
import pylab

# Definimos o número de pontos
p=5

# Criamos X e Y
Y=[]
X=[]
for i in range(p):
    Y.append(cos(i*pi/p))
    X.append(i*pi/p)

# Desenhamos o gráfico
pylab.plot(X,Y)
pylab.show()
```

3. Aquilo a que chamamos embelezar o gráfico é extremamente subjectivo. Aqui vão algumas dicas:

```
#coding=utf8
from math import cos, pi
import pylab
p=1000
Y=[]
X=[]
for i in range(p):
    Y.append(cos(i*pi/p))
    X.append(i*pi/p)

# Cria uma linha de nome cos, com espessura 2, tracejada e vermelha
pylab.plot(X,Y,"--r",label="cos",linewidth=2)

# Dá um título
pylab.title("Grafico do coseno de x")

# Nomeamos o eixo das abcissas de x
pylab.xlabel("x")

# Nomeamos o eixo das ordenadas de y e rodamos o nome
pylab.ylabel("y",rotation="horizontal")

# Criamos uma legenda
pylab.legend()
pylab.show()
```

4. Código:

```
#coding=utf8
# Vamos buscar o coseno e o pi
from math import cos, pi

# Criamos um ficheiro
fic=open("dados.txt","w")

# Definimos o número de pontos
p=1000

# Imprimimos o resultado
for i in range(p):
    s=str(i*pi/p)+" "+str(cos(i*pi/p))+ "\n"
    fic.write(s)

# Fechamos o ficheiro
fic.close()
```

Capítulo 5

Problema do bêbado

Vinho branco ou vinho tinto?

Capítulo 6

Da ordem ao caos

Segundo a física clássica o nosso mundo é determinístico. Ou seja, se nós repetirmos a mesma experiência nas mesmas condições vamos obter exactamente os mesmos resultados. Porém não somos capazes de prever tudo. Porquê? Esta dificuldade em prever certos fenómenos surge, de entre outras causas, do caos.

Para perceber donde surge o caos, propomos aqui um exemplo simples.

6.1 O problema da pirâmide

Imaginemos um pequeno jogo.

Temos uma pirâmide de tubos, tal como mostra a figura 6.1, abertos em baixo e em cima. Deixamos cair uma bola por entre os tubos e deixamo-la cair por entre os tubos até que por fim ela “se decida” a parar numa certa caixa. Agora queremos adivinhar que caixa é essa.

Infelizmente o problema é ainda muito complicado para ser tratado com todo o pormenor em apenas uma semana. Por isso, em 2008, decidimos simplificar o modelo desprezando a gravidade.

6.1.1 Caixa final

Para calcular a que caixa a bola chega utilizamos, em 2008, o seguinte programa:

```
#coding=utf8

x=0.59864 #posição de entrada
b=5.12653 #comprimento do tubo
w=.301 #tangente do ângulo inicial
n=100 #nº de níveis
#a espessura dos tubos é 1
```

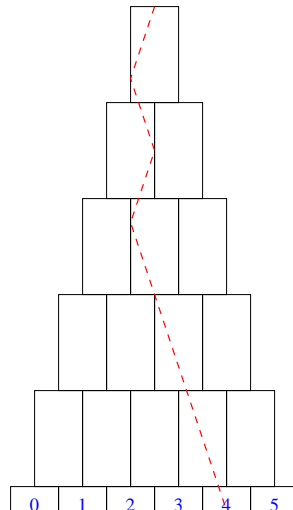


Figura 6.1: Neste exemplo, a bola cai no primeiro tubo e vai batendo nas várias paredes até chegar à caixa 4.

```
num=0 #indicador do número da caixa

# saida(x,w) é uma função auxiliar que
# calcula a posição e o ângulo de saída para cada tubo
# sabendo a posição e o ângulo de entrada neste

def saida(x,w):
    y=w*b+x #posição final se o tubo não tivesse paredes
    n=int(y) #devolve a parte inteira de y
    d=y-n #
    if y>=0:
        if n%2==0:
            wf=w
            xf=d
        else:
            wf=-w
            xf=1-d
    else:
        if n%2==0:
            wf=-w
            xf=-d
        else:
            wf=w
            xf=1+d
    return [xf,wf]

# Calcula fila por fila onde está a bola
```

```

for j in range(n):
    r= saida(x,w)
    if r[0]>0.5:
        x=r[0]-0.5
        num=num+1
    else:
        x=0.5+r[0]
    w=r[1]

```

```

#resultado final
print num

```

6.1.2 Trajectória da bola

Para desenhar a trajectória bastou-nos registar a posição a cada nível e no fim fazer um gráfico:

```

#coding=utf8
import pylab
import math

x=0.59864 #posição de entrada
b=5.12653 #comprimento
w=.301 #tangente do ângulo inicial
n=100 #nº de níveis

#a espessura dos tubos é 1

titulo="Tangente do angulo =" +str(w) #titulo do gráfico
nome="traj_"+str(int(1000*w)) #nome do ficheiro

num=0 #indicador do número da caixa

Pos_x=[] #abscissa = número da caixa
Pos_y=[] #ordenada = nível

#e=0.01

#calculo a posição e o ângulo de saída para cada tubo
#sabendo a posição e o ângulo de entrada
def saida(x,w):
    y=w*b+x
    #int() devolve a parte inteira de y
    n=int(y)
    d=y-n
    if y>=0:
        if n%2==0:
            wf=w
            xf=d

```

```

    else:
        wf=-w
        xf=1-d
    else:
        if n%2==0:
            wf=-w
            xf=-d
        else:
            wf=w
            xf=1+d
    return [xf,wf]

#calcula fila por fila onde está a bola
for j in range(n):
    r= saida(x,w)
    #b=b+e #varia o tamanho do cilindro, aumenta a imprevisibilidade
    if r[0]>0.5:
        x=r[0]-0.5
        num=num+1
    else:
        x=0.5+r[0]
    w=r[1]
    Pos_y.append(n-j) #ordenadas da trajectória
    Pos_x.append(num+(n-j)/2.) #abcissas da trajectória

#desenha a trajectória
pylab.plot(Pos_x,Pos_y)
pylab.axis([0,n,0,n])
pylab.title(titulo)
pylab.savefig(nome)
#pylab.show()

```

Podemos ver na figura 6.2 dois exemplos de trajectórias.

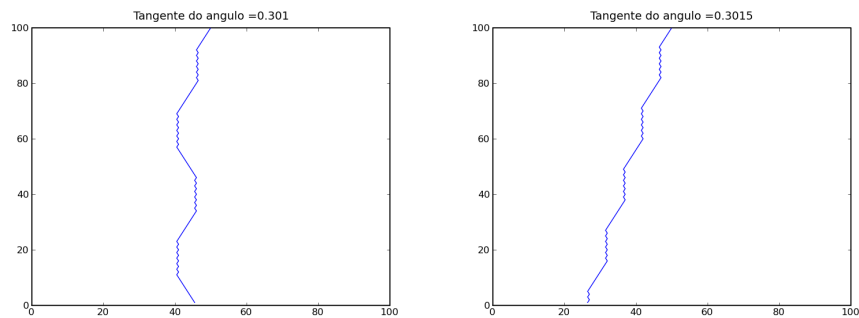


Figura 6.2: Trajectórias para duas configurações quase idênticas.

6.1.3 Variando o ângulo

Para melhor testar a dependência do modelo nas condições iniciais, corremos o programa variando o ângulo inicial. Nesse intuito alteramos o nosso programa:

```
#coding=utf8
import pylab
import math

x=0.59864 #posição de entrada
B=5.12653 #comprimento
w_min=-1.343 #tangente do ângulo inicial mínimo
w_max=1.124 #tangente do ângulo inicial máximo
n=100 #n° de níveis
P=3000 #número de lançamentos

#a espessura dos tubos é 1

Pos_x=[] #abcissa = tangente inicial
Pos_y=[] #ordenada = número da caixa

#e=0.01

#calculo a posição e o ângulo de saída para cada tubo
#sabendo a posição e o ângulo de entrada
def saida(x,w):
    y=w*B+x
    #int() devolve a parte inteira de y
    n=int(y)
    d=y-n
    if y>=0:
        if n%2==0:
            wf=w
            xf=d
        else:
            wf=-w
            xf=1-d
    else:
        if n%2==0:
            wf=-w
            xf=-d
        else:
            wf=w
            xf=1+d
    return [xf,wf]

#percorre os vários ângulos
for k in range(P):
    w=w_min+(w_max-w_min)/P*k
```

```

Pos_x.append(w) #tangente do ângulo de partida
num=0
b=B
#calcula fila por fila onde está a bola
for j in range(n):
    r= saida(x,w)
    #b=b+e #varia o tamanho do cilindro, aumenta a impresibilidade
    if r[0]>0.5:
        x=r[0]-0.5
        num=num+1
    else:
        x=0.5+r[0]
        w=r[1]

Pos_y.append(num) #número da caixa

#desenha o gráfico
pylab.plot(Pos_x,Pos_y,".")
pylab.xlabel("tangente do angulo inicial")
pylab.ylabel("numero da caixa final")
pylab.title("O numero da caixa final em funcao da tangente\ncom a altura da caixa depende
pylab.axis([-1.4,1.2,0,100])
#pylab.savefig("caos")
pylab.show()

```

Isto resulta no gráfico .

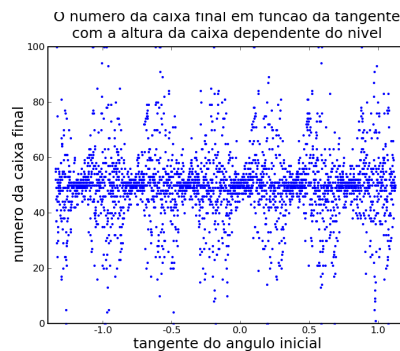


Figura 6.3: Gráfico da variação do número da caixa final em função da tangente do ângulo inicial.

6.1.4 Perguntas em aberto

1. O que acontece se em vez de variarmos o ângulo inicial variarmos a posição inicial?

2. Será que podemos considerar a gravidade? Vai aumentar ou diminuir a imprevisibilidade?
3. Existe alguma forma de calcular a imprevisibilidade? Calcula duas trajectórias começando em condições semelhantes (mas não exactamente iguais) e veja como divergem. Repete a experiência várias vezes.
4. E se o tamanho da caixa fosse aleatório, mas pré-definido?
5. Qual é a probabilidade de a bola cair numa determinada posição?
6. Imaginemos a mesma experiência, mas na qual a bola “escolhe” ir para o tubo à esquerda ou à direita aleatoriamente. Como se comparam os resultados de ambas as experiências?

6.2 Outros modelos

Na preparação deste projecto foram investigados vários modelos com características semelhantes. O modelo da pirâmide foi escolhido por ser o mais adequado às pretensões da Escola de Verão, mas existem outros igualmente interessantes. Nesta secção fazemos uma pequena digressão sobre esses modelos.

6.2.1 Sistemas dinâmicos e coelhos

Numa certa ilha deserta existe uma população de coelhos, a densidade de coelhos no momento n é dada por u_n . É normal considerar que a taxa de natalidade dos coelhos seja proporcional à quantidade de coelhos existentes ($\sim u_n$). Por outro lado, esperamos que a quantidade de coelhos dependa da quantidade de comida disponível. Por sua vez a quantidade de comida depende da concentração de coelhos ($\sim (1 - u_n)$). Podemos então dizer que a concentração na segunda geração de coelhos é descrita pela equação:

$$u_{n+1} = A u_n (1 - u_n)$$

Apesar de parecer tão simples, esta equação esconde propriedades bastante interessantes. Para começar tem dois pontos fixos: $u = 0$, se não há coelhos no início não há procriação; $u = 1 - A^{-1}$. Para $A < 3$ (valor aproximado) o sistema converge sempre, depois tem um intervalo em que o sistema alterna entre dois pontos, três pontos... assim por diante, até que a partir de 3.6 ele começa a visitar todos os pontos entre 0 e 1. Como se pode ver no gráfico 6.4.

O código usado foi:

```
#coding=utf8
```

```
# Este programa simula uma população de coelhos
#
```

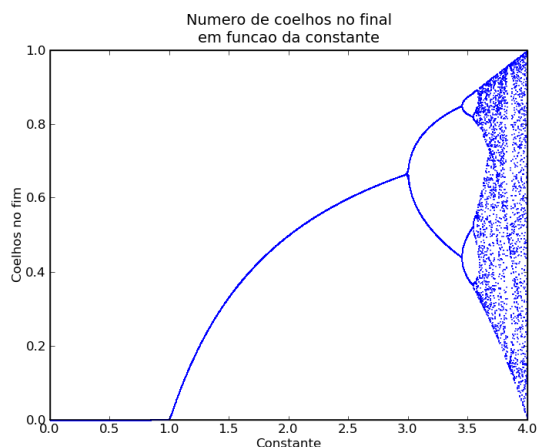


Figura 6.4: Quando se varia a constante A , obtém-se resultados completamente diferentes, desde regiões convergentes até outras que são verdadeiramente caóticas.

```
# A taxa de natalidade dos coelhos numa sociedade é proporcional
# ao número de coelhos e à quantidade de comida, esta é
# proporcional também ao número de coelhos, ficamos assim com uma
# uma equação do tipo  $u \rightarrow A u (1-u)$ 

import pylab

num=200 #quantidade de interações
Q=4 #1-Experiência Simples; 2-Variar número de coelhos;
    #3-Variar constante; 4-Varia ambos

N_A=2000 #número de passos (variarmos a constante)
N_x=20 #número de passos (variarmos num de coelhos)

A_min=0. #constante mínima para 3
A_max=4. #constante máxima para 3
A=[1.0,2.0,3.0] #constante para 1 e 2

n_min=0.1 #número de coelhos mínimo para 2
n_max=0.9 #número de coelhos máximo para 2
n=[0.7] #número de coelhos para 1 e 3

#cálculo da interação seguinte
def inter(a,u):
    return a*u*(1-u)

# Calcula o número de coelhos
# Desenha o gráfico da variação desse número em função do tempo
```

```

def Simples():
    X=range(num)
    Y_num=[]
    g=0
    for a in A:
        for x in n:
            Y_num.append([])
            s="numero="+str(x)+" e A="+str(a)
            for j in range(num):
                x=inter(a,x)
                Y_num[g].append(x)
            pylab.plot(X,Y_num[g],",",label=s)
            g+=1
    pylab.title("Numero de coelhos por interacao.")
    pylab.xlabel("Interacao")
    pylab.ylabel("Numero de coelhos")
    pylab.legend()

#      Calcula o número de coelhos depois de num interações
#      Para diferentes quantidades iniciais de coelhos

def Quantidade():
    for a in A:
        X_in=[]
        Y_fin=[]
        for i in range(N_x):
            x=n_min+i*1.*(n_max-n_min)/N_x
            X_in.append(x)
            for j in range(num):
                x=inter(a,x)
                Y_fin.append(x)
            s="A="+str(a)
            pylab.plot(X_in,Y_fin,",",label=s)
    pylab.title("Numero de coelhos no final\ndependendo do numero inicial")
    pylab.xlabel("Coelhos no inicio")
    pylab.ylabel("Coelhos no fim")
    pylab.legend()

#      Calcula o número de coelhos depois de num interações
#      Para diferentes constantes

def Constante():
    for x in n:
        X_a=[]
        Y_fin=[]
        s="numero inicial = "+str(x)
        for i in range(N_A):
            a=A_min+i*1.*(A_max-A_min)/N_A
            X_a.append(a)
            u=x

```

```

        for j in range(num):
            u=inter(a,u)
            Y_fin.append(u)
        pylab.plot(X_a,Y_fin,"",label=s)
pylab.title("Numero de coelhos no final\nem funcao da constante")
pylab.xlabel("Constante")
pylab.ylabel("Coelhos no fim")
pylab.legend()

#      Calcula o número de coelhos depois de uma interações
#      Para diferentes constantes
#      É repetida para diferentes quantidades iniciais

def Ambos():
    X_a=[]
    Y_fin=[]
    for i in range(N_A):
        a=A_min+(A_max-A_min)*i*1./N_A
        for k in range(N_x):
            x=n_min+k*1.*(n_max-n_min)/N_x
            X_a.append(a)
            for j in range(num):
                x=inter(a,x)
            Y_fin.append(x)
    pylab.plot(X_a,Y_fin,"")
    pylab.title("Numero de coelhos no final\nem funcao da constante")
    pylab.xlabel("Constante")
    pylab.ylabel("Coelhos no fim")

#escolhe que rotina seguir
if Q==1:
    Simples()
elif Q==2:
    Quantidade()
elif Q==3:
    Constante()
else:
    Ambos()

#desenha o gráfico
pylab.savefig("coelhos")
pylab.show()

```

6.2.2 Mapa a 2 dimensões

Um mapa a 2 dimensões consiste num conjunto de duas variáveis. Em cada passo as duas variáveis são actualizadas em simultâneo, vejamos alguns exemplos que apresentam caos:

1. Mapa de Duffing:

$$\begin{cases} x_{n+1} &= y_n \\ y_{n+1} &= -b x_n + a y_n - y_n^3 \end{cases}$$

2. Mapa de Hénon:

$$\begin{cases} x_{n+1} &= y_n + 1 - a x_n^2 \\ y_{n+1} &= b x_n \end{cases}$$

3. Mapa de Tinkerbell:

$$\begin{cases} x_{n+1} &= x_n^2 - y_n^2 + a x_n + b y_n \\ y_{n+1} &= 2 x_n y_n + c x_n + d y_n \end{cases}$$

Fica a cargo dos alunos mais interessados em programar estes sistemas e escolher as informações relevantes a retirar daqui.

6.2.3 Funções por ramos

Uma função por ramos é uma função que tem dois ou mais comportamentos distintos. Quando iteradas várias vezes, algumas delas ganham um carácter imprevisível.

Vejamos alguns exemplos:

1. Mapa tenda:

$$x_{n+1} = \begin{cases} \mu x_n & \text{se } x_n < \frac{1}{2} \\ \mu(1 - x_n) & \text{se } x_n > \frac{1}{2} \end{cases}$$

2. Mapa diático:

$$x_{n+1} = \begin{cases} 2 x_n & \text{se } x_n < \frac{1}{2} \\ 2 x_n - 1 & \text{se } x_n > \frac{1}{2} \end{cases}$$

A programação destes modelos é idêntica às anteriores. Sendo assim, fica como exercício aos eventuais leitores que chegaram até aqui.

6.2.4 Geradores de números aleatórios

Os computadores actuais são incapazes de produzir números aleatórios, uma vez que um computador só utiliza fenómenos determinísticos. No entanto, é possível produzir números aparentemente aleatórios, que são chamados números pseudo-aleatórios. Não é de estranhar que os algoritmos usados sejam caóticos. Vejamos alguns exemplos:

1. Método de Van Neumann:

- (a) Seja v_n um número de n dígitos;
- (b) Calculamos m^2 ;

(c) E isolamos os n dígitos centrais, temos assim o novo número v_{n+1} .

2. Método de Fibonacci, seja m um primo grande:

$$x_{n+1} = (x_n + x_{n-k}) \% m$$

em que k é um número natural.

3. Método de Lehmer, sejam a , b e c números grandes e primos entre si:

$$u_{n+1} = (a u_n + b) \% c$$

O aluno interessado não terá dificuldade em programar estes geradores.

Apêndice A

Linguagens de Programação

Neste apêndice vamos expôr algumas linguagens de programação que achamos útil saber que existem, por uma questão de cultura geral. Antes de passarmos a isso, vamos introduzir mais alguns conceitos.

Código-máquina Na verdade, os computadores falam apenas uma linguagem - código máquina. Esta linguagem não é universal - o código máquina que corre num computador normal não é o mesmo que corre num telemóvel¹, por exemplo. Assim, para ser executado qualquer programa, mesmo escrito noutra linguagem, este tem eventualmente de ser traduzido para código máquina.

Linguagens-compiladas-e-interpretadas A tradução para código máquina pode ser *à priori*, no caso de uma linguagem compilada, sendo o programa distribuído já em código máquina, ou pode ser em tempo real, durante a execução do programa, no caso de uma linguagem interpretada. Cada tipo tem as suas vantagens e desvantagens - por exemplo, as interpretadas são em geral mais lentas, mas quando existem erros no programa estes são mais fáceis de identificar do que nas compiladas. MAL.REVER ISTO.

Alto-nível-e-baixo-nível É frequente classificar as linguagens quanto ao seu nível - quanto mais baixo for o nível de uma linguagem, mas próximo esta a sua maneira de funcionar da do código máquina. Numa linguagem de alto nível esta maneira de funcionar está completamente abstraída, não tendo o programador de se preocupar com os detalhes do funcionamento do computador (idealmente). A diferença será mais aparente quando dermos exemplos de linguagens.

Vamos expôr um programa exemplo para cada uma das linguagens: C, Java, Fortran, Scheme, Perl, Ruby e Python.

¹Um telemóvel é praticamente um computador. Muitos deles até permitem programar em python!

C

```
#include <stdio.h>
int main(void)
{
    printf("Hello, world!\n");
    return 0;
}
C é blablabla
```

Fortran

```
program hello
    print*, 'Hello, world!'
end
Fortran blablabla
```

Java

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello, world!");
    }
}
Java blablabla
```

Perl

```
print "Hello, world!\n";
perl blabla
```

Ruby

```
puts 'Hello, world!'
ruby blablabla
```

Python

```
print 'Hello, world!'
```

Scheme

```
(display "Hello, world!\n")
Scheme blablabla
```

Apêndice B

Bibliotecas úteis e exemplos

No capítulo 3 já foram usadas bibliotecas para poder usar funções matemáticas ou fazer gráficos. Neste capítulo vamos mostrar vários exemplos de código que usam bibliotecas que podem ser úteis em várias situações.

- 1) Gráfico animado
- 2) Mais um gráfico animado
- 2) PyGame - jogo simples?
- 3) Interface gráfico básico com Tkinter