Universidade Federal Fluminense
Escola de Engenharia
Curso de Engenharia de Telecomunicações
Programa de Educação Tutorial
Grupo PET-Tele

Tutorial de Introdução ao Python

(Versão: A2011M01D18)

Niterói - RJ
Janeiro / 2011

Sumário

Introdução

A idéia de um ser humano, enquanto for apenas um pensamento, é algo amorfo, que existe por si mesma, não necessitando de nada, além de si, para descrevê-la. Entretanto para compartilhá-la com outras pessoas precisamos descrever esta idéia com palavras e frases, transformando este pensamento em uma linguagem natural humana.

Computadores e outros dispositivos eletrônicos programáveis possuem sua própria forma de "pensar", isto é, o código binário. Máquinas seguem instruções determinadas pela energização (ou não) de determinadas partes em seus componentes.

Então para transmitir uma idéia para uma máquina devo me comunicar em zeros e uns? A resposta para esta pergunta é um "sim" e um "não". Um "sim" pois, de fato, a "idéia" que será recebida pela máquina será descrita como uma sequência de zeros e uns. O "não" se refere a dizer que embora seja possível traduzir sua idéia para uma linguagem natural da máquina (binário), este não é o processo mais comum. A programação diretamente em binário é algo demasiadamente complexo para humanos, pois a forma básica de expressão do ser humano são as palavras e não sequências numéricas.

Para contornar este problema foram criadas as linguagens de programação, que são pontes entre a linguagem natural humana e a linguagem da máquina (binária), mesclando conceitos das linguagens de máquina e natural humana. As linguagens de programação são classificadas em vários níveis de acordo com sua proximidade com a linguagem humana.

Linguagens de baixo nível tem a função de descrever uma situação mais próxima do "ponto de vista" de uma máquina, ao passo que linguagens de alto nível são concebidas com o propósito de tornar fácil a tradução de um pensamento (algoritmo), se assemelhando com fidelidade 'a linguagem humana que seria necessária para descrever aquela idéia.

Neste tutorial descreveremos a linguagem de programação Python cujo objetivo é se aproximar bastante da linguagem humana, sendo assim intuitiva, fácil e ao mesmo tempo sendo bastante flexível, se adequando ao uso em diversas situações.

1 Características básicas da linguagem

Python é uma linguagem de programação interpretada, de código-fonte aberto e disponível para vários sistemas operacionais. Diz-se que uma linguagem é interpretada se esta não ser precisar compilada (traduzida para uma linguagem da máquina), mas sim "lida" por um outro programa (chamado de interpretador) que traduzirá para a máquina o que seu programa quer dizer.

O interpretador para Python é interativo, ou seja, é possível executá-lo sem fornecer um script (programa) para ele. Ao invés disso, o interpretador disponibilizará uma interface interativa onde é possível inserir os comandos desejados um por um e ver o efeito de cada um deles. Neste tutorial, se encontra a representação "`>>>`" antes da maioria dos comandos apresentados, o que quer dizer que este comando está sendo inserido no interpretador interativo (e sempre abaixo, o resultado gerado).

2 Obtenção e instalação

Caso o usuário esteja utilizando um sistema Linux ou OS X (Apple), o interpretador para Python já vem instalado por padrão, sendo apenas necessário escrever o comando "python" no seu programa de terminal favorito. Para usuários do sistema operacional Windows, o interpretador para Python deve ser baixado através do site http://www.python.org e instalado. Neste

último sistema o usuário encontra um utilitário para fazer o papel de terminal (e editor de python) no Windows, denominado IDLE.

3 Variáveis

Variáveis são formas de se armazenar dados para uso posterior, elas podem ser classificadas em 3 tipos básicos que são mostradas logo abaixo. Quando analisarmos as listas veremos que existem outras variáveis mais complexas.

- *int* - Um número inteiro
- *float* - Um ponto flutuante
- *string* - Uma sequência de caracteres

Ao contrário da maioria das outras linguagens, em Python, não é necessário declarar as variáveis que serão usadas, tampouco definir seu tipo. A própria sintaxe do dado a ser armazenado identifica o tipo da variável para armazená-lo. Por exemplo, caso deseje-se atribuir o valor 3 à variável **A**, basta digitar **A=3**. Python saberá que **A** é um inteiro (tipo "*int*"). Por outro lado, se o valor a ser armazenado fosse 3,2 que é um dado do tipo "ponto flutuante", este deveria ser expresso como **A=3.2**. Observe que, para Python, A=3 e B=3.0 são variáveis de tipos diferentes e isto deve ser levado em conta ao se realizar certos tipos de manipulações de dados.

4 Strings

String é um tipo de objeto formado por uma sequência imutável de caracteres que nos permite trabalhar com textos.

Exemplo:

```
>>> a = "Bom Dia"
>>> print a
Bom Dia
```

Percebemos que elas são delimitadas por aspas, podemos utilizar tanto aspas duplas como as simples. Se utilizarmos aspas duplas, como o mostrado no exemplo acima, podemos usar as simples para aplicações dentro do texto que estamos escrevendo, o contrário também é verdadeiro.

Exemplo:

```
>>> b = 'O lema do governo JK era:\n "Cinquenta anos em cinco."'
>>> print b
O lema do governo JK era:
"Cinquenta anos em cinco."
```

No exemplo acima utilizamos um outro artifício para trabalharmos com strings, o `\n`. Este por sua vez, tem a função de "pular uma linha" e escrever o texto, que está após o `\n`, nessa nova linha. Tanto isso é verdade que no mesmo exemplo, quando usamos o comando print é mostrada a parte do texto que diz: "Cinquenta anos em cinco.", impresso na linha seguinte.

Outro artifício parecido com `\n`, é o `\t` que tem o objetivo de acrescentar ao texto que vem após, um espaço de tabulação.

Há também outra aplicação para as aspas, que consiste na delimitação do texto por três aspas (duplas ou simples).

Exemplo:

```
>>> a = ''' Quatro times do Rio de Janeiro:
Botafogo
Vasco
Fluminense
Flamengo'''
>>> print a
Quatro times do Rio de Janeiro:
Botafogo
Vasco
Fluminense
Flamengo
```

Nessa aplicação para as aspas, quando usamos o comando print, tudo o que estiver entre as três aspas será impresso exatamente da mesma forma como foi escrito.

4.1 *Manipulando Strings*

Pelo fato de uma string ser uma sequência imutável, isso nos dá a possibilidade de manipularmos essa sequência. Consideremos o exemplo abaixo:

```
>>> a = 'matemática'
>>> a[2]+a[-5]+a[-4:]
>>> 'tática'
```

Isso nos mostra que uma string segue uma determinada indexação onde cada caractere assume um endereço que, pode ser acessado colocando o nome da variável, que contém a string, e após, entre os colchetes, o endereço da célula que contém o caractere desejado.

Existe também um recurso conhecido como operador %, que serve para formatar as strings, basicamente são três os tipos de formatação que temos:

- %s - serve para substituir string;
- %d - serve para substituir números inteiros em uma frase destinada a um print;
- %f - serve para substituir floats (números em aritmética de ponto flutuante).

OBS.: As três formatações acima relacionadas são usadas normalmente para aplicações em uma frase destinada a um print.
Exemplo:

```
>>> compra= 'maçã'
>>> tipo='verde'
>>> quilos = 1,5
>>> print 'Maria comprou %f quilos de %s %s .' %(quilos,compra,tipo)
Maria comprou 1,5 quilos de maça verde.
```

OBS.: Como vimos acima o operador % pode ser utilizado para formatação de números também. Com ele é possível também determinar a quantidade de números, após a vírgula de um float.

Exemplo:

```
>>> num=245.47876749
>>> print '%.2f' %(num)
```

```
245.48
```

Percebemos que Python fez uma aproximação do número real, possibilitando que o resultado de uma futura operação seja o mais preciso possível.

5 Operações matemáticas

Além de ser uma poderosa linguagem de programação, Python sabe lidar bem com matemática. Suas capacidades matemáticas se extendem desde operações básicas até operações com números complexos.

Abaixo, vamos começar aprendendo as 4 operações matemáticas básicas. Lembre-se que "**>>>**" significa a digitação de um comando no interpretador.

```
    Soma:
>>>2+3
5
    Subtração:
>>>2-3
-1
    Multiplicação:
>>>2*3
6
```

```
    Divisão:

>>>2/3

0
```

Na divisão, pode-se optar por resultados como números inteiros (ou o inteiro mais próximo) ou resultado exato (um ponto flutuante). Para isso, deve-se fornecer um inteiro (para se obter um inteiro como resultado) ou um ponto flutuante (para se obter o resultado exato).

```
    Divisão:

>>>2.0/3.0

0.66666666666666663
```

A operação aritimética de potenciação também pode ser feita, assim como a sua inversa, a radiciação.

```
    Potenciação:

>>> 2**3

8
```

Não existe um comando específico para a radiciação em Python (exceto para a raiz quadrada, que possui o comando sqrt(x)). Entretanto, para obter a raiz n-ésima de um número basta elevá-lo por $1/n$ onde "n" é o índice da raiz. As regras para a obtenção de números inteiros ou pontos flututante também se aplicam a este caso.

```
    Radiciação:
>>>8**(1.0/3.0)
2.0
```

6 Entrada de Dados

Através do comando ***raw_input*** podemos receber do usuário uma *string*. O número de dígitos da *string* dada pode ser informado através do comando ***len()***.

```
>>> nome = raw_input('Digite seu nome: ')
Digite seu nome: Pedro
>>> nome
'Pedro'
>>> len(nome)
5
>>> sobrenome = raw_input('Agora digite o sobrenome: ')
Agora digite o sobrenome: Albuquerque
```

Operações matemáticas não podem ser feitas com *strings*, apenas com *floats* e inteiros, porém se somarmos *strings*, Python as juntará, num processo chamado concatenação e se multiplicarmos uma *string* ela será repetida.

```
>>> nome + sobrenome
'PedroAlbuquerque'
>>>nome*3
'PedroPedroPedro'
```

Se a priori sabemos que o dado digitado pelo usuário não será uma *string*, podemos utilizar como entrada de dados o comando **input()**:

```
>>> idade = input('Digite sua idade: ')
Digite sua idade: 27
>>> altura = input('Qual a sua altura: ')
Qual a sua altura: 1.75
>>> type(idade)
<type 'int'>
>>> type(altura)
<type 'float'>
```

OBS.: O dado inserido pelo usuário é automaticamente interpretado como um dos tipos de varável, *strings*, *floats* ou *integers*. Para saber como a variável atribuída foi interpretada, basta digitar o comando **type()**.

7 Listas

Listas são sequências de variáveis. Após definidas, podem ser modificadas de várias maneiras, pois são mutáveis.

Para definir uma lista basta digitar:

lista = [9,8,7]

O endereçamento dos componentes começa no 0. Então, se você deseja usar um determinado elemento da lista, basta chamá-lo: **lista[x]**. Com isso, você está se referindo ao elemento de posição x na lista.

Alguns comandos referentes à, listas:

len(lista): informa o tamanho da lista

lista.append(x): adiciona o elemento x no final da sua lista

lista.extend([6,5,4]) : adiciona uma lista inteira no final da sua lista

lista[y]= x : insere o valor x na posição y da lista

Nós já vimos anteriormente que variáveis comuns armazenam um único valor. Entretanto, existem outros tipos de variáveis capazes de armazenar mais de um valor. Em Python, chamamos essas variáveis com capacidade de armazenamento de listas ou vetores. Vamos explicar a sintaxe de uma lista através das linhas de código exibidas abaixo:

```
>>> meses = ['Janeiro', 'Fevereiro', 'Março', 'Abril', 'Maio', 'Junho', 'Julho','Agosto', 'Setembro', 'Outubro', 'Novembro',
'Dezembro']
>>> while 1:
...         mes = input("Escolha um mês (1-12)?")
...         if 1 <= mes <= 12:
...             print 'O mês é ', mes[mes - 1]
```

Ao rodar este programa obteremos a seguinte saída como exemplo:

```
>>> Escolha um mês (1-12)? 5
O mês é Maio
>>> Escolha um mês (1-12)? 12
O mês é Dezembro
>>> Escolha um mês (1-12)?
```

Ainda sobre o programa anterior, primeiramente criamos a nossa lista. A definição de uma lista é análoga a de uma variável qualquer, porém isolando o conteúdo com colchetes. O uso das aspas (duplas ou simples) apenas é necessário caso estejamos inserindo na lista uma *string*, se armazenássemos apenas um número, não seria necessário.

Exemplo:

```
teste = ['vida', 42, 'universo', 6, 'e', 7]
```

Depois inserimos o comando **while 1**: que faz com que o nosso programa entre em *loop*. O programa vai rodar indefinidamente até ser dado o comando Ctrl+D, ou até que o programa seja fechado. Uma vez nesse *loop*, é definida pelo usuário a variável "mes", e depois de um tratamento de erros, feito com a utilização de um comando **if**, é nos devolvido o nome do mês selecionado.

Na última linha do código utilizamos uma propriedade da lista, que é buscar um dado da lista, que se dá escrevendo o nome da lista e entre colchetes o número referente ao local da lista, onde está o dado requerido (lista[número]).

Observe que na nossa linha de código, o comando **print** é dado pelo mês escolhido menos um, ou seja, indexando as listas partindo do zero.

Além de selecionar um elemento de uma lista, temos algumas outras propriedades que apresentaremos em seguida.

7.1 Inserindo um novo dado a uma lista

Para inserir um novo dado a uma lista qualquer, utilizamos um método chamado **.append**:

```
>>>teste = []
>>>teste.append('zero')
>>>teste.append('um')
>>>teste
 ['zero','um']
```

Infelizmente o comando .**append** só consegue adicionar um dado na lista por vez, mas se quisermos adicionar mais dados podemos simplesmente somar listas, multiplicá-las, ou utilizar o método **.extend**:

```
>>>teste.extend(['dois','três',]
>>>soma = ['quatro','cinco']
>>>teste += soma
>>>teste + ['seis']
 ['zero','um','dois','três','quatro','cinco','seis']
>>>teste*2
 [zero','um','dois','três','quatro','cinco','seis',zero','um','dois','três','quatro','cinco','seis']
```

OBS.: o código **teste += soma** é o mesmo que escrever **teste = teste + soma**.

7.2 Impressão dos conteúdos da lista

Ao chamarmos a lista **teste** sem referenciarmos nenhum elemento específico estamos lidando com todos os elementos da lista. Podemos também imprimir somente alguns valores de dentro da lista, necessitando para isso apenas indicar o local:

```
>>>print 'As variavéis da lista teste são: ',teste
As variavéis da lista teste são ['zero','um','dois','três','quatro']
>>>print 'As variavéis 0 e 3 da lista teste são ',teste[0],' e ',teste[3]
As variavéis 0 e 3 da lista teste são zero e três
```

Podemos também utilizar o comando **for**.

```
>>>for valor in teste:
... print valor
...
zero
um
dois
três
quatro
```

7.3 Determinar em que ordem um elemento aparece na lista

O método **.index** mostra em qual posição o item fornacido se encontra na lista. Porém, se o item especificado aparecer mais de uma vez na lista, o método **.index** mostra a posição da primeira ocorrência do valor.

```
>>>print lista.index(”quatro”)
4
```

OBS.: É importante notar que a lista inicia sua contagem a partir do 0 (zero) o que explica o número 4 obtido anteriormente.

7.4 Remover um elemento de uma lista

Para remover um elemento de uma lista utilizamos o comando **del**, referenciando o *index*, ou posição da lista, onde haverá a remoção.

```
>>>print ’Antes a lista ”teste”era: ’,teste
Antes a lista ”teste”era: [’zero’,’um’,’dois’,’três’,’quatro’]
>>>del teste[3]
>>>print ’Agora a lista ”teste”é: ’,teste
Agora a lista ”teste”é: [’zero’,’um’,’dois’,’quatro’]
```

Podemos obter o mesmo efeito utilizando o método **.remove** que procuraria a primeira ocorrência na lista para um dado valor ou *string*:

```
>>>lista + [’um’]
>>>print ’Antes a lista ”teste”era: ’,teste
Antes a lista ”teste”era: [’zero’,’um’,’dois’,’quatro’,’um’]
>>>teste.remove(”um”)
>>>print ’Agora a lista ”teste”é: ’,teste
Agora a lista ”teste”é: [’zero’,’dois’,’quatro’,’um’]
```

7.5 Descobrir o tamanho de uma lista

Para descobrir o tamanho de uma lista, usamos o comando **len()**, como exemplificado no código abaixo:

```
>>>print ’O tamanho da lista teste é: ’, len(teste)
4
```

Nos dizendo que a lista “teste” possui 4 variáveis.

7.6 *Range*

A função **range** gera um vetor contendo números inteiros sequenciais, obedecendo a regra de escrita:

range(início,fim)

É importante observar que o número finalizador descrito acima não é incluído no vetor, por exemplo:

```
>>>vetor = range(1,11)
>>>print vetor
```

O código acima nos dará como saída o vetor: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. Em outras palavras, o último número é excluído do vetor, portanto o 11 não entra na composição deste.

A função **range** aceitará quaisquer números inteiros desde que o número inicial seja maior que o número final, bem como quando apenas o número final é passado para a função, portanto são válidas as construções:

```
>>> range(1, 10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(-32, -20)
[-32, -31, -30, -29, -28, -27, -26, -25, -24, -23, -22, -21]
>>> range(5,21)
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(21, 5)
[ ]
```

No último exemplo foi retornado um vetor vazio, isso se deu porque o elemento inicial era maior que o final.

Outra característica importante deste comando é a de se poder controlar o passo da sequência adicionando uma terceira variável ao range. Isto é, a variação entre um número e seu conseguinte.

```
>>>range(0,24,4)
[0, 4, 8, 12, 16, 20]
```

Exercícios:

1. Faça um programa que leia um vetor de 5 números inteiros e mostre-os.
2. Faça um programa que leia dois vetores com 10 elementos cada. Gere um terceiro vetor de 20 elementos, cujos valores deverão ser compostos pelos elementos intercalados dos dois outros vetores.
3. Faça um programa que leia 4 notas, mostre as notas e a média na tela.
4. Faça um programa que leia um número indeterminado de valores, correspondentes a notas, encerrando a entrada de dados quando for informado um valor igual a -1 (que não deve ser armazenado). Após esta entrada de dados, faça:
    - a) Mostre a quantidade de valores que foram lidos.
    - b) Exiba todos os valores na ordem em que foram informados, um ao lado do outro.
    - c) Calcule e mostre a média dos valores.
    - d) Calcule e mostre a quantidade de valores acima da média calculada.
    - e) Encerre o programa com uma mensagem.

8 Estruturas de controle

Os comandos de Python são executados pelo computador, linha por linha e as estruturas de controle permitem ao programador modificar a ordem em que cada comando será executado bem como se ele será ou não executado.

8.1 *If*

O comando **if** direciona o computador a tomar uma decisão, baseado nas condições determinadas. Se a condição for atendida, um bloco de comandos será executado, caso contrário, o computador executa outros comandos.

```
...         #bloco de comandos 1
>>>if #condição1:
...         #bloco de comandos 2
...(continuação do programa)
...         #bloco de comandos 3
```

OBS.:Se a condição1 dada no if for verdadeira o bloco de comandos 2 será executado, caso contrário o programa passará direto do bloco de comandos 1 para o bloco de comandos 3.

Nessa estrutura podemos utilizar quantas condições foram necessárias, basta repertimos o ***elif*** seguido das condições desejadas. Tendo isso em vista, vejamos um programa de adivinhar um número:

```
>>>num = 23
>>>adv = 0
>>>while adv != num:
...         adv = input('Insira um número: ')
...         if adv < num:
...                 print 'É maior!'
...         elif adv > num:
...                 print 'É menor!'
...         else:
...                 print 'Você acertou!'
...
>>>Insira um número:
```

Como vimos, em Python não é necessário que utilizemos a *tag* **end** para terminarmos uma estrutra. Isto é devido a identação do programa. Em Python, uma estrutura é terminada automaticamente quando a *tag* da próxima linha começa no início da linha.

8.2 *While*

Esta estrutura de controle tem como objetivo executar o bloco de comandos identificado nela repetidamente, enquanto a condição dada, para sua validade, for verdadeira. Para que o bloco de comandos desta condição seja executado de maneira correta, devemos manter uma organização, tudo que pertencer ao bloco do while, deve ter um espaçamento da margem a esquerda no texto, isto ajuda a deixar o código legível e organizado.

```
>>> while #condição for verdadeira :
...         #bloco de comandos pertencentes ao while
...
```

```
>>> #continuação do programa
```

Vejamos, por exemplo, como calcular um número fatorial:

```
>>> resultado = 1
>>> num = input('Entre com um número inteiro: ')
>>> Entre com um número inteiro: 6
>>> num2 = num
>>> while num2 > 1:
...         resultado = resultado * num2
...         num2 = num2 - 1
...
>>> print num, '! é igual a ',resultado
>>> 6! é igual a 720
```

Se a condição estabelecida no **while** for sempre verdadeira, como $2 < 1$, o seu *loop* será infinito, pois a condição será sempre atendida. Caso esse seja seu desejo, pode-se usar também o comando **while** 1 em vez de estabelecer uma condição qualquer (isso porque segundo a lógica booleana, que será vista na nona seção, o python interpreta o valor 1 como verdadeiro).

8.3 *For*

O comando **for**, em Python, difere do que normalmente se vê em outras linguagens de programação, onde esse comando tem a finalidade de realizar uma iteração baseada numa progressão aritmética, percorrendo os números definidos pelo usuário, enquanto em Python a iteração é feita percorrendo os itens de uma sequência, seja ela uma lista ou até mesmo uma *string*. Vamos analisar o código abaixo:

```
>>> for contador in range(1, 11):
...         print contador
```

A estrutura acima utiliza uma variável criada, no caso **contador**, para percorrer cada elemento da lista criada com o comando **range**(1,11), com isso, cada repetição feita pelo *loop* **for** fará com que a variável contador aponte para um diferente valor dentro da lista formada pela função **range** e logo em seguida imprima esse valor.

A saída que teríamos neste caso seria:

```
1
2
3
4
5
6
7
8
9
10
```

Ou seja, um "print" de todos os elementos da lista.

A função **for** também pode ser aplicada em *strings*, observemos o caso abaixo:

```
>>> lista = ['vida', 42, 'o universo', 6, 'e', 7, 'tudo']
>>> for item in lista:
...         print 'O item atual é:',print
```

E obteremos como saída:

```
O item atual é: vida
O item atual é: 42
O item atual é: o universo
O item atual é: 6
O item atual é: e
O item atual é: 7
O item atual é: tudo
```

Exercícios:

1. Exibir uma série de números (1, 2, 3, 4, 5, ...) em um *loop* infinito. O programa deve encerrar-se quando for pressionada uma tecla específica, como um ESC.

2. Obter uma série de números do teclado e ordená-las tanto em ordem ascendente como descendente. Fazer o mesmo com uma série de *strings*.

3. Faça um Programa que peça 2 números inteiros e um número real. Calcule e mostre:

   - a) O produto do dobro do primeiro com metade do segundo .
   - b) A soma do triplo do primeiro com o terceiro.
   - c) O terceiro elevado ao cubo.

4. Faça um programa que receba dois números inteiros e gere os números inteiros que estão no intervalo compreendido por eles.

5. Faça um programa que peça um número inteiro e determine se ele é ou não um número primo. Um número primo é aquele que é divisível somente por ele mesmo e por 1.

9 Dicionário

Um dicionário é um conjunto de elementos que possuem índices, ou seja, dicionários são formados por chaves e seus respectivos valores, onde as chaves são os índices.
Para se declarar um dicionário e os elementos a ele pertecentes, escrevemos:

```
>>>calculos = { 1:'primeiro periodo' , 2:'segundo periodo' , 4:'terceiro periodo' , 8:'quinto periodo' }
>>>print calculos
{ 1:'primeiro periodo' , 2:'segundo periodo' , 4:'terceiro periodo' , 8:'quinto periodo' }
>>>print calculos[8]
'quinto periodo'
>>>print calculos[8]='números complexos'
>>>print calculos
{ 1:'primeiro periodo' , 2:'segundo periodo' , 4:'terceiro periodo' , 8:'números complexos' }
```

Os valores referentes aos índices podem ser mudados.

```
>>>calculos[4] = 'numeros complexos'
```

Abaixo, citaremos alguns métodos dos dicionários:

- ***.keys()*** - Retorna uma lista com as chaves do dicionário.
- ***.values()*** - Retorna uma lista com os valores do dicionário.
- ***.items()*** - Retorna uma lista com as chaves e seus respectivos valores.
- ***.has_key(x)*** - Verifica se o dicionário possui a chave x.

Exercícios:

1. Faça um dicionário que contenha suas refeições e um alimento que esteja contido em cada uma delas. Imprima na tela. Após isso, mude os alimentos pelos seus alimentos favoritos.
2. Faça um dicionário que contenha os meses do ano e um aniversariante por mês. Após, pergunte ao usuário um aniversariante por mês e troque os valores do seu calendário de aniversário pelos do usuário.

10 Funções

As linguagens de programação em geral têm o intuito de automatizar ações tornando-as mais rápidas.

Se houver alguma ação que seja grande e utilizada com frequência, temos a opção de criar uma função que cumpra o seu objetivo, reduzindo o espaço ocupado pelo nosso programa final, além de deixá-lo com uma aparência mais limpa, visto que o tamanho do código irá diminuir. Essas funções também são muito úteis na tarefa de debuggar o seu código, visto que você não precisará vasculhar o código atrás do erro, basta entrar na função e modificá-la. Um exemplo de como podemos diminuir um código está descrito abaixo.

Se em um determinado problema, precisarmos descobrir se dois valores absolutos são iguais podemos utilizar o código 1 descrito abaixo todas as vezes que precisarmos realizar essa descoberta, ou podemos simplesmente usar o código 2 criando uma função que cumpra esse objetivo exigindo apenas que apresentemos os valores que devemos analisar.

Dado que temos os valores:
a = 23 e b = -23

Função 1:

```
>>> if a < 0:
...         a = -a
>>> if b < 0:
...         b = -b
>>> if a == b:
...         print 'Os valores absolutos de ', a, ' e ', b, ' são iguais'
... else:
...         print 'Os valores absolutos de ', a, ' e ', b, ' não são iguais'
```

Função 2:

```
>>>def compara_absolutos(a,b):
...         "Essa função retorna se os valores absolutos das variáveis requeridas são iguais"
...         if a < 0:
...                 a = -a
...         if b < 0:
...                 b = -b
...         if a == b:
...                 print 'Os valores absolutos de ', a, ' e ', b, ' são iguais'
...         else:
...                 print 'Os valores absolutos de ', a, ' e ', b, ' nã são iguais'
...
```

Abaixo apresentamos a sintaxe necessária para criação de funções utilizando a linguagem Python.

```
>>>def funcao(variavel1,variavel2,...,variavelN):
...         bloco de comandos
...         return
```

É que ao chamar uma função, podemos passar a esta alguns parâmetros (valores ou *strings*): **funcao(1234,"pet-tele","UFF-1234")**. Porém em algumas funções ao colocarmos nosso código na seção bloco de comandos, a função automaticamente definirá os parâmetros necessários. Por exemplo, se houver uma função que faça cálculos matemáticos, devolvendo um resultado numérico, será necessário que todas as variáveis chamadas pela função sejam inteiros ou *floats*.

10.1 Variáveis em funções

Em geral, quando estamos eliminando código repetitivo por meio de funções também temos algumas variáveis repetidas neste código. Em Python, as variáveis podem ter tratamentos diferentes em função de onde se encontram. Todas as variáveis que vimos até agora são chamadas de variáveis globais, ou seja, em qualquer momento ou em qualquer parte do código do seu programa, você poderá utilizá-las (seja para ler seu conteúdo ou atribuir valores).

Funções tem um tipo especial de variáveis que são chamadas de variáveis locais. Estas variáveis existem apenas dentro da função, de forma que caso o programador determine uma função e armazene um certo dado em uma variável local, ao término desta função a variável será destruída, não sendo possível recuperar seu valor.

Mas como então, podemos recuperar um valor da variável utilizada emF uma função? Para isto usamos o comando **return** seguido de algum valor ou uma variável. Neste caso, a variável local "esconderá" a variável global, enquanto a função estiver rodando. Um pouco confuso? Então confira o exemplo abaixo:

```
>>>a = 4
>>>def print_func():
...         a = 17
...         print 'in print_func a = ', a
...
```

Agora chame a função **print_func()** e peça o valor "a" utilizando o comando **print**, seu resultado deve ser este:

```
>>>print_func()
in print_func a = 17
>>>print 'a = ', a
a = 4
```

Com isto, podemos concluir que variáveis criadas dentro de uma função (variáveis locais), não afetam as variáveis que estão fora dela (variáveis globais). As variáveis locais existem apenas dentro do espaço limitado pela função, não podendo assim nem ser usada, nem afetar nada fora deste espaço.

10.2 Recursividade

A recursividade é um tipo de iteração (repetição) na qual uma função chama a si mesma repetidamente até que uma condição de saída seja satisfeita. Abaixo temos um exemplo de uma função responsável por calcular o fatorial de números positivos inteiros e demonstra como uma função pode chamar a ela mesma utilizando a propriedade recurssiva.

```
>>>def fatorial(n):
...         if n <= 1:
...                 return 1
...         return n * fatorial(n - 1)
...
>>>print '2! = ',fatorial(2)
2! = 2
>>>print '3! = ',fatorial(3)
3! - 6
>>>print '4! = ',fatorial(4)
4! = 24::
>>>print '5! = ',fatorial(5)
5! = 120
```

Um passo-a-passo de como a função é executada, utilizando o comando **fatorial(n)**, para n = 3:

1. Quando chamamos a função **fatorial(3)**, fornecemos
   'a função o valor 3 para o parâmetro de entrada n.

2. O comando **if** testa se n é menor ou igual a 1, como este não é, a função continua.

3. Nesta parte a função pretende retornar o valor **n*fatorial(n-1)**, ou seja, **3*fatorial(2)**. Entretanto, ela ainda não possui o valor de **fatorial(2)**. Para isso, a função **fatorial** é novamente chamada com n = 2, retornando o valor **2*fatorial(1)**.

4. Como ainda não temos **fatorial(1)**, a função chama a si mesma mais uma vez para calculá-lo.

5. Com isto, atingimos a condição de saída do **if** que está na função **fatorial** e então o valor retornado é 1.

Vamos fazer uma retrospectiva do que aconteceu ao se executar esta função:

fatorial(3)
3 * fatorial(2)
3 * 2 * fatorial(1)
3 * 2 * 1 = 6

Exercícios:

1. Faça um programa, com uma função que necessite de três argumentos, e que forneça a soma desses três argumentos.
2. Faça um programa que receba a matrícula do usuário e informe o período que ele está e em quanto tempo ele irá se formar.
3. Faça um programa, com uma função que necessite de um argumento. A função retorna o valor de caractere 'P', se seu argumento for positivo, e 'N', se seu argumento for zero ou negativo.

11 Módulos

Pensando na reutilização de código, a linguagem Python já possui um conjunto de funções prontas para serem usadas ou agregadas em seus programas. Essas funções estão agrupadas em estruturas denominadas módu los. Para a utilização desses módulos é preciso utilizar o comando **import nome_do_módulo**.

Após ter importado o módulo, qualquer função pertencente a ele pode ser utilizada através do comando **nome_do_módulo.função(argumento)**.

É possível importar do módulo apenas a função desejada. Para isso, utilizamos o comando **from nome_do_módulo import função**, e a função estará disponível para utilização.

Você também pode definir o seu próprio módulo. Defini-se as funções desejadas e ao final, você salva o seu módulo com a extensão .py . Exemplo:

Digite em um editor de texto simples:

```
#IMC.py

def indice(altura,peso):
    imc = peso/(altura**2)
    return imc

def estado(imc):
    if imc < 24.9:
        print 'NORMAL'
    elif 24.9 < imc < 29.9:
        print 'PESO A MAIS'
    elif 29.9 < imc < 40:
        print 'LIGEIRA OBESIDADE'
    elif imc > 40:
```

```
        print 'OBESIDADE'
    else:
        print 'MAGRO DEMAIS'


def pesoideal(peso,altura):
    a = 20*(altura**2)
    b = 24.9*(altura**2)
    print 'Seu peso ideal se encontra entre %f e %f' %(a,b)
```

Agora, salve o seu arquivo como IMC.py . De agora em diante, o módulo já pode ser utilizado por qualquer programa em Python.

Apresentaremos a seguir 3 módulos interessantes e de grande uso para a maioria dos usuários de Python:

11.1 Módulo *Math*

O módulo *math* possui funções matemáticas para números não complexos. Existe um módulo equivalente para números complexos: *cmath*. A distinção destes módulos deve-se ao fato da maioria dos usuários não querer aprender a fundo a teoria dos complexos.

Este módulo contém funções de representação numérica, logaritimicas, exponenciais, hiperbólicas, trigonométricas e conversões angulares e os valores retornados por este módulo são pontos flutuantes.

As constantes *pi* e *e* tem valores definidos nesse módulo, podendo ser usadas diretamente pelo nome, a partir do momento que o módulo é importado.

Abaixo, estão listadas algumas funções desse módulo:

- ***math.factorial(x)*** - Retorna o valor de x tutorial. Caso x seja negativo, retorna um erro.
- ***math.modf(x)*** - Retorna o valor inteiro e o valor fracionário de x.
- ***math.exp(x)*** - Retorna e exponencial de x.
- ***math.log(x,base)*** - Retorna o log de x na base pedida.
- ***math.log1p(x)*** - Retorna o log natural de x.
- ***math.sqrt(x)*** - Retorna a raiz quadrada de x.
- ***math.degrees(x)*** - Converte o ângulo x de radianos para graus.
- ***math.radians(x)*** - Converte o ângulo x de graus para radianos.

As funções trigonométricas sin(x), cos(x), tan(x) e suas inversas também estão contidas nesse módulo, além das funções a seguir:

- ***math.hypot(x,y)*** - Retorna a norma euclidiana entre x e y, ou seja, a hipotenusa do triângulo retângulo de catetos x e y.
- ***math.atan2(y,x)*** - Retorna o valor do arco tangente de y/x.

Os valores retornados das funções trigonométricas estão em radiano.
Para uma lista completa de funções neste módulo, basta pedir um help(math).
Exercícios:

1. Fazer uma calculadora que execute operações de adição, subtração, multiplicação, divisão, quadrado, cubo, raiz quadrada, seno, cosseno, tangente, fatorial, inverso e módulo.
2. Faça um programa que ao receber um valor de raio, retorne a área e perímetro do círculo.
3. Faça um programa que receba 2 valores de catetos e retorne:
   - a) A hipotenusa.
   - b) Os valores dos senos e cossenos dos 3 ângulos do triângulo.
   - c) Todos os valores deverão ser mostrados em radiano e em graus.

11.2 Módulo io - Manipulação de arquivos

Em algumas situações será necessário que se armazene dados ou então que se colete dados de algum arquivo, para isso existe um módulo em python que se comunica com o sistema operacional chamado "io" (input/output). Desse módulo o comando principal é o comando "open", a sintaxe do programa está demonstrado abaixo:

```
>>> open('endereço/nome do arquivo.extensão','modo de abertura')
```

Se estiver utilizando um sistema UNIX e o arquivo estiver na mesma pasta em que foi executado o interpretador não é necessário colocar o endereço do arquivo. E os modos de abertura são mostrados abaixo:

r (read)- Abrir o arquivo somente para leitura;

a (append)- Abrir o arquivo somente para escrita, nesse modo o texto é somado ao arquivo;

w (write)- Abrir o arquivo somente para escrita, nesse modo o texto é substituido

A função *open* possui vários métodos os mais importantes serão mostrados abaixo e depois demonstrados:

close() - fecha o arquivo. Se o arquivo não for fechado, ele continuá como um objeto e não poderá ser modificado fora do interpretador;

closed - Responde com valores booleanos se a conexão foi fechada ou não.

Para criar um novo arquivo se usa o modo *write* (w). Vamos demonstrar agora como criar um arquivo com extensão txt, como todos os métodos desse módulo utilizam a sintaxe mostrada acima, declaramos uma variável que recebe a função *open*.

OBS.: Antes de escrever esse comando verifique se não existe um arquivo com o mesmo nome, pois se houver esse comando substituirá o arquivo por um novo.

```
>>> abrir = open("teste.txt","w")
>>> abrir.close()
>>> editar = open("teste.txt","a")
```

Com isso criamos um arquivo, utilizando o modo **w**, chamado "teste.txt", atribuímos a ele à variável "abrir" e o fechamos com o método ".close". Depois criamos outra variável que possui a qualidade de editar o arquivo sem escrever por cima, ou seja, que utiliza o modo **a**. Agora vamos explorar os métodos existentes quando utilizamos o modo "append". Primeiramente utilizaremos o método ".write", que serve para adicionar uma string ao nosso arquivo.

```
>>>editar.write("Hello World!")
>>>editar.close()
>>>editar.closed
```

OBS.: o comando "closed" confirma se a conexão entre o python e o arquivo foi interrompida. Com esses comandos foi adicionado a string "Hello World!" ao arquivo, podemos confirmar isto abrindo o arquivo e lendo ou então podemos usar o modo de leitura "**r**".

```
>>>ler.open("teste.txt","r")
>>>ler.readline()
>>>ler.colse()
```

Novamente, criamos uma variável, no caso a variável "ler", só que desta vez atribuímos a ela o modo de leitura, depois utilizamos o método ".readline()" que lê o arquivo linha a linha.

OBS.: Depois de alcançar o final do texto ele devolve uma string vazia, para voltar a ler o arquivo depois disso é necessário que você abra outra conexão com o arquivo.

Agora que já sabemos criar, editar e ler um arquivo vamos criar uma lista de presença.

```
>>> #Começaremos criando o arquivo
>>> criar = open("presenca.txt","w")
>>> criar.close()
>>> #Agora vamos escrever as funções responsáveis pela escrita e leitura da nossa lista de presença
>>> #Criamos uma função para adicionar o nome dos alunos à lista
>>> def adicionar_aluno():
>>>         "O nome do aluno presente deve estar entre parênteses "
>>>         #Criamos uma variável para editar o arquivo
>>>         aluno = raw_input("Qual o nome do aluno? \n")
>>>         escrever = open("presenca.txt","a")
>>>         escrever.write(aluno+"\n")
>>>         escrever.close()
>>> #Depois criamos uma função para ler os nomes na lista
>>> def aluno_presente():
>>>         "Deve ser declarado um inteiro correspondente à linha em que o aluno está "
>>>         #Criamos uma variável de para ler o arquivo
>>>         leitura = open("presenca.txt","r")
>>>         #Nesse caso vamos usar o método readlines() e atribuí-la a uma variável, dessa forma a variável será uma lista, em
que, cada elemento desta lista é uma linha do arquivo
>>>         ler = leitura.readlines()
>>>         print ler[1:5]
```

```
>>>     aluno = raw_input("Deseja saber todos os presentes? s/n \n")
>>>     if aluno == "s":
>>>         for nomes in ler:
>>>         print nomes
>>>         elif aluno == "n":
>>>         qual_aluno = input("Qual o nÃºmero do aluno? \n")
>>>         print ler[qual_aluno-1]
>>>     else:
>>>         print "ERRO, digite s ou n"
```

11.3 PySQLite: Manipulação de Bancos de Dados

11.3.1 Introdução

O PySQLite é um módulo de Python que utiliza uma biblioteca em C chamada SQLite capaz de operar um banco de dados usando uma variante da linguagem SQL. Esta biblioteca, ao contrário dos bancos de dados tradicionais, realiza a manipulação direta do arquivo de banco de dados, não necessitando de um servidor (ou um programa-servidor) que intermedie a manipulação do mesmo. Isto faz com que este módulo seja o ideal para a manipulação de pequenos banco de dados, ou em aplicações que necessitem de armazenamento interno de dados. Também é possivel fazer o protótipo de um aplicativo usando o SQLite e em seguida migrar o código para o uso em um banco de dados de maior capacidade como MySQL, PostgreSQL, Oracle, etc.

11.3.2 Comandos básicos

O PySQLite é provido pelo módulo *sqlite*, portanto este deve ser importado antes de ser usado.

```
import sqlite
```

A manipulação do banco de dados é feita, não através de uma aplicação servidora específica para este fim, mas sim manipulando diretamente o arquivo de banco de dados. Para isto, deve ser criado um objeto *connect* que funciona representando o banco de dados que está sendo manipulado:

```
conexao = sqlite3.connect(' /caminho/para/banco_de_dados.db ')
```

OBS.: Caso se necessite do banco de dados apenas temporariamente, pode-se criá-lo na memória, de forma que este será descartado após o encerramento do programa.

```
conexao = sqlite3.connect(':memory:')
```

Em seguida, deve-se criar um objeto cursor, que será a variável que nos permitirá efetuar operações sobre a base de dados. Isto pode ser feito chamando-se o método *cursor()*.

```
cur = conexao.cursor(':memory:')
```

Usando o cursor podemos inserir instruções em SQL para a manipulação do banco de dados, criando novas tabelas, localizando, inserindo, removendo e modificando entradas, como seria feito em um banco de dados tradicional. As instruções em SQL devem ser fornecidas como um parâmetro ao método *execute()* que deve ser aplicado à variavel cursor. Abaixo temos

exemplos de algumas ações:

Criando uma nova tabela no banco de dados:

def criatabela(): ... sql = C̈REATE TABLE contatos(id INTEGER, nome VARCHAR, fone VARCHAR)ë... cur.execute(sql)

Inserindo um item no banco de dados:

def inseredados(): ... insnome = raw_input('Digite o nome do contato:') ... insidade = raw_input('Digite a idade do contato:') ... cur.execute('INSERT INTO contatos VALUES (?,?,?)',(ind, insnome, insidade)) ... connection.commit()

Removendo um item no banco de dados:

def removerdados(): ... idrem = raw_input('Digite o indice:') ... cur.execute('DELETE FROM contatos WHERE id=?',(idrem)) ... connection.commit()

Imprimindo os itens de um banco de dados:

def recuperadados(): ... sql = S̈ELECT * FROM contatosë... cur.execute (sql) ... result = cur.fetchall() ... for contato in result: ... print ë¨11 ... print Ïd: ? Nome: ? Idade: ??̈(contato[0],contato[1],contato[2])

Após efetuar-se cada operação deve-se utilizar o método *commit()* para que as alterações sejam gravadas no banco de dados.

Exercício:

- Desenvolva um programa que funcione como uma lista telefônica. A tabela do banco de dados deverá conter um número índice (inteiro), um nome (string) e um telefone (string). Este programa deverá ter a habilidade de inserir, remover e exibir as entradas registradas.

12 Expressões booleanas

Ao leitor que não estiver acostumado com o título acima, Expressões Booleanas são sentenças lógicas que seguem as leis da Álgebra de Boole. A Álgebra Booleana trabalha com valores lógicos, interpretando esses valores através de números binários, ao oposto do decimal, utilizando o valor 0 para definir uma operação FALSA e o valor 1 para definir uma operação VERDADEIRA. As operações entre valores lógicos são feitas de forma análoga às operações feitas entre conjuntos de elementos, sendo que cada um desses elementos é tratado como uma situação, podendo ser verdadeiros ou falsos.

Em Python os seguintes valores são interpretados como falso:

False None 0 () []

Ou seja, os valores **False** e **None**, o valor numérico 0 e as sequências vazias são denominadas falsas enquantos todos os outros valores possíveis são interpretados como verdadeiro.

Como na maioria das linguagens de programação, temos a possibilidade de utilizar a Álgebra Booleana para realizar testes lógicos usados em estruturas de controle. Com esses testes podemos conferir a veracidade de operações, além de podermos montar funções condicionais **if**, laços de repetição **while** e **for**, entre outros, que podem ser inseridos em qualquer lugar de seu programa. Observemos o exemplo abaixo:

>>> a = 6

>>> b = 7

```
>>> c = 42
>>> print 1, a == 6
>>> print 2, a == 7
>>> print 3, a == 6 and b == 7
>>> print 4, a == 7 and b == 7
>>> print 5, not a == 7 and b == 7
>>> print 6, a == 7 or b == 7
>>> print 7, a == 7 or b == 6
>>> print 8, not (a == 7 and b == 6)
>>> print 9, not a == 7 and b == 6
```

Teríamos como saída o seguinte resultado:

```
1 True
2 False
3 True
4 False
5 True
6 True
7 False
8 True
9 False
```

Para entender a lógica do que foi feito acima, devemos relembrar alguns conceitos de Álgebra Booleana. Abaixo temos uma breve descrição sobre os operadores lógicos:

- Operador "*and*": (em portugês "e") significa que ambas as afirmações devem ser verdadeiras, caso contrário a expressão é falsa.
- Operador "*or*": (em português "ou") significa que se ao menos uma das afirmações for verdadeira, então toda a expressão também o será.
- Operador "*not*": (em português "não") significa uma inversão lógica em uma expressão, o que era verdadeiro, se torna falso, e viceversa.

As tabelas abaixo podem ajudar a clarificar alguns desses conceitos:

Expression	Result
true and true	true
true and false	false
false and true	false
false and false	false
not true	false
not false	true
true or true	true
true or false	true
false or true	true
false or false	false

O programa abaixo mostra um exemplo de aplicação de Expressões Booleanas. Vamos dizer que precisamos de um programa que indique a situação de um aluno ao final do ano, sendo que elas são assim:

- Se o aluno obtiver nota maior ou igual e seis e não tiver mais de 10 faltas, ele passa direto;
- Se o aluno obtiver nota maior ou igual e seis e tiver mais de 10 faltas, ele fica de recuperação por falta;
- Se o aluno obtiver nota maior que quatro e menor que seis e não tiver mais de 10 faltas, ele fica de recuperação por nota;
- Se o aluno obtiver nota menor que quatro, ele repete direto;
- Se o aluno obtiver nota maior que quatro e menor que seis e tiver mais de 10 faltas, ele repete por não obter nota e por excesso de faltas;

Para montar esse programa utilizamos as Expressões Booleanas, já que precisamos fazer a verificação de duas situações, temos que verificar se o aluno tem nota e se ele tem presença.

```
>>> alunos = ['Fred','Suzana','Claudio','Puga','Robson','Gustavo']
>>> nota = [5.4, 6.2, 2.9, 9.9, 7.8, 4.9]
>>> faltas = [9, 5, 15, 2, 11, 12]
>>> contador = 0
>>> for aluno in alunos:
...         if nota[contador] >= 6.0 and faltas[contador] <= 10:
...                 print “Aluno: ”,aluno
...                 print “Nota final: ',nota[contador]
...                 print “Faltas: ”,faltas[contador]
...                 print “Resultado: Passou de ano'
...         elif nota[contador] >= 6.0 and faltas[contador] > 10:
...                 print “Aluno: ”,aluno
...                 print “Nota final: ',nota[contador]
...                 print “Faltas: ”,faltas[contador]
...                 print “Resultado: Recuperação por falta'
...         elif nota[contador] >= 4.0 and nota[contador] < 6.0 and faltas[contador] <= 10:
...                 print “Aluno: ”,aluno
...                 print “Nota final: ',nota[contador]
...                 print “Faltas: ”,faltas[contador]
...                 print “Resultado: Recuperação por nota'
...         elif nota[contador] >= 4.0 and nota[contador] < 6.0 and faltas[contador] > 10:
...                 print “Aluno: ”,aluno
...                 print “Nota final: ',nota[contador]
...                 print “Faltas: ”,faltas[contador]
...                 print “Resultado: Repetiu direto por não obter nota e por excesso de faltas'
...         elif nota[contador] < 4.0:
...                 print 'Aluno: ”,aluno
...                 print “Nota final: ”,nota[contador]
...                 print “Faltas: ”,faltas[contador]
...                 print “Resultado: Repetiu direto por nota”
```

Apêndice A - Módulos

Existem outros módulos que podem ser anexados à sua biblioteca.
Caso o leitor queira saber quais funções cada módulo desse possui, basta digitar o comando *help(nome_do_modulo)*.

abc
aepack (Mac)
aetools (Mac)
aetypes (Mac)
aifc
al (IRIX)
AL (IRIX)
anydbm
applesingle (Mac)
array
ast
asynchat
asyncore
atexit
audioop
autoGIL (Mac)
base64
BaseHTTPServer
Bastion
bdb
binascii
binhex
bisect
bsddb
buildtools (Mac)
bz2
calendar
Carbon
cd (IRIX)
cfmfile (Mac)
cgi
CGIHTTPServer
cgitb
chunk
cmath
cmd
code
codecs
codeop
collections
ColorPicker (Mac)
colorsys
commands (Unix)

compileall
compiler
ConfigParser
contextlib
Cookie
cookielib
copy
copy_reg
cPickle
cProfile
crypt (Unix)
cStringIO
csv
ctypes
curses
datetime
dbhash
dbm (Unix)
decimal
DEVICE (IRIX)
difflib
dircache
dis
distutils
dl (Unix)
doctest
DocXMLRPCServer
dumbdbm
dummy_thread
dummy_threading
EasyDialogs (Mac)
email
encodings
errno
exceptions
fcntl (Unix)
filecmp
fileinput
findertools (Mac)
FL (IRIX)
fl (IRIX)
flp (IRIX)
fm (IRIX)
fnmatch
formatter
fpectl (Unix)
fpformat
fractions
FrameWork (Mac)

ftplib
functools
future_builtins
gc
gdbm (Unix)
gensuitemodule (Mac)
getopt
getpass
gettext
gl (IRIX)
GL (IRIX)
glob
grp (Unix)
gzip
hashlib
heapq
hmac
- hotshot
htmlentitydefs
htmllib
HTMLParser
httplib
ic (Mac)
icopen (Mac)
imageop
imaplib
imgfile
imghdr
imp
imputil
inspect
io
itertools
jpeg (IRIX)
json
keyword
lib2to3
linecache
locale
- logging
macerrors (Mac)
MacOS (Mac)
macostools (Mac)
macpath
macresource (Mac)
mailbox
mailcap
marshal
math

md5
mhlib
mimetools
mimetypes
MimeWriter
mimify
MiniAEFrame (Mac)
mmap
modulefinder
msilib (Windows)
msvcrt (Windows)
multifile
multiprocessing
mutex
Nav (Mac)
netrc
new
nis (Unix)
nntplib
numbers
operator
optparse
os
ossaudiodev (Linux, FreeBSD)
parser
pdb
pickle
pickletools
pipes (Unix)
PixMapWrapper (Mac)
pkgutil
platform
plistlib
popen2
poplib
posix (Unix)
posixfile (Unix)
pprint
pstats
pty (IRIX, Linux)
pwd (Unix)
py_compile
pyclbr
pydoc
Queue
quopri
random
re
readline

repr
resource (Unix)
rexec
rfc822
rlcompleter
robotparser
runpy
sched
ScrolledText (Tk)
select
sets
sgmllib
sha
shelve
shlex
shutil
signal
SimpleHTTPServer
SimpleXMLRPCServer
site
smtpd
smtplib
sndhdr
socket
SocketServer
spwd (Unix)
sqlite3
ssl
stat
statvfs
string
StringIO
stringprep
struct
subprocess
sunau
sunaudiodev (SunOS)
SUNAUDIODEV (SunOS)
symbol
symtable
sys
syslog (Unix)
tabnanny
tarfile
telnetlib
tempfile
termios (Unix)
test
textwrap

thread
threading
time
timeit
Tix
Tkinter
token
tokenize
trace
traceback
tty (Unix)
turtle
types
unicodedata
unittest
urllib
urllib2
urlparse
user
UserDict
UserList
UserString
uu
uuid
videoreader (Mac)
W (Mac)
warnings
wave
weakref
webbrowser
whichdb
winsound (Windows)
wsgiref
xdrlib
xml
xmlrpclib
zipfile
zipimport
zlib

Referências

[1] Labaki , Josué , *Introdução ao Python - Módulo A*, Grupo Python, UNESP-Ilha Solteira.

[2] *The Python Tutorial 2.6, http://python.org/* , 2009.

[3] Hetland , Magnus Lie, *Beginning Python: from novice to professional*, Apress, 2005.

[4] http://www.python.org.br/wiki/ListaDeExercicios