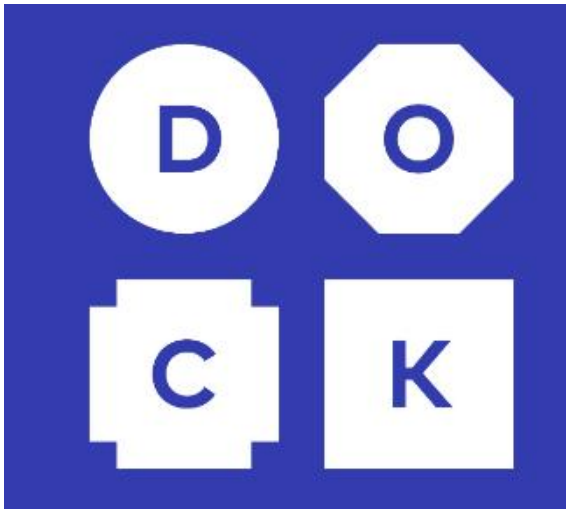
Curso Introdutório de Python



Prefácio

O objetivo deste curso é introduzir os conceitos básicos de programação para pessoas sem experiência em desenvolvimento ou iniciantes que não conheçam a linguagem Python.

O recomendado é cada participante ter acesso a um computador durante o curso para fazer os exercícios. O único modo de aprender programação é programando.

A duração estimada para este curso é de cerca de 36 horas, mas esse tempo pode variar dependendo do tamanho da turma e da disponibilidade de café.

Participe e interaja com o professor, tire suas dúvidas, venha se divertir neste curso, grandes poderes, grandes responsabilidades, já diria o Tio Ben.

Introdução

Python é uma linguagem de programação. Isso significa basicamente duas coisas:

1. existem regras que determinam como as palavras são dispostas, já que é uma linguagem;

2. o texto descreve instruções para o computador realizar tarefas.

Ou seja, podemos escrever um documento - que chamamos de código fonte - em Python para o computador ler e realizar nossos desejos e tarefas. Python tem algumas características interessantes:

• é interpretada, ou seja, o interpretador do Python executa o código fonte diretamente, traduzindo cada trecho para instruções de máquina;

• é de alto nível, ou seja, o interpretador se vira com detalhes técnicos do computador. Assim, desenvolver um código é mais simples do que em linguagens de baixo nível, nas quais o programador deve se preocupar com detalhes da máquina;

• é de propósito geral, ou seja, podemos usar Python para desenvolver programas em diversas áreas. Ao contrário de linguagens de domínio específico, que são especializadas e atendem somente a uma aplicação específica;

• tem tipos dinâmicos, ou seja, o interpretador faz a magia de descobrir o que é cada variável.

Por essas e várias outras características, Python se torna uma linguagem simples, bela, legível e amigável. É uma linguagem utilizada por diversas empresas, como Wikipedia, Google, Yahoo!, CERN, NASA, Facebook, Amazon, Instagram, Spotify.

O desenvolvimento de Python começou no final da década de 1980, por Guido van Rossum. Ele decidiu usar esse nome porque estava lendo um roteiro de Monty Python, um grupo de comédia inglês da década de 1970. A documentação oficial do Python contém muitas referências aos filmes e personagens desse grupo.

1.2 Exemplos

Vamos ver alguns exemplos sobre o uso de Python no mundo real.

1.2.1 BitTorrent

O protocolo Torrent é muito utilizado para transferir quantidades grandes de dados para diversos computadores. O

primeiro programa a implementar esse protocolo foi desenvolvido inteiramente em Python, pela BitTorrent, Inc.

1.2.2 Django

Django é um conjunto de pacotes para desenvolvimento web. E é baseado em Python :)

Um objetivo de Django é desenvolver facilmente websites complexos e que lidam com bancos de dados grandões.

Alguns sites desenvolvidos em Django: Instagram, The Washington Times, Disqus, Mozilla, National Geographic.

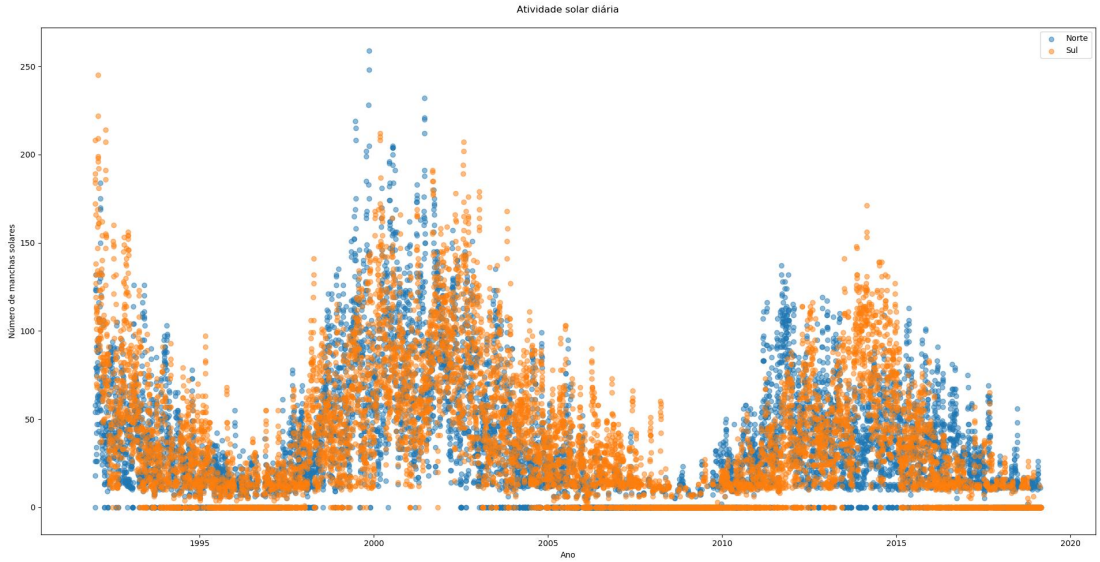
1.2.3 Dropbox

O popular serviço de armazenamento de dados em Nuvem Dropbox tem diversas partes da infraestrutura feita em Python. O aplicativo para computadores é feito em Python e grande parte da infra estrutura dos servidores deles também é.

1.2.4 Estudo sobre erupções solares

Não somente a indústria utiliza Python, muitos pesquisadores utilizam em diversas áreas científicas.

É possível de modo bem simples estudar as erupções solares desde 1992 até hoje. O Observatório Real da Bélgica tem um banco de dados sobre o número de manchas solares, e disponibilizam online para estudos. Veja como é a imagem para visualizar a atividade solar desde 01/01/1992 em cada parte (norte e sul) do Sol:



1.2.5 Física de Partículas

O premio Nobel de 2013 em Física foi para os cientistas que estudaram como as partículas elementares adquirem massa, conhecido como Mecanismo de Higgs. Uma nova partícula foi descoberta em 2012: o Bóson de Higgs.

1.2.6 The Sims 4

O jogo The Sims 4 tem partes feitas em Python 3 . Isso permite o desenvolvimento de mods para o jogo em Python \o/

Hello World

É muito comum, ao apresentar uma nova linguagem, começar com um exemplo simples que mostra na tela as palavras Hello World. Para não perder o costume:

```
>>> print("Hello, World!")
```

Função print()

print() é uma função nativa do Python. Basta colocar algo dentro dos parênteses que o Python se encarrega de fazer a magia de escrever na tela :)

Erros comuns

Usar a letra P maiúscula ao invés de minúscula:

```
>>> Print("Hello, World!")
Traceback (most recent call last):
...
NameError: name 'Print' is not defined
```

Esquecer de abrir e fechar aspas no texto que é passado para a função print():

```
>>> print(Hello, World!)
Traceback (most recent call last):
...
SyntaxError: invalid syntax
```

Esquecer de abrir ou fechar as aspas:

```
>>> print("Hello, World!)
Traceback (most recent call last):
...
SyntaxError: EOL while scanning string literal
```

Começar com aspas simples e terminar com aspas duplas ou vice-versa:

```
>>> print('Hello, World!")
Traceback (most recent call last):
...
SyntaxError: EOL while scanning string literal

Usar espaço ou tabulação (tab) antes do print():
>>> print('Hello, World!')
Traceback (most recent call last):
...
```

```
IndentationError: unexpected indent
>>>
print('Hello, World!')
Traceback (most recent call last):
...
IndentationError: unexpected indent
```

Mas, e se eu precisar usar aspas dentro do texto a ser mostrado na tela? Bem, Caso queira imprimir aspas duplas, envolva tudo com aspas simples e use aspas duplas na parte desejada:

```
>>> print('Python é legal! Mas não o "legal" como dizem pra outras coisas')
```

Caso deseje imprimir aspas simples, faça o contrário (envolva com aspas duplas e use aspas simples onde necessário):

```
>>> print("Python é legal! Mas não o 'legal' como dizem pra outras coisas")
```

E como faz para imprimir um texto em várias linhas? Bom, para isso precisamos lembrar de um caractere especial, a quebra de linha: \n. Esse \n é um caractere especial que significa aqui acaba a linha, o que vier depois deve ficar na linha de baixo. Por exemplo:

```
>>> print('Olha esse textão sobre aspas simples e dúplas.\nIsso aqui é aspas duplas:
‹ → "\nIsso aqui é aspas simples: \''
Olha esse textão sobre aspas simples e dúplas.
Isso aqui é aspas duplas: "
Isso aqui é aspas simples: '
```

Python como calculadora

A linguagem Python possui operadores que utilizam símbolos especiais para representar operações de cálculos, assim como na matemática:

• Soma (+) - `>>> 2 + 3`

Para utilizar números decimais, use o ponto no lugar de vírgula:

```
>>> 3.2 + 2.7
```

• Subtração (–)

```
>>> 6 - 4
```

```
>>> 7 - 8
```

• Multiplicação (*)

```
>>> 7 * 8
56
>>> 2 * 2 * 2
8
```

• Divisão (/)

```
>>> 100 / 20
5.0
>>> 10 / 3
3.3333333333333335
```

E se fizermos uma divisão por zero?

```
>>> 2 / 0
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Como não existe um resultado para a divisão pelo número zero, o Python interrompe a execução do programa (no caso a divisão) e mostra o erro que aconteceu, ou seja, «ZeroDivisionError: divison by zero».

• Divisão inteira (//)

```
>>> 10 // 3
3
>>> 666 // 137
4
>>> 666 / 137
4.861313868613139
```

• Resto da divisão (%)

```
>>> 10 % 2
0
>>> 10 % 3
1
>>> 666 % 137
118
```

Agora que aprendemos os operadores aritméticos básicos podemos seguir adiante. Como podemos calcular 2^{10} ? O jeito mais óbvio seria multiplicar o número dois dez vezes:

```
>>> 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2
1024
```

Porém, isso não é muito prático, pois há um operador específico para isso, chamado de potenciação/exponenciação:

```
**
>>> 2 ** 10
1024
>>> 10 ** 3
1000
>>> (10 ** 800 + 9 ** 1000) * 233
407254001651377825050774086265365912933271559572398924650169906751889900030955189004916347478470698888
```

E a raiz quadrada?

Lembrando que $\sqrt{x} = x^{1/2}$, então podemos calcular a raiz quadrada do seguinte modo:

```
>>> 4 ** 0.5
2.0
```

Mas a maneira recomendada para fazer isso é usar a função sqrt() da biblioteca math:

```
>>> import math
>>> math.sqrt(16)
4.0
```

Na primeira linha do exemplo

√ importamos, da biblioteca padrão do Python, o módulo math e então usamos a sua

função sqrt para calcular 16

E se precisarmos utilizar o número π?

```
>>> math.pi
3.141592653589793
```

Não esqueça que é preciso ter executado import math antes de usar as funções e constantes dessa biblioteca.

Expressões Numéricas

Agora que já aprendemos diversos operadores, podemos combiná-los e resolver problemas mais complexos:

```
>>> 3 + 4 * 2
11
>>> 7 + 3 * 6 - 4 ** 2
9
>>> (3 + 4) * 2
14
>>> (8 / 4) ** (5 - 2)
8.0
```

Quando mais de um operador aparece em uma expressão, a ordem de avaliação depende das regras de precedência.

O Python segue as mesmas regras de precedência da matemática. O acrônimo PEMDAS ajuda a lembrar essa ordem:

1. Parênteses

2. Exponenciação

3. Multiplicação e Divisão (mesma precedência)

4. Adição e Subtração (mesma precedência)

Notação Científica

Notação científica em Python usa a letra e como sendo a potência de 10:

```
>>> 10e6
10000000.0
>>> 1e6
1000000.0
>>> 1e-5
1e-05
```

Também pode ser usada a letra E maiúscula:

```
>>> 1E6
1000000.0
```

Pontos Flutuantes

Uma consideração importante sobre pontos flutuantes (números decimais). Por exemplo:

```
>>> 0.1
0.1
```

É importante perceber que este valor, em um sentido real na máquina, não é exatamente 1/10. Está arredondando a exibição do valor real da máquina.

```
>>> format(0.1, '.50f')
'0.10000000000000005551115123125782702118158340454101'
```

Veja que somente após a 18a casa que há diferença. Isso é mais dígitos do que a maioria das pessoas acham úteis, então o Python mantém o número de dígitos gerenciáveis exibindo um valor arredondado. Este fato se torna aparente assim que você tenta fazer aritmética com esses valores

```
>>> 0.1 + 0.2
0.30000000000000004
>>> 0.7 - 0.2
0.49999999999999994
```

Note que isso é da mesma natureza do ponto flutuante binário, não é um bug no Python e muito menos um bug no seu código. Você verá o mesmo tipo de coisa em todos os idiomas que suportam a aritmética de ponto flutuante de seu hardware (embora alguns idiomas possam não exibir a diferença por padrão ou em todos os modos de saída).

Os erros de representação referem-se ao fato de que a maioria das frações decimais não podem ser representadas exatamente como frações binárias (base 2). Essa é a principal razão pela qual o Python (ou Perl, C, C++, Java, Fortran e muitos outros) geralmente não exibe o número decimal exato que é esperado.

O valor de 1/10 não é exatamente representável como uma fração binária. Quase todas as máquinas atualmente (considerando após novembro de 2000) usam aritmética de ponto flutuante IEEE-754 29 , e quase todas as plataformas mapeiam pontos flutuantes do Python para a «dupla precisão» IEEE-754, que contêm 53 bits de precisão. Portanto, na entrada, o computador se esforça para converter 0.1 na fração mais próxima possível da forma J/2 ** N, onde J é um inteiro contendo exatamente 53 bits.

Sobre Comentários

Caso precise explicar alguma coisa feita no código, é possível escrever um texto (que não será executado), que ajuda a

entender ou lembrar o que foi feito. Esse texto é chamado de comentário, e para escrever um basta utilizar o caractere

#. Exemplo:

```
>>> 3 + 4 # será lido apenas o cálculo, do # para frente o interpretador do Python irá ignorar!
7
>>> # Aqui vai um código só com comentários! Posso falar o que quiser que não será interpretado, lalala, la-le-li-lo-lu. A job we hate to buy things we don't need.
```

Comparações

Os operadores de comparação em Python são:

Operação	Significado
<	menor que
<=	menor igual que
>	maior que
>=	maior igual que
==	igual
!=	diferente

```
>>> 2 < 10
True
>>> 2 > 11
False
>>> 10 > 10
False
>>> 10 >= 10
True
>>> 42 == 24
False
>>> 666 != 137
```

```
True
>>> 8**2 == 60 + 4
True
>>> 100 != 99 + 3
True
```

Variáveis

Variável é um nome que se refere a um valor.

Atribuição

Atribuição é o processo de criar uma nova variável e dar um novo valor a ela. Alguns exemplos de atribuições:

```
>>> numero = 11
>>> numero
11
>>> frase = "Me dá um copo d'água"
>>> frase
"Me dá um copo d'água"
>>> pi = 3.141592
>>> pi
3.141592
```

No exemplo anterior realizamos três atribuições. No primeiro atribuímos um número inteiro à variável de nome numero; no segundo uma frase à variável frase; no último um número de ponto flutuante à pi.

Nomes de Variáveis

Bons programadores escolhem nomes significativos para as suas variáveis - eles documentam o propósito da variável.

Nomes de variáveis podem ter o tamanho que você achar necessário e podem conter tanto letras como números, porém não podem começar com números. É possível usar letras maiúsculas, porém a convenção é utilizar somente letras minúsculas para nomes de variáveis.

```
>>> crieiumavariavelcomnomegiganteeestoucompreguiçadeescrevertudodenovo =
10
>>> crieiumavariavelcomnomegiganteeestoucompreguiçadeescrevertudodenovo #
use TAB para autocompletar =D
10
```

Tentar dar um nome ilegal a uma variável ocasionará erro de sintaxe:

```
>>> 123voa = 10
Traceback (most recent call last):
...
123voa = 10
^
SyntaxError: invalid syntax
>>> ol@ = "oi"
Traceback (most recent call last):
...
ol@ = "oi"
^
SyntaxError: invalid syntax
>>> def = 2.7
Traceback (most recent call last):
...
def = 2.7
^
SyntaxError: invalid syntax
```

123voa é ilegal pois começa com um número. ol@ é ilegal pois contém um caractere inválido (@), mas o que há de errado com def?

A questão é que def é uma palavra-chave da linguagem. O Python possui diversas palavras que são utilizadas na estrutura dos programas, por isso não podem ser utilizadas como nomes de variáveis.

Outro ponto importante: não é possível acessar variáveis que ainda não foram definidas:

```
>>> nao_definida
Traceback (most recent call last):
...
NameError: name 'nao_definida' is not defined
```

Tentar acessar uma variável sem definí-la anteriormente ocasiona em um «erro de nome».

Também podemos atribuir expressões a uma variável:

```
>>>x = 3 * 5 - 2
>>>x
13
>>>y = 3 * x + 10
>>>y
49
>>>z = x + y
>>>z
62
```

```
>>> n = 10
>>> n + 2 # 10 + 2
12
>>> 9 - n # 9 - 10
-1
```

É importante lembrar que para mudar o valor de uma variável é preciso utilizar a atribuição. Nos dois exemplos anteriores não atribuímos as expressões à n, portanto seu valor continuou o mesmo.

Vamos alterar o valor de n:

```
>>> n
10
>>>n = n + 2
>>>n
12
>>>9 - n
-3
```

Outra forma de somar na variável:

```
>>> num = 4
>>> num += 3
>>> num
7
```

Também funciona com multiplicação:

```
>>> x = 2
>>> x *= 3
>>> x
6
```

Atribuição múltipla

Uma funcionalidade interessante do Python é que ele permite atribuição múltipla. Isso é muito útil para trocar o valor de duas variáveis:

```
>>> a = 1
>>> b = 200
```

Para fazer essa troca em outras linguagens é necessário utilizar uma variável auxiliar para não perdemos um dos valores que queremos trocar. Vamos começar da maneira mais simples:

```
>>> a = b # perdemos o valor de a
>>> a
```

```
200
>>> b = a # como perdemos o valor de a, b vai continuar com seu valor original
de 200
>>> b
200
```

A troca é bem sucedida se usamos uma variável auxiliar:

```
>>> a = 1
>>> b = 200
>>> print(a, b)
1 200
>>>aux = a
>>>a = b
>>>b = aux
>>>print(a, b)
200 1
```

Porém, como o Python permite atribuição múltipla, podemos resolver esse problema de uma forma muito mais simples:

```
>>> a = 1
>>> b = 200
>>> print(a, b)
1 200
>>> a, b = b, a
>>> print(a, b)
200 1
```

A atribuição múltipla também pode ser utilizada para simplificar a atribuição de variáveis, por exemplo:

```
>>> a, b = 1, 200
>>> print(a, b)
1 200
>>> a, b, c, d = 1, 2, 3, 4
>>> print(a, b, c, d)
1 2 3 4
>>> a, b, c, d = d, c, b, a
>>> print(a, b, c, d)
4 3 2 1
```

Tipos de objetos

Criamos muitas variáveis até agora. Você lembra o tipo de cada uma? Para saber o tipo de um objeto ou variável, usamos a função type():

```
>>> x = 1
>>> type(x)
```

```
<class 'int'>
>>> y = 2.3
>>> type(y)
<class 'float'>
>>> type('Python')
<class 'str'>
>>> type(True)
<class 'bool'>
```

Python vem com alguns tipos básicos de objetos, dentre eles:

• bool: verdadeiro ou falso.

• int: números inteiros.

• float: números reais.

• complex: números complexos.

• str: strings (textos).

• list: listas. Estudaremos em breve o que são.

• dict: dicionários. Estudaremos em breve o que são.

Buscando ajuda rapidamente

Está com dúvida em alguma coisa? Use a função help() e depois digite o que você busca.

```
>>> help()
Welcome to Python 3.6's help utility!
If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/3.6/tutorial/.
Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules. To quit this help utility and
return to the interpreter, just type "quit".
To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics". Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".
help>
You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)". Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
```

E para buscar ajuda em uma coisa específica?

```
>>> help(len)
Help on built-in function len in module builtins:
len(obj, /)
Return the number of items in a container.
```

Para sair do ambiente de ajuda, pressione a tecla q.

A documentação oficial do Python contém toda a referência sobre a linguagem, detalhes sobre cada função e alguns exemplos.

Strings (sequência de caracteres)

Strings são tipos que armazenam uma sequência de caracteres:

```
>>> "Texto bonito"
'Texto bonito'
>>> "Texto com acentos de cedilhas: hoje é dia de caça!"
'Texto com acentos de cedilhas: hoje é dia de caça!'
```

As strings aceitam aspas simples também:

```
>>> nome = 'Silvio Santos'
>>> nome
'Silvio Santos'
```

Nota: Strings aceitam os dois tipos de aspas, desde que seja consistente. Se começou com uma, termine com aquela!

```
>>> cor_da_caneta = "azul brilhante'
File "<stdin>", line 1
cor_da_caneta = "azul brilhante'
^
SyntaxError: EOL while scanning string literal
Também é possível fazer algumas operações com strings:
>>> nome * 3
'Silvio SantosSilvio SantosSilvio Santos'
>>> nome * 3.14
Traceback (most recent call last):
...
TypeError: can't multiply sequence by non-int of type 'float'
>>> canto1 = 'vem aí, '
>>> canto2 = 'lá '
>>> nome + ' ' + canto1 + canto2 * 6 + '!!'
'Silvio Santos vem aí, lá lá lá lá lá lá !!'
```

Para strings em várias linhas, utilize 3 aspas:

```
>>> string_grande = '''Aqui consigo inserir um textão com várias linhas, posso
‹ → iniciar em uma...
... e posso continuar em outra
... e em outra
... e mais uma
... e acabou.'''
>>> string_grande
'Aqui consigo inserir um textão com várias linhas, posso iniciar em uma...\ne
posso
‹ → continuar em outra\ne em outra\ne mais uma\ne acabou.'
>>> print(string_grande)
```

Caso queira um texto que dentro tem aspas, como Me dá um copo d'água, é necessário utilizar aspas duplas para formar a string:

```
>>> agua = "Me dá um copo d'água"
>>> agua
"Me dá um copo d'água"
```

E também é possível utilizar aspas simples, duplas e triplas ao mesmo tempo! Olha só:

```
>>> todas_as_aspas = """Essa é uma string que tem:
... - aspas 'simples'
... - aspas "duplas"
... - aspas '''triplas'''
... Legal né?"""
>>> todas_as_aspas
'Essa é uma string que tem:\n- aspas \'simples\'\n- aspas "duplas"\n- aspas
\'\'\
‹ → 'triplas\'\'\'\nLegal né?'
>>> print(todas_as_aspas)
Essa é uma string que tem:
- aspas 'simples'
- aspas "duplas"
- aspas '''triplas'''
Legal né?
```

Tamanho

A função embutida len() nos permite, entre outras coisas, saber o tamanho de uma string:

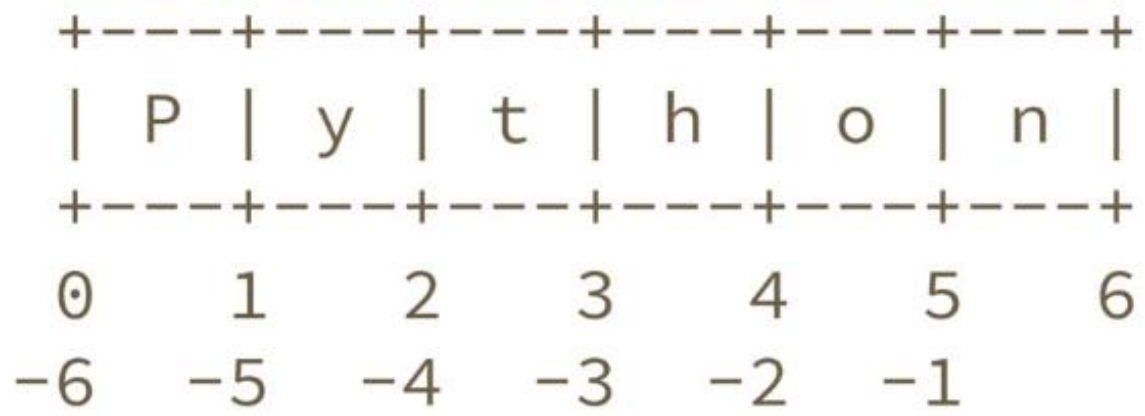
```
>>> len('Abracadabra')
11
>>> palavras = 'Faz um pull request lá'
>>> len(palavras)
22
```

Assim, vemos que a palavra Abracadabra tem 11 letras.

Índices

Como visto anteriormente, o método len() pode ser utilizado para obter o tamanho de estruturas, sejam elas strings, listas, etc. Esse tamanho representa a quantidade de elementos na estrutura.

Para obter somente um caractere de dentro dessas estruturas, deve-se utilizar o acesso por índices, no qual o índice entre colchetes [ ] representa a posição do elemento que se deseja acessar.




```
>>> palavra = 'Python'
>>> palavra[0] # primeira
'P'
>>> palavra[5] # última
'n'
```

Índices negativos correspondem à percorrer a estrutura (string, lista, . . .) na ordem reversa:

```
>>> palavra[-1] # última também
'n'
>>> palavra[-3] # terceira de tras pra frente
'h'
```

Fatias

Se, ao invés de obter apenas um elemento de uma estrutura (string, lista, . . .), deseja-se obter múltiplos elementos, deve-se utilizar slicing (fatiamento). No lugar de colocar o índice do elemento entre chaves, deve-se colocar o índice do primeiro elemento, dois pontos (:) e o próximo índice do último elemento desejado, tudo entre colchetes. Por exemplo:

```
>>> frase = "Aprender Python é muito divertido!"
>>> frase[0]
'A'
>>> frase[5]
'd'
>>> frase[0:5] # do zero até o antes do 5
'Apren'
>>> frase[:] # tudo!
'Aprender Python é muito divertido!'
>>> frase
'Aprender Python é muito divertido!'
>>> frase[6:] # Se omitido o segundo índice significa 'obter até o final'
'er Python é muito divertido!'
>>> frase[:6] # se omitido o primeiro indice, significa 'obter desde o começo'
'Aprend'
>>> frase[2:-3] # funciona com números negativos também
'render Python é muito diverti'
>>> frase[0:-5]
'Aprender Python é muito diver'
>>> frase[0:-6]
'Aprender Python é muito dive'
>>> frase[0:-7]
'Aprender Python é muito div'
>>> frase[2:-2]
'render Python é muito divertid'
```

É possível controlar o passo que a fatia usa. Para isso, coloca-se mais um dois pontos (:) depois do segundo índice e o tamanho do passo:

```
>>> frase[::1] # do começo, até o fim, de 1 em 1. Ou seja, tudo do jeito que
ja era,
‹ → não faz diferença nenhuma.
'Aprender Python é muito divertido!'
>>> frase[::2] # do começo, até o fim, de 2 em 2
'Arne yhnémiodvrio
>>> frase[2:-2:2] # Do terceiro, até o ante penúltimo, de 2 em dois
'rne yhnémiodvri'
```

Resumindo: para fazer uma fatia de nossa string, precisamos saber de onde começa, até onde vai e o tamanho do passo.

```
fatiável[começo : fim : passo]
```

Nota: As fatias incluem o índice do primeiro elemento e não incluem o elemento do índice final. Por isso que frase[0:-1] perde o último elemento.

Caso o final da fatia seja antes do começo, obtemos um resultado vazio:

```
>>> frase[15:2]
''
```

E se quisermos uma fatia fora da string?

```
>>> frase[123:345]
''
```

Mas e se o final da fatia for mais para frente que o tamanho da string? Não tem problemas, o Python vai até o onde der:

```
>>> frase[8:123456789]
' Python é muito divertido!'
>>> frase[8:]
' Python é muito divertido!'
>>> frase[123456789]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Tamanhos negativos de passo também funcionam. Passos positivos significam para frente e passos negativos significam para trás:

```
>>> "Python"[::-1]
'nohtyP'
```

Quando usamos passos negativos, a fatia começa no fim e termina no começo e é percorrida ao contrário. Ou seja, invertemos a ordem. Mas tome cuidado:

```
>>> "Python"[2:6]
'thon'
>>> "Python"[2:6:-1]
''
>>> "Python"[6:2]
''
>>> "Python"[6:2:-1]
'noh'
```

No caso de "Python"[6:2], o começo é depois do fim. Por isso a string fica vazia.

No caso de "Python"[2:6:-1], o começo é o índice 6, o fim é o índice 2, percorrida ao contrário. Ou seja, temos uma string vazia ao contrário, que continua vazia.

Quando fazemos "Python"[6:2:-1], o começo é o índice 2, o fim é o índice 6, percorrida ao contrário. Lembre que o índice final nunca é incluído. Ou seja, temos a string hon a ser invertida. O que resulta em noh.

Formatação de strings

A formatação de string nos permite criar frases dinâmicas, utilizando valores de quaisquer variáveis desejadas. Por exemplo:

```
>>> nome = input('Digite seu nome ')
Digite seu nome Silvio Santos
>>> nome
'Silvio Santos'
>>> frase = 'Olá, {}'.format(nome)
>>> frase
'Olá, Silvio Santos'
```

Vale lembrar que as chaves {} só são trocadas pelo valor após a chamada do método str.format():

```
>>> string_a_ser_formatada = '{} me formate!'
>>> string_a_ser_formatada
'{} me formate!'
>>> string_a_ser_formatada.format("Não") # também podemos passar valores diretamente
‹ → para formatação, apesar de ser desncessário
'Não me formate!'
```

A string a ser formatada não é alterada nesse processo, já que não foi feita nenhuma atribuição:

```
>>> string_a_ser_formatada
'{} me formate!'
```

É possível formatar uma quantidade arbitrária de valores:

```
>>> '{} x {} = {}'.format(7, 6, 7 * 6)
'7 x 6 = 42'
>>> palavra = 'Python'
>>> numero = 10
>>> booleano = False
>>> '{} é {}. E as outras linguagens? {}'.format(palavra, numero, booleano)
'Python é 10. E as outras linguagens? False'
```

Além disso, também é possível usar nomes para identificar quais valores serão substituídos:

```
>>> '{a}, {a}, {a}. {b}, {b}, {b}'.format(a='oi', b='tchau')
'oi, oi, oi. tchau, tchau, tchau'
```

Alternativa ao .format()

Uma maneira mais recente de formatar strings foi introduzida a partir da versão 3.6 do Python: PEP 498 – Literal String Interpolation, carinhosamente conhecida como fstrings e funciona da seguinte forma:

```
>>> nome = 'Silvio'
>>> f'Olá, {name}.'
'Olá, Silvio.'
```

É também possível fazer operações:

```
>>> f'4654 * 321 é {4654 * 321}'
'4654 * 321 é 1493934'
```

Separação de Strings

Se tivermos a frase Sílvio Santos vem aí, oleoleolá! e quisermos separar cada palavra, como fazer?

Pode-se usar o fatiamento:

```
>>> frase = "Sílvio Santos vem aí, oleoleolá!"
>>> frase[:6]
'Sílvio'
>>> frase[7:13]
'Santos'
>>> frase[14:17]
'vem'
>>> frase[18:21]
'aí,'
>>> frase[22:]
'oleoleolá!'
```

Mas também podemos usar a função split():

```
>>> frase.split()
['Sílvio', 'Santos', 'vem', 'aí,', 'oleoleolá!']
```

Nota: É possível transformar uma string em número, dado que seja um número:

```
>>> numero = int("2")
>>> numero
2
```

Nota: A volta também é possível:

```
>>> numero_string = str(1900)
>>> numero_string
```

```
'1900'
>>> type(numero_string)
<class 'str'>
```

Lendo valores do teclado

Em Python também é possível ler do teclado as informações digitadas pelo usuário. E isso é feito por meio da função embutida input() da seguinte forma:

```
>>> valor_lido = input("digite um valor: ")
```

digite um valor: 10

```
>>> type(valor_lido)
<class 'str'>
# deve-se notar que o valor lido é SEMPRE do tipo string
```

A função input() «termina» de ser executada quando pressionamos enter.

Nota: O valor lido é sempre do tipo string.

Mas, como realizar operações com os valores lidos?

```
>>> valor_lido + 10 # para trabalhar com esse valor, é preciso converter para
o tipo
‹ → correto
Traceback (most recent call last):
...
TypeError: must be str, not int
```

Para poder fazer isso pode-se usar os operadores int() e float(), que converte o valor lido para o tipo de dado esperado:

```
>>> valor_lido = int(input("digite um valor inteiro: "))
```

digite um valor inteiro: 10

```
>>> type(valor_lido)
<class 'int'>
>>> valor_lido + 10
20

>>> valor_lido = float(input("digite um valor decimal: "))
digite um valor decimal: 1.5
>>> valor_lido - 1
0.5
```

Tudo o que for digitado no teclado, até pressionar a tecla enter, será capturado pela função input(). Isso significa que podemos ler palavras separadas por um espaço, ou seja, uma frase inteira:

```
>>> frase = input()
Rosas são vermelhas, violetas são azuis, girassóis são legais.
>>> frase
'Rosas são vermelhas, violetas são azuis, girassóis são legais.'
```

Listas

Listas são estruturas de dados capazes de armazenar múltiplos elementos.

Declaração

Para a criação de uma lista, basta colocar os elementos separados por vírgulas dentro de colchetes [ ], como no exemplo abaixo:

```
>>> nomes_frutas = ["maçã", "banana", "abacaxi"]
>>> nomes_frutas
['maçã', 'banana', 'abacaxi']
>>> numeros = [2, 13, 17, 47]
>>> numeros
[2, 13, 17, 47]
```

A lista pode conter elementos de tipos diferentes:

```
>>> ['lorem ipsum', 150, 1.3, [-1, -2]]
['lorem ipsum', 150, 1.3, [-1, -2]]
>>> vazia = []
>>> vazia
[]
```

Índices

Assim como nas strings, é possível acessar separadamente cada item de uma lista a partir de seu índice:

```
>>> lista = [100, 200, 300, 400, 500]
>>> lista[0] # os índices sempre começam em 0
100
>>> lista[2]
300
>>> lista[4] # último elemento
500
>>> lista[-1] # outra maneira de acessar o último elemento
```

```
500
```

Conforme visto anteriormente, ao utilizar um índice negativo os elementos são acessados de trás pra frente, a partir do final da lista:

```
>>> lista[-2] # penúltimo elemento
400
>>> lista[-3] # terceiro
300
>>> lista[-4] # segundo
200
>>> lista[-5] # primeiro
100
```

Ou pode-se acessar através de slices (fatias), como nas strings (Página 49):

```
>>> lista[2:4] # da posição 2 até a 4 (não inclusa)
[300, 400]
>>> lista[:3] # até a posição 3 (não incluso)
[100, 200, 300]
>>> lista[2:] # da posição 2 até o final
[300, 400, 500]
>>> lista[:] # do começo até o final
[100, 200, 300, 400, 500]
```

Tentar acessar uma posição inválida de uma lista causa um erro:

```
>>> lista[10]
Traceback (most recent call last):
...
IndexError: list index out of range
>>> lista[-10]
Traceback (most recent call last):
,,,
IndexError: list index out of range
```

Podemos avaliar se os elementos estão na lista com a palavra in:

```
>>> lista_estranha = ['duas palavras', 42, True, ['batman', 'robin'], -0.84,
'hipófise']
>>> 42 in lista_estranha
True
>>> 'duas palavras' in lista_estranha
True
>>> 'dominó' in lista_estranha
False
>>> 'batman' in lista_estranha[3] # note que o elemento com índice 3 também
é uma
True
```

É possível obter o tamanho da lista utilizando o método len():

```
>>> len(lista)
5
>>> len(lista_estranha)
6
>>> len(lista_estranha[3])
2
```

Removendo itens da lista

Devido à lista ser uma estrutura mutável, é possível remover seus elementos utilizando o comando del:

```
>>> lista_estranha
['duas palavras', 42, True, ['batman', 'robin'], -0.84, 'hipófise']
>>> del lista_estranha[2]
>>> lista_estranha
['duas palavras', 42, ['batman', 'robin'], -0.84, 'hipófise']
>>> del lista_estranha[-1] # Remove o último elemento da lista
>>> lista_estranha
['duas palavras', 42, ['batman', 'robin'], -0.84]
```

Trabalhando com listas

O operador + concatena listas:

```
>>>a = [1, 2, 3]
>>>b = [4, 5, 6]
>>>c = a + b
>>>c
[1,2, 3, 4, 5, 6]
```

O operador * repete a lista dado um número de vezes:

```
>>> [0] * 3
[0, 0, 0]
>>> [1, 2, 3] * 2
[1, 2, 3, 1, 2, 3]
```

O método append() adiciona um elemento ao final da lista:

```
>>> lista = ['a', 'b', 'c']
>>> lista
['a', 'b', 'c']
>>> lista.append('e')
>>> lista
['a', 'b', 'c', 'e']
```

Temos também o insert(), que insere um elemento na posição especificada e move os demais elementos para direita:

```
>>> lista.insert(3, 'd')
>>> lista
['a', 'b', 'c', 'd', 'e']
```

Aviso: Cuidado com lista.insert(-1, algo)! Nesse caso, inserimos algo na posição -1 e o elemento que estava previamente na posição -1 é movido para a direita:

```
>>> lista.insert(-1, 'ç')
>>> lista
['a', 'b', 'c', 'd', 'ç', 'e']
```

Use append() caso queira algo adicionado ao final da lista.

extend() recebe uma lista como argumento e adiciona todos seus elementos a outra:

```
>>> lista1 = ['a', 'b', 'c']
>>> lista2 = ['d', 'e']
>>> lista1
['a', 'b', 'c']
>>> lista2
['d', 'e']
>>> lista1.extend(lista2)
>>> lista1
['a', 'b', 'c', 'd', 'e']
```

lista2 não é modificado:

```
>>> lista2
['d', 'e']
```

O método sort() ordena os elementos da lista em ordem ascendente:

```
>>> lista_desordenada = ['b', 'z', 'k', 'a', 'h']
>>> lista_desordenada
['b', 'z', 'k', 'a', 'h']
>>> lista_desordenada.sort()
>>> lista_desordenada # Agora está ordenada!
['a', 'b', 'h', 'k', 'z']
```

Para fazer uma cópia de uma lista, devemos usar o método copy():

```
>>> lista1 = ['a', 'b', 'c']
>>> lista2 = lista1.copy()
```

```
>>> lista1
['a', 'b', 'c']
>>> lista2
['a', 'b', 'c']
>>> lista2.append('d')
>>> lista1
['a', 'b', 'c']
>>> lista2
['a', 'b', 'c', 'd']
```

Se não usarmos o copy(), acontece algo bem estranho:

```
>>> lista1 = ['a', 'b', 'c']
>>> lista2 = lista1
>>> lista1
['a', 'b', 'c']
>>> lista2
['a', 'b', 'c']
>>> lista2.append('d')
>>> lista1
['a', 'b', 'c', 'd']
>>> lista2
['a', 'b', 'c', 'd']
```

Tudo o que for feito com lista2 nesse exemplo também altera lista1 e vice-versa.

Função range()

Aprendemos a adicionar itens a uma lista mas, e se fosse necessário produzir uma lista com os números de 1 até 200?

```
>>> lista_grande = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13] # ???
>>> lista_grande
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
```

Em Python existe a função embutida range(), com ela é possível produzir uma lista extensa de uma maneira bem

simples:

```
>>> print(list(range(1, 200)))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38,
39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56,
57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74,
75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92,
```

```
93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108,
109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122,
123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136,
137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150,
151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164,
165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178,
179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192,
193, 194, 195, 196, 197, 198, 199]
```

Além disso, o range() também oferece algumas coisas interessantes. Por exemplo, imprimir os números espaçados de 5 em 5, entre 0 e 30:

```
>>> print(list(range(0, 30, 5)))
[0, 5, 10, 15, 20, 25]
```

Mas, como na maior parte das vezes apenas queremos uma lista começando em 0 e indo até o número desejado, a

função range() também funciona da seguinte maneira:

```
>>> print(list(range(10)))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Nota: O intervalo do range() é aberto, ou seja, quando passamos o valor 10, ele vai até o 9 (n - 1). Caso deseje criar a lista até o 10 de fato, deve-se passar o valor 11.

Mas por que precisamos transformar o range() em list? O que acontece se não fizermos isso?

```
>>> print(range(200))
range(0, 200)
```

Mas o que é que essa função retorna?

```
type(range(200))
<class 'range'>
```

AHA! A função range() retorna algo do tipo range, por isso precisamos transformar em uma lista para vermos todos os números no print()!

Dicionários

Dicionário é uma coleção de itens (chamados chaves) e seus respectivos significados (chamados de valores):

{chave: valor}

Cada chave do dicionário deve ser única! Ao contrário de listas, dicionários, não podem ter chaves repetidas.

Nota: As chaves devem ser únicas.

Declaração

Declaramos um dicionário colocando entre colchetes {} cada chave e o seu respectivo valor, da seguinte forma:

```
>>> telefones = {"ana": 123456, "yudi": 40028922, "julia": 4124492}
>>> telefones
{'ana': 123456, 'yudi': 40028922, 'julia': 4124492}
```

No caso acima, a chave "ana", por exemplo, está relacionada ao valor 123456. Cada par chave-valor é separado por uma vírgula , .

Função dict()

A função dict() constrói um dicionário. Existem algumas formas de usá-la:

• Com uma lista de listas:

```
>>>lista1 = ["brigadeiro", "leite condesado, achocolatado"]
>>>lista2 = ["omelete", "ovos, azeite, condimentos a gosto"]
>>>lista3 = ["ovo frito", "ovo, óleo, condimentos a gosto"]
>>>lista_receitas = [lista1, lista2, lista3]
>>> print(lista_receitas)
[['brigadeiro', 'leite condesado, achocolatado'], ['omelete', 'ovos,
azeite,condimentos a gosto'], ['ovo frito', 'ovo, óleo, condimentos a
gosto']]
>>> receitas = dict(lista_receitas)
>>> print(receitas)
{'brigadeiro': 'leite condesado, achocolatado', 'omelete': 'ovos,
azeite,condimentos a gosto', 'ovo frito': 'ovo, óleo, condimentos a
gosto'}
```

• Atribuindo os valores diretamente:

```
>>> constantes = dict(pi=3.14, e=2.7, alpha=1/137)
>>> print(constantes)
{'pi': 3.14, 'e': 2.7, 'alpha': 0.0072992700729927005}
```

Neste caso, o nome das chaves deve ser um identificador válido. As mesmas regras de nomes de variáveis se aplicam.

• Usando as chaves { }:

```
>>> numerinhos = dict({"um": 1, "dois": 2, "três": 3})
>>> print(numerinhos)
{'um': 1, 'dois': 2, 'três': 3}
```

E nesse caso se não houvesse a função dict(), o resultado seria exatamente o mesmo. . .

Chaves

Acessamos um determinado valor do dicionário através de sua chave:

```
>>> capitais = {"SP": "São Paulo", "AC": "Rio Branco", "TO": "Palmas",
"RJ": "Rio de Janeiro", "SE": "Aracaju", "MG": "Belo Horizonte"}
>>> capitais["MG"]
'Belo Horizonte'
```

Até o momento, usamos apenas strings, mas podemos colocar todo tipo de coisa dentro dos dicionários, incluindo listas e até mesmo outros dicionários:

```
>>> numeros = {"primos": [2, 3, 5], "pares": [0, 2, 4], "ímpares": [1,
3, 5]}
>>> numeros["ímpares"]
[1, 3, 5]
```

Mesmo que os pares chave-valor estejam organizados na ordem que foram colocados, não podemos acessá-los por índices como faríamos em listas:

```
>>> numeros[2]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 2
```

O mesmo erro ocorre se tentarmos colocar uma chave que não pertence ao dicionário:

```
>>> numeros["negativos"]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'negativos'
```

Assim como os valores não precisam ser do tipo string, o mesmo vale para as chaves:

```
>>> numeros_por_extenso = {2: "dois", 1: "um", 3: "três", 0: "zero"}
>>> numeros_por_extenso[0]
'zero'
>>> numeros_por_extenso[2]
'dois'
```

Nota: Listas e outros dicionários não podem ser usados como chaves por serem de tipos mutáveis.

Adicionando e removendo elementos

Podemos alterar o valor relacionado a uma chave da seguinte forma:

```
>>> pessoa = {"nome": "Cleiton", "idade": 34, "família": {"mãe": "Maria", "pai": "Enzo"}}
>>> pessoa["idade"]
34
>>> pessoa["idade"] = 35
>>> pessoa["idade"]
35
```

Para adicionar um elemento novo à um dicionário, podemos simplesmente fazer o seguinte:

```
>>>meses = {1: "Janeiro", 2: "Fevereiro", 3: "Março"}
>>>meses[4] = "Abril"
meses
{1:"Janeiro", 2: "Fevereiro", 3: "Março", 4: "Abril"}
```

Aqui nos referimos a uma chave que não está no dicionário e associamos um valor a ela. Desta forma, adicionando esse conjunto chave-valor ao dicionário.

Removemos um conjunto chave-elemento de um dicionário com o comando `` `del` ``:

```
>>>meses
{1:"Janeiro", 2: "Fevereiro", 3: "Março", 4: "Abril"}
>>>del(meses[4])
meses
{1:"Janeiro", 2: "Fevereiro", 3: "Março"}
```

Para apagar todos os elementos de um dicionário, usamos o método clear:

```
>>> lixo = {"plástico": ["garrafa", "copinho", "canudo"], "papel": ["folha amassada", "guardanapo"], "orgânico": ["batata", "resto do bandeco", "casca de banana"]}
>>> lixo
{"plástico": ["garrafa", "copinho", "canudo"], "papel": ["folha amassada", "guardanapo"], "organico": ["batata", "resto do bandeco", "casca de banana"]}
>>> lixo.clear()
>>> lixo
{}
```

Função list()

A função list() recebe um conjunto de objetos e retorna uma lista. Ao passar um dicionário, ela retorna uma lista contendo todas as suas chaves:

```
>>> faculdades = {"FAFUPI": "Faculdade de Funilaria e Pintura",
"IAA":"Instituto de Assuntos Aleatórios", "EESC": "Escola de Engenharia
do Seo Carlos", "FMP": "Faculdade Manobrista de Postit", "INDI":
"Instituto de Nomes De Institutos"}
>>> list(faculdades)
[FAFUPI, IAA, EESC, FMP, INDI]
```

Função len()

A função len() retorna o número de elementos («tamanho») do objeto passado para ela. No caso de uma lista, fala quantos elementos há. No caso de dicionários, retorna o número de chaves contidas nele:

```
>>> faculdades
{"FAFUPI": "Faculdade de Funilaria e Pintura", "IAA":"Instituto de
Assuntos Aleatórios", "EESC": "Escola de Engenharia
do Seo Carlos", "FMP": "Faculdade Manobrista de Postit", "INDI":
"Instituto de Nomes De Institutos"}
>>> len(faculdades)
5
```

Você pode contar o número de elementos na lista gerada pela função list() para conferir:

```
>>> len(list(faculdades))
5
```

Método get()

O método get(chave, valor) pode ser usado para retornar o valor associado à respectiva chave! O segundo parâmetro <valor> é opcional e indica o que será retornado caso a chave desejada não esteja no dicionário:

```
>>> faculdades.get("EESC")
'Escola de Engenharia do Seo Carlos'
Dá para ver que ele é muito parecido com fazer assim:
>>> faculdades["EESC"]
'Escola de Engenharia do Seo Carlos'
Mas ao colocarmos uma chave que não está no dicionário:
>>> faculdades.get("Poli", "Não tem!")
'Não tem!'
```

```
>>> faculdades["Poli"]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'Poli'
```

Alguns métodos

```
• O método items() pode ser comparado com o inverso da função dict():
>>> pessoa = {"nome": "Laurito", "RA": "2422211334", "curso":
"mixologista"}
>>> pessoa.items()
dict_items([('curso', 'mixologista'), ('nome','Laurito'), ('RA',
2422211334)])
```

Usando a função list() nesse resultado, obtemos:

```
>>> pessoa.items()
dict_items([('curso', 'mixologista'), ('nome', 'Laurito'), ('RA',
2422211334)])
>>> itens = list(pessoa.items())
>>> itens
[('curso', 'mixologista'), ('nome', 'Laurito'), ('RA', 2422211334)]
```

Experimente usar a função ``dict()`` na lista ``itens``!

• O método values() nos retorna os valores do dicionário:

```
>>> pessoa.values()
dict_values(['mixologista', 'Laurito', 2422211334])
>>> valores = list(pessoa.values())
>>> valores
['mixologista', 'Laurito', 2422211334]
```

• O método keys() nos retorna as chaves do dicionário:

```
>>> pessoa.keys()
dict_keys(['curso', 'nome', 'RA'])
>>> chaves = list(pessoa.keys())
>>> chaves
['curso', 'nome', 'RA']
```

Repare que nesse último obtemos o mesmo que se tivéssemos usado a função list():

```
>>> list(pessoa)
['curso', 'nome', 'RA']
```

Ordem dos elementos

Dicionários não tem sequência dos seus elementos. As listas têm. Dicionários mapeiam um valor a uma chave. Veja este exemplo:

```
>>> numerinhos = dict({"um": 1, "dois": 2, "três": 3})
>>> numeritos = {"três": 3, "dois": 2, "um": 1}
>>> numerinhos == numeritos
True
>>> numeritos
{'três': 3, 'dois': 2, 'um': 1}
>>> numerinhos
{'um': 1, 'dois': 2, 'três': 3}
```

Vemos que numerinhos e numeritos têm as mesmas chaves com os mesmos valores e por isso são iguais. Mas quando imprimimos cada um, a ordem que aparece é a que os itens foram inseridos.

Está no dicionário?

Podemos checar se uma chave está ou não em um dicionário utilizando o comando in. Voltando para o dicionário que contem as faculdades:

```
>>> faculdades
{"FAFUPI": "Faculdade de Funilaria e Pintura", "IAA":"Instituto de
Assuntos Aleatórios", "EESC": "Escola de Engenharia
do Seo Carlos", "FMP": "Faculdade Manobrista de Postit", "INDI":
"Instituto de Nomes De Institutos"}
>>> "FAFUPI" in faculdades
True
>>> "ESALQ" in faculdades
False
```

E checamos se uma chave não está no dicionário com o comando not in:

```
>>> faculdades
{"FAFUPI": "Faculdade de Funilaria e Pintura", "IAA":"Instituto de
Assuntos Aleatórios", "EESC": "Escola de Engenharia
do Seo Carlos", "FMP": "Faculdade Manobrista de Postit", "INDI":
"Instituto de Nomes De Institutos"}
>>> "IFSC" not in faculdades
False
>>> "ESALQ" not in faculdades
True
```

Condicionais

O tipo de dado boleano (bool) refere-se a uma unidade lógica sobre a qual podemos realizar operações, particularmente úteis para o controle de fluxo de um programa.

A unidade boleana assume apenas 2 valores: Verdadeiro (True) e Falso (False).

Nota: Essa estrutura binária é a forma com a qual o computador opera (0 e 1).

```
>>> True
True
>>> type(False)
<class 'bool'>
```

Qualquer expressão lógica retornará um valor boleano:

```
>>> 2 < 3
True
>>> 2 == 5
False
```

Os operadores lógicos utilizados em programação são:

- `>  : maior a, por exemplo 5 > 3`
- `<  : menor a`
- `>= : maior ou igual a`
- `<= : menor ou igual a`
- `== : igual a`
- `!= : diferente de`

Para realizar operações com expressões lógicas, existem:

and (e): opera segundo a seguinte tabela:

Valor1	Valor2	Resultado
Verdadeiro	Verdadeiro	Verdadeiro
Verdadeiro	Falso	Falso
Falso	Verdadeiro	Falso
Falso	Falso	Falso

• or (ou):

Valor1	Valor2	Resultado
Verdadeiro	Verdadeiro	Verdadeiro
Verdadeiro	Falso	Verdadeiro
Falso	Verdadeiro	Verdadeiro
Falso	Falso	Falso

• not (não):

Valor1	Resultado
Verdadeiro	Falso
Falso	Falso

```
>>> 10 > 3 and 2 == 4
False
>>> 10 > 3 or 2 == 4
True
>>> not not not 1 == 1
False
```

Assim como os operadores aritméticos, os operadores boleanos também possuem uma ordem de prioridade:

• not tem maior prioridade que and que tem maior prioridade que or:

```
>>> not False and True or False
True
>>> 10 > 3 or 2 == 4
True
>>> not not not 1 == 1
False
```

Assim como os operadores aritméticos, os operadores boleanos também possuem uma ordem de prioridade:

• not tem maior prioridade que and que tem maior prioridade que or:

```
>>> not False and True or False
True
```

Estruturas de controle

As estruturas de controle servem para decidir quais blocos de código serão executados.

Exemplo:

Se estiver nublado:
    Levarei guarda-chuva
Senão:
    Não levarei

Nota: Na linguagem Python, a indentação (espaço dado antes de uma linha) é utilizada para demarcar os blocos de código, e são obrigatórios quando se usa estruturas de controle.

```
>>> a = 7
>>> if a > 3:
...   print("estou no if")
... else:
...   print("cai no else")
...
estou no if

>>> valor_entrada = 10
>>> if valor_entrada == 1:
...     print("a entrada era 1")
... elif valor_entrada == 2:
...     print("a entrada era 2")
... elif valor_entrada == 3:
...     print("a entrada era 3")
... elif valor_entrada == 4:
...     print("a entrada era 4")
... else:
...     print("o valor de entrada não era esperado em nenhum if")
...
o valor de entrada não era esperado em nenhum if
```

Estruturas de repetição

As estruturas de repetição são utilizadas quando queremos que um bloco de código seja executado várias vezes.

Em Python existem duas formas de criar uma estrutura de repetição:

• O for é usado quando se quer iterar sobre um bloco de código um número determinado de vezes.

• O while é usando quado queremos que o bloco de código seja repetido até que uma condição seja satisfeita.

Ou seja, é necessário que uma expressão boleana dada seja verdadeira. Assim que ela se tornar falsa, o while para.

Nota: Na linguagem Python a indentação é obrigatória. assim como nas estruturas de controle, as estruturas de repetição também precisam.

```
>>> # Aqui repetimos o print 3 vezes
>>> for n in range(0, 3):
...     print(n)
...
0
1
2
>>> # Aqui iniciamos o n em 0, e repetimos o print até que seu valor
seja maior ou igual a 3
>>> n = 0
>>> while n < 3:
...     print(n)
...     n += 1
...
0
1
2
```

O loop for em Python itera sobre os itens de um conjunto, sendo assim, o range(0, 3) precisa ser um conjunto de elementos. E na verdade ele é:

```
>>> list(range(0, 3))
[0, 1, 2]
```

Para iterar sobre uma lista:

```
>>> lista = [1, 2, 3, 4, 10]
>>> for numero in lista:
...     print(numero ** 2)
...
1
4
9
16
100
```

Isso se aplica para strings também:

```
>>> # Para cada letra na palavra, imprimir a letra
>>> palavra = "casa"
>>> for letra in palavra:
...     print(letra)
...
c
a
s
a
```

Em dicionários podemos fazer assim:

```
>>> gatinhos = {"Português": "gato", "Inglês": "cat", "Francês": "chat",
"Finlandês": "Kissa"}
>>> for chave, valor in gatinhos.items():
...     print(chave, "->", valor)
...
Português -> gato
Inglês -> cat
Francês -> chat
Finlandês -> Kissa
```

Para auxiliar as estruturas de repetição, existem dois comandos:

- break: É usado para sair de um loop, não importando o estado em que se encontra.

- continue: Funciona de maneira parecida com a do break, porém no lugar de encerrar o loop, ele faz com que todo o código que esteja abaixo (porém ainda dentro do loop) seja ignorado e avança para a próxima iteração.

Veja a seguir um exemplo de um código que ilustra o uso desses comandos. Note que há uma string de documentação no começo que explica a funcionalidade.

```
"""
Esse código deve rodar até que a palavra "sair" seja digitada.
* Caso uma palavra com 2 ou menos caracteres seja digitada, um aviso
deve ser exibido e o loop será executado do início (devido ao
continue), pedindo uma nova palavra ao usuário.
* Caso qualquer outra palavra diferente de "sair" seja digitada, um
aviso deve ser exibido.
* Por fim, caso a palavra seja "sair", uma mensagem deve ser exibida
e o loop deve ser encerrado (break).
"""
>>> while True:
...     string_digitada = input("Digite uma palavra: ")
...     if string_digitada.lower() == "sair":
...         print("Fim!")
...         break
...     if len(string_digitada) < 2:
...         print("String muito pequena")
...         continue
...     print("Tente digitar \"sair\"")
...
Digite uma palavra: oi
Tente digitar "sair"
Digite uma palavra: ?
String muito pequena
Digite uma palavra: sair
Fim!
```

Funções

Função é uma sequência de instruções que executa uma operação de computação. Ao definir uma função, você especifica o nome e a sequência de instruções. Depois, pode utilizar ("chamar") a função pelo nome.

A ideia é similar às funções matemáticas! Mas funções em uma linguagem de programação não realizam necessariamente apenas cálculos.

Vimos o type(), um exemplo de função:

```
>>> type(23)
<class 'int'>
>>> type('textinho')
<class 'str'>
```

Definindo funções

Se quisermos uma função que ainda não existe em Python, temos que definí-la.

Criando uma função simples:

```
def NOME_DA_FUNÇÃO(LISTA DE PARÂMETROS):
    COMANDOS
```

Aviso: Coloque os dois pontos após definir a função!

Nota: Faça a indentação nas linhas abaixo da definição da função!

```
>>> def soma():
...     print(1 + 1)
...
>>> soma()
2
>>> def soma():
...     return 1 + 1
...
>>> soma()
2
```

Qual a diferença entre utilizar print() e return() aqui em cima?!?

```
>>> def imprime_letra():
...     print("If you didn't care what happened to me. And I didn't care for
you")
...
>>> imprime_letra()
If you didn't care what happened to me. And I didn't care for you

>>> type(imprime_letra)
<class 'function'>

>>> def repete_letra():
...     imprime_letra()
...     imprime_letra()
...
>>> repete_letra()
If you didn't care what happened to me. And I didn't care for you
If you didn't care what happened to me. And I didn't care for you
```

Funções com argumentos

Queremos somar 3 com um número qualquer que insiro na função. Bora lá:

```
>>> def soma_valor(x):
...     return 3 + x
...
>>> soma_valor(5)
8
>>> z = soma_valor(10)
>>> z
13
```

Que sem graça! Quero somar dois números quaisquer!

```
>>> def soma_dois_numeros(x, y):
...     return x + y
...
>>> soma_dois_numeros(7, 4)
11
```

Tenho dificuldade com a tabuada do 7! Ajude-me!

```
>>> def tabuada_do_7():
... for i in range(11):
...    print (7 * i)
...
```

```
>>> tabuada_do_7()
0
7
14
21
28
35
42
49
56
63
70
```

Mais tá legal isso! Quero a tabuada do 1 ao 10 agora! Bora!

```
>>> def tabuadas():
...     for i in range(1, 11):
...         for j in range(1, 11):
...             print("{} * {} = {}".format(i, j, i * j))
...
>>> tabuadas()
1 * 1 = 1       3 * 1 = 1       5 * 1 = 1       7 * 1 = 1       9 * 1 = 1
1 * 2 = 2       3 * 2 = 2       5 * 2 = 2       7 * 2 = 2       9 * 2 = 2
1 * 3 = 3       3 * 3 = 3       5 * 3 = 3       7 * 3 = 3       9 * 3 = 3
1 * 4 = 4       3 * 4 = 4       5 * 4 = 4       7 * 4 = 4       9 * 4 = 4
1 * 5 = 5       3 * 5 = 5       5 * 5 = 5       7 * 5 = 5       9 * 5 = 5
1 * 6 = 6       3 * 6 = 6       5 * 6 = 6       7 * 6 = 6       9 * 6 = 6
1 * 7 = 7       3 * 7 = 7       5 * 7 = 7       7 * 7 = 7       9 * 7 = 7
1 * 8 = 8       3 * 8 = 8       5 * 8 = 8       7 * 8 = 8       9 * 8 = 8
1 * 9 = 9       3 * 9 = 9       5 * 9 = 9       7 * 9 = 9       9 * 9 = 9
1 * 10 = 10     3 * 10 = 10     5 * 10 = 10     7 * 10 = 10     9 * 10 = 10
2 * 1 = 1       4 * 1 = 1       6 * 1 = 1       8 * 1 = 1       10 * 1 = 1
2 * 2 = 2       4 * 2 = 2       6 * 2 = 2       8 * 2 = 2       10 * 2 = 2
2 * 3 = 3       4 * 3 = 3       6 * 3 = 3       8 * 3 = 3       10 * 3 = 3
2 * 4 = 4       4 * 4 = 4       6 * 4 = 4       8 * 4 = 4       10 * 4 = 4
2 * 5 = 5       4 * 5 = 5       6 * 5 = 5       8 * 5 = 5       10 * 5 = 5
2 * 6 = 6       4 * 6 = 6       6 * 6 = 6       8 * 6 = 6       10 * 6 = 6
2 * 7 = 7       4 * 7 = 7       6 * 7 = 7       8 * 7 = 7       10 * 7 = 7
2 * 8 = 8       4 * 8 = 8       6 * 8 = 8       8 * 8 = 8       10 * 8 = 8
2 * 9 = 9       4 * 9 = 9       6 * 9 = 9       8 * 9 = 9       10 * 9 = 9
2 * 10 = 10     4 * 10 = 10     6 * 10 = 10     8 * 10 = 10     10 * 10 = 10
```