

# INTRODUÇÃO À LINGUAGEM PYTHON



05/03/2019

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 maior a apresentação da Linguagem de Programação Python. Essa linguagem tem como principal característica o ecletismo, permitindo ao usuário o desenvolvimento de aplicações de toda sorte, desde aplicações que envolvam processamento científico até aplicações comerciais, para rodar no desktop, tablet ou smartphone, envolvendo banco de dados, 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

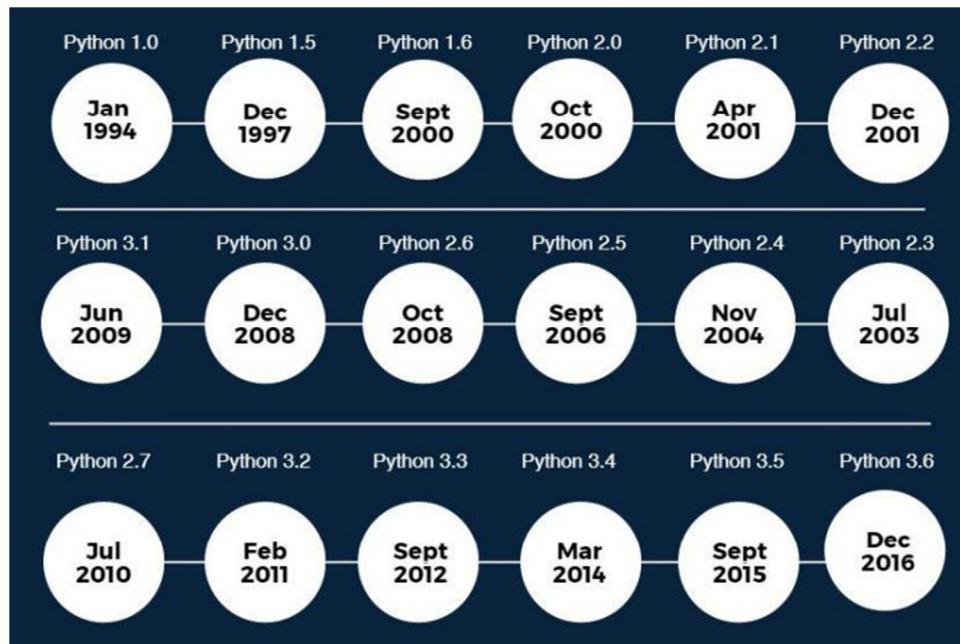
A linguagem Python começou a ser desenvolvida por Guido van Rossum em 1989 (lançada oficialmente em 1999) na Centrum Wiskunde & Informatica (CWI), na Holanda, como sucessora da linguagem ABC (inspirada na SETL<sup>1</sup>) que era capaz de lidar com exceção e interagir com o sistema operacional Amoeba 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 não é pela serpente Python, embora o ícone utilize duas cobras estilizadas.

Linha do tempo do lançamento das versões:



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

<sup>1</sup> 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 altamente extensível. Essa modularidade compacta tornou-a particularmente popular, pois pode-se 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 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. **Grátis:** a ling. Python é gratuita para download e uso. Isso significa que você pode baixá-la gratuitamente e usá-la para criar sua aplicação.
7. **Permite a 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 suporta o tratamento de exceções, o que significa que podemos escrever menos código propenso a erros e testar vários cenários que possam provocar uma exceção mais tarde.
8. **Recursos avançados:** geradores e abrangência de lista (*list comprehension*). Nós cobriremos esses recursos mais tarde.
9. **Gerenciamento automático de memória:** a memória é limpa e liberada automaticamente. Você não precisa se preocupar em limpar a memória.

## LISTA DE PALAVRAS-CHAVE EM PYTHON 2.7

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

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

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

## COMO INSTALAR O PYTHON?

Você pode instalar o Python em qualquer sistema operacional, como Windows, Mac OS X, Linux/Unix e outros. Para instalar o Python em seu sistema operacional, acesse <https://www.python.org/downloads/>. Você verá uma tela como aquela mostrada na figura seguinte.

The screenshot shows the Python.org homepage. The top navigation bar includes links for Python, PSF, Docs, PyPI, Jobs, and Community. Below the navigation is a search bar with a 'GO' button. The main content area features a large Python logo. A navigation bar below the logo includes links for About, Downloads (which is highlighted with a red box), Documentation, Community, Success Stories, News, and Events. To the left of the main content, there is a code snippet example:

```
# 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')]
```

To the right of the code example is a section titled "Compound Data Types" with a brief description and a link to "More about lists in Python 3". Below the code example are five small numbered boxes (1, 2, 3, 4, 5). 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 with the 'Downloads' tab selected. The main content area features a large illustration of two packages hanging from parachutes. The text "Download the latest version for Windows" is prominently displayed, followed by a button labeled "Download Python 3.7.2". Below this, there are links for "Looking for Python with a different OS? Python for [Windows](#), [Linux/UNIX](#), [Mac OS X](#), [Other](#)" and "Want to help test development versions of Python? [Pre-releases](#), [Docker images](#)". There is also a note for "Looking for Python 2.7? See below for specific releases".

Below the main content, there is a section titled "Looking for a specific release?" with a table showing Python releases by version number. The table has columns for "Release version", "Release date", and "Click for more". The row for "Python 2.7.15" is highlighted with a red box. The table entries are:

Release version	Release date	Click for more
Python 2.7.15	2018-05-01	<a href="#">Download</a> <a href="#">Release Notes</a>
Python 3.6.5	2018-03-28	<a href="#">Download</a> <a href="#">Release Notes</a>

## Python 2.7.15

Release Date: 2018-05-01

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

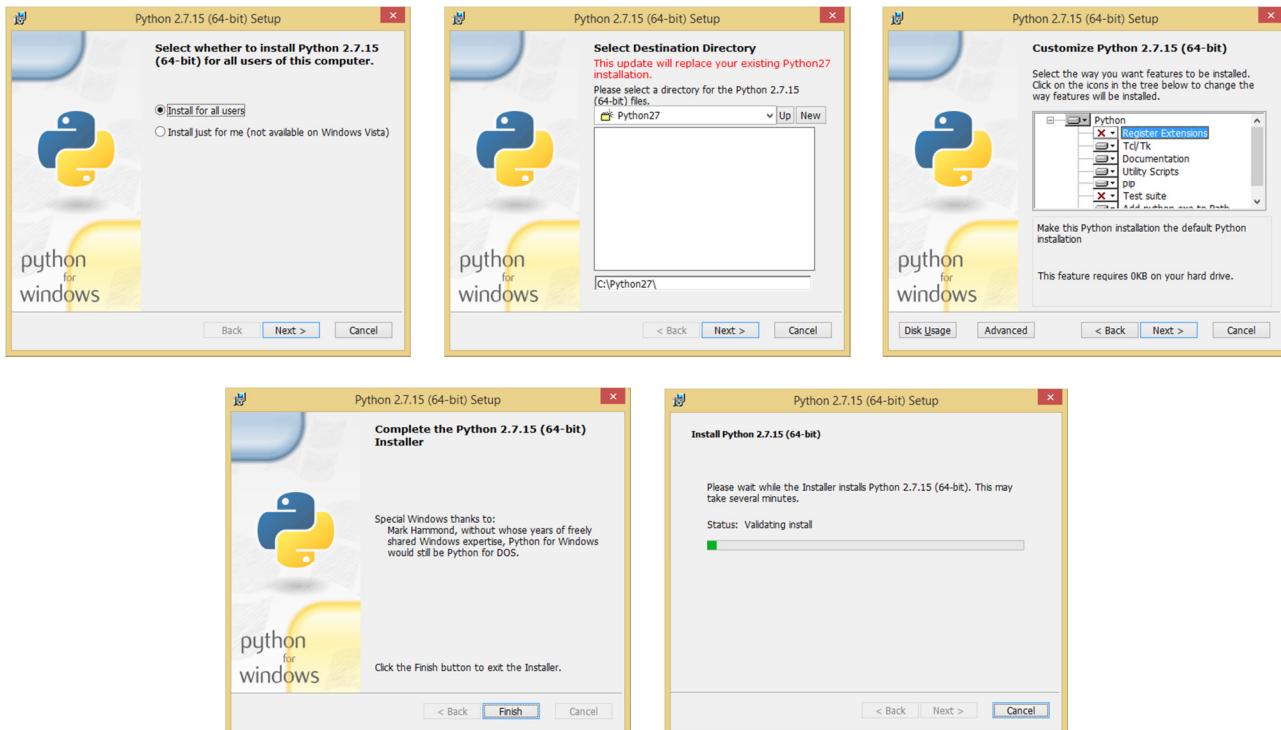
### Files

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	223b71346316c3ec7a8dc8bff5476d84	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	0ffa44a86522f9a37b916b361eebc552	20246528	SIG
Windows x86 MSI installer	Windows		023e49c9fba54914ebc05c4662a93ff	19304448	SIG
					19,3 MB

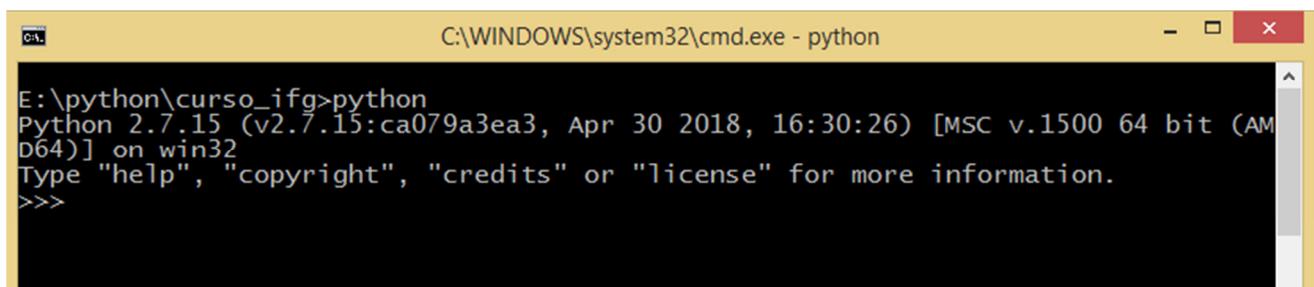
Este é o site oficial da ling. Python. A página Web detectará o sistema operacional instalado no seu computador, e com base nisso, recomendará a versão adequada para você baixar. Como estou usando o Windows-64, foram dadas as opções de download para Python-2 e Python-3 para Windows.

Neste curso usaremos a versão 2.7, portanto recomendo que você baixe a versão mais recente do Python-2 (à época da escrita desse texto: **Python 2.7.15**, como mostrado na captura de tela anterior).

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

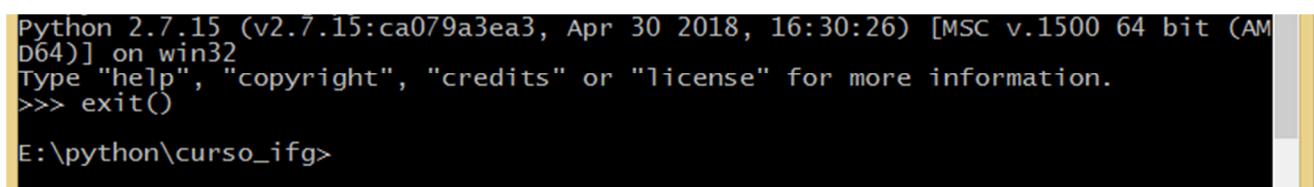


Após a instalação inicie uma janela de execução de comandos de linha (terminal) e carregue o interpretador Python instalado:



A screenshot of a Windows Command Prompt window titled "C:\WINDOWS\system32\cmd.exe - python". The window shows the Python interpreter starting at the command line "E:\python\curso\_ifg>python". The output text is:  
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 use a função `exit()` (finalizada com [Enter]).



A screenshot of a Windows Command Prompt window showing the Python interpreter exiting. The window shows the Python interpreter starting at the command line "E:\python\curso\_ifg>python". The output text is:  
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

## PACOTES

11. Numpy
12. Scipy
13. Matplotlib
14. 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 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 simples
... 'Fleury'
'Fleury'
>>>
```

```
>>> # uma frase
... 'Bem-vindos ao IFG - Curso de Extensão 2019/1'
'Bem-vindos ao IFG - Curso de Extensão 2019/1'
>>>
```

```
>>> # erro de syntax
... 'Valor de 'x' no programa?'
  File "<stdin>", line 2
    'Valor de 'x'; no programa?'
      ^
SyntaxError: invalid syntax
>>>
```

A razão para o erro acima é porque a aspa simples delimitou duas *strings* e um caractere 'x' ficou sem delimitação entre elas. Você pode usar combinações de aspas duplas e simples para obter a declaração correta.

```
>>> "Valor de 'x' no programa?"
"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 pegar tudo, 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!

- ### • *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.

**Formatação básica:** A formatação posicional simples é provavelmente o caso de uso mais comum. Use-o se a ordem dos seus argumentos não for alterada e você tiver poucos elementos que queira concatenar.

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

```
>>> print '%s, %.1s.' % ('Fleury', 'Cláudio')
Fleury, C.
>>> # estilo novo
... print '{1}, {0:.1}'.format('Cláudio', 'Fleury')
Fleury, C.
>>>
```

Com a nova formatação de estilo, é possível (e no Python 2.6, mesmo obrigatório) 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.

```
>>> # estilo novo
... print '{1}, {0}'.format('Cláudio', 'Fleury')
Fleury, Cláudio.
>>>
```

Com números inteiros e reais:

```
>>> print '%d %f' % (54, 2.334343)
54 2.334343
>>> # estilo novo
... print '{:d} {:.2f}'.format(54, 2.334343)
54 2.334343
>>>
>>> print '{:6.2f}'.format(2.334343)
    2.33
>>>
```

#### 4. Intervalo

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

Os intervalos podem ser construídos de duas formas:

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

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

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

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

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

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

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

```

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

```

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

## 5. Lista:

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 para o primeiro valor armazenado. Cada elemento na lista tem seu lugar definido na lista, o que permite a duplicação de elementos na lista, com cada elemento tendo seu próprio lugar memória.

A lista é uma ferramenta útil para preservar uma sequência de dados e para iterar sobre ela (acessar elementos lista).

```

>>> 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*

```
>>> lista_2d = [['12', 3, True], [0.1, 'ok', 22], [False, 15, True]]
>>> print lista_2d
[['12', 3, True], [0.1, 'ok', 22], [False, 15, True]]
>>> print lista_2d[0][1]
3
>>>
```

```
>>> for linha in lista_2d:
...     for elem in linha:
...         print elem,
...     print
...
12 3 True
0.1 ok 22
False 15 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 'Peso Total: ', soma
Peso Total: 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 elem in pesos:
...     if type(elem) == int:
...         soma += elem
...
>>> print 'Peso Total: ', soma
Peso Total: 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 a minha cor favorita!
vermelho
verde
roxo
>>>
```

- Alteração da Lista

```
>>> pesos = [56, 73, 92, 'ok', 32, 45, True, "Vida"]
>>> print 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,"alegre")
>>> print pesos
[56, 73, 'alegre', 92, 'ok', 32, 45, True, 'Vida']
>>> pesos.remove(45)
>>> pesos
[56, 73, 'alegre', 92, 'ok', 32, True, 'Vida']
>>> del pesos[1]
>>> pesos
[56, 'alegre', 92, 'ok', 32, True, 'Vida']
>>>
>>>
>>> pesos
[56, 'alegre', 92, 'ok', 32, True, 'Vida']
>>> ultimo = pesos.pop()    # retorna último elemento e retira da lista
>>> print pesos, '\n', ultimo
[56, 'alegre', 92, 'ok', 32, True]
Vida
>>>
```

- Divisão de String em Itens da Lista

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

- Concatenando Itens da Lista

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

- Tamanho da Lista (quantidade de itens)

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

- Revertendo os Itens da Lista

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

- *List Comprehension*

Além das operações de sequência e métodos de lista, o Python inclui uma operação mais avançada chamada de inclusão em lista (*list comprehension*).

*List Comprehension* é uma construção oriunda da programação funcional, e equivale à notação matemática:

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

Ou seja,  $R$  é o conjunto formado por  $x$  dividido por 2 para todo  $x$  no conjunto dos números naturais, se  $x$  for maior ou igual a zero e menor ou igual a 5.

*List Comprehension* nos permite criar listas usando notação mais compacta do que seria possível usando *loops* de repetição.

```
>>> 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]
>>> R
[-5, -5, -4, -4, -3, -3, -2, -2, -1, -1, 0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5]
>>> 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]
```

## 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
<b>all()</b>	Retorna <b>True</b> se todas as chaves do dicionário são verdadeiras (ou se o dicionário está vazio)
<b>any()</b>	Retorna <b>True</b> se qualquer chave do dicionário é verdadeira. Se o dicionário está vazio, retorna <b>False</b>
<b>len()</b>	Retorna o comprimento (a quantidade de itens) do dicionário
<b>cmp()</b>	Compara os itens de dois dicionários
<b>sorted()</b>	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]
>>>
```

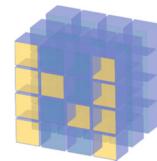
## 9. Entrada do Usuário

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

## PACOTES



NumPy

## 11. 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])
array([4, 5, 6])
```

Criando outro arranjo (matriz) 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 (linhas e 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 no lugar (a função "resize" altera a forma da matriz no lugar, ao contrário de "reshape").

```
>>> 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((3,3))
>>> z
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

Criando uma matriz diagonal com valores diagonais unitários:

```
>>> diag = np.eye(4)
>>> diag
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
>>> print diag
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
```

Criando um **array** com repetição de lista e outro usando função de repetição:

```
>>> r1 = np.array([1,2,3]*7)
>>> print r1
[1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3]
>>> r2 = np.repeat([1,2,3],3)
>>> print r2
[1 1 1 2 2 2 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([[ 0.30086379,  0.93115915,  0.42666759],
       [ 0.65619448,  0.09246489,  0.27937376]])
>>> b = np.random.rand(2,3)
>>> b
array([[ 0.36839724,  0.85417235,  0.91562177],
       [ 0.7308914 ,  0.2701699 ,  0.29323732]])
```

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

```
>>> pv = np.vstack([a,b])
>>> pv
array([[ 0.30086379,  0.93115915,  0.42666759],
       [ 0.65619448,  0.09246489,  0.27937376],
       [ 0.36839724,  0.85417235,  0.91562177],
       [ 0.7308914 ,  0.2701699 ,  0.29323732]])
```

Agora, horizontalmente:

```
>>> ph = np.hstack([a,b])
>>> ph
array([[ 0.30086379,  0.93115915,  0.42666759,  0.36839724,  0.85417235,
         0.91562177],
       [ 0.65619448,  0.09246489,  0.27937376,  0.7308914 ,  0.2701699 ,
         0.29323732]])
```

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

```
>>> a = np.array([[1,2],[3,4]])
>>> 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:

```
>>> # 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:

```
>>> 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
array([[1, 2],
       [3, 4]])
>>> a.sum()
10
>>> a.max()
4
>>> a.min()
1
>>> a.mean()
2.5
```

Tabuada de 5 e indexação e fatiamento:

```
>>> tab5 = np.arange(0,11)*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 linha e a quarta e quinta colunas da matriz **d**. Observe que a numeração das linhas e colunas começam em 0 (zero):

```
>>> d[3,3:6]
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([[ 9, 13, 14],
       [14, 15,  8],
       [16, 16,  9]])
```

Muito bem, até agora vimos como criar, como acessar e como manipular matrizes em Numpy. Na próxima parte, veremos o pacote PANDAS que é construído sobre o pacote Numpy. Esse pacote facilita muito a manipulação e análise de dados em Python, além de oferecer estruturas de dados e operações para tabelas numéricas e séries temporais.

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>