

INTRODUÇÃO À LINGUAGEM PYTHON



02/04/2019

Curso de Extensão

Este curso é dirigido aos alunos do IFG, tanto do ensino médio técnico quanto do superior (Engenharias). Em nível introdutório, tem-se por objetivo apresentar a Linguagem de Programação Python. Essa linguagem tem como principal característica o ecletismo, permitindo ao usuário o desenvolvimento de aplicativos de toda sorte, desde o processamento de dados científicos até aplicações comerciais, para rodar no *desktop*, *tablet* ou *smartphone*, envolvendo banco de dados, servidores via Web ou não, manipulando grande massa de dados (*big data*).

Goiânia, Fev/2019

Cláudio A. Fleury

Introdução à Linguagem Python

CURSO DE EXTENSÃO

O QUE É PYTHON?

É uma linguagem de programação de alto nível, interpretada, interativa, versátil, de código aberto e legível aos seres humanos, orientada a objetos/imperativa/funcional/estruturada e de uso geral. Possui um sistema de tipificação dinâmica de variáveis, gerenciamento automático de memória e uma biblioteca padrão abrangente. Como outras linguagens dinâmicas, o Python é frequentemente usado como uma linguagem de *script*, mas também pode ser compilado em programas executáveis.

Nos exemplos a seguir, a entrada e a saída de comandos/respostas são diferenciadas pela presença ou ausência de ***prompts*** (`>>>` e `...`): para reproduzir o exemplo, você deve digitar os comandos após o ***prompt*** `>>>`. As linhas que não começam com um ***prompt*** são geradas pelo interpretador, são as respostas aos comandos. Observe que um ***prompt*** secundário sozinho em uma linha num exemplo significa que você deve digitar uma linha em branco; isso é usado para encerrar um comando de várias linhas.

Muitos dos exemplos nesta apostila, mesmo aqueles inseridos no ***prompt*** interativo, incluem comentários. Comentários em Python começam com o caractere ***hash*** '#', e se estendem até o final da linha física. Um comentário pode aparecer no início de uma linha ou após um espaço em branco ou código, mas não dentro de uma ***string***. Um caractere ***hash*** dentro de uma ***string*** é apenas mais um caractere da ***string***. Como os comentários são para esclarecer o código e não são interpretados pelo Python, eles podem ser omitidos ao se digitar os exemplos.

```
>>> # Exemplo de script Python
>>> porta = 1                               # porta aberta
>>> texto = "Deixe seu comentario em #comPythonehmaisfacil"
```

USANDO O PYTHON COMO UMA CALCULADORA:

O interpretador funciona como uma calculadora simples: você pode digitar uma expressão e aperta [Enter] para ter o valor calculado. A sintaxe da expressão é direta: os operadores +, -, *, / e / funcionam como na maioria das outras linguagens (Pascal, Java ou C). Os parênteses '()' podem ser usados para alterar a hierarquia de resolução dos operadores. Por exemplo:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6)/4
5
>>> 8/5   # divisão de operandos inteiros retorna resultado inteiro (Vs.2.7)
1
>>> 8./5 # divisão de operandos reais retorna resultado real (Vs.2.7)
1.6
>>> 5**2 # 5 ao quadrado
25
```

O sinal de igual (=) é usado para atribuir um valor a uma variável. Depois de uma atribuição nenhum resultado é exibido antes do próximo ***prompt***:

```
>>> largura = 10  
>>> altura = 5 * 9  
>>> largura * altura  
450
```

Se uma variável não for "definida" (atribuída um valor), tentar usá-la causará um erro:

```
>>> n          # tentativa de acessar uma variável não definida  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'n' is not defined  
>>>
```

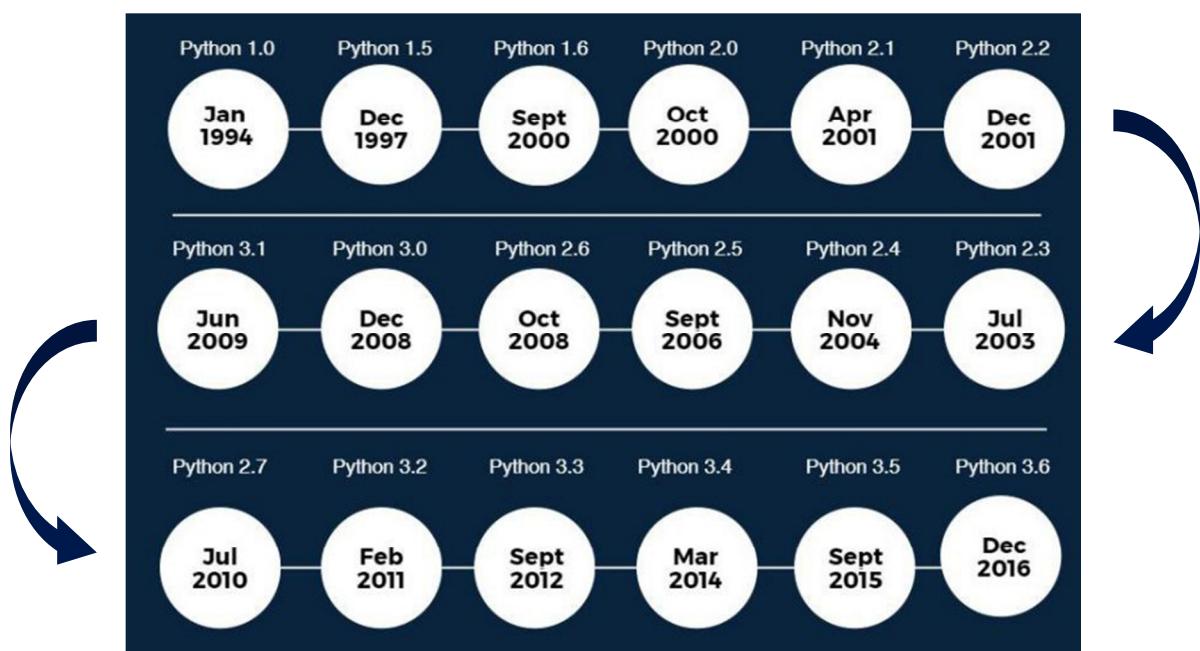
HISTÓRICO

A linguagem Python começou a ser desenvolvida por **Guido van Rossum** em 1989 (lançada oficialmente em 1999) na Centrum Wiskunde & Informatica (CWI), Holanda, como sucessora da linguagem ABC (inspirada na SETL¹) que era capaz de lidar com exceções e interagir com o sistema operacional Amoeba que ele estava ajudando a desenvolver.

Guido era fã do grupo humorístico **Monty Python**, criador do programa de comédia *Monty Python's Flying Circus* na televisão inglesa BBC (1969-1974). Ele quis homenagear o grupo dando o nome Python à linguagem.

Então, a denominação da linguagem não é devida à serpente Python, embora o ícone utilizado para representação sejam duas cobras estilizadas.

Linha do tempo do lançamento das versões:



Fonte: <http://www.trytoprogram.com/python-programming/history-of-python/>

¹ Linguagem de programação de "altíssimo nível", baseada na teoria matemática de conjuntos. Foi originalmente desenvolvida por Jacob Theodore Schwartz no Courant Institute of Mathematical Sciences na NYU no fim dos anos 1960. Lambert Meertens passou um ano com o grupo SETL na NYU antes de finalizar o projeto da ling. ABC.

FILOSOFIA

A filosofia central da linguagem Python inclui os seguintes preceitos:

- Bonito é melhor que feio.
- Explícito é melhor que implícito.
- Simples é melhor que complexo.
- Complexo é melhor que complicado.
- Legibilidade importa.

CARACTERÍSTICAS

Em vez de ter todas as suas funcionalidades incorporadas em seu núcleo, o Python foi projetado para ser facilmente extensível. Essa modularidade compacta tornou-a popular, pois se pode adicionar interfaces programáveis a aplicativos existentes. A intenção de Rossum era projetar uma linguagem com um pequeno núcleo, uma grande biblioteca padrão e um interpretador facilmente extensível... Isso foi resultado de suas frustrações com a ling. ABC, que adotava uma abordagem oposta.

PRINCIPAIS CARACTERÍSTICAS DA LINGUAGEM PYTHON:

1. **Legível e Interpretada:** Python é uma linguagem muito legível e cada instrução é traduzida individualmente e executada antes da instrução seguinte.
2. **Fácil de aprender:** Aprender Python é fácil por ela ser uma linguagem expressiva e de alto nível.
3. **Multiplataforma:** está disponível para execução em vários sistemas operacionais, tais como: Mac-OS, MS-Windows, Linux, Unix, Oracle Solaris etc.
4. **Open Source:** Python é uma linguagem de programação de código aberto.
5. **Grande Biblioteca Padrão:** a ling. Python vem com uma grande biblioteca padrão com códigos e funções úteis que podem ser usados enquanto se escreve código em Python.
6. **Gratuita:** a ling. Python é gratuita para download e uso.
7. **Manipulação de Exceção:** Uma exceção é um evento que pode ocorrer durante a execução do programa e que interrompe o fluxo normal do programa. A ling. Python permite o tratamento de exceções, o que significa que podemos escrever códigos menos propenso a erros e testar vários cenários que possam provocar uma exceção mais tarde.
8. **Recursos Avançados:** geradores e abrangência de lista (*list comprehension*). Veremos esses recursos mais tarde.
9. **Gerenciamento Automático de Memória:** a memória é limpa e liberada automaticamente. Você não precisa se preocupar em liberar memória em seus códigos.

PALAVRAS-CHAVE - PYTHON 2.7

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Fonte: <https://www.programiz.com/python-programming/keyword-list>

```
>>> from keyword import kwlist
>>> print(kwlist) # Vs. 2.7.9
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class',
'continue', 'def', 'del', 'elif', 'else', 'except', 'exec', 'finally', 'for',
'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'not', 'or', 'pass',
'print', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

INSTALANDO O INTERPRETADOR PYTHON

Você pode instalar o interpretador Python OFICIAL em “qualquer” Sistema Operacional..., tais como **Windows, Mac OS X, Linux/Unix, Solaris** e outros. Para fazer a instalação do Python em seu Sistema Operacional, acesse www.python.org/downloads/. Você terá a tela mostrada na figura seguinte.

The screenshot shows the Python.org homepage. At the top, there is a navigation bar with links for Python, PSF, Docs, PyPI, Jobs, and Community. Below the navigation bar is a search bar with a 'GO' button and a 'Socialize' button. The main content area features the Python logo and a brief introduction: "Python is a programming language that lets you work quickly and integrate systems more effectively. » Learn More". A red arrow points to the "Downloads" button in the top navigation bar, which is highlighted with a red box. To the right of the "Downloads" button, there is a snippet of Python code demonstrating list comprehensions and the enumerate function.

The screenshot shows the "Downloads" page for Windows on the Python.org website. The top navigation bar is identical to the homepage. The main content area features a large illustration of two boxes descending from the sky on parachutes. Below the illustration, there is a section titled "Download the latest version for Windows" with a "Download Python 3.7.2" button. Below this, there are links for "Python for Windows, Linux/UNIX, Mac OS X, Other" and "Pre-releases, Docker images". A red arrow points to the "Python 2.7.15" link in the "Release version" column of a table at the bottom of the page. The table also lists "Python 3.6.5" and provides download links and release notes for each version.

Release version	Release date	Click for more
Python 2.7.15	2018-05-01	Download Release Notes
Python 3.6.5	2018-03-28	Download Release Notes

Python 2.7.15

Release Date: 2018-05-01

Python 2.7.15 is a bugfix release in the Python 2.7 series.

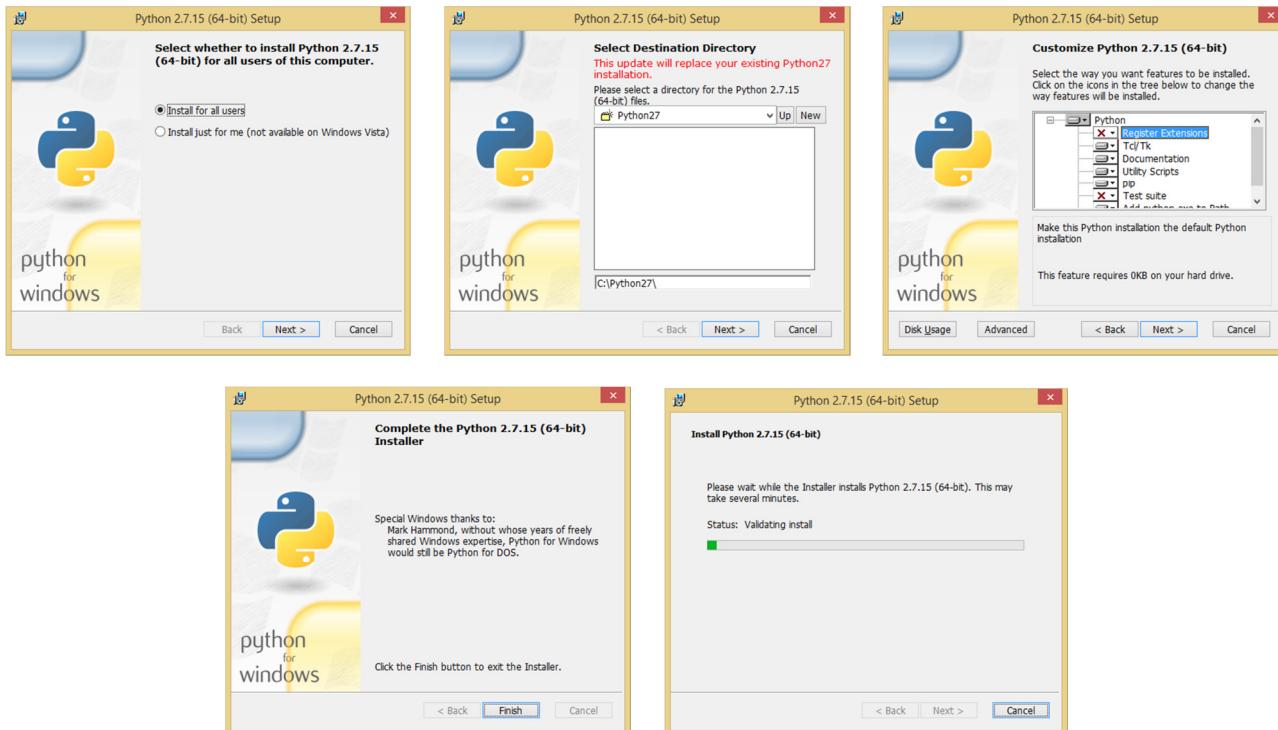
Files

Version	Operating System	Description	MD5 Sum	File Size	GPG
Gzipped source tarball	Source release		045fb3440219a1f6923fefdabde63342	17496336	SIG
XZ compressed source tarball	Source release		a80ae3cc478460b922242f43a1b4094d	12642436	SIG
macOS 64-bit/32-bit installer	Mac OS X	for Mac OS X 10.6 and later	9ac8c85150147f679f213addd1e7d96e	25193631	SIG
macOS 64-bit installer	Mac OS X	for OS X 10.9 and later	223b71346316c3ec7a8dc0bff5476d84	23768240	SIG
Windows debug information files	Windows		4c61ef61d4c51d615cbe751480be01f8	25079974	SIG
Windows debug information files for 64-bit binaries	Windows		680bf74bad3700e6b756a84a56720949	25858214	SIG
Windows help file	Windows		297315472777f28368b052be734ba2ee	6252777	SIG
Windows x86-64 MSI installer	Windows	for AMD64/EM64T/x64	0ffa44a86522f9a37b916b361ebebc552	20246528	SIG
Windows x86 MSI installer	Windows		023e49c9fba54914ebc05c4662a93ff	19304448	SIG
					19,3 MB

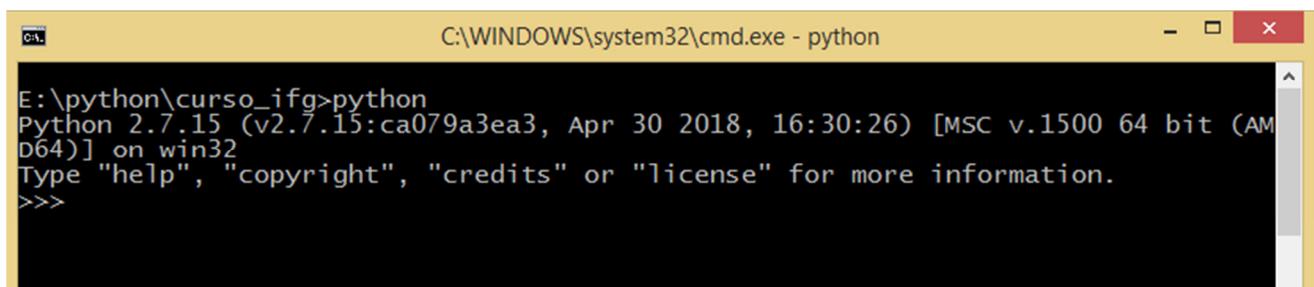
Este é o site oficial da linguagem Python. A página web detectará o sistema operacional instalado no seu computador, e recomendará a versão adequada a ser baixada. Como estou usando o Windows-64 no meu notebook, foram dadas as opções de download para **Python-2** e **Python-3** para Windows.

Neste curso usaremos a versão 2.7 da ling. Python, portanto recomendo que você baixe a versão mais recente do **Python-2** (à época da escrita desse texto era a versão **Python 2.7.15** - ver figura anterior).

As etapas de instalação são bem simples. Você só precisa escolher o diretório para instalação e clicar para avançar nas próximas etapas: botão [Next >].

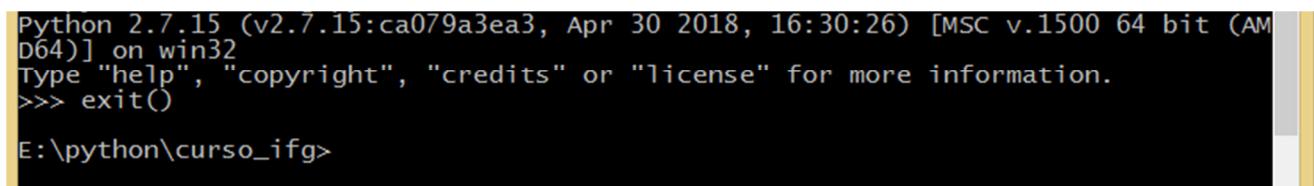


Após a instalação inicie uma janela de execução de comandos de linha (**terminal**) e carregue o interpretador Python instalado:



A screenshot of a Windows Command Prompt window titled "C:\WINDOWS\system32\cmd.exe - python". The window contains the following text:
E:\python\curso_ifg>python
Python 2.7.15 (v2.7.15:ca079a3ea3, Apr 30 2018, 16:30:26) [MSC v.1500 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>

Para abandonar o interpretador Python, use o atalho de teclado [Ctrl-Z]+[Enter] ou a função `exit()` (finalize a entrada com a tecla [Enter]).



A screenshot of a Windows Command Prompt window showing the Python interpreter exiting. The window contains the following text:
Python 2.7.15 (v2.7.15:ca079a3ea3, Apr 30 2018, 16:30:26) [MSC v.1500 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
E:\python\curso_ifg>

TÓPICOS BÁSICOS

1. [Variáveis e Operações](#)
2. [Tipos de Dados e Conversões](#)
3. [String](#)
4. [Intervalo](#)
5. [Lista](#)
6. [Tupla](#)
7. [Conjunto](#)
8. [Dicionário](#)
9. [Entrada de Dados](#)
10. [Controle de Fluxo de Execução](#)
11. [Função Definida pelo Usuário](#)
12. [Funções Incorporadas](#)
13. [Arquivos de E/S](#)
14. [Exceções](#)

PACOTES

15. [Numpy](#)
16. [Scipy](#)
17. [Matplotlib](#)
18. [Pandas](#)

1. Variáveis e Operações:

Um sinal de igual ‘=’ é usado para atribuir (armazenar) um valor a uma variável, como ocorre na grande maioria das linguagens de programação de computadores.

Sempre que pressionarmos qualquer tecla numérica, letra ou instrução no console do Python seguida da tecla [Enter], a resposta do interpretador Python será mostrada na linha seguinte. Mas, se atribuirmos um valor a uma variável, à esquerda do sinal de igual ‘=’, e pressionarmos [Enter], nenhuma informação será exibida na linha seguinte. Em vez disso, a informação digitada do lado direito do sinal de igual será armazenada na variável.

Uma variável é como uma pequena caixa na memória do computador, na qual se armazena qualquer informação. Quando declaramos uma variável, o computador aloca determinada memória para armazenar essa variável. O endereço de memória de cada variável é único.

Operador de Atribuição (criação de variável)

No interpretador Python 2.7.15 digite:

```
>>> x = 'Python + Machine Learning = Sucesso'
>>> print x
Python + Machine Learning = Sucesso
>>>
>>> ML = 2019
>>> print ML
2019
```

```
>>> xl = ML + 10
>>> print xl
2019
>>>
```

No exemplo anterior: a *string* 'Python + Machine Learning = Sucesso' é armazenada numa variável chamada *x*. Na próxima figura, a instrução *print* apresenta o valor armazenado na variável *x*. Depois foi atribuído o valor 2019 à variável *ML*, e então mostrado na tela. Na linha seguinte foi feito uma operação aritmética de soma *ML* + 10 e o resultado atribuído à variável *xl*.

Operadores Aritméticos: soma (+), subtração (-), multiplicação (*), divisão (/), resto da divisão inteira (%), divisão inteira (//), potenciação (**). A prioridade de execução segue os padrões matemáticos. Usando parênteses pode-se definir que parte da operação será feita antes da operação.

Oper.	Uso	Descrição
+	a + b	Soma os operandos em ambos lados do operador.
-	a - b	Subtrai o operando do lado direito do operando do lado esquerdo.
*	a * b	Multiplica os operandos em ambos lados do operador.
/	a / b	Divide o operando do lado esquerdo pelo operando do lado direito do operador (Python 2.7: retorna valor inteiro se operandos são inteiros).
%	a % b	Retorna o resto da divisão inteira do operando do lado esquerdo pelo operando do lado direito do operador.
**	a ** b	Retorna a potenciação do operador do lado esquerdo elevado ao operador do lado direito do operador.
//	a // b	Retorna o quociente da divisão dos operandos, onde o resultado é do tipo inteiro. Mas se um dos operandos for negativo, o resultado é flutuante, isto é, arredondado para longe de zero (em direção ao infinito negativo).

```
>>> 12-4*9
-24
>>> (12-4)*9
72
>>> 12-(4*9)
-24
```

Atenção: divisão com valores inteiros apresenta resultado também inteiro. Para obter resultado fracionário um dos operandos deve ser do tipo real (*float*).

```
>>> 8/6
1
>>> 8./6
1.333333333333333
>>> 12.5/3
4.166666666666667
>>> 12.5//3
4.0
>>> 13 % 5
3
>>> 2 ** 10
1024
>>>
```

Operadores Relacionais

Oper.	Uso	Descrição
<code>==</code>	<code>a == b</code>	Retorna True se os operandos são iguais, e retorna False caso contrário
<code>!=</code>	<code>a != b</code>	Retorna True se os operandos são diferentes, e retorna False caso contrário
<code><></code>	<code>a <> b</code>	Retorna True se o operando do lado esquerdo do operador for maior que o operando do lado direito, e retorna False caso contrário
<code>></code>	<code>a > b</code>	Retorna True se o operando do lado esquerdo do operador for menor que o operando do lado direito, e retorna False caso contrário
<code><</code>	<code>a < b</code>	Retorna True se o operando do lado esquerdo do operador for maior ou igual ao operando do lado direito, e retorna False caso contrário
<code>>=</code>	<code>a >= b</code>	Retorna True se o operando do lado esquerdo do operador for menor ou igual ao operando do lado direito, e retorna False caso contrário
<code><=</code>	<code>a <= b</code>	Retorna True se o operando do lado esquerdo do operador for menor que o operando do lado direito, e retorna False caso contrário

Operadores Lógicos

Oper.	Uso	Descrição
<code>and</code>	<code>a and b</code>	Retorna True se os operandos são True , e retorna False caso contrário
<code>or</code>	<code>a or b</code>	Retorna True se um ou outro operando é True , e retorna False caso contrário
<code>not</code>	<code>not a</code>	Complementa o valor do operando booleano 'a'

Operadores bit a bit: atua nos operandos no nível de bits.

Oper.	Uso	Descrição
<code>&</code>	<code>a & b</code>	Operação E: copia um bit para o resultado se ele existe em ambos operandos.
<code> </code>	<code>a b</code>	Operação OU: copia um bit para o resultado se ele existe pelo menos em um dos operandos.
<code>^</code>	<code>a ^ b</code>	Operação XOU: copia um bit para o resultado se ele estiver ligado (1) em um dos operandos, mas não em ambos.
<code>~</code>	<code>~a</code>	Operação Complemento 1: comuta os bits do operando.
<code><<</code>	<code>a << b</code>	Operação Desloc. à Esq.: desloca para esquerda os bits do operando à esquerda do operador da quantidade de bits indicada pelo operador à direita.
<code>>></code>	<code>a >> b</code>	Operação Desloc. à Dir.: desloca para direita os bits do operando à esquerda do operador da quantidade de bits indicada pelo operador à direita.

Operadores de Pertinência: indica a existência/pertinência de um elemento num iterável.

Oper.	Uso	Descrição
<code>in</code>	<code>x in y</code>	Retorna True se o operando à esquerda do operador estiver presente na sequência à direita do operador.
<code>not in</code>	<code>x not in y</code>	Retorna True se o operando à esquerda do operador não estiver presente na sequência à direita do operador

Operadores de Identidade: indica a existência/pertinência de um elemento num iterável.

Oper.	Uso	Descrição
<code>is</code>	<code>x is <tipo></code>	Retorna True se ambos os operandos se referirem ao mesmo tipo de objeto..
<code>is not</code>	<code>x is not <tipo></code>	Retorna False se as variáveis de cada lado do operador apontarem para o mesmo tipo de objeto e True de outra forma.

2. Tipos de Dado e Conversões (casting, moldagem)

Dados em Python podem ser um dos cinco tipos a seguir: Números, None, Sequências (listas, tuplas), Conjuntos ou Mapeamentos.

Conversão de Tipo significa converter variáveis de um tipo de dado em outro. O Python possui algumas funções internas para conversão de tipos. Até agora vimos exemplos de variáveis de tipos de dados inteiros, reais (ponto flutuante) e de *strings* (cadeias de caracteres). Para converter estes tipos, as funções são, respectivamente - `int()`, `float()`, `str()`.

Conversão para Inteiros: A função `int()` é usada para converter *strings* ou *floats* em inteiros.

```
>>> int('2533')
2533
>>> int('2533.45')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '2533.45'
>>>
```

Observe que se a *string* não resultar num valor inteiro então uma mensagem de erro de valor será mostrado ao se solicitar a conversão de tipo para inteiro.

```
>>> int(2533.45)
2533
>>>
```

Conversão para Strings: Use a função `str()` sem qualquer restrição para gerar uma cadeia de caracteres.

```
>>> str(2533.45)
'2533.45'
```

Exercício: Qual é a explicação para a seguinte mensagem de erro?

```
>>> str(2.533,45)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: str() takes at most 1 argument (2 given)
>>>
```

Quando escrevemos múltiplas variáveis na instrução `print`, conversões de *strings* devem ser usadas.

```
>>> print 'Real = ' + str(255.45) + ' Inteiro = ' + str(54)
Real = 255.45 Inteiro = 54
>>>
```

Conversão para Reais: A função `float()` é usada para converter de *strings* ou inteiros em valores reais.

```
>>> float(54)
54.0
>>> float('54.221')
54.221
>>>
```

3. String

As *strings* são usadas para armazenar informações de texto, tais como nome, endereço, mensagens etc. O Python monitora todos os elementos da *string* como uma sequência de caracteres. Por exemplo, o Python entende que a *string* "IFG" é uma sequência de letras numa ordem específica. Isso significa que poderemos usar a indexação para capturar letras específicas (como a primeira ou a última letra).

índices positivos	0	1	2	3	4	5
	P	y	t	h	o	n
índices negativos	-6	-5	-4	-3	-2	-1

- *Criando Strings*

Para criar uma *string* em Python você precisa usar aspas simples ou aspas duplas. Por exemplo (usando comentário na primeira linha):

```
>>> # palavra única
... 'IFG'
'IFG'
>>> # uma frase
... 'Bem-vindos ao Curso de Extensão do IFG 2019-1'
'Bem-vindos ao Curso de Extensão do IFG 2019-1'
>>> # erro de syntax
... 'Valor de 'x' no programa?'
  File "<stdin>", line 2
    'Valor de 'x' no programa?'
      ^
SyntaxError: invalid syntax
>>>
```

A causa do erro na *string* acima é a aspa simples delimitando duas *strings* e o caractere 'x' ficou sem delimitação entre elas. Solução: use combinações de aspas duplas e simples para obter a declaração correta.

```
>>> # erro de syntax
... "Valor de 'x' no programa?"
```

- *Imprimindo Strings*

Usar a sequência de caracteres (*string*) no prompt (>>>) do interpretador mostrará automaticamente seu valor, mas a maneira correta de exibir as *strings* na sua saída é usar a instrução de impressão: *print*.

```
>>> u"Aprendizado de Máquina"
u'Aprendizado de Máquina'
>>> print u"Aprendizado de Máquina"
Aprendizado de Máquina
>>>
```

- *Diferenças na Impressão em Python 2 e 3*

Na versão 2 a impressão é realizada por uma instrução (sentença, comando) enquanto que na versão 3 é uma função que faz a impressão: *print()*.

```
>>> print 'Aprendizado de Máquina'  
Aprendizado de Máquina  
>>> print 'Vamos aprender!'  
Vamos aprender!  
>>> print 'Use \n para imprimir uma nova linha'  
Use  
    para imprimir uma nova linha  
>>> print '\n'  
  
>>> print 'Entendeu?'  
Entendeu?  
>>>
```

Na versão 3 você imprime da seguinte forma: `print('Olá Mundo!')`. Se você quer usar esta funcionalidade no Python-2, você pode importar o formulário do módulo futuro.

Atenção: depois de importar isso, você não poderá mais escolher o método de declaração de impressão. Então, escolha o que você preferir, dependendo da sua instalação do Python e continue com ele.

```
>>> from __future__ import print_function  
>>> print(34)  
34  
>>> print 'Entendeu?'  
  File "<stdin>", line 1  
    print 'Entendeu?'  
          ^  
SyntaxError: invalid syntax  
>>>
```

Uma instrução **future** é uma diretiva para o compilador de que um módulo específico deve ser compilado usando sintaxe ou semântica que estará disponível em uma versão futura do Python. A declaração **future** destina-se a facilitar a migração para versões futuras do Python que introduzem alterações incompatíveis à linguagem. Ela permite o uso dos novos recursos por módulo antes do lançamento no qual o recurso se torna padrão.

É como dizer "Como esse é o Python v2.7, use essa função **print** diferente que também foi adicionada ao Python v2.7, depois que ela foi adicionada no Python 3. Então, meu 'print' não será mais uma instrução (por exemplo **print "mensagem"**) mas uma função (por exemplo, **print("mensagem")**). Dessa forma, quando meu código é executado em Python 3, **print** não irá quebrar o programa (dar pau!).

Também podemos usar uma função chamada **len()** para verificar o tamanho de uma string.

```
>>> len('Entendeu?')  
9  
>>>
```

- *Indexação e Fatiamento de Strings*

Sabemos que **strings** são sequências de caracteres, o que significa que o Python pode usar índices para acessar partes da sequência.

Em Python, usamos colchetes [] depois de um objeto para trazer o conteúdo do índice. Também devemos notar que, para o Python, a indexação começa em 0 (zero), ou seja, o

primeiro caractere de uma *string* é referenciado pelo índice 0. Vamos criar um novo objeto chamado 's' e realizar alguns exemplos de indexação.

```
>>> # atribuindo uma seq. de caracteres a um objeto 's'
... s = 'Bom dia Bia!'
>>> # verificando
... s
'Bom dia Bia!'
>>> # imprimindo o objeto
... print s
Bom dia Bia!
>>>
```

Acessando um caractere da sequência de caracteres...

```
>>> # primeiro caractere
... s[0]
';B
>>> # segundo caractere
... s[1]
';o
>>> # sétimo caractere
... s[6]
';a
>>> # último caractere
... s[-1]
';|
>>> # penúltimo caractere
... s[-2]
';a
>>>
```

Acessando partes da sequência de caracteres usando fatiamento (*slicing*)

```
>>> # acessando partes da string
... s[0:3]
'Bom'
>>> s[:3]
'Bom'
>>> s[4:len(s)]
'dia Bia!'
>>> s[4:]
'dia Bia!'
>>>
```

Observe o primeiro fatiamento acima: `s[0:3]`. Aqui estamos dizendo ao Python para acessar os caracteres de `s`, do índice 0 até 3, não incluindo o índice 3 (quarta posição). Você notará esse comportamento muitas vezes em Python, onde as declarações geralmente estão no contexto "até, mas não incluindo".

Também podemos usar a notação de índice e fatia para capturar elementos de uma sequência com um determinado incremento (o padrão é passo unitário). Por exemplo, podemos usar dois dois-pontos em uma linha e, em seguida, um número especificando a frequência para acessar os elementos. Por exemplo:

```
>>> # usando um passo diferente no acesso aos caracteres
... s[::-2]
'BmdaBa'
>>>
```

Acessando em ordem inversa:

```
>>> s[::-1]  
'Baid moB'>>>
```

- ### • Propriedades da Strings

É importante notar que as *strings* têm uma propriedade conhecida como **imutabilidade**. Isso significa que, uma vez que uma *string* é criada, os elementos dentro dela não podem ser alterados ou substituídos. Por exemplo:

```
>>> s
'Bom dia Bia!'
>>> s[0] = 'C'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
```

Observe como o erro nos diz diretamente o que não podemos fazer: não admite atribuição do item!

Algo que podemos fazer é concatenar *strings*!

```
>>> s + ' Vc pode me servir um cafezinho?'  
'Bom dia Bia! Vc pode me servir um cafezinho?'  
>>> s  
'Bom dia Bia!'  
>>> t = s + ' Vc pode me servir um cafezinho?'  
>>> print t  
Bom dia Bia! Vc pode me servir um cafezinho?  
>>>
```

Podemos usar o símbolo de multiplicação para criar repetição de caracteres!

Ajuntando (concatenando) Strings:

```
>>> livre = ['Terça', 'Quinta', 'Sábado']
>>> todos = ' - '.join(livre)
>>> print todos
Terça - Quinta - Sábado
>>>
```

- **Métodos Nativos de Strings**

Objetos em Python geralmente possuem métodos internos. Esses métodos são funções associadas ao objeto (aprenderemos sobre isso com muito mais profundidade posteriormente) que podem executar ações ou comandos no próprio objeto.

Nós acessamos os métodos de um objeto usando um ponto '.' e depois o nome do método: **objeto.método(parâmetros)**, onde os parâmetros são argumentos extras que podemos passar ao método. Não se preocupe com os detalhes se não fizeram sentido agora. Mais tarde

criaremos nossos próprios objetos e métodos! Aqui estão exemplos de métodos internos dos objetos *strings*:

```
>>> s
'Bom dia Bia!'
>>> s.upper()
'BOM DIA BIA!'
>>> s.lower()
'bom dia bia!'
>>> # fatia uma string por espaços em branco (caractere padrão)
... s.split()
['Bom', 'dia', 'Bia!']
>>> s.split('dia')
['Bom ', ' Bia!']
>>>
```

Existem muitos mais métodos para objetos *strings* do que os abordados aqui.

- *Formatação de Impressão*

Python tem formatadores de *string* impressionantes. Vamos mostrar os casos de uso mais comuns cobertos pelas API's de estilo de formatação de *strings*, antiga e nova.

À moda antiga:	À moda nova:
nome = "Matheus" prof = "Programador"	
titulo = "%s, o %s" % (nome, prof)	titulo = "{}, o {}".format(nome, prof)
print titulo	
'Matheus, o Programador'	

Formatação básica: A formatação posicional simples é provavelmente o caso de uso mais comum. Seu uso é mais indicado quando a ordem dos argumentos não precisa ser alterada e você tiver poucos elementos que queira concatenar.

Como os elementos não são representados por algo tão descritivo quanto um nome, esse estilo simples deve ser usado apenas para formatar uma quantidade relativamente pequena de elementos.

À moda antiga:	À moda nova:
print '%s, %s.' % ('Fleury', 'Cláudio')	print '{}, {}'.format('Fleury', 'Cláudio')
Fleury, Cláudio.	
print '%s, %.1s.' % ('Fleury', 'Cláudio')	print '{}, {:.1}'.format('Fleury', 'Cláudio')
Fleury, C.	

Com a nova formatação de estilo, é possível (e obrigatório no Python 2.6) dar aos espaços reservados um índice posicional explícito. Isso permite reorganizar a ordem de exibição sem alterar os argumentos. Esta operação não está disponível na formatação antiga.

```
>>> # à moda nova
... print {1}, {0}'.format('Cláudio', 'Fleury')
Fleury, Cláudio.
>>>
```

Formatação de números inteiros (%d) e reais (%f):

```
>>> # à moda antiga
... print '%d %f' % (54, 2.334343)
54 2.334343
>>> print '%4d %6.3f' % (54, 2.3349)
 54 2.335
>>> # à moda nova
... print '{1:f} {0:d}'.format(54, 2.334343)
2.334343 54
>>> print '{:6.2f}'.format(2.334343)
   2.33
>>>
```

Código comumente encontrado nas outras linguagens:

```
>>> usuario = 'Maria'
>>> if usuario == 'Maria':
...   print '-----'
...   print usuario
...   print '-----'
...
-----
Maria
-----
>>>
```

Código pythonico:

```
>>> usuario = 'Maria'
>>> if usuario == 'Maria':
...   print '{0}\n{1}\n{0}'.format('-'*30, usuario)
...
-----
Maria
-----
>>>
```

4. Intervalo

O tipo **range** (intervalo de valores) representa uma sequência imutável de números e é comumente usado para repetir um determinado número de vezes em laços de repetição **for**.

Os intervalos podem ser construídos de duas formas:

- **range(final)**
- **range(início, final[, passo])**

Os argumentos para o construtor de **range** devem ser inteiros (seja o **int** nativo ou qualquer outro objeto que implemente o método especial **__index__**). Se o argumento **passo** for omitido, o padrão será 1 (um). Se o argumento **início** for omitido, o padrão será 0 (zeros). Se o **passo** for zero, um erro **ValueError** será gerado.

Para passo positivo, o conteúdo de um intervalo r é determinado pela fórmula:

$$r[i] = \text{início} + \text{passo} * i, \text{ onde } i \geq 0 \text{ e } r[i] < \text{final}$$

Para passo negativo, o conteúdo de um intervalo r é determinado pela fórmula:

$$r[i] = \text{início} + \text{passo} * i, \text{ onde } i \geq 0 \text{ e } r[i] > \text{final}$$

Um objeto **range** estará vazio se $r[0]$ não atender à restrição de valor. Os intervalos suportam índices negativos, mas estes são interpretados como indexação a partir do final da sequência determinada pelos índices positivos.

```
>>> r = range(0, 20, 2)
>>> r
range(0, 20, 2)
>>> 11 in r
False
>>> 10 in r
True
>>> r.index(10)
5
>>> r[5]
10
>>> r[:5]
range(0, 10, 2)
>>> r[-1]
18
```

Comparação de objetos do tipo **range** para igualdade (`==`) ou diferença (`!=`) são realizados como na comparação de sequências. Ou seja, dois objetos de intervalo são considerados iguais se eles representam a mesma sequência de valores. Observe que dois objetos de intervalo que se compararam como iguais podem ter diferentes atributos de início, final e passo, por exemplo: `range(0) == range(2,1,3)` ou `range(0,3,2) == range(0,4,2)`.

5. Lista:

As listas **são coleções de objetos heterogêneos**, os quais podem ser de qualquer tipo, inclusive outras listas. As listas podem ser homogêneas (dados de um mesmo tipo) ou heterogêneas, o que as tornam ferramentas muito poderosas no Python. Uma única lista pode conter dados de qualquer tipo, tais como: números inteiros, números reais (ponto flutuante), strings, ou qualquer outro objeto. As listas também são muito úteis na implementação de pilhas e filas. As listas são mutáveis e, portanto, podem ser alteradas mesmo após sua criação.

No Python, ***list*** é um **tipo de recipiente** (container) de estrutura de dados que é usado para armazenar vários dados ao mesmo tempo. Ao contrário dos conjuntos (**sets**), a lista é ordenada e tem uma contagem definida. Os elementos em uma lista são indexados de acordo com uma sequência definida e a indexação de uma lista é feita com o índice 0 (zero) para o primeiro valor armazenado. Cada elemento da lista tem seu lugar definido na lista, o que permite a existência de elementos duplicados na lista, com cada elemento tendo seu próprio lugar na memória.

A lista é uma ferramenta útil para preservar uma sequência de dados e para "iterar" sobre ela (acesso a cada um de seus elementos).

```
>>> # exemplo de uma lista em Python
... placas = ['RPi', 'BeagleBone', 'Arduino']
>>> # mostrando o primeiro e terceiro elementos da lista
... print placas[0], placas[2]
Rpi Arduino
>>> print placas
['RPi', 'BeagleBone', 'Arduino']
>>>
```

- *Lista Vazia*

```
>>> lista_vazia = []
>>> print lista_vazia
[]
>>>
```

- *Lista Misturada (heterogênea)*

```
>>> mix = [3.141592653589, -5, 'Tudo certo?', True, '*']
>>> print mix
[3.141592653589, -5, 'Tudo certo?', True, '*']
>>>
```

- *Lista 2D*

```
>>> lista2D = [['12', 3, True],
...               [0.1, 'ok', 22],
...               [-5, 'Tudo certo?', True]]
>>> print lista2D
[['12', 3, True], [0.1, 'ok', 22], [-5, 'Tudo certo?', True]]
>>> print lista2D[0][1]
3
>>> for linha in lista2D:
...     for elem in linha:
...         print elem,
...     print
...
12 3 True
0.1 ok 22
-5 Tudo certo? True
>>>
```

- *Fatiamento de Lista*

```
>>> num = range(1,10)
>>> print num
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print num[0:3]      # acessa do 1o. ao 3o. elementos
[1, 2, 3]
>>> print num[2:-2]     # acessa do 3o. ao penúltimo elementos
[3, 4, 5, 6, 7]
>>>
```

- *Soma dos Números da Lista (loop)*

```
>>> pesos = [56, 73, 92, 32, 45]
>>> soma = 0
>>> for peso in pesos:
...     soma += peso
...
>>> print 'Soma dos Pesos: ', soma
Soma dos Pesos: 298
```

E se a lista contiver elementos não numéricos?

```
>>> pesos = [56, 73, 92, 'ok', 32, 45, True, "vida"]
>>> soma = 0
>>> for peso in pesos:
...     soma += peso
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: unsupported operand type(s) for +=: 'int' and 'str'
>>>
```

O interpretador Python indica erro para a tentativa de somar elemento do tipo 'str' com 'int'.

Prevenindo-se do referido erro:

```
>>> pesos = [56, 73, 92, 'ok', 32, 45, True, "vida"]
>>> soma = 0
>>> for peso in pesos:
...     if type(peso) == int:          # verificação do conteúdo do elemento
...         soma += peso
...
>>> print 'Soma dos Pesos: ', soma
Soma dos Pesos: 298
```

- Apresentação da Lista usando laço de repetição (loop)

```
>>> cores = ["azul","amarelo","vermelho","verde","roxo"]
>>> for cor in cores:
...     print cor
...     if cor == "amarelo":
...         print "Amarelo é a minha cor favorita!"
...
azul
amarelo
Amarelo é minha cor favorita!
vermelho
verde
roxo
>>>
```

Exercícios:

1. Criar uma lista com os números inteiros positivos múltiplos de 7, de 7 a 70. Mostrar os elementos da lista separados, um por linha.
2. Idem, porém mostrar apenas os elementos cuja soma de dígitos seja ímpar.
3. Idem, porém para múltiplos até 700, mostrados em sequência.
4. Idem, porém para múltiplos até N (entrada do usuário).

- Alteração da Lista

```
>>> pesos = [56, 73, 92, 'ok', 32, 45, True, "vida"]
```

```
>>> pesos[0] = 100
>>> print pesos
[100, 73, 92, 'ok', 32, 45, True, "vida"]
>>>
```

- Incluindo e Excluindo itens da Lista

```
>>> pesos = [56, 73, 92, 'ok', 32, 45, True, "vida"]
>>> pesos.insert(2,"alegria")
>>> print pesos
[56, 73, 'alegria', 92, 'ok', 32, 45, True, "vida"]
>>> pesos.remove(45)
>>> pesos
[56, 73, 'alegria', 92, 'ok', 32, True, "vida"]
>>> del pesos[1]
>>> pesos
[56, 'alegria', 92, 'ok', 32, True, "vida"]
>>> ultimo = pesos.pop() # retorna último elemento, retirando-o da lista
>>> print pesos, '\n', ultimo
[56, 'alegria', 92, 'ok', 32, True]
"vida"
>>>
```

- Divisão de String em Itens da Lista (split() é um método da classe String)

```
>>> sedans = 'focus/jetta/408/cruze/civic/corolla/mercedes c180/sentra/cerato'
>>> lista_carros = sedans.split("/") # sedans é uma variável string
>>> print lista_carros
['focus', 'jetta', '408', 'cruze', 'civic', 'corolla', 'mercedes c180', 'sentra',
'cerato']
```

- Concatenando Itens da Lista (join() é um método da classe String)

```
>>> ' '.join(lista_carros)
'focus jetta 408 cruze civic corolla mercedes c180 sentra cerato'
>>> ', '.join(lista_carros)
'focus, jetta, 408, cruze, civic, corolla, mercedes c180, sentra, cerato'
>>> '|'.join(lista_carros)
'focus | jetta | 408 | cruze | civic | corolla | mercedes c180 | sentra | cerato'
>>>
```

- Tamanho da Lista (quantidade de itens) (len() é uma função incorporada)

```
>>> len(lista_carros)
9
>>>
```

- Acessando os Itens da Lista em Ordem Reversa

```
>>> lista_carros
['focus', 'jetta', '408', 'cruze', 'civic', 'corolla', 'mercedes c180', 'sentra',
'cerato']
>>> lista_carros.reverse()
>>> lista_carros
['cerato', 'sentra', 'mercedes c180', 'corolla', 'civic', 'cruze', '408', 'jetta',
'focus']
```

```
>>>
```

- *List Comprehension*

Sintaxe: [expressão **for** item **in** lista]

Além das operações de sequência e os métodos da classe *list*, o Python inclui uma operação mais avançada chamada de abrangência de lista (*list comprehension*). Trata-se de uma maneira rápida de filtrar uma lista com base em um ou mais critérios estabelecidos pelo usuário.

List Comprehension é uma construção oriunda da programação funcional, e equivale à seguinte descrição matemática:

$$R = \left\{ \frac{x}{2}, \forall x \in \mathbb{N}, 0 \leq x \leq 5 \right\}$$

Leia-se: *R* é o conjunto formado por todos os números do conjunto dos números naturais divididos por 2, desde que o número seja maior ou igual a zero e menor ou igual a 5.

```
>>> r = [x/2 for x in range(6)]
>>> r
[0, 0, 1, 1, 2, 2]
>>>
```

List Comprehension permite criar listas usando notação mais compacta do que seria possível usando laços de repetição com o comando **for**. Tenha cuidado, no entanto, pois *list comprehension* nem sempre é resposta para tudo. É fácil se deixar levar e escrever *lists comprehension* complexas e difíceis de ler. Às vezes, escrever mais código é melhor, especialmente se isso ajudar na legibilidade. Lembre-se: simples é melhor que complexo, e legibilidade importa!

```
>>> range(11)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> faixa = range(-10,11)
>>> faixa
[-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> R = [x/2 for x in faixa if x >= 0 and x <= 5]
>>> R
[0, 0, 1, 1, 2, 2]
>>> R = [x/2. for x in faixa if x >= 0 and x <= 5]
>>> R
[0.0, 0.5, 1.0, 1.5, 2.0, 2.5]
```

Exemplo:

```
>>> letras = ['a', 'b', 'c', 'd']
>>> print letras
['a', 'b', 'c', 'd']
>>> letras_maiusc = []
>>> for letra in letras:
...     letras_maiusc.append(letra.upper())
...
>>> print letras_maiusc
['A', 'B', 'C', 'D']
>>>
```

Usando *list comprehension*:

```
>>> letras = ['a', 'b', 'c', 'd']
>>> letras_maiusc = [letra.upper() for letra in letras]
>>> print letras_maiusc
['A', 'B', 'C', 'D']
>>>
```

Filtrando alguns valores, por exemplo: não incluir as letras maiúsculas 'A' e 'E'.

```
>>> letras = ['a', 'b', 'c', 'd']
>>> letras_maiusc = [x.upper() for x in letras if x not in ['a', 'e']]
>>> print letras_maiusc
['B', 'C', 'D']
>>>
```

Gerar uma lista de sedans com nome do modelo em maiúsculas e começando com 'c':

```
>>> [x.upper() for x in lista_carros if x[0]=='c']
['CRUZE', 'CIVIC', 'COROLLA', 'CERATO']
```

Idem, porém somente com a primeira letra do nome do modelo em maiúscula:

```
>>> [x[0].upper() + x[1:] for x in lista_carros if x[0]=='c']
['Cruze', 'Civic', 'Corolla', 'Cerato']
```

Agora usando o método `capitalize()` da classe `String` para colocar em maiúscula a primeira letra:

```
>>> [x.capitalize() for x in lista_carros if x[0]=='c']
['Cruze', 'Civic', 'Corolla', 'Cerato']
>>>
```

Métodos (funções) da classe *list*:

<code>append()</code>	acrescenta um único elemento ao final da lista
<code>clear()</code>	elimina todos os elementos da lista
<code>copy()</code>	retorna uma cópia superficial/rasa (<i>shallow</i>) de uma lista
<code>count()</code>	retorna a quantidade de ocorrências do elemento na lista
<code>extend()</code>	acrescenta elementos de uma lista a outra lista
<code>index()</code>	retorna o menor índice do elemento na lista
<code>insert()</code>	insere elementos à lista
<code>pop()</code>	remove o elemento da posição dada ou o último se não indicado a posição
<code>remove()</code>	remove um elemento da lista
<code>reverse()</code>	reverte a ordem dos elementos da lista
<code>sort()</code>	classifica (ordena) os elementos de uma lista

Outras **funções incorporadas** e compatíveis com listas:

<code>any()</code>	verifica se algum elemento de um iterável é True
<code>all()</code>	retorna True quando todos os elementos de um iterável é True
<code>ascii()</code>	retorna uma String contendo representação imprimível

<code>bool()</code>	converte um valor para Boolean
<code>enumerate()</code>	retorna um Object do tipo Enumerate
<code>filter()</code>	constrói um <i>iterator</i> a partir dos elementos que são True
<code>iter()</code>	retorna um <i>iterator</i> para um objeto
<code>list()</code>	cria uma lista
<code>len()</code>	retorna o comprimento de um Object
<code>max()</code>	retorna o maior elemento de um Object
<code>min()</code>	retorna o menor elemento de um Object
<code>map()</code>	aplica uma função a um iterável e retorna uma lista
<code>reversed()</code>	retorna um <i>iterator</i> de uma sequência revertida
<code>slice()</code>	cria um objeto <i>slice</i> especificado por <code>range()</code>
<code>sorted()</code>	retorna uma lista classificada a partir de um dado <i>iterable</i>
<code>sum()</code>	soma os itens de um <i>Iterable</i>
<code>zip()</code>	retorna um <i>Iterator</i> de tuplas

Exemplo: Encontrar números comuns a duas listas, usando laço **for**.

```
>>> lista_a = [0, 1, 2, 3, 4] # lista A
>>> lista_b = [2, 3, 4, 5]     # lista B
>>> lista_c = []              # números em comum
>>> for elem_a in lista_a:
...     for elem_b in lista_b:
...         if elem_a == elem_b:
...             lista_c.append(elem_a)
...
>>> print lista_c
[2, 3, 4]
>>>
```

Agora usando *list comprehension* (abrangência de lista):

```
>>> lista_a = [0, 1, 2, 3, 4] # lista A
>>> lista_b = [2, 3, 4, 5]     # lista B
>>> lista_d = [a for a in lista_a for b in lista_b if a == b]
>>> print lista_d
[2, 3, 4]
>>>
```

Operador **in**:

Se você quiser apenas verificar se existe um valor dentro de um **objeto iterável** (lista, tupla ou dicionário), a maneira mais rápida é fazer é usando o operador **in**:

Código não pythônico:

```
>>> cidade = 'Paris'
>>> achou = False
>>> if cidade == 'Nairobi' or cidade == 'Kampala' or cidade == 'Paris':
...     achou = True
...
...
```

```
>>> print achou  
True  
>>>
```

Código pythonico:

```
>>> cidade = 'Paris'  
>>> achou = cidade in ['Nairobi', 'Kampala', 'Paris']  
>>> print achou  
True  
>>>
```

Outro exemplo:

```
>>> dias_livres = ["Segunda", "Sexta"]  
>>> desejado = "Sábado"  
>>> if desejado in dias_livres:  
...     print("Eu estou disponível no(a)", desejado,"!")  
... else:  
...     print("Desculpe, mas tenho compromisso no(a)", desejado,"!")  
...  
Desculpe, mas tenho compromisso no Sábado!  
>>>
```

Exercício: Retornar os números não comuns das listas: lista_a = [0, 1, 2, 3, 4] e lista_b = [2, 3, 4, 5], usando **for** e depois **list comprehension**.

6. Tupla:

Uma tupla é uma sequência imutável de objetos Python. Tuplas são sequências, assim como as listas. As diferenças entre tuplas e listas são: as tuplas não podem ser alteradas diferentemente das listas, e as tuplas usam parênteses como delimitadores enquanto as listas usam colchetes. Criar uma tupla é tão simples quanto colocar diferentes valores separados por vírgulas e delimitados por parênteses.

- Construindo tuplas

A construção de uma tupla usa () com elementos separados por vírgulas. Por exemplo:

```
>>> tup = (15, 34, 92, 22)  
>>> print tup  
(15, 34, 92, 22)  
>>>
```

Assim como para as listas, a quantidade de elementos pode ser obtida com a função **len()**:

```
>>> len(tup)  
4  
>>>
```

Com objetos de tipos diversos:

```
>>> tur = (True, 232, "Matric.", -223.33, False, 11)
>>> print tur[2]
Matric.
>>> tur[3]
-223.33
>>>
```

Todas as operações de acesso a elementos, usadas para listas também valem para tuplas:

```
>>> tur[-1]
11
>>> tur[::-1]
(11, False, -223.33, 'Matric.', 232, True)
>>>
```

- *Métodos Básicos de Tupla*

Tuplas têm métodos embutidos, mas não tantos quanto aos métodos disponíveis para listas.

```
>>> # Use .count() para contar o número de vezes que um valor aparece
... tur.count(22)
0
>>> tur.count(232)
1
>>>
```

- *Imutabilidade*

Só relembrando, as tuplas são imutáveis, não se pode modificar uma tupla depois que ela já estiver definida...

```
>>> tur[1]
232
>>> tur[1] = 500
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
>>> tur.append(500)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

- *Quando usar tuplas*

Você pode estar se perguntando: "Por que se preocupar em usar tuplas quando elas têm menos métodos disponíveis?" Para ser honesto, as tuplas não são usadas tão frequentemente quanto as listas, mas são usadas quando a imutabilidade é necessária. Se no seu programa você está passando um objeto e precisa ter certeza de que ele não será alterado, então a tupla será a solução. Ela fornece uma forma conveniente de se obter integridade de dados.

7. Conjunto:

A estrutura **conjunto** (*set*, em inglês) pode ser usada na execução de operações de conjuntos matemáticos, tais como: união, interseção, diferença simétrica etc.

Um **conjunto** é uma **coleção não ordenada de elementos únicos** e que podem ser alterados, acrescentados ou excluídos.

- Criando um Conjunto

Podemos usar a função `set()` ou um par de chaves `{ }` para criar um conjunto vazio, ou podemos ainda criá-los com elementos a partir de outras estruturas iteráveis (lista, tupla, dicionário etc).

Exemplo: Criação e inserção de elementos num conjunto.

```
>>> A = set()                                # conjunto vazio, ou: A = {}  
>>> A  
set([])
```

- Acrescentando Elemento(s) ao Conjunto:

```
>>> A.add('ok')  
>>> A  
set(['ok'])  
>>>  
>>> A.add(1)  
>>> A.add(1)  
>>> A  
set([1, 'ok'])  
>>>  
>>> A.add(-8)  
>>> A  
set([-8, 1, 'ok'])  
>>> A.add(5)  
>>> A                               # coleção não ordenada  
set([-8, 1, 'ok', 5])
```

Observe que os elementos não guardam qualquer ordem no conjunto e que elementos já existentes não são repetidos ao serem adicionados mais de uma vez ao conjunto.

Exemplo: Supressão de elementos repetidos de uma lista ou tupla:

```
>>> lista = [22,22,23,25,25,25,26,26,27,28,28]  
>>> print set(lista)  
set([22, 23, 25, 26, 27, 28])  
>>>  
>>> tupla = tuple(lista)  
>>> tupla  
(22, 22, 23, 25, 25, 25, 26, 26, 27, 28, 28)  
>>>  
>>> B = set(tupla)  
>>> print B  
set([22, 23, 25, 26, 27, 28])
```

- Removendo Elemento(s) de um Conjunto

Um item específico pode ser removido do conjunto usando os métodos: **discard()** e **remove()**. A diferença entre esses métodos é que, ao usar o **discard(item)** se o **item** não existir no conjunto, então ação ocorre, enquanto o método **remove(item)** gerará um erro nessa situação.

```
>>> B.discard(23)
>>> B
set([22, 25, 26, 27, 28])
>>> B.discard(23)
>>> B.remove(23)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 23
>>>
```

- *União de Conjuntos*

A união dos conjuntos **A** e **B** é um conjunto de todos os elementos de ambos os conjuntos. União é realizada usando operador ‘|’ ou o método **union()**.

```
>>> A = {3,4,5}
>>> A
set([3, 4, 5])
>>> B = {4,5,6}
>>> print A | B           # união dos conjuntos A e B
set([3, 4, 5, 6])
>>>
>>> print A.union(B)      # união dos conjuntos A e B
set([3, 4, 5, 6])
>>>
```

- *Interseção entre Conjuntos*

A interseção entre **A** e **B** é um conjunto de elementos comuns a ambos os conjuntos. A interseção é realizada usando o operador ‘&’ ou o método **intersection()**.

```
>>> B.intersection(A)
set([4, 5])
>>> print B & A
set([4, 5])
```

- *Diferença entre Conjuntos*

A diferença entre os conjuntos **A** e **B** é representada por **A - B**, e é um conjunto formado pelos elementos que estão apenas em **A**, mas não em **B**. Da mesma forma, **B - A** é um conjunto formado pelos elementos que estão em **B** e que não estão em **A**. A diferença é realizada em Python pelo operador ‘-’ ou pelo método **difference()**.

```
>>> A - B                  # ou: A.difference(B)
set([3])
>>> print B - A            # ou: B.difference(A)
set([6])
>>>
>>> homens = {"Pedro", "Matheus"}
>>> mulheres = {"Ana", "Camila"}
```

```
>>> familia = homens | mulheres
>>> familia
set(['Matheus', 'Pedro', 'Camila', 'Ana'])
>>> empregado = {"Pedro"}
>>>
>>> desempregado = familia - empregado
>>> desempregado
set(['Matheus', 'Camila', 'Ana'])
>>> empregado - familia
set([])
```

- *Diferença Simétrica entre Conjuntos*

Diferença simétrica entre os conjuntos **A** e **B** é um conjunto de elementos em **A** e **B**, exceto aqueles que são comuns a ambos. A diferença simétrica é realizada usando o operador '^' ou usando o método **symmetric_difference()**.

```
>>> A.symmetric_difference(B)
set([3, 6])
>>> print A ^ B, B ^ A
set([3, 6]) set([3, 6])
>>>
>>> familia ^ empregado
set(['Matheus', 'Camila', 'Ana'])
>>> familia - empregado
set(['Matheus', 'Camila', 'Ana'])
>>>
```

Exercício: Mostre os números incomuns das listas **lista_a** = [0, 1, 2, 3, 4] e **lista_b** = [2, 3, 4, 5] usando **conjuntos**.

- *Acessando Elementos do Conjunto:*

Diferentemente das estruturas Lista e Tupla, os elementos do Conjunto não podem ser acessados via indexação:

```
>>> B
set([4, 5, 6])
>>> B[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing
>>> for elem in B:
...   print elem,
...
4, 5, 6
>>>
```

Funções incorporadas para conjuntos: **all()**, **any()**, **enumerate()**, **len()**, **max()**, **min()**, **sorted()** e **sum()**.

- *Conjuntos Imutáveis: **frozenset***

É uma estrutura que possui características de um conjunto, mas seus elementos não podem ser alterados/acrescidos/removidos depois de atribuídos. Enquanto as tuplas são listas imutáveis, os **frozensets** são conjuntos imutáveis.

Conjuntos não podem ser usados como chaves de dicionário, por serem mutáveis. Por outro lado, os **frozensets** podem ser usados como chaves de um dicionário. Podem ser criados usando a função **frozenset()**.

```
>>> A = frozenset([1, 2, 3, 4])
>>> B = frozenset([3, 4, 5, 6])
>>> A, B
(frozenset([1, 2, 3, 4]), frozenset([3, 4, 5, 6]))
>>> A ^ B, A | B
frozenset([1, 2, 5, 6]), frozenset([1, 2, 3, 4, 5, 6])
>>> A.add(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
>>>
```

Exercício: Mostre a quantidade de vogais existente numa string, usando conjunto(s).

8. Dicionário:

São estruturas de dados que implementam mapeamentos. Um mapeamento é uma coleção de associações entre pares de valores. De certa forma, um mapeamento é uma generalização do acesso de dados por índices usados em vetores e matrizes, exceto que num mapeamento os índices (ou chaves) podem ser de qualquer tipo de dado imutável, inclusive *string*.

O dicionário é uma coleção não ordenada de itens. Enquanto outros tipos de dados compostos (arranjos) têm apenas um valor para cada índice, um dicionário tem um par **chave:valor**. Os dicionários são otimizados para recuperar valores quando a **chave** for conhecida.

Se você estiver familiarizado com outras linguagens, pode pensar em dicionários como tabelas de espalhamento (*hash tables*).

- Criando um Dicionário

Um dicionário em Python consiste de uma chave e um valor associado. Esse valor pode ser praticamente qualquer objeto do Python.

Criar um dicionário é tão simples quanto colocar itens separados por vírgula entre chaves `{ }`. Um item tem uma **chave** e um **valor** correspondente, e é expresso como um par **chave : valor**. Embora os valores possam ser de qualquer tipo de dado e possam se repetir, as chaves devem ser do tipo imutável (*string*, número ou tupla) e devem ser exclusivas.

```
>>> # criando um dicionário vazio
... dici = {}
>>> # dicionário com chaves do tipo numérico 'inteiro'
```

```
... z = { 1 : 'manga', 2 : 'bola', 3 : 1220 }
>>> z[1]
'manga'
>>> z[1] = 'banana'
'banana'
>>> z
{ 1 : 'banana', 2 : 'bola', 3 : 1220 }
>>> z[3]
1220

>>> # criando dicionário com chaves mistas (vários tipos de dados)
... y = { 'nome': 'Maria', 'notas': [6,7,4], 5: 'ok' }
>>> y[5]
'ok'
>>> y['nome']
'Maria'
>>> y['notas']
[6, 7, 4]
>>> y['notas'][0]
[6]
```

Também podemos criar um dicionário usando a função incorporada `dict()` que requer como parâmetro uma lista de tuplas, cada uma com um par **chave:valor**, ou uma sequência de itens no formato **chave=valor**.

```
>>> # criando dicionário com a função dict()
... w = dict([('nome','Maria'), ('notas',[6,7,4]), (5,'ok')])
>>> w
{5: 'ok', 'notas': [6, 7, 4], 'nome': 'Maria'}
>>>
>>> a = dict(x=1,y=2)
>>> a
{'y': 2, 'x': 1}
>>> a['x']
1
>>>
```

Nós também podemos criar chaves via operador de atribuição. Por exemplo, se começássemos com um dicionário vazio, poderíamos adicionar elementos continuamente:

```
>>> d = {}           # dicionário vazio
>>> d['animal'] = 'gato'  # acrescenta o elemento 'animal':'gato'
>>> d['idade'] = 6      # acrescenta o elemento 'idade':6
>>> d
{'idade': 6, 'animal': 'gato'}
>>>
```

- Acessando Elementos de um Dicionário

Dicionário são estruturas mutáveis. Podemos adicionar novos itens ou alterar o valor dos itens existentes usando o operador de atribuição. Se a chave já estiver presente, o valor será atualizado, caso contrário, um novo par **chave : valor** será adicionado ao dicionário.

```
>>> d
{'idade': 6, 'animal': 'gato'}
>>> d['idade'] = 5
>>> d['cor'] = 'branco'
>>> d['nome'] = 'Teo'
>>> d
{'idade': 5, 'cor': 'branco', 'animal': 'gato', 'nome': 'Teo'}
>>>
```

- *Excluindo Elementos de um Dicionário*

Podemos remover um item específico em um dicionário usando o método **`pop()`**. Este método remove o item com a chave fornecida e retorna o valor.

O método **`popitem()`** pode ser usado para remover e retornar um item arbitrário (chave, valor) do dicionário. Todos os itens podem ser removidos de uma vez usando o método **`clear()`**. Também podemos usar a função incorporada **`del()`** para remover itens individuais ou o dicionário inteiro.

```
>>> y
{5: 'ok', 'notas': [6, 7, 4], 'nome': 'Maria'}
>>> y.pop(5)
'ok'
>>> y
{'notas': [6, 7, 4], 'nome': 'Maria'}
>>> del y['notas']
>>> y
{'nome': 'Maria'}
>>>

>>> d
{'idade': 5, 'cor': 'branco', 'animal': 'gato', 'nome': 'Teo'}
>>> del d
>>> d
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'd' is not defined
>>>
```

Outros tantos métodos estão disponíveis para uso com dicionários: `clear()`, `copy()`, `fromkeys(seq[, v])`, `get(key[,d])`, `items()`, `keys()`, `pop(key[,d])`, `popitem()`, `update([other])`, `values()`.

```
>>> notas = {}.fromkeys(['matematica','espanhol','portugues'],0)
>>> notas
{'portugues': 0, 'espanhol': 0, 'matematica': 0}
>>> for disciplina in notas.items():
...     print disciplina
...
('portugues', 0)
('espanhol', 0)
('matematica', 0)
>>> list(sorted(notas.keys()))
['espanhol', 'matematica', 'portugues']
>>>
```

- *Aninhamento de Dicionários*

O Python é poderoso com sua flexibilidade de aninhar objetos e na chamada de métodos. Vamos ver um dicionário aninhado dentro de outro dicionário:

```
>>> d = {'chave1':{'ch_aninh':{'ch_sub_aninh':1000}}}
>>> d
{'chave1': {'ch_aninh': {'ch_sub_aninh': 1000}}}
>>>
>>> d['chave1']['ch_aninh']['ch_sub_aninh']
1000
>>>

>>> pessoas = {1: {'nome':'Maria', 'idade':25, 'sexo':'Fem.', 'cor':'branca'}, 2: {'nome':'Luiz', 'idade':35, 'sexo':'Masc.'} }
>>>
>>> print pessoas
{1: {'idade': 25, 'sexo': 'Fem.', 'cor': 'branca', 'nome': 'Maria'}, 2: {'idade': 35, 'sexo': 'Masc.', 'nome': 'Luiz'}}
>>>
>>> print pessoas[1]['nome'] + ', ', pessoas[2]['nome']
Maria, Luiz
>>>
>>> pessoas[3] = {'nome':'Ana','idade':19,'sexo':'Fem.','casada':False}
>>> print pessoas
{1: {'idade': 25, 'sexo': 'Fem.', 'cor': 'branca', 'nome': 'Maria'}, 2: {'idade': 35, 'sexo': 'Masc.', 'nome': 'Luiz'}, 3: {'idade': 19, 'sexo': 'Fem.', 'casada': False, 'nome': 'Ana'}}
>>>
>>> for id, info in pessoas.items():
...     print u"\nIdentificação:", id
...     for carac in info:
...         print carac, ': ', info[carac]
...
Identificação: 1
idade : 25
sexo : Fem.
nome : Maria

Identificação: 2
idade : 35
sexo : Masc.
nome : Luiz

Identificação: 3
idade : 19
sexo : Fem.
casada : False
nome : Ana
```

- *Abrangência de Dicionário*

A abrangência do dicionário é uma maneira elegante e concisa de criar um novo dicionário a partir de um objeto iterável em Python. A abrangência do dicionário consiste num par **chave:valor**

seguido por uma instrução dentro de chaves { }. Aqui está um exemplo para fazer um dicionário em que cada item é um par formado por um número e seu cubo.

```
>>> cubos = {x:x*x*x for x in range(6)}
>>> print cubos
{0: 0, 1: 1, 2: 8, 3: 27, 4: 64, 5: 125}
```

Código equivalente usando a sentença **for**:

```
>>> cubos = {}
>>> for x in range(6):
...     cubos[x] = x*x*x
...
>>> print cubos
```

Uma abrangência de dicionário pode, opcionalmente, conter mais instruções **for** ou **if**. Uma instrução **if** pode filtrar itens para formar o novo dicionário. Veja um exemplo para criar um dicionário com apenas itens ímpares.

```
>>> quadrados_impares = {x: x*x for x in range(11) if x%2 == 1}
>>> print quadrados_impares
{1: 1, 3: 9, 9: 81, 5: 25, 7: 49}
>>>
```

- *Teste de Associação ao Dicionário*

Podemos testar se uma chave está num dicionário ou não usando a palavra-chave **in**. Observe que o teste de associação é válido somente para chaves, não para valores.

```
>>> print quadrados_impares
{1: 1, 3: 9, 9: 81, 5: 25, 7: 49}
>>>
>>> print 1 in quadrados_impares
True
>>> print 2 in quadrados_impares
False
>>> print 25 in quadrados_impares
False
>>>
```

- *Iterando por um Dicionário*

Usando um laço **for**, podemos iterar por cada chave do dicionário.

```
>>> print quadrados_impares
{1: 1, 3: 9, 9: 81, 5: 25, 7: 49}
>>>
>>> for i in quadrados_impares:
...     print quadrados_impares[i],
...
1 9 81 25 49
>>>
```

- *Funções e Métodos para Dicionários*

Funções incorporadas, tais como, **all()**, **any()**, **len()**, **cmp()**, **sorted()** etc, são comumente usadas com o dicionário para realizar diferentes tarefas.

Função	Descrição
<code>all()</code>	Retorna <code>True</code> se todas as chaves do dicionário são verdadeiras (ou se o dicionário está vazio)
<code>any()</code>	Retorna <code>True</code> se qualquer chave do dicionário é verdadeira. Se o dicionário está vazio, retorna <code>False</code>
<code>len()</code>	Retorna o comprimento (a quantidade de itens) do dicionário
<code>cmp()</code>	Compara os itens de dois dicionários
<code>sorted()</code>	Retorna uma nova lista ordenada de chaves/valores do dicionário

Método	Descrição
<code>copy()</code>	Retorna uma cópia profunda do dicionário, um novo objeto é criado na memória contendo todos os pares do dicionário fonte da cópia
<code>fromkeys()</code>	Retorna um novo dicionário cujas chaves são os elementos de lista (prim. Parâmetro) e cujos valores são todos iguais a valor (seg. parâmetro)
<code>update()</code>	Atualiza um dicionário com os elementos de outro. Os itens do outro dic são adicionados um a um ao dicionário original

Exemplos:

```

>>> print quadrados_impares
{1: 1, 3: 9, 9: 81, 5: 25, 7: 49}
>>> print len(quadrados_impares)
5
>>> print sorted(quadrados_impares)           # somente as chaves
[1, 3, 5, 7, 9]
>>> print sorted(quadrados_impares.values()) # somente os valores
[1, 9, 25, 49, 81]

>>> antonimos = {'sobe':'desce', 'certo':'errado', 'verdadeiro':'falso'}
>>> antônimos
{'sobe':'desce', 'certo':'errado', 'verdadeiro':'falso'}
>>> opostos = antônimos
>>> opostos
{'sobe':'desce', 'certo':'errado', 'verdadeiro':'falso'}
>>> opostos['certo'] = 'incerto'
>>> antonimos
{'certo': 'incerto', 'sobe': 'desce', 'verdadeiro': 'falso'}
>>> copia = antonimos.copy()                  # cópia profunda
>>> copia
{'certo': 'incerto', 'sobe': 'desce', 'verdadeiro': 'falso'}
>>> copia['certo'] = 'duvidoso'
>>> antonimos
{'certo': 'incerto', 'sobe': 'desce', 'verdadeiro': 'falso'}
>>> copia
{'certo': 'duvidoso', 'sobe': 'desce', 'verdadeiro': 'falso'}
>>> opostos
{'certo': 'incerto', 'sobe': 'desce', 'verdadeiro': 'falso'}
>>> id(antonimos), id(opostos), id(copia)
(111542608, 111542608, 111542320)
>>>

```

```

>>> x = {"Carla": [1,2], "Maria": [3,4]}
>>> y = x.copy()                                     # cópia profunda
>>> id(x), id(y)
(111524720, 111528960)
>>> y['Carla'] = [0]
>>> print x, '\n', y
{'Carla': [1, 2], 'Maria': [3, 4]}
{'Carla': [0], 'Maria': [3, 4]}
>>>

>>> y["Mariana"] = [5,6]
>>> y
{'Carla': [0], 'Mariana': [5, 6], 'Maria': [3, 4]}
>>> x["Carla"] = x["Carla"] + [3]                  # ou x["Carla"] += [3]
>>> x
{'Carla': [1, 2, 3], 'Maria': [3, 4]}
>>> y
{'Carla': [0], 'Mariana': [5, 6], 'Maria': [3, 4]}
>>> id(x), id(y)
(111524720, 111528960)

>>> z = {"a":1, "b":2, "c":3}
>>> q = {"z":9, "b":7}
>>> z.update(q)
>>> z
{'a': 1, 'c': 3, 'b': 7, 'z': 9}
>>> q
{'z': 9, 'b': 7}
>>>

```

- *Matriz Esparsa e Dicionário*

Considere a seguinte matriz esparsa:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

Uma representação dessa matriz usando uma lista terá muitos zeros:

```

>>> matriz = [ [0, 0, 0, 1, 0],
              [0, 0, 0, 0, 0],
              [0, 2, 0, 0, 0],
              [0, 0, 0, 0, 0],
              [0, 0, 0, 3, 0] ]

```

Uma alternativa para economizar memória é usarmos um dicionário. Para as chaves, nós podemos usar tuplas que contêm os índices da linha e da coluna:

```
>>> esparsa = {(0,3): 1, (2, 1): 2, (4, 3): 3}
```

Nós precisamos apenas de três pares **chave:valor**, para armazenar os valores diferente de zero da matriz. Cada chave é uma tupla, e cada valor é um número inteiro.

Para acessarmos um elemento da matriz armazenada na lista utilizamos o operador de indexação [] :

```
>>> matriz[0][3]
1
```

Note que a sintaxe da representação de um dicionário não é a mesma usada na representação das listas. Em vez de usarmos dois índices inteiros, nós usamos apenas um índice, que nesse caso, é uma tupla de dois valores inteiros.

Mas temos um problema com essa alternativa... Se tentarmos buscar um elemento zero, obteremos um erro, pois não existe uma entrada no dicionário para a chave especificada, 0 (zero):

```
>>> matriz[1][3]
0
>>> esparsa[0,3]
1
>>> esparsa[1,3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: (1,3)
>>>
```

O método **get()** resolve esse problema:

```
>>> esparsa.get((0,3), 0)
1
```

O primeiro parâmetro do **get()** é a chave buscada, e o segundo parâmetro é o valor que **get()** retornará caso a chave não exista no dicionário. Exemplo:

```
>>> esparsa.get((1,3),0)
0
>>>
```

Exercícios:

1. Dada uma palavra, só com letras minúsculas, digitada pelo usuário, mostre a quantidade de letras na palavra.
2. Dada uma palavra, só com letras minúsculas, digitada pelo usuário, mostre a ocorrência de cada letra da palavra (histograma = contagem de frequência), usando lista.
3. Dada uma palavra, só com letras minúsculas, digitada pelo usuário, mostre a ocorrência de cada letra da palavra (histograma = contagem de frequência), usando dicionário.

9. Entrada de Dados

Python fornece funções nativas que pegam entradas realizadas via teclado. A mais simples é chamada **raw_input()**. Ao executar esta função, o programa para e espera o usuário digitar alguma coisa. Quando o usuário finaliza a entrada de dados ao pressionar a tecla [Enter], o programa prossegue e a função **raw_input()** retorna a digitação do usuário sempre como **string**:

```
>>> entrada = raw_input()
Curso de Introdução ao Python
>>> print entrada
Curso de Introdução ao Python
```

Antes de usar `raw_input()` é bom exibir uma mensagem para o usuário dizendo o que ele deve entrar. Esta mensagem é uma como se fosse uma pergunta (*prompt*):

```
>>> nome = raw_input("Qual é o seu nome? ")
Qual é o seu nome? Arthur, Rei dos Bretões!
>>> print nome
Arthur, Rei dos Bretões!
```

Se a entrada esperada for um valor numérico inteiro, então podemos usar a função `input()`.

A função `input()` do Python também lê dados informados pelo usuário via teclado. Se a informação a ser entrada pelo usuário for uma *string* então a informação deve ser digitada com os delimitadores de *string* (par de aspas ou apostrofes), mas se a informação for numérica então nenhum delimitador será necessário.

```
>>> nome = input('Digite o seu sobrenome: ')
Digite o seu sobrenome: 'Fleury'
>>> idade = input('Digite a sua idade: ')
Digite a sua idade: 19
>>> print nome, 'tem', idade, 'anos.'
Fleury tem 19 anos.
>>>
```

Atente-se para o fato de que *strings* devem ser delimitadas por aspas ("") ou apóstrofos ('') na entrada de dados com a função `input()`.

```
>>> pergunta = "Velocidade de percurso (km/h): "
>>> velocidade = input(pergunta)
Velocidade de percurso (km/h): 80
>>>
```

Se o usuário digitar uma *string* de números, ela será convertida num inteiro e atribuída à variável **velocidade**. Infelizmente, se o usuário digitar um caractere que não seja um dígito numérico, o programa trava:

```
>>> velocidade = input(pergunta)
Qual... é a velocidade de vôo de uma andorinha?
De qual você fala, uma andorinha Africana ou uma Europeia?
SyntaxError: invalid syntax
```

Para evitar esse tipo de erro, geralmente é bom usar `raw_input()` para pegar uma *string* e, então, usar funções de conversão para outros tipos de dados: `int()`, `float()`, `long()`.

10. Controle de Fluxo de Execução

Condisional

Para poder escrever programas úteis, quase sempre precisamos da habilidade de checar condições e mudar o comportamento do programa de acordo com elas. As instruções condicionais oferecem essa habilidade. A forma mais simples é a instrução **if** (se):

```
if x > 0:  
    print x, "é positivo"
```

A expressão booleana depois da instrução **if** é chamada de condição. Se ela é verdadeira (**True**), então a instrução indentada² é executada. Se não, nada acontece, e segue-se com a execução do comando seguinte ao **if**.

As instruções compostas em Python são constituídas de um cabeçalho e de um bloco de comandos separados por um caractere ':' como mostrado a seguir:

```
INSTRUÇÃO-COMPOSTA:  
    PRIMEIRO COMANDO  
    ...  
    ÚLTIMO COMANDO
```

A primeira instrução não endentada marca o final do bloco.

Não existe limite para a quantidade de instruções que podem aparecer aninhadas numa instrução **if**. Ocionalmente, é útil ter um corpo sem nenhuma instrução (usualmente, como um delimitador de espaço para código que você ainda escreverá!). Nesse caso, você pode usar o comando **pass**, que indica ao Python: “passe por aqui sem fazer nada” ou “siga em frente...”.

```
if a < b:  
    pass  
print a, b  
...
```

Estruturas Alternativas

Simples:

```
if condição:  
    comando(s) # executados se condição é True
```

Composta:

```
if condição:  
    comando(s) # executados se condição é True  
else:  
    comando(s) # executados se condição é False
```

Encadeada:

```
if condição1:
```

² Recuar o texto em relação à margem esquerda da folha; inserir espaços entre a margem e o começo da linha de um parágrafo.

```

    comando(s)           # executados se condição1 é True
elif condição2:
    comando(s)           # executados se condição2 é True
...
elif condiçãoN:
    comando(s)           # executados se condiçãoN é False
else:
    comando(s)           # executados se todas as condições são False

```

Exemplo:

```

if a < b:
    print a, "é menor que", b
elif a > b:
    print a, "é maior que", b
else:
    print a, "e", b, "são iguais"

```

A expressão **elif** é uma abreviação de “**else if**” (“senão se”). No Python não existe uma instrução do tipo **escolha/caso** (switch, na ling. C/C++), mas pode-se usar:

```

if escolha == 'A':
    funcao_A()
elif escolha == 'B':
    funcao_B()
elif escolha == 'C':
    funcao_C()
else:
    print "Escolha não disponível."

```

Estruturas de Repetição

Em geral, as instruções são executadas sequencialmente: a primeira instrução em uma função é executada primeiro, seguida pela segunda e assim por diante. Pode haver situações em que você precise executar um bloco de comandos várias vezes. As linguagens de programação fornecem várias estruturas de controle que permitem caminhos de execução mais complexos.

Uma delas apresenta um teste da condição de manutenção do laço de repetição no início da estrutura: **while** (enquanto).

Nesse caso o(s) comando(s) a serem repetidos são colocados de forma indentada após o caractere ‘:’, e podem ser executados nenhuma ou várias vezes, enquanto a condição for verdadeira (**True**).

```

Inicialização
while condição:
    comando(s)

```

Uma instrução **while** na linguagem de programação Python executa repetidamente um ou mais comandos indentados, enquanto a **condição** for verdadeira. A **condição** pode ser qualquer expressão e **True** é qualquer valor diferente de zero. O loop itera enquanto a condição for verdadeira.

No Python, todas as instruções recuadas pela mesma quantidade de caracteres espaços após uma construção de programação ':' são consideradas parte de um único bloco de código. Python usa recuo (endentação) como método de agrupar instruções.

Exemplo:

```
>>> curso = 'Python'
>>> indice = 0
>>> while indice < len(curso):
...     letra = curso[indice]
...     print letra,
...     indice = indice + 1
...
P y t h o n
```

Usar um índice para percorrer um conjunto de valores é uma tarefa comum em programação, de modo que o Python oferece uma sintaxe alternativa simplificada - o laço de repetição **for**:

```
>>> for letra in curso:
...     print letra, ',',
...
P y t h o n
>>>
```

11. Função Definida pelo Usuário

Função é um grupo de instruções relacionadas que executam uma tarefa específica. As funções ajudam a dividir os programas em partes menores e modulares. À medida que o programa se torna maior, as funções o tornam mais organizado e gerenciável, além de evitar a repetição de comandos e gerar códigos reutilizáveis.

```
def nome_funcao(parametros):
    """ string documental para descrever o que faz a função """
    comando(s) indentado(s)
```

A primeira *string* após o cabeçalho da função é chamada de **docstring**. Ela é usada para resumir o que faz a função. Embora opcional, a documentação é uma boa prática de programação. Geralmente, usamos aspas triplas para que a docstring possa se estender por várias linhas. Esta *string* estará disponível para nós como atributo **__doc__** da função.

A instrução **return** é usada para encerrar a função e voltar para o local de onde foi chamada.

Exemplo:

```
import math
def polar(x, y):
    """ converte a coordenada retangular (x,y) em polar (mod,fase) """
    mod = math.sqrt(x*x + y*y)
    fase = math.atan2(y,x)*180./math.pi           # ângulo em graus
    return mod,fase

z,teta = polar(4, 4)
```

```

print "4 + j4 = %.2f|%.1f (grau)" % (z, teta)

>>> runfile('E:/python/apostila2.py', wdir=r'E:/python')
4 + j4 = 5.66|45.0 (grau)
>>> print polar.__doc__
Converte a coordenada retangular (x,y) em polar (mod,fase)
>>>

```

Funções que definimos para executar determinada tarefa específica são referidas como **funções definidas pelo usuário**. A maneira como definimos e chamamos funções em Python já é discutida.

Funções que vêm prontas com o Python são chamadas de **funções internas** (ou incorporadas).

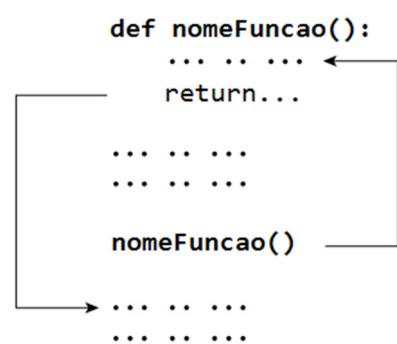
Funções escritas por outras pessoas na forma de biblioteca são denominadas funções de biblioteca.

Funcionamento das Funções

Depois de definida pelo usuário (comando **def**) a função poderá ser chamada no script sempre que for necessário.

Escopo de Variáveis e Passagem de Parâmetros

Escopo de uma variável se refere ao local de um programa no qual a variável é reconhecida. Parâmetros e variáveis definidos dentro de uma função não são visíveis de fora dessa função. Por isso, diz-se que variáveis desse tipo têm **escopo local**. Tempo de vida de uma variável é o período em que a variável existe até ser tirada da memória. O tempo de vida de variáveis dentro de uma função é o mesmo que a função é executada. Elas são destruídas quando a função é encerrada. Assim, uma função não se lembra do valor de uma variável relativa às suas chamadas anteriores.



```

def minha_funcao():
    x = 10
    print "Valor de 'x' dentro da funcao:", x

    x = 20
    minha_funcao()
    print "Valor de 'x' fora da funcao:", x

>>> runfile('E:/python/apostila2.py', wdir=r'E:/python')
Valor de 'x' dentro da funcao: 10
Valor de 'x' fora da funcao: 20
>>>

```

Vantagens das Funções Definidas pelo Usuário

- As funções definidas pelo usuário ajudam a decompor um programa grande em pequenos trechos, o que torna o programa mais fácil para se entender, manter e depurar.
- Se ocorrer código repetido em um programa. A função pode ser usada para incluir esses códigos e executar quando necessário, chamando a função definida pelo usuário.
- Programadores de grandes projetos dividem a carga de trabalho criando diferentes funções.

```

def comum(lista1, lista2):
    '''Retorna uma lista com os elementos em comum.'''
    A = set(lista1)
    B = set(lista2)
    return list(A & B)

x = [1,2,3,4,5]
y = [3,4,5,6,7,9]
z = range(0,50,5)
print "Comum entre x e y:", comum(x,y)
print "Comum entre x, y e z:", comum(comum(x,y),z)
>>> runfile('E:/python/apostila2.py', wdir=r'E:/python')
Comum entre x e y: [3, 4, 5]
Comum entre x, y e z: [5]

```

Exercícios:

1. Calcule e mostre o mínimo múltiplo comum (mmc) de dois números (menor inteiro positivo que é perfeitamente divisível pelos dois números dados).
2. Calcule e mostre o máximo divisor comum (mdc) de dois números (menor inteiro positivo que é perfeitamente divisível pelos dois números dados)

12. Funções Incorporadas

O interpretador Python possui várias **funções incorporadas** que estão sempre disponíveis – não dependem de importações (carga prévia). Elas estão listadas em ordem alfabética a seguir:

Funções Incorporadas				
<code>abs()</code>	<code>divmod()</code>	<code>input()</code>	<code>open()</code>	<code>staticmethod()</code>
<code>all()</code>	<code>enumerate()</code>	<code>int()</code>	<code>ord()</code>	<code>str()</code>
<code>any()</code>	<code>eval()</code>	<code>isinstance()</code>	<code>pow()</code>	<code>sum()</code>
<code>basestring()</code>	<code>execfile()</code>	<code>issubclass()</code>	<code>print()</code>	<code>super()</code>
<code>bin()</code>	<code>file()</code>	<code>iter()</code>	<code>property()</code>	<code>tuple()</code>
<code>bool()</code>	<code>filter()</code>	<code>len()</code>	<code>range()</code>	<code>type()</code>
<code>bytearray()</code>	<code>float()</code>	<code>list()</code>	<code>raw_input()</code>	<code>unichr()</code>
<code>callable()</code>	<code>format()</code>	<code>locals()</code>	<code>reduce()</code>	<code>unicode()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>long()</code>	<code>reload()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>map()</code>	<code>repr()</code>	<code>xrange()</code>
<code>cmp()</code>	<code>globals()</code>	<code>max()</code>	<code>reversed()</code>	<code>zip()</code>
<code>compile()</code>	<code>hasattr()</code>	<code>memoryview()</code>	<code>round()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hash()</code>	<code>min()</code>	<code>set()</code>	
<code>delattr()</code>	<code>help()</code>	<code>next()</code>	<code>setattr()</code>	
<code>dict()</code>	<code>hex()</code>	<code>object()</code>	<code>slice()</code>	
<code>dir()</code>	<code>id()</code>	<code>oct()</code>	<code>sorted()</code>	

Funções por categorias: matemática, repres. numérica, string, estr. de dados, funcional, outras.

Fonte: docs.python.org/2/library/functions.html#

enumerate(iterable, start=0)

Retorna um objeto enumerado. O parâmetro **iterable** pode ser uma sequência, um iterador ou algum outro objeto que suporte iteração³. O método **next()** do iterador retornado por **enumerate()** retorna uma tupla contendo uma contagem (do início, parâmetro **start**, cujo padrão é 0) e os valores obtidos da iteração pela sequência:

```
>>> estacoes = ['Primavera', 'Verao', 'Outono', 'Inverno']
>>> list(enumerate(estacoes))
[(0, 'Primavera'), (1, 'Verao'), (2, 'Outono'), (3, 'Inverno')]
>>> list(enumerate(estacoes,start=1))
[(1, 'Primavera'), (2, 'Verao'), (3, 'Outono'), (4, 'Inverno')]
```

Equivalente a:

```
def enumerate(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
        n += 1
```

Outro exemplo:

```
>>> frutas = ('manga', 'banana', 'laranja')
>>> for cont, fruta in enumerate(frutas):
...     print "Fruta %d: %s" % (cont,fruta)
...
Fruta 0: manga
Fruta 1: banana
Fruta 2: laranja
>>>
```

Outro exemplo:

```
>>> dias_sem = ["Segunda", "Terça", "Quarta", "Quinta", "Sexta"]
>>> for i, dia in enumerate(dias_sem,1):
...     print("{} é o {}º dia útil da semana".format(dia, i))
...
Segunda é o 1º dia útil da semana
Terça é o 2º dia útil da semana
Quarta é o 3º dia útil da semana
Quinta é o 4º dia útil da semana
Sexta é o 5º dia útil da semana
>>>
```

filter(funcao, iteravel)

A função **filter()** cria uma lista a partir dos elementos da estrutura de dados **iteravel**, para os quais a aplicação da **funcao** retorna **True**. Um iterável pode ser uma sequência, um contêiner que suporta iteração ou um iterador. Se **iteravel** for uma *string* ou uma *tupla*, o resultado também será

³ **Iteração**: substantivo feminino. 1. ato de iterar; repetição. 2. ÁLGEBRA: processo de resolução de uma equação mediante operações em que sucessivamente o objeto de cada uma é o resultado da que a precede. 3. COMPUT.: é o processo de repetição de uma ou mais ações; cada iteração se refere a apenas uma instância da ação.

desse mesmo tipo; caso contrário, será uma lista. Se **funcão** não for especificada ou for **None**, a função **identity()** é assumida, ou seja, todos os elementos iteráveis que forem **False** não serão considerados.

Observe que a função **filter()** é equivalente à seguinte *list comprehension*:

- se **funcão** for especificada: `[item for item in iterável if função(item)]`
- se **funcão** não for especificada: `[item for item in iterável if item]`

```
>>> lista = [8, 9, -1, -3, 3, -5, -4, 5, -4, 2, 5, 91, -11, 5, 10, 93, -75]
>>> positivos = []
>>> for item in lista:
...     if item > 0:
...         positivos.append(item)
...
>>> print positivos
[8, 9, 3, 5, 2, 5, 91, 5, 10, 93]
>>> negativos = [x for x in lista if item < 0]
>>> print negativos
[-1, -3, -5, -4, -11, -75]
```

Agora vamos mostrar o ‘jeito Python’ de fazer a mesma coisa, mas de uma forma pouco mais compacta, usando uma função definida pelo usuário e denominada **posit()**:

```
>>> lista = [8, 9, -1, -3, 3, -5, -4, 5, -4, 2, 5, 91, -11, 5, 10, 93, -75]
>>> def posit(x):      # função retorna True para x positivo
...     return x > 0
...
>>> positivos = filter(posit, lista)
>>> positivos
[8, 9, 3, 5, 2, 5, 91, 5, 10, 93]
>>>
```

Agora sem usar uma função específica em **filter**:

```
>>> lista = [1, 'a', 0, False, True, '0']
>>> result = filter(None, lista)
>>> print 'Elementos mantidos:', result
Elementos filtrados: [1, 'a', True, '0']
>>>
```

map(function, iterable, ...)

Aplica **function** a cada item de **iterable** e retorne uma lista com os resultados.

O mapeamento consiste em aplicar uma função a todos os itens de uma sequência, gerando outra lista contendo os resultados e com o mesmo tamanho da lista inicial.

Se argumentos iteráveis adicionais forem passados, a função deve tomar tantos quantos argumentos que serão aplicados aos itens de todos os iteráveis em paralelo. Se um iterável for menor que outro, então ele será estendido com itens **None**. Se a função não for especificada (**None**), a função identidade é assumida; se houver vários argumentos, **map()** retorna uma lista que consiste em tuplas contendo os itens correspondentes de todos os iteráveis (um tipo de

operação de transposição). Os argumentos iteráveis podem ser sequência ou qualquer objeto iterável; o resultado é sempre uma lista.

```
>>> import math
>>> quad = [1, 4, 9, 16, 25]
>>> result = map(math.sqrt, lista1)
>>> print result
[1.0, 2.0, 3.0, 4.0, 5.0]
>>>
```

Ao chamar a função `map(math.sqrt, lista1)`, estamos solicitando ao interpretador para que execute a função `math.sqrt` (square root, do inglês: raiz quadrada) usando como entrada cada um dos elementos da lista `quad`, e inserindo o resultado na lista retornada como resultado da função `map()`, a lista `result`.

Podemos facilmente substituir uma chamada a `map()` com *list comprehensions*. O código anterior poderia ser substituído por:

```
>>> result = [math.sqrt(x) for x in quad]
>>> print result
[1.0, 2.0, 3.0, 4.0, 5.0]
>>>
```

Expressões lambda

Lambda é uma função anônima composta apenas de expressões. As funções *lambda* são expressas apenas numa linha, e podem ter o resultado atribuído a uma variável. Funções *lambda* são muito usadas em programação funcional.

No exemplo da função `filter()`, tivemos que definir uma nova função (`posit`) para ser usada só dentro da função `filter()`, sendo chamada uma vez para cada elemento filtrado. Ao invés de definir uma nova função de usuário com a instrução `def`, podemos definir uma função válida somente enquanto durar a execução do `filter()`. Não é necessário nem nomear tal função, sendo, portanto chamada de função anônima ou função *lambda*.

Considere o exemplo seguinte:

```
>>> valores = [10, 4, -1, 3, 5, -9, -11]
>>> print filter(lambda x: x > 0, valores)
[10, 4, 3, 5]
>>>
```

Definimos uma função anônima que recebe um parâmetro de entrada (a variável `x`) e retorna o resultado da operação relacional `x > 0`: `True` ou `False`.

Podemos também usar uma função *lambda* no exemplo mostrado para a função `reduce()`:

```
>>> cinco = [1, 2, 3, 4, 5]
>>> soma = reduce(lambda x, y: x + y, cinco)
```

```
>>> print soma  
15  
>>>
```

No código acima, definimos uma função anônima que recebe dois parâmetros de entrada e retorna a soma deles.

```
reduce(function, iterable[, initializer])
```

Aplica **function** de dois parâmetros, aos dois primeiros elementos de **iterable**, depois aplica novamente **function** usando como parâmetros de entrada o resultado do primeiro par e o terceiro elemento de **iterable**, seguindo assim até o final da sequência. O resultado final da redução é apenas um elemento.

Exemplo:

```
reduce(lambda x, y: x + y, [1,2,3,4,5]) faz o seguinte cálculo (((1+2)+3)+4)+5).
```

O argumento da esquerda, **x**, é o valor acumulado e o argumento da direita, **y**, é o valor de atualização do **iterable**. Se **initializer** opcional estiver presente, ele será colocado antes dos itens do **iterable** no cálculo e servirá como padrão quando o **iterable** estiver vazio. Se o **initializer** não for fornecido e **iterable** contiver apenas um item, o primeiro item será retornado.

Aproximadamente se equivale a:

```
def reduce(function, iterable, initializer=None):  
    it = iter(iterable)  
    if initializer is None:  
        try:  
            initializer = next(it) # primeiro valor do iterable  
        except StopIteration:  
            raise TypeError('reduce() de sequência vazia e sem valor inicial')  
    accum_value = initializer  
    for x in it:  
        accum_value = function(accum_value, x)  
    return accum_value
```

Para somar os valores de uma determinada lista:

```
>>> import operator # necessário para obter a operação de soma  
>>> cinco = [1, 2, 3, 4, 5]  
>>> dez = range(11)  
>>> print reduce(operator.add, cinco)  
>>> print reduce(operator.add, dez)  
15  
55  
>>>
```

É claro que, para realizar a soma de todos os elementos de uma sequência, é muito mais simples utilizarmos a função **sum()**:

```
>>> print sum(cinco), '\n', sum(dez)
15
55
>>>
```

A função **reduce()** pode ser usada para calcular o fatorial de um valor *n*:

```
# Calcula o fatorial de n
>>> def fat(n):
...     return reduce(lambda x,y: x*y, range(1,n))
...
>>> print fat(7)
720
>>>
```

13. Arquivos de E/S

Saída de Dados em Dispositivo Padrão

A maneira mais simples de produzir uma saída na tela é usando a instrução **print**, na qual pode-se passar zero ou mais expressões separadas por vírgulas. Essa função converte as expressões numa *string* e grava o resultado na saída padrão, da seguinte forma:

```
>>> print "Python é uma grande linguagem de programação, ", "não é mesmo?"
```

Isso produz o seguinte resultado em seu dispositivo padrão de saída de dados (tela):

Python é uma grande linguagem de programação, não é mesmo?

Entrada de Dados em Dispositivo Padrão

O Python também fornece duas funções internas para ler uma linha de texto da entrada de dados padrão (teclado). Essas funções são: **raw_input()** e **input()** (vistas anteriormente).

- **raw_input([msg])**: lê uma linha da entrada padrão e a retorna como uma *string* (removendo o último caractere de linha nova, *new-line*).
- **input([msg])**: é equivalente a **raw_input()**, exceto pelo fato de assumir que a entrada é uma expressão válida do Python e retorna o resultado avaliado para você, ou seja, você pode usar ctes ou variáveis com informações numéricas ou *strings*.

E/S de Dados em Arquivos

Até agora fizemos leituras e escritas em entrada e saída padrões. Agora, vamos usar arquivos de dados reais. O Python fornece funções básicas e métodos necessários para manipular arquivos. Podemos fazer a maior parte da manipulação de arquivos usando um objeto de arquivo (**file**).

Abertura do Arquivo: Antes de ler ou escrever um arquivo, é preciso abri-lo usando a função `open()` do Python. Essa função cria um objeto de arquivo que será utilizado na chamada de outros métodos de suporte associados a ele.

```
objeto_arq = open(nome_arq [, modo_acesso][, buffer])
```

<code>nome_arq</code>	string com o nome do arquivo a ser acessado.
<code>modo_acesso</code>	determina o modo no qual o arquivo será aberto: leitura/escrita/anexação etc. Este parâmetro é opcional e o modo de acesso padrão é leitura (<code>r</code>).
<code>buffer</code>	Se o valor do buffer for 0, não existirá nenhum buffer. Se o valor do buffer for 1, o buffer de linha é criado ao acessar um arquivo. Se o valor do buffer for um inteiro maior que 1, a ação do buffer será executada com o tamanho do buffer indicado. Se for negativo, o tamanho do buffer é o padrão do sistema (default).

Tabela de Modos de Acesso a arquivos:

<code>r</code>	Abre um arquivo apenas para leitura. O ponteiro do arquivo é colocado no início do arquivo. Este é o modo padrão.
<code>rb</code>	Abre um arquivo para leitura somente em formato binário. O ponteiro do arquivo é colocado no início do arquivo. Este é o modo padrão.
<code>r+</code>	Abre um arquivo para leitura e gravação. O ponteiro do arquivo colocado no início do arquivo.
<code>rb+</code>	Abre um arquivo para leitura e gravação em formato binário. O ponteiro do arquivo colocado no início do arquivo.
<code>w</code>	Abre um arquivo apenas para gravação. Sobrescreve o arquivo se o arquivo existir. Se o arquivo não existir, cria um novo arquivo para gravação.
<code>wb</code>	Abre um arquivo para gravação somente em formato binário. Sobrescreve o arquivo se o arquivo existir. Se o arquivo não existir, cria um novo arquivo para gravação.
<code>w+</code>	Abre um arquivo para escrita e leitura. Sobrescreve o arquivo existente se o arquivo existir. Se o arquivo não existir, cria um novo arquivo para leitura e gravação.
<code>wb+</code>	Abre um arquivo para escrita e leitura em formato binário. Sobrescreve o arquivo existente se o arquivo existir. Se o arquivo não existir, cria um novo arquivo para leitura e gravação.
<code>a</code>	Abre um arquivo para anexar. O ponteiro do arquivo está no final do arquivo, se o arquivo existir. Ou seja, o arquivo está no modo de acréscimo. Se o arquivo não existir, ele criará um novo arquivo para gravação.
<code>ab</code>	Abre um arquivo para anexar em formato binário. O ponteiro do arquivo está no final do arquivo, se o arquivo existir. Ou seja, o arquivo está no modo de acréscimo. Se o arquivo não existir, ele criará um novo arquivo para gravação.
<code>a+</code>	Abre um arquivo para anexar e ler. O ponteiro do arquivo está no final do arquivo, se o arquivo existir. O arquivo é aberto no modo de acréscimo. Se o arquivo não existir, ele cria um novo arquivo para leitura e gravação.
<code>ab+</code>	Abre um arquivo para anexar e ler em formato binário. O ponteiro do arquivo está no final do arquivo, se o arquivo existir. O arquivo é aberto no modo de acréscimo. Se o arquivo não existir, ele cria um novo arquivo para leitura e gravação.

Atributos do objeto `file`: Depois que um arquivo é aberto tem-se um objeto `file`, pelo qual se pode obter várias informações relacionadas ao arquivo aberto.

<code>obj_file.closed</code>	Retorna True se o arquivo for fechado, caso contrário, False .
<code>obj_file.mode</code>	Retorna o modo de acesso com o qual o arquivo foi aberto.
<code>obj_file.name</code>	Retorna o nome do arquivo.
<code>obj_file.softspace</code>	Retorna False se o espaço for explicitamente requerido com <code>print</code> , True caso contrário.

Métodos do objeto `file`:

O método `close()` libera qualquer informação não escrita e fecha o objeto `file`, após o qual nenhuma escrita mais pode ser feita. O Python fecha automaticamente um arquivo quando o objeto de referência de um arquivo é reatribuído a outro arquivo. É uma boa prática usar o método `close()` para fechar um arquivo.

`objeto_arq.close()`

Exemplo:

```
# abertura/fechamento de um arquivo de dados
fo = open("qq.txt", "wb")
print "Nome do arquivo : ", fo.name
print "Fechado ou não : ", fo.closed
print "Modo de abertura : ", fo.mode
print "Flag de Softspace: ", fo.softspace
fo.close()
```

Resultado:

```
>>> fo = open("qq.txt", "wb")
>>> print "Nome do arquivo : ", fo.name
Nome do arquivo : qq.txt
>>> print "Fechado ou não : ", fo.closed
Fechado ou não : False
>>> print "Modo de abertura : ", fo.mode
Modo de abertura : wb
>>> print "Flag de Softspace: ", fo.softspace
Flag de Softspace: 0
>>> fo.close()
>>>
```

O método `write()` grava qualquer *string* em um arquivo aberto. É importante observar que as *strings* Python podem ter dados binários e não apenas texto. O método `write()` não adiciona um caractere de nova linha ('\n') ao final da *string*.

`objeto_arq.write(string)`

Exemplo:

```
# abre um arquivo
fo = open("qq.txt", "wb")
fo.write("Python é uma grande linguagem.\nNão é mesmo?\n")
# fecha o arquivo aberto
fo.close()
```

Vamos visualizar na tela o conteúdo do arquivo recém-criado usando um comando do S.O.:

Para S.O. Windows:	>>> !type qq.txt
Para S.O. Linux:	>>> !cat qq.txt
Resultado:	Python é uma grande linguagem. Não é mesmo?

O método **read()** lê uma *string* de um arquivo aberto. É importante observar que as *strings* Python podem ter dados binários, além dos dados de texto.

Exemplo:

```
fo = open("qq.txt", "r+")
msg = fo.read(10)
print "String lida: ", msg
fo.close() # fecha arquivo aberto

String lida: Python é
```

O método **tell()** informa a posição atual dentro do arquivo. Em outras palavras, a próxima leitura ou gravação ocorrerá nesse número de bytes a partir do início do arquivo.

O método **seek(offset [, from])** altera a posição atual do arquivo. O argumento **offset** indica o número de bytes a serem movidos. O argumento **from** especifica a posição de referência de onde os bytes devem ser movidos: 0, significa usar o início do arquivo como posição de referência; e 1 significa usar a posição atual como a posição de referência e; 2, significa que a referência será o final do arquivo.

Exemplo:

```
fo = open("qq.txt", "r+")
msg = fo.read(10) # lê apenas 10 bytes
print "String lida: ", msg
posicao = fo.tell() # verifica a posição corrente
print "Posição corrente no arquivo: ", posicao
# Reposiciona o ponteiro no início do arquivo novamente
posicao = fo.seek(0, 0)
msg = fo.read(50)
print "Denovo, a String lida: ", msg
# Close open file
fo.close() # fecha arquivo aberto
```

Resultado:

```
>>> fo = open("qq.txt", "r+")
>>> msg = fo.read(10) # lê apenas 10 bytes
>>> print "String lida: ", msg
String lida: Python é
>>> posicao = fo.tell() # verifica a posição corrente
>>> print "Posição corrente no arquivo: ", posicao
Posição corrente no arquivo: 8
>>> posicao = fo.seek(0, 0)
>>> msg = fo.read(50)
>>> print "Denovo, a String lida: ", msg
Denovo, a String lida: Python é uma grande linguagem.
```

```
Não é mesmo?  
>>> fo.close()  
>>>
```

```
# fecha arquivo aberto
```

Exercícios:

1. Fazer um programa para criar uma agenda de compromissos: evento, dia, horário, local, obs devem ser cadastrados para cada compromisso. Usuário deve conseguir inserir, alterar, excluir e visualizar compromisso(s) do dia, da semana e do mês.
2. Fazer um programa para cadastrar ofertas de produtos de supermercados, armazenando: supermercado, produto (categoria, descrição, quantidade, unidade de medida, preço), data. Usuário deve conseguir inserir, alterar, excluir e visualizar oferta(s) do dia, classificadas por categoria.

14. Exceções

O Python fornece dois recursos muito importantes para lidar com situações em que erro(s) inesperado(s) ocorra(m) em seus programas Python e para incluir recursos de depuração neles.

- Manipulação de Exceção (*exception handling*)
- Afirmações (*assertions*)

O que é Exceção em Python?

É um evento que ocorre durante a execução de um programa e que interrompe o fluxo normal de execução das instruções do programa. Em geral, quando o interpretador Python encontra uma situação com a qual ele não pode lidar, ele gera uma exceção. Uma exceção é um objeto Python que representa um erro. Quando um script Python provoca uma exceção, ele deve manipular a exceção imediatamente, pois caso contrário, ele será encerrado.

Tratando uma Exceção

Se seu script tiver algum código suspeito que possa provocar uma **exceção**, então você pode defender seu programa colocando o código suspeito em um bloco **try**: Depois desse bloco inclua uma instrução **except**: seguida por um bloco de código que manipula o problema da maneira mais elegante possível.

```
try:  
    operação suspeita do seu script  
    ....  
except Exception1:  
    se existir Exception1, então execute este bloco.  
except Exception2:  
    se existir Exception2, então execute este bloco.  
    ....  
else:  
    se não for nenhuma das exceções anteriores, então execute este bloco.  
    ....
```

Exemplo: Abre um arquivo, grava conteúdo no arquivo e sai corretamente porque não há problema algum.

```
try:  
    arq = open("arqteste.txt", "w")  
    arq.write("Arquivo teste para tratamento de exceção!!")  
except IOError:  
    print "Erro: não pode encontrar arquivo, ou os dados de leitura."  
else:  
    print "Conteúdo escrito no arquivo com sucesso."  
arq.close()
```

Conteúdo escrito no arquivo com sucesso.

>>>

Exemplo: Tentativa de abrir um arquivo inexistente para leitura, então uma exceção será gerada.

```
try:  
    arq = open("arqfantasma.txt", "r")  
    arq.read(10)  
except IOError:  
    print "Erro: não pode encontrar arquivo, ou os dados de leitura."  
else:  
    print "Conteúdo lido do arquivo com sucesso."  
arq.close()
```

Erro: não pode encontrar arquivo, ou os dados de leitura.

>>>

Pode-se também usar a instrução **except** sem exceções definidas:

```
try:  
    operação suspeita do seu script  
    ....  
except:  
    se existir alguma exceção, então execute este bloco.  
    ....  
else:  
    se não existir exceção então execute este bloco.  
    ....
```

Esse tipo de instrução **try-except** captura todas as exceções que ocorrem. A utilização desse tipo de instrução **try-except** não é considerada uma boa prática de programação, justamente por ela capturar todas as exceções, e não ajudar o programador a identificar a causa do problema que está ocorrendo...

Outra possibilidade para a instrução **except** é usar várias exceções numa única cláusula:

```
try:  
    operação suspeita do seu script  
    ....
```

```

except (Excecao1[, Excecao2[, ... ExcecaoN]]):
    se for uma das exceções da lista, então execute este bloco.
    ....
else:
    se não existir exceção então execute este bloco.
    ....

```

Você pode usar um bloco **finally**: junto com um bloco **try**: O bloco **finally** é o lugar para se colocar qualquer código que deve ser executado de qualquer forma, quer o bloco de teste (**try**) tenha gerado uma exceção ou não.

```

try:
    arq = open("arqteste.txt", "w")
    arq.write("Teste de gravação em arquivo p/ tratamento de exceção!!")
finally:
    print "Erro: arquivo ou dados não localizados!!"

```

Quando uma exceção é lançada por uma instrução dentro do bloco **try**, a execução passa imediatamente para o bloco **finally** e depois a exceção é levantada novamente e é tratada nas instruções **except** se presentes na próxima camada superior da instrução **try-except**.

```

try:
    arq = open("arqteste.txt", "w")
    try:
        arq.write("Teste de grav. em arquivo p/ tratamento de exceção!!")
    finally:
        print "Indo fechar o arquivo..."
        arq.close()
    except IOError:
        print "Erro: não pode encontrar arquivo, ou os dados de leitura."

```

Você pode gerar exceções arbitrariamente, ou seja, quando você precisar, e de várias maneiras usando a instrução **raise**.

```
raise [Excecao [, args [, traceback]]]
```

Exceção é o tipo de exceção (por exemplo, **NameError**) e o args é um valor para o argumento da exceção. O argumento é opcional; se não for fornecido, o argumento de exceção é **None**. O argumento final, traceback, também é opcional (e raramente usado na prática) e, se presente, é o objeto **traceback** usado na exceção.

Exemplo: Uma exceção pode ser uma *string*, uma classe ou um objeto. A maioria das exceções que o núcleo do Python gera são classes, com um argumento que é uma instância da classe. Definir novas exceções é muito fácil e pode ser feito da seguinte maneira:

```

def nomeFuncao(nivel):
    if nivel < 1:
        raise "Nível não permitido!", nivel
        # O código abaixo disto não será executado
        # se gerarmos a exceção

```

Obs: Para capturar uma exceção, uma cláusula **except** deve referir-se à mesma exceção lançada como objeto de classe ou *string*. Por exemplo, para capturar a exceção definida acima, devemos escrever a cláusula **except** da seguinte forma:

```

try:
    comandos suspeitos
    ....
except "Nível não permitido!":
    tratamento da exceção aqui...
else:
    resto do código aqui...

```

Exceções padronizadas no Python:

N.	Exceção	Motivo da Ocorrência
1	Exception	Classe base para todas as exceções
2	StopIteration	O método next() de um iterador não aponta para nenhum objeto.
3	SystemExit	Função sys.exit() .
4	StandardError	Classe base para todas as exceções internas, exceto StopIteration e SystemExit .
5	ArithmeticError	Classe base para todos os erros que ocorrem no cálculo numérico.
6	OverflowError	Um cálculo excede o limite máximo de um tipo numérico.
7	FloatingPointError	Um cálculo de ponto flutuante falha.
8	ZeroDivisionError	Uma divisão ou módulo por zero (para todos os tipos numéricos).
9	AssertionError	Em caso de falha da declaração do Assert .
10	AttributeError	Em caso de falha de referência a atributo.
11	EOFError	Não há entrada para função raw_input() ou input() e o final do arquivo é atingido.
12	ImportError	Uma declaração de importação falha.
13	KeyboardInterrupt	O usuário interrompe a execução do programa: [Ctrl]+[C].
14	LookupError	Classe base para todos os erros de pesquisa.
15	IndexError	Um índice não é encontrado em uma sequência.
16	KeyError	A chave especificada não é encontrada no dicionário.
17	NameError	Um identificador não é encontrado no espaço de nomes local/global.
18	UnboundLocalError	Ao tentar acessar uma variável local em uma função ou método, mas nenhum valor foi atribuído a ela.
19	EnvironmentError	Classe base p/ todas exceções que ocorrem fora do ambiente Python.
20	IOError	Uma operação de entrada/saída falha, como a instrução print ou a função open() ao tentar abrir um arquivo que não existe.
21	IOError	Por erros relacionados ao sistema operacional.
22	SyntaxError	Há um erro na sintaxe do Python.
23	IndentationError	O recuo não é especificado corretamente.
24	SystemError	O interpretador encontra um problema interno, mas quando esse erro é encontrado, o interpretador Python não sai.
25	SystemExit	O interpretador Python é encerrado usando a função sys.exit() . Se não for tratado no código, faz com que o intérprete saia.
26	TypeError	Uma operação/função tentada é inválida para o tipo de dado indicado
27	ValueError	A função interna de um tipo de dado possui o tipo válido de argumentos, mas os argumentos têm valores inválidos especificados.
28	RuntimeError	Um erro gerado não se enquadra em nenhuma categoria.
29	NotImplementedError	Um método abstrato de uma classe herdada não foi implementado.

Afirmativas

Uma afirmação é uma verificação de sanidade que você ativa durante o desenvolvimento do programa e desativa quando terminar de testar o programa. A maneira mais fácil de pensar em

uma afirmação é compará-la a uma declaração **raise-if** (gera-se) ou, para ser mais preciso, a uma declaração **raise-if-not** (gera-se-não). Uma expressão é testada e, se o resultado for **falso**, uma exceção é gerada (**raised**, levantada). As afirmações são executadas pela instrução **assert**, a mais nova palavra-chave para o Python, introduzida na versão 1.5. Os programadores costumam colocar afirmações no início de uma função para verificar se há uma entrada válida e depois de uma chamada de função para verificar se a saída é válida.

```
assert expressao[, argumentos]
```

Quando uma declaração de afirmação (**assert expressão**) é encontrada, o Python avalia sua expressão (espera-se que o resultado dessa avaliação seja **verdadeira** para que o fluxo de execução siga normalmente). Se a expressão for **falsa**, o Python gerará a exceção **AssertionError**.

As exceções **AssertionError** podem ser capturadas e tratadas como qualquer outra exceção usando a instrução **try-except**, mas se não forem tratadas, elas encerrão o programa e produzirão um **traceback**.

Exemplo: Dado uma função que converte temperatura de graus Kelvin para graus Fahrenheit. Como zero grau Kelvin é definido como a temperatura mais baixa, a função não deve aceitar uma temperatura negativa.

```
def Kelvin2Fahrenheit(Temperature):
    assert (Temperature >= 0), "Mais frio que o zero absoluto?!"
    return ((Temperature-273)*1.8)+32

print Kelvin2Fahrenheit(273)
print int(Kelvin2Fahrenheit(505.78))
print Kelvin2Fahrenheit(-5)

>>> print Kelvin2Fahrenheit(273)
32.0
>>> print int(Kelvin2Fahrenheit(505.78))
451
>>> print Kelvin2Fahrenheit(-5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 2, in Kelvin2Fahrenheit
AssertionError: Mais frio que o zero absoluto?!
>>>
```

Exercícios:

1. Crie uma função que converta temperaturas em Kelvin para Celsius, tal como no exemplo anterior.
2. Crie uma função que calcule a velocidade média de um automóvel, dados a distância (m) e o tempo (s) gasto para percorrê-la. Use afirmação.

PACOTES - MÓDULOS

Um pacote no Python é uma estrutura hierárquica de diretório de arquivos que define um único ambiente de aplicativo Python que consiste em módulos e subpacotes e "subsubpacotes" e assim por diante.

Importações

Até agora, falamos sobre tipos e funções incorporados à linguagem. Mas uma das melhores coisas sobre o Python é o grande número de **bibliotecas** personalizadas de alta qualidade que estão disponíveis para ele. Algumas dessas bibliotecas fazem parte da "biblioteca padrão", o que significa que você pode encontrá-las em qualquer lugar que execute o Python. As outras bibliotecas (não fazem parte da "biblioteca padrão") também podem ser facilmente acrescentadas.

O acesso a esses códigos (módulos/pacotes/bibliotecas) se faz em Python através das importações (comando **import**). Começaremos nosso exemplo importando o pacote/módulo **math** da biblioteca padrão. Um módulo é apenas uma coleção de variáveis (*namespace*) definidas por outra pessoa. Podemos ver todos os nomes definidos no módulo **math** usando a função **dir()**.

```
>>> import math
>>> print 'Tipo da biblioteca "math": {}'.format(type(math))
Tipo da biblioteca "math": <type 'module'>
>>> dir(math)
>>> dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh',
'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e',
' erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
'fsum', 'gamma', 'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
'log10', 'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt',
'tan', 'tanh', 'trunc']
>>>
```

Podemos acessar essas variáveis usando a sintaxe do ponto “.” para separar os nomes do módulo e de seus elementos. Algumas das referem-se a simples valores (constantes), tal como **math.pi**, e outras se referem a funções, tal como **math.log()**:

```
>>> print "O número pi com 4 dígitos significativos = {:.4}".format(math.pi)
O número pi com 4 dígitos significativos = 3.142
>>> math.log(32,2)    # logaritmo de 32 na base 2
5.0
>>>
```

Se você não souber o que faz uma determinada função, então peça ajuda ao interpretador, usando a função **help()**:

```
>>> help(math.log)
Help on built-in function log in module math:

log(...)
    log(x[, base])

    Return the logarithm of x to the given base.
```

```
If the base not specified, returns the natural logarithm (base e) of x.  

>>>
```

Outras Sintaxes para Importação

Se soubermos os nomes das funções em **math** que serão usadas com frequência, podemos importá-las com um *alias* (apelido) mais curto para economizar digitação (embora nesse caso o módulo "**math**" já seja uma palavra bem curta).

```
>>> import math as mt  

>>> mt.pi  

3.141592653589793  

>>>
```

Não seria ótimo se pudéssemos nos referir a todas as variáveis do módulo **math** usando apenas os nomes delas? Ou seja, se pudéssemos nos referir ao número pi apenas citando **pi** em vez de **math.pi** ou **mt.pi**? Boas notícias: podemos fazer isso. Veja um exemplo disso a seguir:

```
>>> from math import *  

>>> print pi, log(32, 2)  

3.141592653589793 5.0  

>>>
```

O comando **import *** torna todas as variáveis do módulo acessíveis diretamente (sem qualquer prefixo com ponto). Más notícias: alguns puristas podem reclamar de você por fazer isso.

Esse tipo de "importação com estrela (asterisco)" pode ocasionalmente levar a situações estranhas e difíceis de depurar (corrigir).

```
>>> from math import *  

>>> from numpy import *  

>>> print pi, log(32, 2)  

Traceback (most recent call last):  

  File "<stdin>", line 1, in <module>  

TypeError: return arrays must be of ArrayType  

>>>
```

O problema neste caso específico é que os módulos **math** e **numpy** têm funções de logaritmo com o mesmo nome: **log**, porém com semânticas diferentes. Porque nós importamos **numpy** por último, sua função **log** sobrescreveu a função **log** importada do módulo **math**.

Uma boa prática de programação, adotada pelos profissionais da área, é importar apenas as variáveis (constantes e funções) específicas de cada módulo que serão necessárias no seu código (script):

```
>>> from math import log, pi  

>>> from numpy import asarray  

>>> print pi, log(32, 2)  

3.14159265359 5.0  

>>>
```

Submódulos

Vimos que os módulos contêm variáveis que podem se referir a funções ou constantes. Lembre-se de que os módulos também podem ter variáveis que se referem a outros módulos (submódulos).

```
>>> print "numpy.random é um", type(numpy.random)
numpy.random é um <type 'module'>
>>> print "Ele contém nomes tais como...", dir(numpy.random)[-15:]
Ele contém nomes tais como... ['set_state', 'shuffle', 'standard_cauchy',
'standard_exponential', 'standard_gamma', 'standard_normal', 'standard_t',
'test', 'triangular', 'uniform', 'vonmises', 'wald', 'warnings', 'weibull',
'zipf']
>>>
```

Então, se nós importamos **numpy** como acima, então para chamar uma função no "submódulo" **random** será preciso o uso de dois pontos:

```
>>> # Lançamento do dado 10 vezes
... lancamentos = numpy.random.randint(1, 6, 10)
>>> lancamentos
array([2, 4, 3, 1, 5, 2, 1, 5, 2, 4])
>>>
```

Ferramentas para Entender Objetos Desconhecidos

No item anterior vimos que ao chamar uma função **numpy** sempre teremos um arranjo (array) como retorno. Nós nunca vimos algo assim antes (não neste curso de qualquer forma). Mas não entre em pânico: temos três funções incorporadas para nos ajudar nessa questão.

1. **type()** (o que é isso?)

```
>>> type(lancamentos)
<type 'numpy.ndarray'>
>>>
```

2. **dir()** (o que posso fazer com isso?)

```
>>> print dir(lancamentos)[-20:]
['round', 'searchsorted', 'setfield', 'setflags', 'shape', 'size', 'sort',
'squeeze', 'std', 'strides', 'sum', 'swapaxes', 'take', 'tofile', 'tolist',
'tostring', 'trace', 'transpose', 'var', 'view']
>>>
>>> # O que quero fazer com os resultados dos lançamentos do dado?
... # Talvez eu queira o valor médio...
... lancamentos.mean()
2.899999999999999
>>>
```

3. **help()** (conte-me mais sobre...)

```
>>> # o atributo "ravel" soa interessante. Eu sempre fui fã de música clássica...
>>> help(lançamentos.ravel)
Help on built-in function ravel:

ravel(...)
    a.ravel([order]) --> Return a flattened array.
    Refer to 'numpy.ravel' for full documentation.

See Also
-----
numpy.ravel : equivalent function
ndarray.flat : a flat iterator on the array.
>>>
>>> # Ok, o que há para saber sobre numpy.ndarray
>>> help(lançamentos)
Help on ndarray object:

class ndarray(builtins.object)
|   ndarray(shape, dtype=float, buffer=None, offset=0,
|           strides=None, order=None)
|
|   An array object represents a multidimensional, homogeneous array
|   of fixed-size items. An associated data-type object describes the
...
>>>
```

15. Numpy



O **NumPy** é um pacote fundamental para computação científica com o Python. Ele contém, entre outras coisas:

- Um poderoso objeto de **arranjo N-dimensional**
- Funções sofisticadas (*broadcasting*)
- Ferramentas para **integrar códigos C/C++ e Fortran**
- Pacotes de funções para **Álgebra Linear, transformada de Fourier** e geração de **números aleatórios**.

Além do óbvio uso científico, o **NumPy** também pode ser usado como um contêiner multidimensional eficiente para dados genéricos. Tipos de dados arbitrários podem ser definidos. Isso permite ao **NumPy** fácil e rápida integração a uma ampla variedade de bancos de dados. A seguir veremos como usar o pacote **Numpy** para manipulação básica de arranjos (array).

Primeiro, precisamos importar **Numpy**, para só então poder usar suas classes, funções e demais recursos.

```
>>> import numpy as np  
>>>
```

Criando um arranjo (vetor) **numpy**:

```
>>> np.array([4,5,6]) # converte uma lista num vetor (array)  
array([4, 5, 6])
```

Criando outro arranjo (matriz ou vetor) **numpy**:

```
>>> np.array([[4,5,6],[7,8,9],[10,11,12]])  
array([[ 4,  5,  6],  
       [ 7,  8,  9],  
       [10, 11, 12]])
```

Verificando a forma do arranjo (quantidade de linhas e de colunas da matriz):

```
>>> mat = np.array([[4,5,6],[7,8,9],[10,11,12]])  
>>> mat.shape  
(3, 3)
```

Criando uma matriz com valores uniformemente espaçados de 2 de 1 a 59:

```
>>> faixa = np.arange(1,60,2)  
>>> faixa  
array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31,  
       33,  
       35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59])  
>>> faixa.reshape(3,10)  
array([[ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19],
```

```
[21, 23, 25, 27, 29, 31, 33, 35, 37, 39],  
[41, 43, 45, 47, 49, 51, 53, 55, 57, 59]])  
>>>
```

Gerando um vetor com 15 valores uniformemente espaçados de 1 a 8:

```
>>> x = np.linspace(1,8,15)  
>>> x  
array([ 1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,  5.5,  6. ,  
       6.5,  7. ,  7.5,  8. ])  
>>>
```

Agora, vamos alterar a forma do vetor na memória (a função "resize" altera a forma da matriz na memória, ao contrário de "reshape" que só muda a forma na apresentação da informação).

```
>>> x.resize(5,3)  
>>> x  
array([[ 1. ,  1.5,  2. ],  
       [ 2.5,  3. ,  3.5],  
       [ 4. ,  4.5,  5. ],  
       [ 5.5,  6. ,  6.5],  
       [ 7. ,  7.5,  8. ]])  
>>>
```

Criando um arranjo com todos os elementos iguais a um e outro com zeros:

```
>>> y = np.ones((4,4))  
>>> y  
array([[ 1.,  1.,  1.,  1.],  
       [ 1.,  1.,  1.,  1.],  
       [ 1.,  1.,  1.,  1.],  
       [ 1.,  1.,  1.,  1.]])  
>>> z = np.zeros((3,2))  
>>> z  
array([[ 0.,  0.],  
       [ 0.,  0.],  
       [ 0.,  0.]])  
>>>
```

Criando uma matriz diagonal unitária:

```
>>> diag = np.eye(5)  
>>> diag  
array([[ 1.,  0.,  0.,  0.,  0.],  
       [ 0.,  1.,  0.,  0.,  0.],  
       [ 0.,  0.,  1.,  0.,  0.],  
       [ 0.,  0.,  0.,  1.,  0.],  
       [ 0.,  0.,  0.,  0.,  1.]])  
>>> print diag  
[[ 1.  0.  0.  0.  0.]
```

```
[ 0.  1.  0.  0.  0.]  
[ 0.  0.  1.  0.  0.]  
[ 0.  0.  0.  1.  0.]  
[ 0.  0.  0.  0.  1.]]  
>>>
```

Criando um **array** via repetição de lista e outro usando função **repeat()** (de repetição):

```
>>> r1 = np.array([1,2,3]*6)  
>>> r1  
array([1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3])  
>>> r2 = np.repeat([1,2,3],6)  
>>> r2  
array([1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3])  
>>>
```

Gerando dois **arrays** de dimensão 2x3, preenchidos com valores aleatórios entre 0 e 1:

```
>>> a = np.random.rand(2,3)  
>>> a  
array([[ 0.15472392,  0.11692319,  0.35272142],  
       [ 0.84370634,  0.43836276,  0.5290161 ]])  
>>> b = np.random.rand(2,3)  
>>> b  
array([[ 0.88079953,  0.85233273,  0.9430226 ],  
       [ 0.34204979,  0.30754441,  0.98637748]])  
>>>
```

Empilhando verticalmente as duas matrizes criadas anteriormente, **a** e **b**:

```
>>> vert = np.vstack([a,b])  
>>> vert  
array([[ 0.15472392,  0.11692319,  0.35272142],  
       [ 0.84370634,  0.43836276,  0.5290161 ],  
       [ 0.88079953,  0.85233273,  0.9430226 ],  
       [ 0.34204979,  0.30754441,  0.98637748]])  
>>>
```

Agora, horizontalmente:

```
>>> hor = np.hstack((a,b))  
>>> hor  
array([[ 0.15472392,  0.11692319,  0.35272142,  0.88079953,  0.85233273,  
         0.9430226 ],  
       [ 0.84370634,  0.43836276,  0.5290161 ,  0.34204979,  0.30754441,  
         0.98637748]])  
>>>
```

Vamos fazer algumas operações elemento a elemento em arranjos de mesmas dimensões:

```

>>> a
array([[1, 2],
       [3, 4]])
>>> b = -a
>>> b
array([[-1, -2],
       [-3, -4]])
>>> a + b
array([[0, 0],
       [0, 0]])
>>> a * b
array([[ -1, -4],
       [ -9, -16]])
>>> a**3
array([[ 1,  8],
       [27, 64]])
>>>

```

Multiplicação matricial:

```

>>> a.dot(b)
array([[ -7, -10],
       [-15, -22]])
>>>

```

Transposição matricial:

```

>>> a.T
array([[1, 3],
       [2, 4]])
>>>

```

Agora, vamos verificar o tipo de dados dos elementos na matriz:

```

>>> a.dtype
dtype('int32')
>>> b.dtype
dtype('int32')
>>>

```

Mudando o tipo de dados da matriz, de inteiro para ponto flutuante (*float*):

```

>>> c = a.astype('f')
>>> c
array([[ 1.,  2.],
       [ 3.,  4.]], dtype=float32)
>>> print c
[[ 1.  2.]
 [ 3.  4.]]

```

```
>>>
```

Agora, vamos ver algumas funções matemáticas aplicadas aos elementos de uma matriz, começando pela soma, depois valores máximo e mínimo, e por fim a média:

```
>>> a.sum()  
10  
>>> a.max()  
4  
>>> a.min()  
1  
>>> a.mean()  
2.5  
>>>
```

Tabuada de 5, indexação e fatiamento:

```
>>> tab5 = np.arange(0,11)*5 # ou: np.arange(0,51,5)  
>>> tab5  
array([ 0,  5, 10, 15, 20, 25, 30, 35, 40, 45, 50])  
>>> tab5[1:5]  
array([ 5, 10, 15, 20])  
>>> tab5[-1]  
50  
>>> tab5[::-1]  
array([50, 45, 40, 35, 30, 25, 20, 15, 10, 5, 0])
```

Criando um arranjo bidimensional a partir de um vetor:

```
>>> d = np.arange(25)  
>>> d  
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,  
       17, 18, 19, 20, 21, 22, 23, 24])  
>>> d.reshape(5,5)  
array([[ 0,  1,  2,  3,  4],  
       [ 5,  6,  7,  8,  9],  
       [10, 11, 12, 13, 14],  
       [15, 16, 17, 18, 19],  
       [20, 21, 22, 23, 24]])  
>>> d.shape  
(25,)  
>>>
```

Mudando a forma do arranjo na memória:

```
>>> d = np.arange(25)  
>>> d  
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
```

```

    17, 18, 19, 20, 21, 22, 23, 24])
>>> d.resize(5,5)
>>> d
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
>>> d.shape
(5,5)
>>>

```

Acessando a quarta e quinta colunas da quarta linha da matriz **d**. Observe que a numeração das linhas e colunas começam em 0 (zero):

```

>>> d[3,3:5]
array([18, 19])
>>>

```

Selecionando os valores maiores que 15 da matriz **d**, e atribuindo valor zero:

```

>>> d[d>15]
array([16, 17, 18, 19, 20, 21, 22, 23, 24])
>>> d[d>15] = 0
>>> d
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15,  0,  0,  0,  0],
       [ 0,  0,  0,  0,  0]])
>>>

```

Criando uma matriz 3x3 preenchida com números inteiros aleatórios entre 5 e 20:

```

>>> e = np.random.randint(5,20,(3,3))
>>> e
array([[14,  5, 18],
       [19, 18, 10],
       [18,  5, 19]])
>>>

```

Muito bem, até agora vimos como criar, como acessar e como manipular arranjos numéricos (vetores e matrizes) em **Numpy**. Na próxima seção, veremos os pacotes SCIPY, MATPLOTLIB e PANDAS, todos construídos a partir do pacote **Numpy**.

O pacote **SciPy** possui funções úteis à manipulação de sinais na análise de sistemas de Engenharia, em geral.

O pacote **Matplotlib** é um pacote que permite a apresentação de dados na forma gráfica. Possui uma extensa gama de tipos de gráficos, muito útil também na exibição de sinais contínuos e discretos.

O pacote **Pandas** facilita muito a manipulação e análise de dados em Python, além de oferecer estruturas de dados e operações para tabelas numéricas e séries temporais.

16. Scipy ([documentação: docs.scipy.org](http://docs.scipy.org))

SciPy é uma coleção de algoritmos matemáticos e funções de conveniência construídos sobre a extensão **Numpy** do Python. Ele acrescenta poder significativo à sessão interativa do Python, fornecendo ao usuário comandos e classes de alto nível para manipular e visualizar dados. Com o SciPy, uma sessão Python interativa torna-se um ambiente de processamento de dados e prototipagem do sistema que rivaliza com sistemas como MATLAB, IDL, Octave, R-Lab e SciLab.

As aplicações científicas que usam o **SciPy** se beneficiam do desenvolvimento de módulos adicionais em inúmeros nichos do cenário de software por desenvolvedores em todo o mundo. Tudo, desde programação paralela até sub-rotinas e classes de base de dados e da Web, foi disponibilizado para o programador Python. Todo esse poder está disponível além das bibliotecas matemáticas do **SciPy**.

Por questões de conveniência e brevidade, assumimos que os pacotes principais (**Numpy**, **Scipy** e **Matplotlib**) foram importados como:

```
>>> import numpy as np  
>>> import matplotlib as mpl  
>>> import matplotlib.pyplot as plt
```

Organização do Scipy

Subpacote	Descrição
cluster	Algoritmos de agrupamento
constants	Constantes físicas e matemáticas
fftpack	Rotinas de transformada rápida de Fourier
integrate	Integração e solucionadores de equação diferencial ordinária
interpolate	Interpolação e suavização Splines
io	Entrada e saída
linalg	Álgebra Linear
misc	Funções diversas: factorial, comb, lena, ...
ndimage	Processamento de imagem N-dimensional
odr	Regressão de distância ortogonal
optimize	Rotinas de optimização e localização de raízes
signal	Processamento de sinais
sparse	Rotinas de matrizes esparsas e associadas
spatial	Algoritmos de estruturas de dados espaciais
special	Funções especiais
stats	Funções e distribuições estatísticas

Os subpacotes do **Scipy** precisam ser importados separadamente, por exemplo:

```
>>> from scipy import linalg, optimize
```

Para usar funções de alguns dos módulos do **Scipy**, use:

```
>>> from scipy import algum_modulo
>>> algum_modulo.alguma_funcao()
```

O nível superior de **Scipy** também contém funções de **Numpy** e **numpy.lib.scimath**. No entanto, é melhor usá-las diretamente do módulo **Numpy**.

Truques de índice

Existem algumas instâncias de classe que fazem uso especial da funcionalidade do fatiamento para fornecer meios eficientes na construção de arranjos (vetores, matrizes). Vamos ver a operação de **np.mgrid**, **np.ogrid**, **np.r_** e **np.c_** para construir arranjos (arrays) rapidamente.

Por exemplo, em vez de escrever algo como o seguinte:

```
>>> a = np.concatenate(([3], [0]*5, np.arange(-1, 1.002, 2/9.0)))
>>> a
array([ 3.          ,  0.          ,  0.          ,  0.          ,  0.          ,
       0.          , -1.          , -0.77777778, -0.55555556, -0.33333333,
      -0.11111111,  0.11111111,  0.33333333,  0.55555556,  0.77777778,  1. ])
```

Com o comando **r_** use o seguinte para obter o mesmo efeito:

```
>>> a = np.r_[3,[0]*5,-1:1:10j]
```

Essa forma facilita a digitação e torna o código mais legível. Observe como os objetos são concatenados e a sintaxe de fatiamento é (ab)usada para construir intervalos. O outro termo que merece uma pequena explicação é o uso do número complexo **10j** como o tamanho do passo na sintaxe do fatiamento. Esse uso não padrão permite que o número seja interpretado como o **número de pontos** a serem produzidos no intervalo, **em vez de um tamanho de passo** (note que teríamos usado a notação inteira longa, **10L**, mas essa notação pode desaparecer em Python quando valores inteiros tornarem-se unificados). Esse uso não padrão pode ser desagradável para alguns, mas dá ao usuário a capacidade de construir rapidamente vetores complicados de uma forma muito legível. Quando o número de pontos é especificado dessa maneira, o ponto final é inclusivo.

O "r" significa concatenação de linha (**row**), porque se os objetos entre vírgulas forem matrizes bidimensionais, eles serão empilhados por linhas (e, portanto, devem ter colunas proporcionais). Existe um comando equivalente **c_** que empilha arrays bidimensionais por colunas, mas funciona de forma idêntica a **r_** para arrays unidimensionais.

A função **mgrid** no seu caso mais simples, é usada para construir intervalos de faixas unidimensionais como substituto conveniente para a função **arange**. Também usa números complexos na indicação do tamanho do passo para indicar o número de pontos a serem colocados entre os pontos finais (inclusive). O propósito real dessa função, entretanto, é produzir **N** matrizes, **N**-dimensionais que forneçam matrizes de coordenadas para um volume **N**-dimensional. A maneira mais fácil de entender isso é com um exemplo de uso da função:

```
>>> np.mgrid[0:5,0:5]
```

```

array([[0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1],
       [2, 2, 2, 2, 2],
       [3, 3, 3, 3, 3],
       [4, 4, 4, 4, 4]],

      [[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]]])
>>> np.mgrid[0:5:4j,0:5:4j]
array([[[ 0.        ,  0.        ,  0.        ,  0.        ],
       [ 1.66666667,  1.66666667,  1.66666667,  1.66666667],
       [ 3.33333333,  3.33333333,  3.33333333,  3.33333333],
       [ 5.        ,  5.        ,  5.        ,  5.        ]],

      [[[ 0.        ,  1.66666667,  3.33333333,  5.        ],
       [ 0.        ,  1.66666667,  3.33333333,  5.        ],
       [ 0.        ,  1.66666667,  3.33333333,  5.        ],
       [ 0.        ,  1.66666667,  3.33333333,  5.        ]]]])
>>>

```

Tendo matrizes de malha como este é, por vezes, muito útil. No entanto, nem sempre é necessário avaliar apenas algumas funções N -dimensionais numa grade, devido às regras de transmissão de arranjos do **Numpy** e do **SciPy**. Se este for o único propósito da geração de uma grade de malha, então você deve usar a função **ogrid** que gera uma grade “aberta” usando **newaxis** para criar N matrizes N -dimensionais, onde somente uma dimensão em cada arranjo tem comprimento maior que 1. Isso economizará memória e produzirá o mesmo resultado se a única finalidade da grade de malha for gerar pontos de amostra para avaliação de uma função N -dimensional.

Polinômios

Existem duas maneiras (intercambiáveis) de lidar com polinômios unidimensionais no **SciPy**. A primeira é usar a classe **poly1d** do **Numpy**. Essa classe aceita coeficientes ou raízes polinomiais para inicializar um polinômio. O objeto polinomial pode então ser manipulado em expressões algébricas, integrado, diferenciado, avaliado e até mostrado como um polinômio:

```

>>> from numpy import poly1d
>>> p = poly1d([3,4,5])
>>> print p
2
3 x + 4 x + 5
>>> print p*p
4      3      2
9 x + 24 x + 46 x + 40 x + 25
>>> print p.integ(k=6)
3      2
1 x + 2 x + 5 x + 6
>>> print p.deriv()
6 x + 4
>>> p([4, 5])           # avaliação para x = 4 e x = 5
array([ 69, 100])

```

A outra maneira de lidar com polinômios é como matriz de coeficientes, com o primeiro elemento da matriz fornecendo o coeficiente da maior potência. Existem funções explícitas para adicionar,

subtrair, multiplicar, dividir, integrar, diferenciar e avaliar polinômios representados como sequências de coeficientes.

Funções de Vetorização

O **NumPy** fornece uma classe vetorializada para converter uma função Python comum, que aceita escalares e retorna escalares, em uma “**função vetorializada**” com as mesmas regras de transmissão que outras funções **Numpy** (ou seja, as funções Universais ou **ufuncs**). Por exemplo, suponha que você tenha uma função Python chamada **somasubtrai** definida como:

```
>>> def somasubtrai(a,b):
...     if a > b:
...         return a - b
...     else:
...         return a + b
>>>
```

que define uma função que atua em duas variáveis escalares e retorna um resultado escalar. A classe **vectorize** pode ser usada para “vetorizar” esta função:

```
>>> vec_somasubtrai = np.vectorize(somasubtrai)
```

Retorna uma função que recebe argumentos matriciais e retorna um resultado matricial:

```
>>> vec_somasubtrai([0,3,6,9],[1,3,5,7])
array([1, 6, 1, 2])
>>>
```

Esta particular função poderia ter sido escrita em forma vetorial sem o uso de **vectorize**. No entanto, funções que empregam rotinas de otimização ou integração provavelmente só podem ser vetorializadas usando **vectorize**.

Manipulação de Tipos

Observe a diferença entre **np.iscomplex/np.isreal** e **np.iscomplexobj/np.isrealobj**. As funções **np.iscomplex/np.isreal** são baseadas em matrizes e retornam matrizes de uns e zeros fornecendo o resultado do teste elementar. As funções **np.iscomplexobj/np.isrealobj** são baseadas em objetos e retornam um escalar descrevendo o resultado do teste em todo o objeto.

Frequentemente, é necessário obter apenas a parte real ou a imaginária de um número complexo. Enquanto números complexos e matrizes têm atributos que retornam esses valores, se não tiver certeza se o objeto será ou não de valor complexo, é melhor usar as funções **np.real** e **np.imag**. Essas funções são funcionam para qualquer coisa que possa ser transformada em um arranjo **Numpy**. Considere também a função **np.real_if_close** que transforma um número de valor complexo com pequena parte imaginária em um número real.

Para verificar se um número é ou não escalar ((**long**) **int**, **float**, **complex** ou matriz de rank⁴-0) use a função **np.isscalar** que retorna 1 ou 0.

⁴ Posto ou Característica de uma matriz - número de linhas ou colunas linearmente independentes da matriz.

Para garantir que objetos sejam do tipo **Numpy** existe um dicionário de funções no **SciPy** específicas para isto: **np.cast**. As chaves do dicionário são os tipos de dados para os quais se deseja a conversão e os valores do dicionário armazenam funções para executar as conversões.

```
>>> np.pi
3.141592653589793
>>> type(np.pi)
<type 'float'>
>>> np.cast['f'](np.pi)
array(3.1415927410125732, dtype=float32)
>>> np.cast['i'](np.pi)
array(3)
>>> np.cast['l'](np.pi)
array(3)
>>> sorted(np.cast.items())
[(<type 'numpy.bool_'>, <function <lambda> at 0x02A2FF30>),
 (<type 'numpy.object_'>, <function <lambda> at 0x02A380F0>),
 (<type 'numpy.string_'>, <function <lambda> at 0x02A38170>),
 (<type 'numpy.unicode_'>, <function <lambda> at 0x02A381F0>),
 (<type 'numpy.void'_>, <function <lambda> at 0x02A382B0>),
 (<type 'numpy.int8'_>, <function <lambda> at 0x02A2FCF0>),
 (<type 'numpy.int16'_>, <function <lambda> at 0x02A2FD70>),
 (<type 'numpy.int32'_>, <function <lambda> at 0x02A2FDF0>),
 (<type 'numpy.int32'_>, <function <lambda> at 0x02A2FEB0>),
 (<type 'numpy.int64'_>, <function <lambda> at 0x02A2FF70>),
 (<type 'numpy.uint8'_>, <function <lambda> at 0x02A38070>),
 (<type 'numpy.uint16'_>, <function <lambda> at 0x02A38130>),
 (<type 'numpy.uint32'_>, <function <lambda> at 0x02A381B0>),
 (<type 'numpy.uint32'_>, <function <lambda> at 0x02A38270>),
 (<type 'numpy.uint64'_>, <function <lambda> at 0x02A382F0>),
 (<type 'numpy.float16'_>, <function <lambda> at 0x02A2FD30>),
 (<type 'numpy.float32'_>, <function <lambda> at 0x02A2FDB0>),
 (<type 'numpy.float64'_>, <function <lambda> at 0x02A2FE70>),
 (<type 'numpy.float64'_>, <function <lambda> at 0x02A2FEF0>),
 (<type 'numpy.datetime64'_>, <function <lambda> at 0x02A38030>),
 (<type 'numpy.timedelta64'_>, <function <lambda> at 0x02A380B0>),
 (<type 'numpy.complex64'_>, <function <lambda> at 0x02A2FFB0>),
 (<type 'numpy.complex128'_>, <function <lambda> at 0x02A38230>),
 (<type 'numpy.complex128'_>, <function <lambda> at 0x02A2FE30>)]
```

Outras Funções Úteis

Para processar a fase use a função **angle** e **unwrap**. Além disso, as funções **linspace** e **logspace** retornam amostras igualmente espaçadas em uma escala linear ou logarítmica.

Por fim, é útil estar ciente dos recursos de indexação do **Numpy**. Deve-se mencionar a função **select** que estende a funcionalidade do **where** para incluir várias condições e várias opções.

```
>>> select(listacond, listaescolha, default = 0).
```

select é uma forma vetorizada da declaração **if** múltipla. Permite a construção rápida de uma função que retorna uma matriz de resultados com base em uma lista de condições. Cada elemento da matriz de retorno é retirado da matriz numa lista de opções correspondente à primeira condição na listacond que é verdadeira. Por exemplo:

```

>>> x = np.r_[-2:3]
>>> x
array([-2, -1,  0,  1,  2])
>>> np.select([x > 3, x >= 0], [0, x+2])
array([0, 0, 2, 3, 4])
Algumas funções úteis adicionais também podem ser encontradas no módulo scipy.misc. Por exemplo: factorial e comb que calculam  $n!$  e  $n!/k!$  ( $n - k$ )! usando aritmética inteira exata (graças ao objeto inteiro Long) ou usando a precisão de ponto flutuante e a função gama. Outra função que retorna uma imagem comum usada no processamento de imagens: lena.
>>> import scipy.misc as sm
>>> import pylab as pl
>>> pl.gray()
>>> pl.imshow(sm,lena())
<matplotlib.image.AxesImage object at 0x0C068290>
>>> pl.show()
>>>

```

Transformada de Fourier (scipy.fftpack)

A análise de Fourier é um método usado para expressar um sinal como uma soma de componentes periódicas e para recuperar o sinal desses componentes. Quando o sinal e sua transformada de Fourier são substituídos por contrapartes discretizadas, ela é chamada de **Transformada Discreta de Fourier** (DFT).

A DFT se tornou um dos pilares da computação numérica, em parte devido a um algoritmo muito rápido de cálculo, chamado **Fast Fourier Transform** (FFT)⁵, que já era conhecido por Gauss (1805) e foi trazido à luz, em sua forma atual, por Cooley e Tukey (1965).

A FFT $X[k]$ de comprimento N , da sequência de comprimento- N $x[n]$ é definida como:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j2\pi \frac{k \cdot n}{N}}$$

E a FFT inversa é definida como:

$$x[n] = \sum_{k=0}^{N-1} X[k] \cdot e^{j2\pi \frac{k \cdot n}{N}}$$

Essas transformações podem ser calculadas por meio das funções **fft()** e **ifft()**, respectivamente.

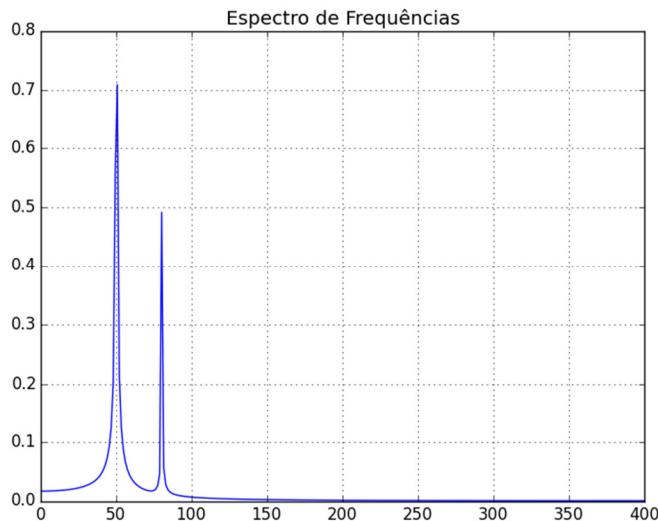
```

>>> from scipy.fftpack import fft, ifft
>>> x = np.array([1.0, 2.0, 1.0, -1.0, 1.5])
>>> X = fft(x)
>>> X
array([ 4.5 +0.j,  2.08155948-1.65109876j,
       -1.83155948+1.60822041j, -1.83155948-1.60822041j,
       2.08155948+1.65109876j])
>>> Xinv = ifft(X)
>>> Xinv
array([ 1.0+0.j,  2.0+0.j,  1.0+0.j, -1.0+0.j,  1.5+0.j])

```

⁵ Usuários para quem a velocidade das rotinas FFT é crítica devem considerar a instalação do pacote PyFFTW.

O exemplo representa a FFT da soma de dois senos nas frequências de 50 e 80 Hz:



Processamento de Sinais (scipy.signal)

A caixa de ferramentas de processamento de sinais contém atualmente algumas funções de filtragem, um conjunto limitado de ferramentas de projeto de filtros e alguns algoritmos de interpolação B-spline para dados unidimensionais e bidimensionais. Embora os algoritmos B-spline possam tecnicamente ser colocados sob a categoria do subpacote **interpolate**, eles são incluídos aqui porque eles só trabalham com dados igualmente espaçados e fazem uso intenso da teoria de filtragem e do formalismo da função de transferência para fornecer uma transformação B-spline rápida. Para entender esta seção, você precisará entender que um sinal no SciPy é uma matriz de números reais ou complexos.



Filtragem

Filtragem é um nome genérico dado à operação realizada por qualquer sistema que modifica um sinal de entrada de alguma forma. No **Scipy** um sinal pode ser visto como um arranjo (**array**) **Numpy** (vetor ou matriz).

Existem diferentes tipos de filtros para diferentes tipos de operações e, em geral, eles estão divididos em duas grandes categorias de operação de filtragem: linear e não linear. **Filtros lineares** podem sempre ser reduzidos a uma multiplicação de matrizes **Numpy** achatadas (todos os elementos da matriz são colocados numa única linha) por uma matriz resultante apropriada em outra matriz **Numpy** achatada.

```
# Retorna uma cópia do array colapsado em (reduzido a) uma dimensão.
numpy.ndarray.flatten(order='C')

>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])                      # construído por linha
>>> a.flatten('F')
```

```
array([1, 3, 2, 4])           # construído por coluna
```

É claro que isto não é usualmente a melhor forma para calcular o filtro, pois as matrizes e vetores envolvidos podem ser enormes. Por exemplo, a filtragem de uma imagem 512x512 com este método precisaria da multiplicação de uma matriz $512^2 \times 512^2$ por um vetor de 512^2 elementos. Apenas o armazenamento dessa matriz Numpy padrão, seriam necessários 68.719.476.736 elementos (68 bilhões). Usando armazenamento de 4 bytes por elemento (valor de precisão simples) seriam necessários mais de 256 GB de memória. Na maioria das aplicações os elementos dessa matriz são nulos e um método diferente para cálculo da saída deve ser empregado.

Convolução/Correlação

Muitos filtros lineares também possuem a propriedade de **invariância ao deslocamento**. Isto significa que a operação de filtragem é a mesma em locais diferentes do sinal e implica que a matriz de filtragem pode ser construída a partir do conhecimento de uma linha (ou coluna) da matriz sozinha. Neste caso, a multiplicação da matriz pode ser realizada usando transformadas de Fourier.

Seja $x[n]$ um sinal unidimensional indexado pelo inteiro n . Convolução de dois sinais unidimensionais pode ser expressa por:

$$y[n] = \sum_{k=-\infty}^{\infty} x[k].h[n-k]$$

Esta equação só pode ser implementada diretamente se limitarmos as sequências a sequências de durações finitas para que possam ser armazenadas num computador. Vamos escolher $n = 0$ para ser o ponto inicial de ambas as sequências, e seja $K + 1$ o valor para o qual $x[n] = 0$ para todo $n \geq K + 1$ e $M + 1$ seja o valor para o qual $h[n] = 0$ para todo $n \geq M + 1$, então a expressão de **convolução discreta** será:

$$y[n] = \sum_{k=\max(n-M,0)}^{\min(n,K)} x[k].h[n-k]$$

A convolução discreta de duas sequências finitas de comprimentos $K + 1$ e $M + 1$, respectivamente, resulta numa sequência finita de comprimento $K + M + 1 = (K + 1) + (M + 1) - 1$.

A convolução unidimensional é implementada no **SciPy** com a função **convolve**. Esta função tem como entradas os sinais x , h , e dois flags opcionais **mode** e **method** e retorna o sinal y .

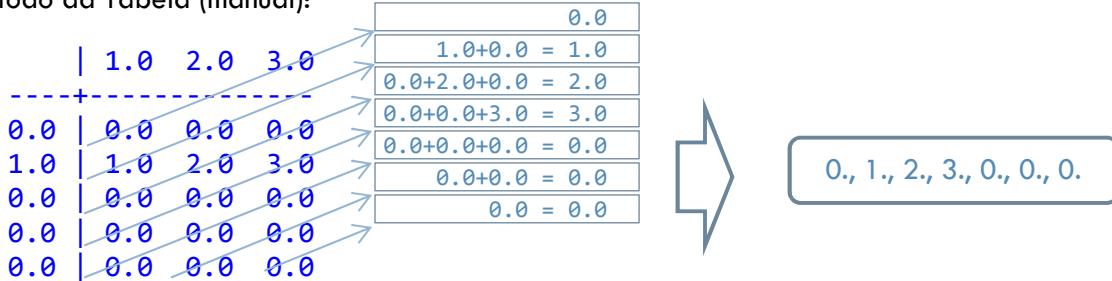
O flag **mode** permite especificar qual parte do sinal de saída será retornada. O valor padrão **full** retorna o resultado cheio, ou seja, todos os valores calculados na convolução (inclusive nulos). Se o flag for **same**, somente os K valores centrais serão retornados, começando em $y[(M - 1)/2]]$, de modo que a saída tenha o mesmo comprimento do primeiro sinal das entradas. Se o flag for **valid** então somente os $K - M + 1$ valores de saída são retornados.

O segundo flag "método", opcional, determina como a convolução é computada, seja através da abordagem da transformada de Fourier com **fftconvolve** ou através do método direto. Normalmente é selecionado o método esperado mais rápido. O método da transformada de Fourier tem ordem $O(N \log N)$ enquanto o método direto tem ordem $O(N^2)$. Dependendo da

constante O e do valor de N , um desses métodos pode ser mais rápido. O valor padrão "auto" executa um cálculo aproximado e escolhe o método esperado mais rápido, enquanto os valores "direto" e "fft" forçam a computação com os outros dois métodos. O código abaixo mostra um exemplo simples de convolução de 2 sequências:

```
>>> x = np.array([1.0, 2.0, 3.0])
>>> h = np.array([0.0, 1.0, 0.0, 0.0, 0.0])
>>> signal.convolve(x, h)
array([ 0.,  1.,  2.,  3.,  0.,  0.])
>>> signal.convolve(x, h, 'same')
array([ 2.,  3.,  0.])
```

Método da Tabela (manual):



Essa mesma função **convolve** pode, na verdade, usar matrizes N -dimensionais como entradas e retornar a convolução N -dimensional das duas matrizes, conforme mostrado no exemplo seguinte. Os mesmos flags de entrada também estão disponíveis para esse caso.

```
>>> x = np.array([[1.,1.,0.,0.],[1.,1.,0.,0.],[0.,0.,0.,0.],[0.,0.,0.,0.]])
>>> h = np.array([[1.,0.,0.,0.],[0.,1.,0.,0.],[0.,0.,1.,0.],[0.,0.,0.,0.]])
>>> signal.convolve(x, h)
array([[ 1.,  1.,  0.,  0.,  0.,  0.],
       [ 1.,  1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.]]))
```

A correlação é muito semelhante à convolução, troca-se apenas o sinal de menos por um mais. Portanto, a correlação cruzada dos sinais x e y , é dada por:

$$w[n] = \sum_{k=-\infty}^{\infty} y[k].x[n+k]$$

Para sinais de comprimento finito com $y[n] = 0$ fora do intervalo $[0, K]$ e $x[n] = 0$ fora do intervalo $[0, M]$, o somatório seria:

$$w[n] = \sum_{k=\max(n-M,0)}^{\min(n,K)} y[k].x[n+k]$$

A função **scipy.correlate()** implementa essa operação. Flags equivalentes aos da função **convolve()** estão disponíveis para esta operação para retornar a sequência completa de comprimento $K + M + 1$ (**full**) ou uma sequência com o mesmo tamanho da maior sequência iniciando em $w[-K + |M - 1|/2]$ (**same**) ou uma sequência onde os valores dependem de todos

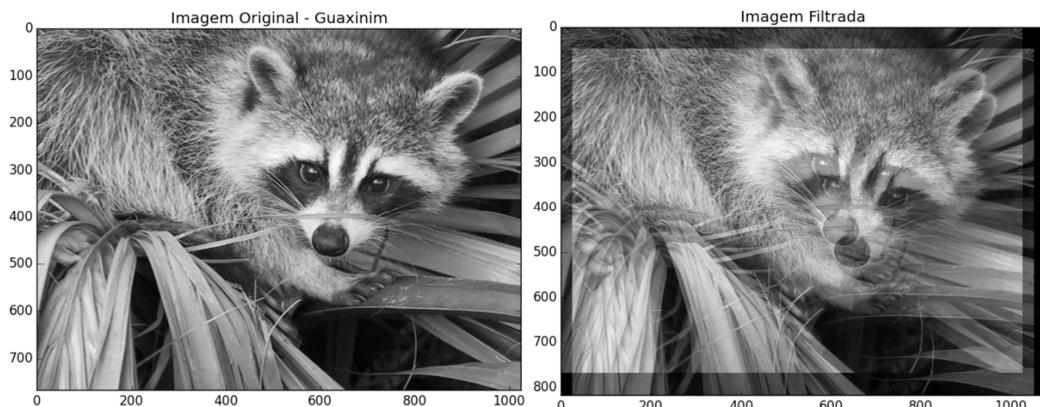
os valores da menor sequência (**valid**). Esta opção final retorna os $K - M + 1$ valores da sequência $w[M - K]$ até $w[0]$, inclusive.

Quando $N = 2$, correlacionar e/ou convoluir pode ser usado para construir filtros de imagem arbitrários para executar ações como desfoque, melhoramento e detecção de borda para uma imagem.

```
import numpy as np
from scipy import signal, misc
import matplotlib.pyplot as plt
imagem = misc.face(gray=True)
w = np.zeros((50, 50))
w[0][0] = 1.0
w[49][25] = 1.0
imagem_nova = signal.fftconvolve(imagem, w)

plt.figure(); plt.imshow(imagem); plt.gray()
plt.title('Imagen Original - Guaxinim')

plt.figure(); plt.imshow(imagem_nova); plt.gray()
plt.title('Imagen Filtrada'); plt.show()
```



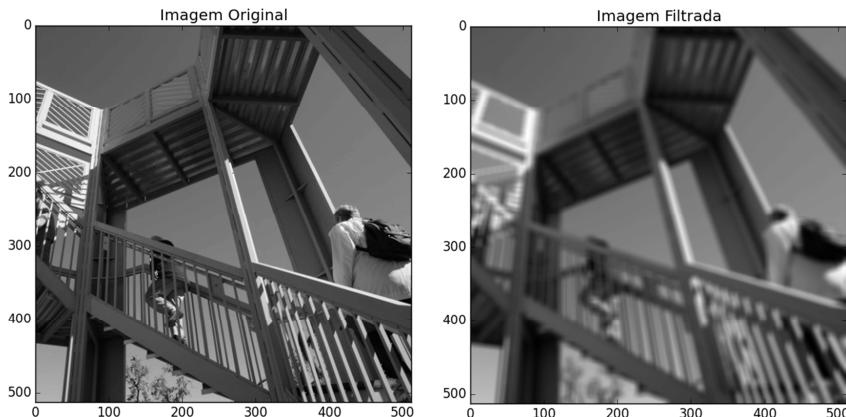
O cálculo da convolução no domínio do tempo, como acima, é usado principalmente para filtrar quando um dos sinais é muito menor do que o outro ($K \gg M$), caso contrário, a filtragem linear é calculada com mais eficiência no domínio de frequência, usando a função **fftconvolve()**. Por padrão, **convolve()** estima o método mais rápido usando **choose_conv_method()**.

Se a função de filtro $w[n, m]$ puder ser fatorada de acordo com: $h[n, m] = h_1[n]h_2[m]$, a convolução pode ser calculada por meio da função **sepfir2d()**.

Exemplo: Seja um filtro gaussiano $h[n, m] \propto e^{x^2-y^2} = e^{x^2} \cdot e^{y^2}$ que é frequentemente usado para borrar imagens.

```
import numpy as np
from scipy import signal, misc
import matplotlib.pyplot as plt
img = misc.ascent()
w = signal.gaussian(10, 10.0)
nova = signal.sepfir2d(img, w, w)
plt.figure(); plt.imshow(img); plt.gray(); plt.title('Imagen Original')
```

```
plt.figure(); plt.imshow(nova); plt.gray(); plt.title('Imagen Filtrada')
plt.show()
```



17. Matplotlib

Trata-se de uma excelente biblioteca para traçado de gráficos de dados científicos em 2D e 3D. Algumas vantagens desta biblioteca:

- Fácil de começar a usar;
- Suporte a etiquetas e textos formatados em LATEX;
- Ótimo controle de todos os elementos de uma figura, incluindo tamanho e DPI (*Dots Per Inch – Pontos Por Polegada*);
- Saída de alta qualidade em vários formatos, incluindo PNG, PDF, SVG, EPS;
- GUI para explorar figuras de forma interativa e suporte a geração de arquivos de figuras sem cabeçalhos (útil para trabalhos com arquivos de lotes de comandos - *batch*).

Uma das principais características do **matplotlib** é que todos os aspectos da figura podem ser controlados via programação (ou seja, sem precisar usar a GUI). Isso é importante para a reprodutibilidade, e conveniente quando é necessário regenerar a figura com dados atualizados ou alterar sua aparência.

O **Matplotlib** é incluído automaticamente como parte do espaço de nomes do **pylab** interativo, mas se você precisar importá-lo em seu próprio namespace (por exemplo, em um script ou módulo não interativo), assim como usamos a abreviação **np** para **NumPy** e **pd** para **Pandas**, usaremos algumas abreviações costumeiras para a importação do **Matplotlib**:

```
import matplotlib as mpl
import matplotlib.pyplot as plt
```

18. Pandas

O **Pandas** é uma API⁶ de análise de dados, orientada a colunas. É uma ótima ferramenta para manipular e analisar dados de entrada, e é usada por muitos frameworks ML⁷ para tratamento de dados. Embora uma introdução abrangente à API **Pandas** certamente teria muitas páginas, os principais conceitos são bem diretos. Para uma referência mais completa, o site **Pandas docs** contém extensa documentação e muitos tutoriais.

Nossos objetivos principais em relação ao pacote **Pandas** são:

- Apresentar as estruturas de dados **DataFrame** e **Series**.
- Acessar e manipular dados num **DataFrame** e **Series**.
- Importar dados CSV⁸ para um **DataFrame**.
- Reindexar um **DataFrame** para misturar dados.

Conceitos Básicos

A linha a seguir importa a API Pandas e mostra a versão da API instalada no seu sistema:

```
>>> from __future__ import print_function
>>> import pandas as pd
>>> pd.__version__

'0.15.2'
>>>
```

Sobre a primeira linha do script acima: considerando a versão 2.6+ do Python, você pode importar a função de impressão do Python-3 usando: **from __future__ import print_function**, com o objetivo de compatibilizar seu script com as futuras versões do Python.

No Python 3, a **instrução** de impressão foi alterada para uma **função**, ou seja, você deve usar:

```
>>> print('Olá mundão!')
Olá mundão!
>>>
```

Isso também funciona no Python 2, desde que você tenha colocado a instrução **from __future__ import print_function** no seu script. Lembre-se de que depois dessa instrução a versão 2.6+ do comando **print** já não poderá ser mais usada:

```
>>> from __future__ import print_function
>>> print 'Olá mundão!'
      File "<stdin>", line 1
        print 'Olá mundão!'
          ^
SyntaxError: invalid syntax
>>>
```

⁶ Application Programming Interface, do inglês: Interface de Programação de Aplicativos

⁷ Machine Learning, do inglês: Aprendizado de Máquina

⁸ Comma Separated Values, do inglês: Valores Separados por Vírgula

As estruturas de dados primárias em Pandas são implementadas como duas classes:

- **DataFrame**, que você pode imaginar como uma tabela de dados relacionais, com linhas e colunas nomeadas.
- **Series**, que é uma única coluna. Um **DataFrame** contém uma ou mais **Series** e um nome para cada **Series**.

O quadro de dados é uma abstração comumente usada na manipulação de dados.

Implementações semelhantes existem em Spark e R.

Uma maneira de criar uma **Series** é construir um objeto da classe **Series**. Por exemplo:

```
pd.Series(['Sao Francisco', 'Sao Jose', 'Sacramento'])
0    Sao Francisco
1        Sao Jose
2      Sacramento
dtype: object
>>>
```

Objetos **DataFrame** podem ser criados passando-se os nomes das colunas numa string de mapeamento do tipo dicionário (dict) para suas respectivas **Series**. Se a série não corresponder em comprimento, os valores omissos serão preenchidos com valores especiais NA/Nan. Exemplo:

```
>>> nomes_cid = pd.Series(['Sao Francisco', 'Sao Jose', 'Sacramento'])
>>> populacao = pd.Series([852469, 1015785, 485199])
>>> pd.DataFrame({ 'Nome da Cidade': nomes_cid, 'Populacao': populacao })
   Nome da Cidade  Populacao
0  Sao Francisco     852469
1      Sao Jose     1015785
2    Sacramento      485199
>>>
```

Mas na maioria das vezes, você carrega um arquivo inteiro num **DataFrame**. O exemplo a seguir carrega um arquivo com dados de habitação da Califórnia. Execute a seguinte célula para carregar os dados e criar definições de recursos:

```
>>> hab_calif_df = pd.read_csv("https://download.mlcc.google.com/mledu-
datasets/california_housing_train.csv", sep=",")
>>> hab_calif_df.describe()
   longitude      latitude  housing_median_age  total_rooms \
count  17000.000000  17000.000000  17000.000000  17000.000000
mean    -119.562108     35.625225     28.589353  2643.664412
std      2.005166     2.137340     12.586937  2179.947071
min    -124.350000    32.540000     1.000000    2.000000
25%    -121.790000    33.930000     18.000000  1462.000000
50%    -118.490000    34.250000     29.000000  2127.000000
75%    -118.000000    37.720000     37.000000  3151.250000
max    -114.310000    41.950000     52.000000  37937.000000

   total_bedrooms  population  households  median_income \
count  17000.000000  17000.000000  17000.000000  17000.000000
mean      539.410824    1429.573941     501.221941     3.883578
std       421.499452    1147.852959     384.520841     1.908157
```

```

min           1.000000    3.000000    1.000000    0.499900
25%          297.000000   790.000000   282.000000   2.566375
50%          434.000000  1167.000000  409.000000  3.544600
75%          648.250000  1721.000000  605.250000  4.767000
max          6445.000000 35682.000000 6082.000000 15.000100

      median_house_value
count      17000.000000
mean       207300.912353
std        115983.764387
min        14999.000000
25%       119400.000000
50%       180400.000000
75%       265000.000000
max       500001.000000
>>>

```

O exemplo acima usou **DataFrame.describe()** para mostrar estatísticas interessantes sobre um **DataFrame**. Outra função útil é o **DataFrame.head()**, que exibe os primeiros registros de um **DataFrame**:

```

>>> hab_calif_df.head()
   longitude  latitude  housing_median_age  total_rooms  total_bedrooms \
0     -114.31      34.19                  15         5612            1283
1     -114.47      34.40                  19         7650            1901
2     -114.56      33.69                  17         720             174
3     -114.57      33.64                  14         1501            337
4     -114.57      33.57                  20         1454            326

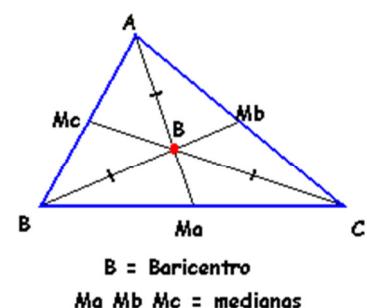
   population  households  median_income  median_house_value
0        1015         472      1.4936            66900
1        1129         463      1.8200            80100
2         333         117      1.6509            85700
3         515         226      3.1917            73400
4         624         262      1.9250            65500
>>>

```

Lembrete:

Mediana: substantivo feminino

1. GEOMETRIA: segmento de reta que liga o vértice de um triângulo retângulo ao meio do lado oposto.
2. ESTATÍSTICA: valor que divide um conjunto de valores ordenados em partes iguais (o valor do meio de um conjunto de dados).

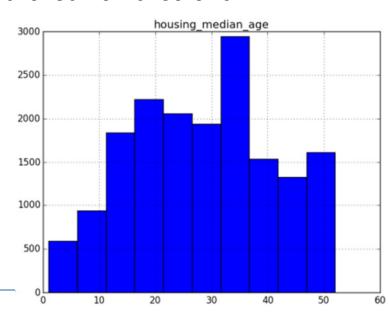


Outra característica poderosa do pacote **Pandas** é a representação gráfica. Por exemplo, **DataFrame.hist()** permite estudar rapidamente a distribuição de valores numa coluna:

```

>>> hab_calif_df.hist('housing_median_age')
array([[<matplotlib.axes._subplots.AxesSubplot object
at 0x0988C4D0>]], dtype=object)
>>>

```



Acessando Dados

Você pode acessar dados do **DataFrame** usando operações conhecidas de dicionários/listas do Python:

```
>>> cidades = pd.DataFrame({ 'Nome Cidade': nomes_cid, 'Populacao': populacao })
>>> print(type(cidades['Nome Cidade']))
<class 'pandas.core.series.Series'>
>>> cidades['Nome Cidade']
0    Sao Francisco
1        Sao Jose
2      Sacramento
Name: Nome Cidade, dtype: object
>>> cidades
   Nome Cidade  Populacao
0  Sao Francisco     852469
1      Sao Jose     1015785
2  Sacramento       485199
>>> cidades['Nome Cidade'][2]
'Sacramento'
>>> print(type(cidades['Nome Cidade'][2]))
<type 'str'>
>>> cidades[0:2]
   Nome Cidade  Populacao
0  Sao Francisco     852469
1      Sao Jose     1015785
>>>
```

Além disso, o **Pandas** oferece uma API extremamente rica para seleção e indexação avançada que é muito extensa para ser abordada aqui.

Manipulando Dados

Você pode aplicar as operações aritméticas básicas do Python a uma **Serie**. Por exemplo:

```
>>> populacao
0    852469
1    1015785
2    485199
dtype: int64
>>> populacao/1000.
0    852.469
1    1015.785
2    485.199
dtype: float64
>>>
```

Uma **Serie Pandas** pode ser usada como argumento para a maioria das funções do **NumPy**:

```
>>> import numpy as np
>>> np.log(populacao)
0    13.655892
```

```

1    13.831172
2    13.092314
dtype: float64
>>>

```

Para transformações mais complexas em coluna única, você pode usar o método **Series.apply**. Assim como a função **map** do Python, o método **Series.apply** aceita como argumento uma função **lambda**, que é aplicada a cada valor. O exemplo abaixo cria uma nova **Series** que indica se a população da cidade é superior a um milhão:

```

>>> populacao.apply(lambda val: val > 1000000)
0    False
1     True
2    False
dtype: bool
>>>

```

Modificar um **DataFrames** também é simples. Por exemplo, o código a seguir adiciona duas **Series** a um **DataFrame** existente:

```

>>> cidades['Area (milhas2)'] = pd.Series([46.87, 176.53, 97.92])
>>> cidades['Densidade Populacional'] = cidades['Populacao'] / cidades['Area (milhas2)']
>>> cidades
      Nome Cidade  Populacao  Densidade Populacional  Area (milhas2)
0   Sao Francisco    852469        18187.945381      46.87
1       Sao Jose     1015785        5754.177760      176.53
2   Sacramento      485199        4955.055147      97.92
>>>

```

Exercício:

Modifique a tabela de cidades adicionando uma nova coluna booleana que seja verdade (**True**) se, e somente se, as duas opções forem verdadeiras:

A cidade recebeu o nome de um santo.

A cidade tem uma área maior que 50 milhas quadradas.

Nota: As séries booleanas são combinadas usando os operadores bit a bit, em vez dos tradicionais booleanos. Por exemplo, ao executar a lógica ‘e’ use **&** em vez de **and**.

Solução:

```

>>> cidades['Grande e Nome Santo'] = (cidades['Area (milhas2)'] > 50) &
cidades['Nome Cidade'].apply(lambda nome: nome.startswith('Sao'))
>>> cidades
      Nome Cidade  Populacao  Densidade Populacional  Area (milhas2) \
0   Sao Francisco    852469        18187.945381      46.87
1       Sao Jose     1015785        5754.177760      176.53
2   Sacramento      485199        4955.055147      97.92

      Grande e Nome Santo
0                False
1                 True

```

```
2           False  
>>>
```

Índices

Os objetos **Series** e **DataFrame** também definem uma propriedade **index** que designa um valor de identificador para cada item da **Serie** ou linha do **DataFrame**. Por padrão, na construção, o pacote **Pandas** atribui valores de índice que refletem a ordenação dos dados de origem. Uma vez criados, os valores do índice são estáveis; isto é, eles não mudam quando os dados são reordenados.

```
>>> cidades.index  
Int64Index([0, 1, 2], dtype='int64')  
>>>
```

Chame a função **DataFrame.reindex()** para reorganizar manualmente as linhas. Por exemplo, o seguinte comando tem o mesmo efeito da classificação por nome da cidade:

```
>>> cidades  
      Nome Cidade  Populacao  Densidade Populacional  Area (milhas2)  \  
0   Sao Francisco    852469        18187.945381      46.87  
1       Sao Jose     1015785        5754.177760      176.53  
2   Sacramento      485199        4955.055147      97.92  
  
      Grande e Nome Santo  
0               False  
1               True  
2               False  
>>> cidades.reindex([2, 0, 1])  
      Nome Cidade  Populacao  Densidade Populacional  Area (milhas2)  \  
2   Sacramento      485199        4955.055147      97.92  
0   Sao Francisco    852469        18187.945381      46.87  
1       Sao Jose     1015785        5754.177760      176.53  
  
      Grande e Nome Santo  
2               False  
0               False  
1               True  
>>>
```

Reindexar é uma ótima maneira de embaralhar (randomizar) um **DataFrame**. No exemplo seguinte, pegamos o índice, que é semelhante a um vetor, e o passamos para a função **random.permutation** do **NumPy**, que embaralha seus valores no lugar. Chamar a reindexação com esse vetor embaralhado faz com que as linhas do **DataFrame** sejam embaralhadas da mesma maneira. Execute o comando seguinte várias vezes!

```
>>> cidades.reindex(np.random.permutation(cidades.index))  
      Nome Cidade  Populacao  Densidade Populacional  Area (milhas2)  \  
1       Sao Jose     1015785        5754.177760      176.53  
2   Sacramento      485199        4955.055147      97.92  
0   Sao Francisco    852469        18187.945381      46.87  
  
      Grande e Nome Santo  
1               True
```

```

2           False
0           False
>>> cidades.reindex(np.random.permutation(cidades.index))
   Nome Cidade Populacao Densidade Populacional Area (milhas2) \
1    Sao Jose    1015785      5754.177760      176.53
0  Sao Francisco     852469      18187.945381      46.87
2  Sacramento      485199      4955.055147      97.92

  Grande e Nome Santo
1           True
0          False
2          False
>>>

```

Exercício:

- O método de reindexação permite valores de índice que não estão nos valores de índice do **DataFrame** original. Experimente e veja o que acontece se você usar esses valores! Por que você acha que isso é permitido?

Solução:

Se o vetor de entrada de reindexação incluir valores que não estão nos valores originais do índice **DataFrame**, a função **reindex** incluirá novas linhas para esses índices "ausentes" e preencherá todas as colunas correspondentes com valores NaN:

```

>>> cidades.reindex([0, 4, 5, 2])
   Nome Cidade Populacao Densidade Populacional Area (milhas2) \
0  Sao Francisco     852469      18187.945381      46.87
4            NaN        NaN            NaN        NaN
5            NaN        NaN            NaN        NaN
2  Sacramento      485199      4955.055147      97.92

  Grande e Nome Santo
0           False
4          False
5          False
2          False
>>>

```

Esse comportamento é desejável porque os índices são geralmente cadeias de caracteres extraídos dos dados reais (consulte a documentação sobre [reindex do Pandas](#) para obter um exemplo em que os valores de índice são nomes de navegadores).

Nesse caso, permitir índices "ausentes" facilita a reindexação usando uma lista externa, já que não se precisa preocupar-se com a limpeza da entrada.

Fontes Consultadas:

<https://pyformat.info/>
<https://hackernoon.com/numpy-with-python-for-data-science-16ff2f646591>
<https://www.computerhope.com/unix/pylibbi.htm>
<https://www.python.org/community/logos/>
<https://www.programiz.com/python-programming/dictionary#methods>
<https://www.programiz.com/python-programming/examples/lcm>
https://colab.research.google.com/notebooks/mlcc/intro_to_pandas.ipynb#scrollTo=oa5wfZT7VHJl
<https://www.kaggle.com/harunshimanto/python-bootcamp-part-1>
<https://www.kaggle.com/colinmorris/working-with-external-libraries>
https://www.tutorialspoint.com/python/python_files_io.htm
https://www.tutorialspoint.com/python/python_exceptions.htm
<https://pythonacademy.com.br/blog/list-comprehensions-no-python>
<https://www.scipy.org/getting-started.html>
<https://www.sanfoundry.com/python-problems-solutions/>
https://aprendendo-computacao-com-python.readthedocs.io/en/latest/capitulo_07.html#travessia-e-o-loop-for

Proposta para próximo curso: **Python Avançado**

- POO (classes/objetos)
- Programação CGI (Web)
- Acesso a BD
- Redes
- Envio de E-mail
- *Multithreading*
- Processamento XML
- Programação GUI
- Extensões adicionais (**C**, C++, Java etc)

*Resposta aos Exercícios***Item 8:**

1.

```
# Contando letras de uma palavra
p1 = 'anticonstitucionalíssimamente'
p2 = 'pneumoultramicroscópicossilicovulcanoconiótico'
print "Qtde. de letras: p1 = %d, p2 = %d" % (len(p1), len(p2))
```

2.

```
# Ocorrências de letras de uma palavra - sol. com lista
ocor = [0]*26
orig = ord('a')
for letra in p1:
    ocor[ord(letra) - orig] += 1
for asc,freq in enumerate(ocor):
    print "Letra %s = %d" % (chr(asc+ord('a')), freq)
```

3.

```
# Ocorrências de letras de uma palavra - sol. com dicionário
ocor = {}
for letra in p1:
    ocor[letra] = ocor.get(letra,0) + 1
print ocor           # ordem alfabética: print sorted(ocor.items())
```