# Curso de Python

Gustavo Noronha Silva

11 de agosto de 2005

# Sumário

# Nota de Copyright

# Agradecimentos

Muito obrigado a Allan Douglas <allan_douglas@gmx.net> pela grande ajuda na escrita desse curso.

# Capítulo 1

# Introdução

A programação nos dias atuais está centrada em uma briga de gigantes que pretendem impôr seus padrões para substituir a linguagem conhecida como "padrão da indústria", C++.

Correndo por fora estão linguagens relativamente pouco conhecidas e aclamadas, talvez por serem fruto do esforço de grandes mestres da computação (como as primeiras linguagens) e não por grandes empresas.

Python é uma dessas linguagens e traz uma simplicidade indiscutível, ao mesmo tempo em que, apesar de ser uma linguagem interpretada, é extremamente veloz.

Aqueles que já programam há tempos com linguagens cheias de controle sintático vão certamente se sentir perdidos num primeiro contato, mas perceberão que Python se adapta muito bem ao modelo mental do programador e vai se sentir falando com a máquina em pouco tempo.

# Capítulo 2

# Revisão de Conceitos

Python é uma linguagem simples, mas é construída em uma base teórica e técnica muito complexa. Ela eleva a orientação a objetos, em alguns casos, ao extremo.

Vamos dar uma revisada em alguns conceitos importantes que nos subsidiarão no aprendizado de Python.

## 2.1 Linguagem Interpretada vs Compilada

Python, como já foi dito, é uma linguagem interpretada, como Perl, Shell Script, Batch Scripts, entre outras. Isso significa que não é necessária a compilação do código para que ele seja executado e isso trás várias vantagens e desvantagens embutidas.

Linguagens compiladas normalmente são mais rápidas, porque o código já está num formato que o computador entende. Linguagens interpretadas costumam funcionar de uma ou outra maneira:

- Compilação Just-In-Time
- Interpretação pura ou em Bytecode

O Python pode funcionar das duas formas. Vamos usar mais o segundo modelo durante o curso, mas não se esqueça de conferir o compilador JIT do Python.

## 2.2 Tipagem Forte

Python é uma linguagem de tipagem forte. Isso significa que se uma variável adquire um determinado tipo não deixa mais de ser daquele tipo a menos que seja recriada. Isso o torna diferente de um script Shell, por exemplo, em que nunca se sabe o tipo exato de uma variável.

Apesar da sua tipagem ser forte, a declaração de variáveis não é necessária e a simples atribuição de um valor serve para criar ou recriar uma variável. Leve isso em conta quando programar. Tome muito cuidado com os nomes das variáveis.

## 2.3 Orientação a Objeto

Dou uma ênfase especial a esse conceito, pois já vi muita gente dizer que uma linguagem é orientada a objetos "porque você pode criar interfaces gráficas"[1]. Não tem nada a ver.

Uma linguagem orientada a objetos coloca como centro nervoso do programa um ou mais objetos de determinada classe, ao contrário das linguagens estruturadas, em que o processo, ou as estruturas de dados são o centro e você chama funções que atuam sobre esses elementos.

Isso não significa, é claro, que não se pode criar aplicações com interfaces gráficas com Python ou com qualquer outra linguagem orientada ou não a objetos.

Python é uma linguagem orientada a objetos, e nela quase tudo é um objeto. Até mesmo as variáveis que representam os tipos mais básicos, como inteiro e caractere são objetos, têm seus métodos e propriedades.

[1] Outro conceito extremamente errado e muito difundido é que C é para aplicações comuns e C++ para interfaces gráficas. Talvez seja exatamente uma derivação do conceito errado que eu cito aqui, já que C++ é orientada a objetos e C não.

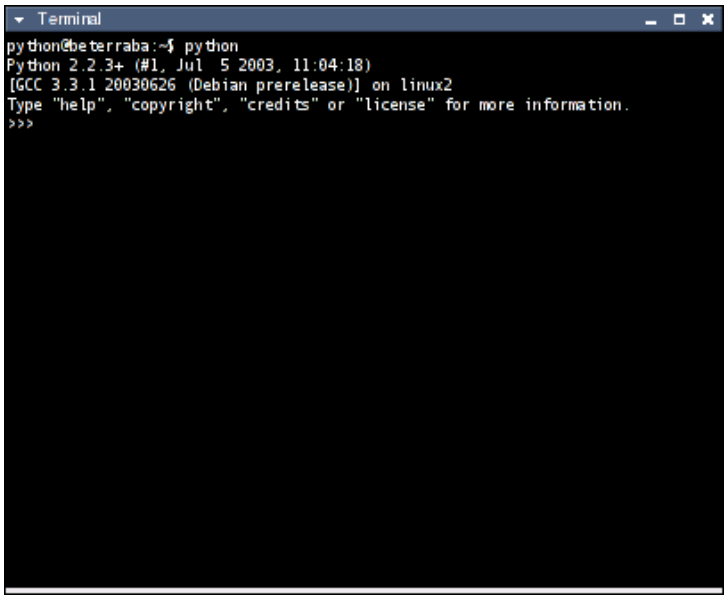
# Capítulo 3

# Mão na massa!

Vamos então começar! A primeira coisa a fazer é nos familiarizarmos com o interpretador Python, que pode nos ser muito útil a qualquer momento. Veremos como descobrir informações sobre um determinado elemento e como usar o interpretador para testarmos pequenos trechos de código.

## 3.1 O Interpretador Python

O interpretador é o programa que executa código Python. Para executá-lo basta abrir um terminal e digitar 'python'. Você verá algo desse tipo:




O que vemos nas três primeiras linhas é a apresentação do interpretador. Ele está dizendo "oi!" =D. Basicamente ele informa de que versão do Python

se trata, com que compilador foi compilado e sugere algumas chamadas básicas caso tenha problemas ou dúvidas.

O >>> é o prompt do interpretador. Podemos sair programando em Python agora mesmo. O interpretador vai executar o código que escrevermos na hora, e poderemos ver o resultado.

Já que o interpretador nos disse "oi" não sejamos mal-educados, vamos responder a ele, digitando o seguinte:

```
>>> print 'Olá, Python!'
Olá, Python!
```

Ótimo! Podemos ver que 'print' serve para mostrar mensagens na tela. Você pode usar aspas simples ou duplas para delimitar a mensagem. A função 'print' é uma excessão entre as funções do Python, já que ela não precisa de parenteses. Note, também, que não há um caractere delimitador da chamada (como "*;*" em C e Pascal).

## 3.2 Módulos, as bibliotecas do Python

No Python chamamos as coleções de código que fornecem extensões para a linguagem de módulos. Pode-se fazer uma associação às bibliotecas usadas em C (e, na verdade, algumas vezes os módulos são bibliotecas). Para usá-los, temos que importá-los. Mas cuidado com a balança comercial, hein! (duh)

O Python procura sempre proteger o chamado *espaço de nomes* e, portanto, sempre que você importar um módulo terá de usar seu nome para chamar funções e acessar propriedades que estão dentro dele. Isso pode ser familiar para quem lida com Java ou C#. Vejamos as três formas de importar módulos:

```
>>> import os
>>> os.getcwd ()
'/home/kov'
>>> from os import getcwd
>>> getcwd ()
'/home/kov'
>>> from os import *
>>> getcwd ()
'/home/kov'
```

A primeira, "import os", importa o módulo como um todo, mas exige que sempre que você quiser acessar algo que pertence ao módulo você tenha que adicionar "os." antes da função ou propriedade. O segundo, "from os import getcwd", importa somente aquela função determinada; isso pode usar menos memória e não é mais necessário usar "os." antes de chamar a função; a terceira forma é como a segunda, mas ao invés de importar uma só função, importa todas.

## 3.3 Se virando no Python

Este documento não pretende ser uma referência completa sobre o Python. Então, como obter ajuda? Como descobrir quais são as funcionalidades presentes nesta linguagem? Como "se virar"?

### 3.3.1 A função dir()

O Python tem uma função chamada *dir()*. Ela fornece à linguagem a capacidade de reflexão, presente em linguagens como Java. Isso significa que você pode listar o conteúdo de módulos e qualquer outro tipo de objeto[1]. Isso ajuda a saber sobre o que queremos ajuda, por exemplo. Vamos ver como funciona:

```
>>> dir (__builtins__)
(...)
'reduce', 'reload', 'repr', 'round', 'setattr', 'slice',
'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'unichr',
'unicode', 'vars', 'xrange', 'zip']
>>> import sys
>>> dir (sys)
(...)
'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setdlopenflags',
'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin',
'stdout', 'version', 'version_info', 'warnoptions']
>>> dir("umastring")
(...)
rip', 'replace', 'rfind', 'rindex', 'rjust', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']
```

Isso significa que o espaço de nomes *__builtins__*, carrega aquelas funções ou propriedades[2], que o módulo sys tem aquelas listadas, e que você pode usar aqueles métodos em qualquer string.

### 3.3.2 PyDoc

**pydoc** é um software que acompanha a distribuição oficial do Python. Ele permite acesso à documentação presente nas *docstrings*[3] dos objetos.

Há vários modos de utiliza-lo, um dos mais úteis é chamando-o para que monte um pequeno servidor HTTP. Digite em um terminal:

```
\$ pydoc -p 1234
pydoc server ready at http://localhost:1234/
```

[1] Módulos são objetos, depois de importados, e quase tudo em Python é um objeto, até uma string!

[2] Uma propriedade, normalmente, é uma variável.

[3] As docstrings são textos incluídos no início da definição de uma classe ou função documentando-as. Veja mais detalhes na seção 5.4 do capítulo sobre Funções, página 28.

Ele criará um servidor que pode ser acessado pelo endereço http://localhost:1234. A página é produzida automaticamente, e lista os módulos Python instalados. Clicando-se em um dos módulos pode-se visualizar quais são a funções (métodos) disponíveis, sua sintaxe e também uma breve descrição.

### 3.3.3 Nosso amigo help()

O interpretador Python oferece um mecanismo de ajuda simples e eficiente. Ele provê uma função help(), que permite acessar parte da documentação oficial Python e o PyDoc.

Ele poderá ser chamado para uma sessão interativa:

```
>>> help()
(Apresentação)
help>
```

*help>* é o prompt do help. Tudo o que você digitar a partir de agora, será interpretado pelo help. Digite 'quit' para sair. Para visualizar a documentação *docstring* de um módulo, basta digitar o nome do módulo:

```
help> os
Help on module os:

NAME
    os - OS routines for Mac, DOS, NT, or Posix depending on what system
         we're on.
(...)
```

Use as setas do teclado para ir para baixo/cima e aperte 'q' para sair. Para saber quais são os modulos disponíveis, digite 'modules':

```
help> modules

Please wait a moment while I gather a list of all available modules...

ArrayPrinter        asyncore            linuxaudiodev       sgmllib
BaseHTTPServer      atexit              locale              sha
Bastion             audiodev            logging (package)   shelve
CDROM               audioop             macpath             shlex
(...)
```

Para procurar um módulo por palavra-chave, digite 'module palavra' e o help retornará uma lista de modulos que correspondem àquela palavra.

Além da documentação dos módulos, o help permite que você obtenha ajuda em determinados tópicos. Os assuntos são variados, vão desde a descrição dos tipos básicos até como fazer o *debugging* de um programa. Digite 'topics' para ver quais são os topicos disponíveis.

Esqueceu qual a sintaxe do *if* ou de outra palavra-chave? Não tema! O help também oferece um rápido acesso à *gramática* das palavras-chave, com uma breve descrição de seu uso. No help, digite 'keywords' para saber quais são as palavras-chaves. E para acessar sua documentação, é só digitar o nome:

```
help> if
  7.1 The if statement



  The if statement is used for conditional execution:

        if_stmt   ::=    "if" expression[1] ":" suite[2]
                         ( "elif" expression[3] ":" suite[4] )*
                         ["else" ":" suite[5]]
(...)
```

O help() pode ser chamado fora de um sessão interativa. Para obter a documentação de um módulo ou função, é necessário, primeiramente, importá-lo:

```
>>> import os
>>> help(os.open)
Help on built-in function open:

open(...)
    open(filename, flag [, mode=0777]) -> fd

    Open a file (for low level IO).
```

Para acessar a ajuda das *palavras-chaves* e dos *tópicos*, é preciso chamar o help delimitando o nome do tópico ou palavra-chave com aspas:

```
>>> help('TRUTHVALUE')
  2.2.1 Truth Value Testing

  Any object can be tested for truth value, for use in an if or while
  condition or as operand of the Boolean operations below. The following
  values are considered false:
(...)
```

## 3.4 Variáveis e mais sintaxe básica

Vamos continuar usando o interpretador. Agora vamos ver um pouco sobre variáveis. Variáveis são estruturas simples que servem para guardar dados. Como eu já disse, não é necessário declarar variáveis em Python. Vamos dar uma olhada então:

```
>>> variavel = 1
>>> print variavel
1
>>> variavel
1
```

A primeira linha coloca 1 na variável de nome *variavel*. A segunda linha mostra *variavel* na tela e a terceira retorna o valor de *variavel*. É muito importante fazer distinção entre essas duas últimas.

Um programador deve saber que normalmente toda "afirmação" que é feita em um programa é, na verdade, uma expressão, na maioria das linguagens. Em Python, sempre. Sempre existe uma avaliação do valor e um retorno do valor. O sentido prático disso é que o segundo *1* não seria realmente mostrado na tela em um programa de verdade, ele seria retornado para nada.

As operações com números são feitas como em outras linguagens:

```
>>> soma = 1 + 1
>>> print soma
2
>>> mult = soma * 3
>>> print mult
6
```

Já que estamos lidando com variáveis, vamos ver como usar o print para coisas mais complexas. Vamos mostrar variáveis com ele. Existem várias formas. Observe:

```
>>> soma = 2 + 2
>>> print "A soma é: " + str(soma)
A soma é: 4
>>> print "A soma é, na verdade: %d" % (soma)
A soma é, na verdade: 4
```

A primeira forma deve ser comum a quem programa em VB ou PHP, por exemplo. A segunda deve causar boas lembranças aos programadores de C =). Eu, particularmente, prefiro a segunda forma, mas a primeira nos ensina duas coisas importantes: 1) é possível somar strings (e, veremos mais a frente, outros tipos de dados, também!) e 2) os tipos em Python são fortes, você tem que converter o inteiro para string se quiser concatenar os dois.

Vamos analizar melhor a segunda. Entre aspas temos o que chamamos de *formato*, que é a definição do que aparecerá na tela. A string "%d" significa que ali será colocado um inteiro. Depois do formato colocamos um separador (um "%") e, entre parênteses uma lista separada por vírgulas, das variáveis que queremos que substituam os códigos. Vamos ver mais um exemplo:

```
>>> meunome="Gustavo"
>>> minhaidade=20
>>> print "Oi, eu sou %s e tenho %d anos!" % (meunome, minhaidade)
Oi, eu sou Gustavo e tenho 20 anos!
```

## 3.5 Condições e Estruturas de Repetição

Antes de começarmos a seção propriamente dita, é necessário entender como o Python marca o início e o final de blocos. Aqueles acostumados com C e Pascal estarão acostumados com coisas do tipo:

```
if (variavel == 10)
{
   printf ("É 10!!!\n");
}
```

Ou

```
if (variavel = 10) then
begin
   writeln ("É 10!!!");
end;
```

O Python, para o desgosto de alguns, **não** tem estruturas sintáticas de abertura e fechamento de blocos. O Python usa a indentação para definir o início e término de blocos. O problema aqui é que muitos programadores não têm hábito de fazer uma indentação consistente. A vantagem do Python é que ele obriga o programador a ter uma indentação consistente. =)

Portanto, para começar um bloco de condição, é necessário um nível de indentação. Para indentar, pressione a tecla tab. Um exemplo:

```
>>> variavel = 10
>>> if variavel == 10:
...     print "É 10!!"
...
É 10!!
```

Duas coisas importantes a serem observadas: 1) quando o Python espera que você inicie um bloco, o interpretador muda o prompt para "..." e 2) para mostrar que um bloco acabou (no interpretador) basta dar enter sem indentar uma linha depois de ter escrito seu bloco. Num programa de verdade não é necessário adicionar uma linha em branco para terminar um bloco, basta não indentar a linha seguinte. Nós veremos isso melhor mais a frente.

Outra coisa importante: como eu disse inicialmente, no Python toda "afirmativa" é uma expressão. Mas, diferentemente do C, o Python não aceita atribuições em contextos que devem ser somente usados para expressões de condição. Por exemplo, como vimos, o Python usa o operador "==" para comparações. Se eu fizer *if variavel = 10* o interpretador irá emitir um erro, porque atribuições não são permitidas em um if.

Você pode fazer testes de condição mais complexos com Python, também:

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

```
>>> b = 6
>>> if variavel == 10 and a == 2:
...     print "aaa"
... elif b == 6:
...     print "bbb"
... else:
...     print "ccc"
...
aaa
```

Outros operadores podem ser usados, como "*or*" para relação entre comparações, e "! =", "<", ">", "<=" e assim por diante, para relação entre variáveis e valores.

Um loop *for* (para) em Python se parece muito com o estilo Pascal de *for*. Ele exige um tipo de dado mais complexo, que nós veremos mais para frente, que é a lista. Mas por enquanto vamos ver o básico do for:

```
>>> for contador in range (0, 3):
...     print contador
...
0
1
2
```

A função *range* serve para retornar uma lista que contenha o intervalo dado pelos números especificados. Preste muita atenção. O primeiro número que você especificar sempre entra no intervalo, o último nunca. Em matemática poderíamos representar isso assim: $[0, 3[$.

Um *while* segue, essencialmente a mesma lógica de qualquer outro *while* que eu já vi:

```
>>> contador = 3
>>> while contador > 0:
...     print "Python!"
...     contador = contador - 1
...
Python!
Python!
Python!
```

## 3.6 Usando um editor

Sim! Vamos parar de usar o interpretador diretamente para criar nossos códigos e vamos passar para um editor, o que nos permitirá salvar nossos programas. Isso não significa que vamos esquecer o interpretador daqui para frente: muito pelo contrário! O interpretador vai ser quem rodará, efetivamente, o programa

criado, e também o usaremos para obter ajuda e testar pequenos pedaços de código enquanto programamos.

Podemos usar qualquer editor que seja capaz de salvar texto puro. Isso significa qualquer editor, provavelmente. Muitas pessoas têm a idéia errada de que é necessário um programa específico para programar. Muitas pessoas também cometem o erro de chamar esses programas de "compiladores". Na verdade esses programas são chamados IDE's (Integrated Development Environmet). Há diversos IDE's para Python e nós conheceremos um mais à frente. No entanto, por agora, vamos escolher qualquer editor simples e começar!

Algumas convenções para as quais devemos atentar: os arquivos executáveis Python normalmente não têm extensão, para manter a coerência com o resto do sistema. Os que têm usam ".py" como extensão. Os arquivos podem ser "byte-compilados"[4] e, nesse caso, receberão a extensão ".pyc".

Outra convenção importante é que todo script Python deve começar com uma linha contendo a "string mágica" (#!) e a localização do interpretador, para que o sistema saiba como executar o script[5]. Normalmente essa linha será:

```
#!/usr/bin/python
```

Essa é uma boa hora para dizer, também, que em Python os comentários são feitos com o símbolo #. Tudo que vier depois de um # e antes de uma quebra de linha é um comentário.

Então vamos colocar nosso primeiro exemplo para funcionar... abra seu editor e escreva o seguinte:

```
#!/usr/bin/python

# define que a variavel 'variavel' conterá o valor 10
variavel = 10

if variavel == 10:     # se variável for igual a 10
   while variavel > 0: # enquanto a variável for maior que 0
      print variavel   # mostrar a variável
      variavel = variavel - 1 # e, claro, diminuir seu valor

# fim do programa!!
```

Os comentários estão aí só pra você aprender a usá-los =). Eles não são, é claro, necessários. Depois de salvar o arquivo com um nome qualquer, "teste1.py" por exemplo, abra um terminal e execute os seguintes comandos:

```
$ chmod +x teste1.py
$ ./teste1.py
```

[4] Pré-processados pelo interpretador para se tornar o "assembly" usado por ele, aumentando a velocidade de processamento sem perder a portabilidade.

[5] Isso não tem validade no Windows... somente em sistemas Unix-like, como o GNU/Linux, o MacOSX, FreeBSD, HP-UX e outros, já que isso é uma convenção Unix.

O primeiro comando dá permissões de execução ao arquivo, o segundo o executa. O resultado deve ser:

```
python@beterraba:~$ ./teste1.py
10
9
(...)
1
python@beterraba:~$
```

Hora da brincadeira!! Pegue seu editor preferido e faça testes com o que já aprendeu da linguagem!

# Capítulo 4

# Tipos de Dados

O Python fornece, além dos tipos básicos de dados, alguns tipos mais avançados e complexos que podem ser usados com muita facilidade. Quem já teve que usar uma lista encadeada sabe o quanto é complicado implementar esse tipo de estrutura do zero. O Python facilita muito as coisas, e nós vamos explorar essas capacidades agora!

## 4.1 Inteiros e Ponto flutuante

Lidar com números em Python é bem fácil. Como já vimos, o Python detecta o tipo do dado no momento em que ele é atribuído à variável. Isso significa que:

```
>>> inteiro = 10
>>> ponto_flutuante = 10.0
>>> print 'um inteiro: %d, um ponto flutuante: %f' % (inteiro, ponto_flutuante)
um inteiro: 10, um ponto flutuante: 10.000000
```

## 4.2 Strings

Lidar com strings em Python é muito simples. O que faz com que o Python "detecte" o tipo string são as aspas, simples ou duplas. Vejamos:

```
>>> meu_nome = "Gustavo"
>>> meu_nick = 'kov'
>>> print "Meu nome é: %s, meu nick é %s!" % (meu_nome, meu_nick)
Meu nome é: Gustavo, meu nick é kov!
```

Como disse anteriormente, quase tudo em Python é um objeto. Isso significa que uma string tem métodos! Sim! Observe:

```
>>> nome = "kov"
>>> nome.upper ()
```

```
'KOV'
>>> "kov".upper ()
'KOV'
```

Note, no entanto, uma coisa muito importante: uma string em Python é **sempre** uma constante! Você nunca modifica uma string, o que você pode fazer é reatribuí-la. Isso significa que se você quiser que a string continue em letras maiúsculas dali para frente você deve fazer "nome = nome.upper ()". Nós veremos mais sobre como conhecer os métodos disponíveis.

Para acessar partes específicas da string, você pode usar a notação de "fatias". É importante entender bem essa notação, porque ela é muito útil em várias oportunidades e com alguns outros tipos de dados.

```
>>> nome = 'Software Livre'
>>> nome
'Software Livre'
>>> nome[1]
'o'
>>> nome[1:5]
'oftw'
>>> nome[:-1]
'Software Livr'
```

## 4.3 Listas e Tuplas

Agora o tipo de dados que serve como vetor e listas encadeadas. Chamamos de lista uma lista que pode ser modificada dinamicamente e de tupla uma lista imóvel.

Vamos ver as operações principais com listas:

```
>>> lista = [] # os [] fazem com que 'lista' seja iniciada como lista
>>> lista.append ('kov')
>>> lista = ['kov', 'spuk', 'o_0']
>>> len(lista)
3
>>> lista.insert (2, 'agney')
>>> print lista
['kov', 'spuk', 'agney', 'o_0']
>>> lista.pop ()
'o_0'
>>> print lista
['kov', 'spuk', 'agney']
>>> lista.remove ('spuk')
>>> print lista
['kov', 'agney']
```

Note que todas as operações básicas de lista encadeada são contempladas pela implementação de listas do Python. A terceira linha mostra como inicializar uma variável com uma lista pré-criada. A quarta mostra como obter o tamanho da lista usando a função len() [1]. Nesse exemplo usamos strings na lista, mas uma lista pode conter qualquer tipo de dado, inclusive listas, possibilitando a criação simples de árvores, inclusive.

Algo importante de se notar é que dá para usar a lista do Python como pilha, usando os métodos append() e pop(). Outra derivação importante desse exemplo é ver que uma lista em Python é indexada, como um vetor. Podemos usar "lista[0]" para acessar o primeiro ítem da lista[2], e podemos usar esse índice para inserir nós em posições específicas.

As listas têm outros métodos de extremo interesse, como o sort(), que serve para ordenar a lista. É sentar e explorar as possibilidades!

As tuplas funcionam quase do mesmo modo que as listas, com a excessão de que são listas que não são modificáveis. Elas são bastante usadas como retorno de algumas funções, portanto é bom explorá-las. Você reconhece uma tupla por sua similaridade com uma lista, mas ao invés de usar [] uma tupla usa parênteses. Um exemplo rápido:

```
>>> tupla = (1, 35)
>>> tupla[0]
1
>>> tupla
(1, 35)
```

## 4.4 Dicionários

Quem já ouviu falar de *Hash tables* vai se sentir familiarizado com os *dicionários* rapidamente. Os dicionários em Python são, literalmente, estruturas de dados que permitem "traduzir" uma chave para um valor. Seu uso é muito simples:

```
>>> linguagem = {}
>>> linguagem['nome'] = 'Python'
>>> linguagem['tipo'] = 'scripting'
>>> linguagem[2] = 100
>>> linguagem[2]
100
>>> linguagem['tipo']
'scripting'
```

[1] Note que a função len() pode ser usada em outros tipos de dados, como strings, por exemplo

[2] Em Python, como em C, índices sempre começam do 0.

## 4.5 Conversão / "Casting"

Python usa tipagem forte. Isso significa que se uma variável contém um inteiro ela não pode simplesmente ser usada como se fosse uma variável de ponto_flutuante. Para esse tipo de coisas é necessário fazer uma conversão. Aqui estão exemplos:

```
>>> a = 10
>>> str (a)
'10'
>>> float (a)
10.0
>>> int ('25')
25
```

Note que uso aspas na palavra *Casting* porque o que acontece, na verdade é que um novo objeto é criado e não o mesmo objeto é "adaptado" como acontece nos castings de C, por exemplo.

# Capítulo 5

# Funções

## 5.1 Sintaxe Básica

Funções em python tem uma sintaxe muito parecida com qualquer outra linguagem. O uso de parênteses é obrigatório, mesmo que não aja argumentos, ao contrário de algumas, com a única excessão da função interna *print*.

Declarar uma função em Python é muito simples:

```
>>> def minha_funcao (argumento):
...     print argumento
...
>>> minha_funcao ('123kokoko!')
123kokoko!
```

Como pode ser visto, o bloco que implementa a função também é demarcado pela indentação.

## 5.2 Passagem Avançada de Argumentos

Os argumentos não têm tipo declarado, o que dá uma grande flexibilidade. Os argumentos podem ter valores padrão, também, e você pode dar valores específicos para argumentos opcionais fora da localização deles, nesses casos. Vejamos:

```
>>> def mostra_informacoes (obrigatorio, nome = 'kov', idade=20):
...     print "obrigatorio: %s\nnome: %s\nidade: %d" % (obrigatorio, nome, idade)
...
>>> mostra_informacoes ('123')
obrigatorio: 123
nome: kov
idade: 20
>>> mostra_informacoes ('123', idade = 10)
```

```
obrigatorio: 123
nome: kov
idade: 10
```

## 5.3 Retorno da função

Para retornar um (ou mais) valores em uma função basta usar a declaração *return*. Você pode, inclusive, retornar diversos valores e, como na passagem de parâmetros, não precisa declarar o tipo de dado retornado:

```
>>> def soma_e_multiplica (x, y):
...     a = x + y
...     b = x * y
...     return a, b
...
>>> c, d = soma_e_multiplica (4, 5)
>>> print c
9
>>> print d
20
```

## 5.4 Documentando

Uma característica interessante do Python é a possibilidade de documentar código no próprio código, sem usar ferramentas externas. Basta colocar uma string envolvida por sequências de três aspas:

```
>>> def funcao (x):
...     """
...     funcao (x) -> inteiro
...
...     recebe um inteiro, multiplica ele por si mesmo
...     e retorna o resultado
...     """
...     return x * x
...
>>> help (funcao)
Help on function funcao in module __main__:

funcao(x)
    funcao (x) -> inteiro

    recebe um inteiro, multiplica ele por si mesmo
    e retorna o resultado
```

# Capítulo 6

# Classes

Classes são como idéias. São um dos pilares da orientação a objetos. Se pensarmos como filósofos podemos associar um objeto (ou uma instância, o que é mais correto no caso do Python) a um ente concreto, um indivíduo. O exemplo clássico é pensar em *ser humano*, uma classe, e *José da Silva*, um exemplar dessa classe, uma instância dessa classe.

## 6.1 Criando estruturas em Python

No Pascal são conhecidos como *registros*, no C como estruturas ou *structs*. O Python por incrível que pareça não tem uma estrutura de dado como essa. Para implementar algo similar a um registro é necessário usar uma classe.

A sintaxe é muito simples:

```
>>> class registro:
...     nome = ""
...     idade = 0
...
>>> r = registro ()
>>> r.nome = "kov"
>>> r.idade = 10
>>> print r.nome
kov
```

A única diferença existente entre uma classe e uma estrutura, quando se usa assim, é que você pode criar novos "membros" dentro da classe a qualquer momento, bastando atribuir algo.

## 6.2 Métodos

Definir métodos em classes Python é uma questão de definir uma função dentro da classe (a indentação marca o bloco, lembre-se). O importante é que o nome

que você passar como primeiro argumento para um método representa o próprio objeto (análogo ao *this* de outras linguagens), mas não precisa ser passado como argumento quando for chamar o método. Um exemplo:

```
>>> class humano:
...     def falar (self, oque):
...             self.imprimir (oque)
...     def imprimir (self, oque):
...             print oque
...
>>> eu = humano ()
>>> eu.falar ("qualquer coisa")
qualquer coisa
```

Existe um método especial para toda classe, chamado *__init__*. Esse método é análogo ao *construtor* de outras linguagens. Ele é chamado quando você instancia um objeto, e os argumentos passados na chamada de instanciação são passados para esse método.

## 6.3 Herança

O Python permite herança de classes. Podemos, então, definir uma nova classe que herda as características de *registro* e *humano*, que definimos acima assim:

```
>>> class coisa (registro,humano):
...     def coisar (self):
...             print "eu coiso, tu coisas, ele coisa..."
...
>>> c = coisa ()
>>> c.falar ("abc")
abc
```

É claro, é possíve herdar uma classe apenas, bastando colocar seu nome entre parênteses.

# Capítulo 7

# Arquivos / IO

Lidar com arquivos em Python é coisa muito simples. Uma das primeiras coisas que eu procuro fazer quando aprendo uma linguagem é abrir, ler e gravar em um arquivo, então vamos ver aqui como fazer. Um arquivo aberto é um objeto, para criar esse objeto usando a função *open*, que será muito familiar para programadores da linguagem C:

```
>>> arq = open ('/tmp/meuarquivo.txt', 'w')
>>> arq.write ('meu teste de escrever em arquivo\nmuito legal')
>>> arq.close ()
```

O primeiro argumento de *open* é o nome do arquivo, e o segundo é o modo como ele será aberto. Os modos possíveis são: "w" para apagar o que existir e abrir para escrita (cria o arquivo se não existir); "w+" faz o mesmo que "w", mas abre para escrita e leitura; "r", abre para leitura; "a" abre o arquivo para escrita, mantendo o conteúdo e posiciona o cursor no final do arquivo; "a+" faz o mesmo que "a", mas abre para leitura e escrita.

O método *write* da instância do objeto *arq* está sendo usado para gravar a string no arquivo. Você pode usar a mesma notação usada com print entre os parâmetros do método (na verdade você pode usar a notação do print em qualquer lugar que receba uma string).

Para ler um arquivo:

```
>>> arq = open ('/tmp/meuarquivo.txt')
>>> arq.read ()
'meu teste de escrever em arquivo\nmuito legal'
>>> arq.readline ()
''
>>> arq.seek(0)
>>> arq.readline ()
'meu teste de escrever em arquivo\n'
```

Como podemos ver, não é necessário especificar o modo de abertura quando só se quer ler. A função *open* assume o segundo argumento como sendo "r". O

método read() lê o arquivo completo, como podemos ver, e qualquer tentativa de leitura resulta em uma string vazia, como vemos na quarta linha. O método seek() serve para posicionar o cursor no arquivo e a readline() serve para ler uma linha apenas. O método close() seria usado nesse caso, também.

# Capítulo 8

# Leitura Recomendada

O Gustavo Barbieri escreveu um curso de Python que vai mais fundo em alguns conceitos e pode ser uma ótima referência para continuar indo em frente no seu aprendizado.

Veja em `http://www.gustavobarbieri.com.br/python/aulas_python/`.