

INTRODUÇÃO À LINGUAGEM PYTHON



12/03/2019

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?

Trata-se de 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  
>>>
```

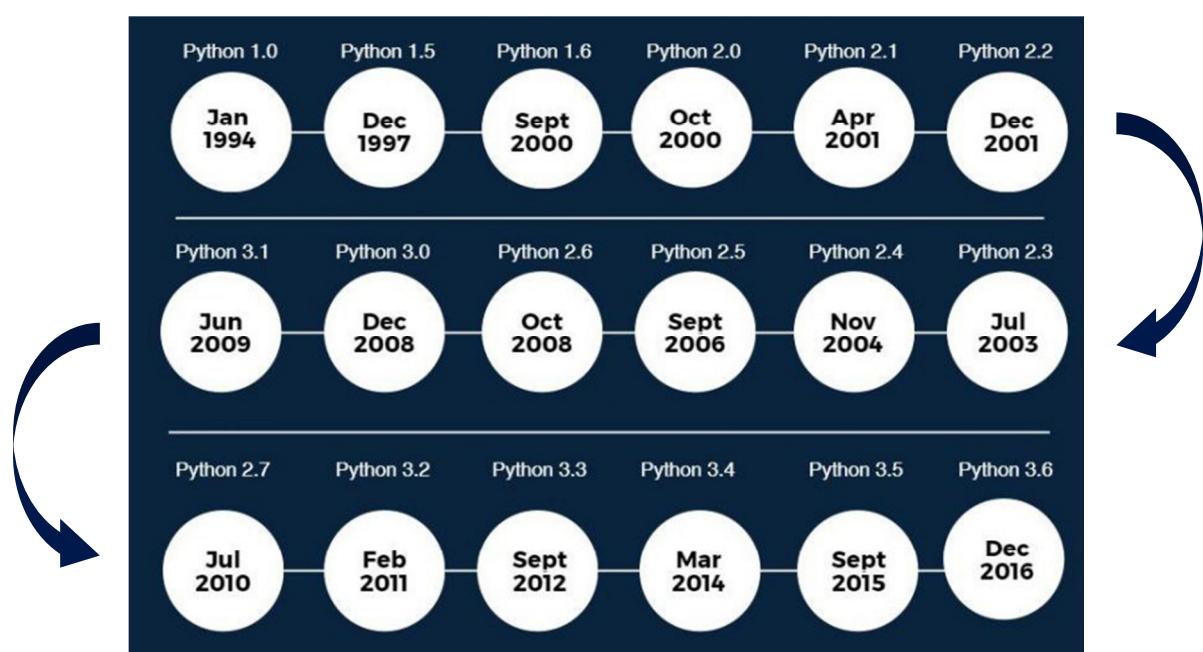
HISTÓRICO

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 particularmente popular, pois se pode adicionar interfaces programáveis a aplicativos existentes. A visão de Rossum era a de 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.

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, pois é uma linguagem de programação expressiva e de alto nível.
3. **Multiplataforma:** a ling. Python está disponível para execução em vários sistemas operacionais, tais como: Mac, Windows, Linux, Unix 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.

LISTA DE PALAVRAS-CHAVE EM PYTHON 2.7

O diagrama mostra uma grade de 7x5 de palavras-chave Python. As palavras-chave estão dispostas nas seguintes posições:

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	

As palavras-chave 'lambda', 'try', 'yield' e 'with' estão circundadas por caixas vermelhas. Existem três linhas azuis que conectam as palavras-chave 'lambda', 'try' e 'yield' entre si, formando um triângulo.

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 PYTHON

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

The screenshot shows the Python.org homepage. The navigation bar at the top includes 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 the word 'python™'. A red box highlights the 'About' tab in the secondary navigation menu, which also includes 'Downloads', 'Documentation', 'Community', 'Success Stories', 'News', and 'Events'. To the right of the menu, there is a code snippet demonstrating Python list comprehensions:

```
# Python 3: List comprehensions
>>> fruits = ['Banana', 'Apple', 'Lime']
>>> loud_fruits = [fruit.upper() for fruit in
fruits]
>>> print(loud_fruits)
['BANANA', 'APPLE', 'LIME']

# List and the enumerate function
>>> list(enumerate(fruits))
[(0, 'Banana'), (1, 'Apple'), (2, 'Lime')]
```

Below the code snippet, a section titled 'Compound Data Types' discusses lists. At the bottom of the page, a banner states: 'Python is a programming language that lets you work quickly and integrate systems more effectively. [» Learn More](#)'.

The screenshot shows the Python.org homepage again, but this time the 'Downloads' tab is highlighted with a red box. To the right of the menu, there is a large illustration of two boxes descending from the sky, each attached to a yellow and white striped parachute. Below the illustration, text provides links for different Python versions and release dates:

- Download Python 3.7.2
- Looking for Python with a different OS? Python for [Windows](#), [Linux/UNIX](#), [Mac OS X](#), [Other](#)
- Want to help test development versions of Python? [Pre-releases](#), [Docker images](#)
- Looking for Python 2.7? See below for specific releases

Below this, a section titled 'Looking for a specific release?' lists Python releases by version number:

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.

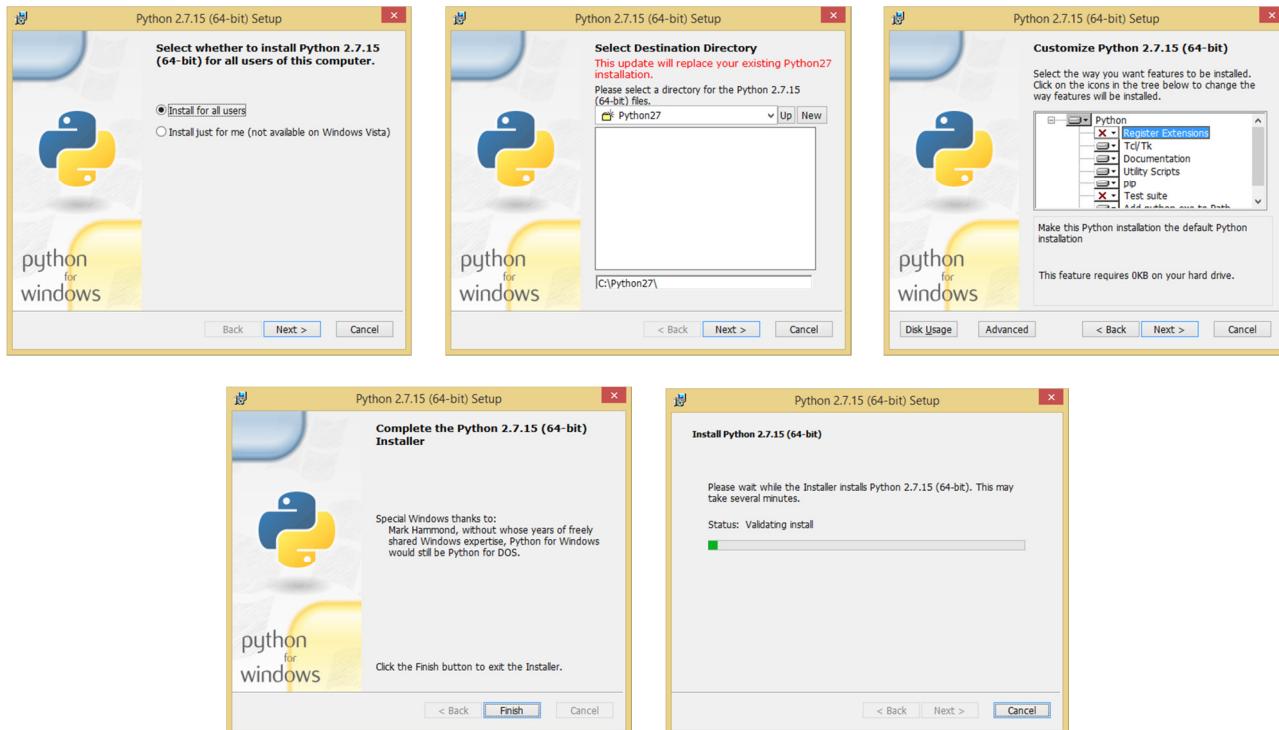
Files

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

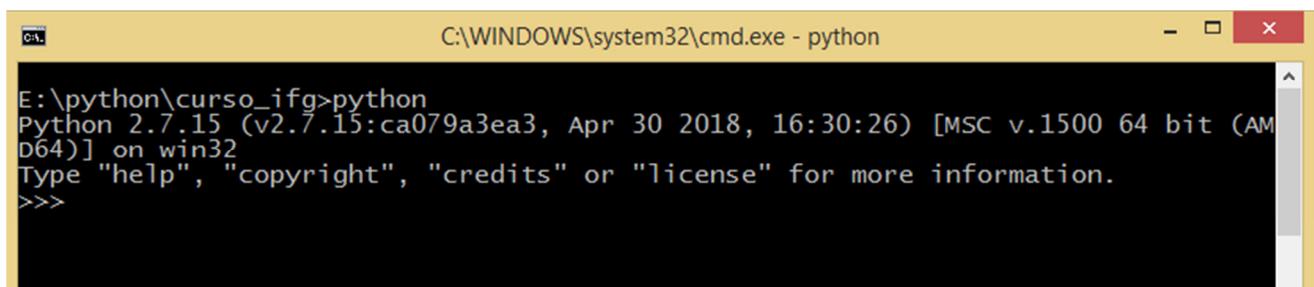
Este é o site oficial da ling. Python. A página web detectará o sistema operacional instalado no seu computador, e recomendará a versão adequada a ser baixada. Como estou usando o Windows-64 no meu notebook, foram dadas as opções de download para Python-2 e Python-3 para Windows.

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: **Python 2.7.15** - ver captura de tela 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 >].

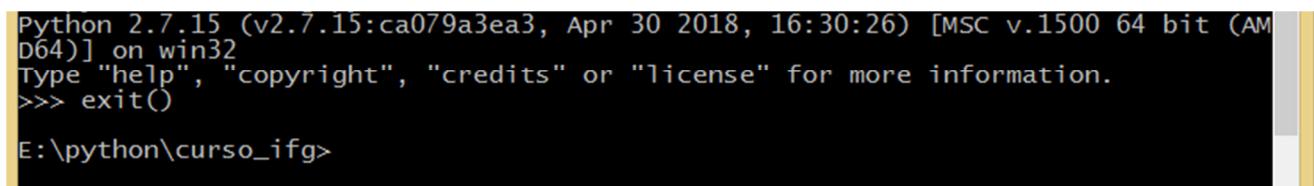


Após a instalação inicie uma janela de execução de comandos de linha (**terminal**) e carregue o interpretador Python instalado:



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 Aritméticas
2. Conversão de Tipo
3. String
4. Intervalo
5. Lista
6. Tupla
7. Conjunto
8. Dicionário
9. Entrada do usuário
10. Controle de Fluxo de Execução
11. Funções Incorporadas

PACOTES

12. Numpy
13. Scipy
14. Matplotlib
15. Pandas

1. Variáveis e Operações Aritméticas:

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.

No interpretador Python 2.7.15 digite:

```
>>> X = u'Python é útil no Machine Learning (ML)'
>>> print X
Python é útil no Machine Learning (ML)
>>>
```

```
>>> ML = 5
>>> print ML
5
>>>
```

```
>>> XL = ML + 10
>>> print XL
15
>>>
```

No exemplo acima, inicialmente, 'Python é útil no Machine Learning (ML)' (uma *string*) é armazenada numa variável X. Na próxima figura, a instrução **print** apresenta o valor armazenado na variável X. Depois é atribuído 5 à variável ML, e então mostrado na tela. Na linha seguinte foi feito uma operação aritmética de soma: ML + 10.

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.

```
>>> 12-4+9  
17  
>>> (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  
>>>
```

2. Conversão de Tipo de Dado (Casting, moldagem)

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 sintaxe
... '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 sintaxe
... "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 M\xaelquina'
>>> 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]
'i
>>> # 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]
'!aIB aid 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 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

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 descriptivo quanto um nome, esse estilo simples deve ser usado apenas para formatar uma quantidade relativamente pequena de elementos.

À moda antiga:	À moda nova:
<code>print '%s, %s.' % ('Fleury', 'Cláudio')</code>	<code>print '{}, {}'.format('Fleury', 'Cláudio')</code>
	Fleury, Cláudio.
<code>print '%s, %.1s.' % ('Fleury', 'Cláudio')</code>	<code>print '{}, {:.1}'.format('Fleury', 'Cláudio')</code>
	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 pythônico:

```
>>> 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

comparam como iguais podem ter diferentes atributos de índice, final e passo, por exemplo:
`range(0) == range(2,1,3)` ou `range(0,3,2) == range(0,4,2)`.

5. Lista:

Listas **são coleções de objetos heterogêneas**, 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 ou reais (ponto flutuante), strings, ou qualquer objeto. As listas também são muito úteis para implementar 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 em Python 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 na 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 (acessar cada um de seus elementos).

```
>>> placas = ['RPI', 'BeagleBone', 'Arduino']
>>> print placas[0], placas[2]
Rpi Arduino
>>> print placas
['RPI', 'BeagleBone', 'Arduino']
>>>
• Lista Vazia

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

• Lista Misturada

>>> mix = [3.14, -5, 'Tudo certo?', True, '*']
>>> print mix
[3.14, -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]
[1, 2, 3]
>>> print num[2:-2]
[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 tentativa de soma de 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:
...         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
>>>
```

- 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

```
>>> carros = 'focus/jetta/408/cruze/civic/corolla/mercedes c180/sentra/cerato'
>>> lista_carros = carros.split("/")
>>> print lista_carros
['focus', 'jetta', '408', 'cruze', 'civic', 'corolla', 'mercedes c180', 'sentra',
'cerato']
```

- Concatenando Itens da Lista

```
>>> ' '.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(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*

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 = [x.upper() for x 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']  
>>>
```

Métodos (funções) da classe *list*:

append()	acrescenta um único elemento à lista
clear()	elimina todos os elementos da lista.
copy()	retorna uma cópia superficial/rasa (<i>shallow</i>) de uma lista
count()	retorna ocorrências do elemento na lista
extend()	acrescenta elementos de uma lista a outra lista
index()	retorna o menor índice do elemento na lista
insert()	insere elementos à lista
pop()	remove o elemento no índice dado ou o último se não indicado o índice
remove()	remove elementos da lista
reverse()	reverte a ordem dos elementos da Lista
sort()	classifica (ordena) os elementos de uma lista

Outras funções incorporadas e disponíveis para aplicação às listas:

any()	verifica se algum elemento de um iterável é True
all()	retorna True quando todos os elementos de um iterável é True
ascii()	retorna uma String contendo representação imprimível
bool()	converte um valor para Boolean
enumerate()	retorna um Object do tipo Enumerate
filter()	constrói um iterator a partir dos elementos que são True
iter()	retorna um iterator para um objeto
list()	cria uma lista
len()	retorna o comprimento de um Object
max()	retorna o maior elemento de um Object
min()	retorna o menor elemento de um Object
map()	aplica uma função a um iterável e retorna uma lista
reversed()	retorna um iterator de uma sequência revertida

<code>slice()</code>	cria um objeto <code>slice</code> especificado por <code>range()</code>
<code>sorted()</code>	retorna uma lista classificada a partir de um dado <code>iterable</code>
<code>sum()</code>	soma os itens de um <code>Iterable</code>
<code>zip()</code>	retorna um <code>Iterator</code> 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 pythônico:

```
>>> 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 Sábado!")
... else:
...     print("Desculpe, mas tenho compromisso no Sábado!")
...
Desculpe, mas tenho compromisso no Sábado!
>>>
```

Exercício: Retornar os números não comuns das listas, 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:

Conjuntos podem ser usados para executar 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 construir os usando a função `set()`. Vamos fazer um conjunto para ver como funciona:

```
>>> A = set()
>>> A
set(())
>>> A.add('ok')
>>> A
set(['ok']) ..

>>> A.add(1)
>>> A
set([1, 'ok'])
>>> A.add(-8)
>>> A
set([-8, 1, 'ok'])
>>> A.add(5)
>>> A
set([-8, 1, 'ok', 5])
>>> A.add(1)
>>> A
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.

Supressão de elementos repetidos de uma lista ou tupla:

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

- 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}
>>> # a operação de união é realizada pelo operador |
... print A | B
set([3, 4, 5, 6])
>>>
```

```
>>> A.union(B)
set([3, 4, 5, 6])
>>>
```

- *Interseção de Conjuntos*

A interseção entre A e B é um conjunto de elementos comuns em ambos os conjuntos. A interseção é realizada usando o operador '&'. O mesmo pode ser feito usando o método ***intersection()***.

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

- *Diferença entre Conjuntos*

Diferença entre A e B, ($A - B$), é um conjunto de elementos que estão apenas em A, mas não em B. Da mesma forma, ($B - A$) é um conjunto de elementos em B, mas não em A. A diferença é realizada usando operador '-' ou o método ***difference()***.

```
>>> A - B
set([3])
>>> print B - A
set([6])
>>> A.difference(B)
set([3])
>>> B.difference(A)
set([6])
>>>
```

- *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 B ^ A
set([3, 6])
>>>
```

8. Dicionário:

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

Se você estiver familiarizado com outras linguagens, pode pensar nesses dicionários como *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 dentro de chaves {} separados por vírgula. 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.

```
>>> # dicionário vazio
... dici = {}
>>> # dicionário com chaves do tipo 'inteiro'
... z = {1:'maçã', 2:'bola', 3:4545}
>>> z[1]
'ma\x87\xc6'
>>> z[1] = 'banana'
>>> z[1]
'banana'
>>>
>>> z
{1: 'banana', 2: 'bola', 3: 4545}
>>> z[3]
4545
>>>
```

```
>>> # dicionário com chaves mistas
... y = {'nome':'Maria', 'notas':[4,6,3], 5:'ok'}
>>> y[5]
'ok'
>>> y['nome']
'Maria'
>>> y['notas']
[4, 6, 3]
>>> y['notas'][0]
4
>>>
```

Também podemos criar um dicionário usando a função interna **dict()**.

```
>>> # usando a função interna dict()
... w = dict([(1,'banana'),(2,'broa')])
>>> w
{1: 'banana', 2: 'broa'}
>>>
```

Nós também podemos criar chaves por atribuição. Por exemplo, se começássemos com um dicionário vazio, poderíamos adicioná-lo continuamente:

```
>>> d = { }
>>> d['animal'] = 'Gato'
>>> d['idade'] = 6
>>> d
{'idade': 6, 'animal': 'Gato'}
>>>
```

- Acessando Elementos de um Dicionário

Dicionário são 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
{'idade': 5, 'cor': 'branco', 'animal': 'Gato'}
```

- Removendo Elementos de um Dicionário

Podemos remover um item específico em um dicionário usando o método **pop()**. Este método remove um 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 o comando **del** para remover itens individuais ou o dicionário inteiro.

```
>>> y
{5: 'ok', 'notas': [4, 6, 3], 'nome': 'Maria'}
>>> y.pop(5)
'ok'
>>> y
{'notas': [4, 6, 3], 'nome': 'Maria'}
>>> del y['notas']
>>> y
{'nome': 'Maria'}
>>>
```

```
>>> d
{'idade': 5, 'animal': 'Gato'}
>>> 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: **clear()**, **copy()**, **fromkeys(seq[, v])**, **get(key[,d])**, **items()**, **keys()**, **pop(key[,d])**, **popitem()**, **update([other])**, **values()**.

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

- Aninhamento de Dicionários

O Python é poderoso com sua flexibilidade de aninhar objetos e chamar métodos neles. 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.'}, 2:
{'nome':'Luiz', 'idade':35, 'sexo':'Masc.'} }
>>>
>>> print pessoas
{1: {'idade': 25, 'sexo': 'Fem.', '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.', '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 chave in info:
...         print chave, ': ', info[chave]
...
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 cada chave de um 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 embutidas como **all()**, **any()**, **len()**, **cmp()**, **sorted()** etc são comumente usadas com o dicionário para realizar diferentes tarefas.

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

```
>>> 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]
>>>
```

9. Entrada do Usuário

Use a função **input()** do Python para aceitar a entrada de dados feita pelo usuário. Você pode escrever programas em Python que aceitam a entrada do usuário. Você pode perguntar ao usuário seu nome, sua idade ou qualquer outra coisa. A entrada do usuário pode ser usada em seu programa de várias maneiras. Você pode simplesmente imprimir sua entrada de volta na tela, pode inseri-la em um banco de dados ou fazer com que o programa faça coisas diferentes dependendo da entrada recebida.

```
>>> nome = input("Qual é o seu nome?")
Qual é o seu nome? 'Washington'
>>> print nome
Washington
>>>
```

10. Controle de Fluxo de Execução

11. 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 iteração². 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
```

² **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.

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(function, iterable)`

Constrói uma lista a partir dos elementos **iterable** para os quais a aplicação da **function** retorna **True**. iterável pode ser uma sequência, um contêiner que suporta iteração ou um iterador. Se **iterable** for uma string ou uma tupla, o resultado também terá esse tipo; caso contrário, será sempre uma lista. Se **function** for **None**, a função **identity** é assumida, ou seja, todos os elementos iteráveis que forem **False** serão removidos.

Observe que a função **filter(function, iterable)** é equivalente à seguinte *list comprehension* `[item for item in iterable if function(item)]` se **function** for especificada, e equivale a `[item for item in iterable if item]` se **function** não for especificada.

```
>>> 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]
>>>
```

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

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

```

...
>>> listapositivos = filter(posit, lista)
>>> listapositivos
[8, 9, 3, 5, 2, 5, 91, 5, 10, 93]
>>>

```

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

```

>>> # lista aleatória
>>> lista = [1, 'a', 0, False, True, '0']
>>> filtrada = filter(None, lista)
>>> print 'Elementos filtrados:', filtrada
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]
>>>

```

```
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
>>>
```

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.

PACOTES - MÓDULOS

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)

```
>>> # o atributo "ravel" soa interessante. Eu sempre fui fã de música clássica...
```

```
>>> help(lancamentos.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, diga-me tudo o que há para saber sobre numpy.ndarray
>>> help(lancamentos)
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
...
>>>
```

12. Numpy



O NumPy é o 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 de dados genéricos. Tipos de dados arbitrários podem ser definidos. Isso permite que o NumPy integre-se de forma fácil e rápida a uma ampla variedade de bancos de dados. Ótimo, vamos ver como usar a biblioteca Numpy para manipulação básica de arranjos (*array*).

Primeiro, precisamos importar numpy, para então 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])
>>> r2 = np.repeat([1,2,3],6)
>>> r2
array([1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 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.

13. Scipy

14. 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
```

15. Pandas

O **Pandas** é uma API³ de análise de dados, orientada a colunas. É uma ótima ferramenta para manipular e analisar dados de entrada, e é usada por muitos frameworks ML⁴ 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 CSV⁵ 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!  
>>>
```

³ Application Programming Interface, do inglês: Interface de Programação de Aplicativos

⁴ Machine Learning, do inglês: Aprendizado de Máquina

⁵ Comma Separated Values, do inglês: Valores Separados por Vírgula

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
>>>
```

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 \
0           2.339519       37.854722            3.524324      6.000000
1           2.354514       37.854722            3.471576      6.000000
2           2.329514       37.854722            3.549324      6.000000
3           2.339519       37.854722            3.524324      6.000000
4           2.354514       37.854722            3.471576      6.000000
```

```

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**:

```

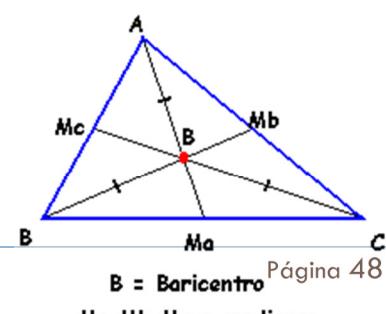
>>> hab_calif_df.head()
    longitude  latitude  housing_median_age  total_rooms  total_bedrooms \
0       -114.31      34.19                  15           5612            1283
1       -114.47      34.40                  19           7650            1901
2       -114.56      33.69                  17           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:

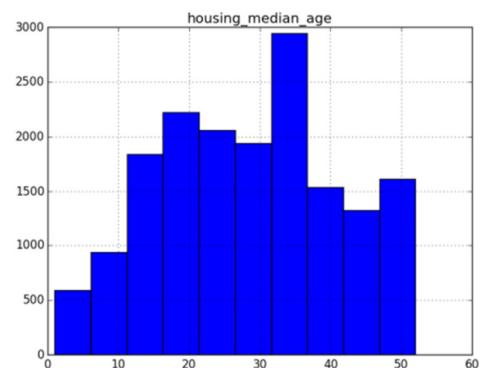
Mediana: substantivo feminino



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.

Em construção...

Fontes:

- <https://www.kaggle.com/harunshimanto/python-bootcamp-part-1>
- <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://colab.research.google.com/notebooks/mlcc/intro_to_pandas.ipynb#scrollTo=oa5wfZT7VHJI
- <https://www.kaggle.com/colinmorris/working-with-external-libraries>