Aprendendo Python: Conceitos Básicos

Asimov Academy

ASIMOV

Conteúdo

01. Como Aprender Python?

Bem-vindos à apostila do curso “Aprendendo Python: Conceitos Básicos” da Asimov Academy! Nesta apostila, nosso foco é aprender a linguagem de programação **Python**.

Cada capítulo acompanha uma das aulas do curso e introduz um conceito novo. Dessa forma, iremos construindo nosso conhecimento de forma incremental. Ao longo da apostila, são abordados diversos conceitos que qualquer programador de Python utiliza diariamente ao escrever seus códigos.

Dicas para estudar Python

Reforçamos aqui o nosso principal mantra dentro da Asimov:

Pratique!

O consenso entre programadores é que só aprendemos a programar a partir da **prática!** Portanto, tente realmente digitar todos os exemplos demonstrados, e resolver os desafios antes de olhar as respostas. Volte para as aulas anteriores e revisem o conteúdo se necessário. Utilize também a nossa comunidade caso você esteja com dificuldades.

Dito tudo isso: lembre-se de que **parte do processo de aprendizado, especialmente para quem é autodidata, é conseguir encontrar as respostas das suas perguntas**. Os melhores programadores não são aqueles que decoraram todo o código, mas sim aqueles que conseguem encontrar a resposta de qualquer dúvida rapidamente, seja através do Google, ChatGPT, ou consultando o material das aulas.

Como se escreve código Python

Como de fato escrevemos código Python?

Tudo começa com um **código-fonte**, que é um arquivo com comandos na linguagem Python.

Existem programas muito utilizados para ler e escrever códigos. Estes programas são chamados coletivamente de IDEs (do inglês *Integrated Development Environment*, ou Ambiente de Desenvolvimento Integrado).

Uma IDE é um ambiente montado e configurado para nos ajudar a escrever código. As mais famosas IDEs para Python, atualmente, são VS Code e PyCharm.

Dito isso, a configuração inicial de uma IDE não é exatamente simples. Por isso, vamos utilizar outra IDE feita especialmente para o ensino de Python, chamada **Mu**. Dessa forma, conseguiremos focar em aprender código Python o mais rápido possível!

Preciso mesmo usar o Mu?

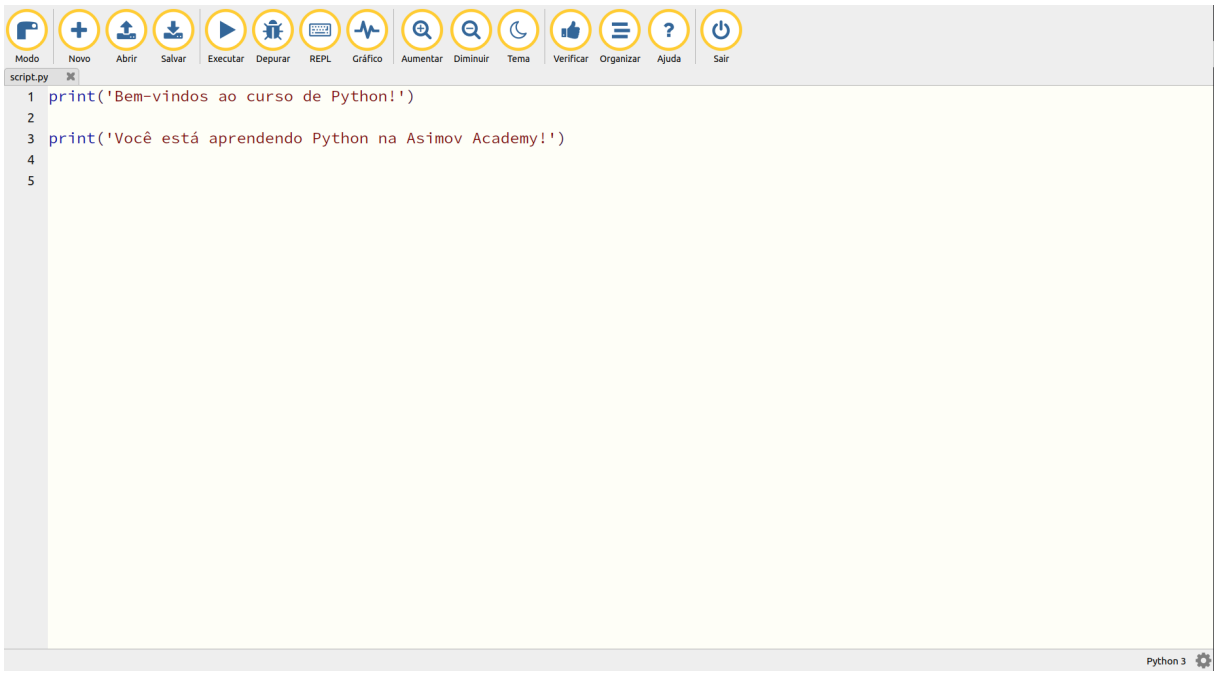
Caso você já saiba usar VS Code ou PyCharm, ou quer se desafiar e usar estes programas logo de cara, não tem problema. Os códigos e os exemplos vão funcionar da mesma forma.

A diferença é que IDEs avançadas possuem diversas funcionalidades que, para quem está começando, pode dificultar o aprendizado. Também existirão algumas diferenças no layout dos botões entre as IDEs avançadas e o Mu. Mas a opção é inteiramente sua.

Vamos começar então a programar em Python com o Mu!

02. Conhecendo a IDE Mu

O programa Mu é uma IDE voltada especificamente para o aprendizado de Python. Ela simplifica bastante a parte mais complicada de instalação e setup de Python, pois já inclui sua instalação própria de Python, embutida no programa.



Instalação

Baixe o Mu a partir do site: https://codewith.mu/. Há links para a versão para Windows, Mac e Linux.

Criando o primeiro script: `Hello World`

Quando estamos aprendendo linguagens de programação, a tradição é escrever o `Hello World` como o primeiro script. Este código simplesmente exibe o texto "Hello World!" no output.

Para escrevermos este código no Mu, escreva o conteúdo abaixo dentro da janela principal:

```
print('Hello World!')
```

Em seguida, clique em `Salvar` e salve o código como um arquivo Python (extensão `.py`) no seu computador. Nomeie o script de `hello_world.py`.

Aperte o botão `Executar` e veja o resultado em tela! No Mu, o código permanece ativo mesmo após execução, então clique em `Parar` para finalizar a execução.

A função `print()`

`print()` é uma **função** em Python. Quando executamos ("chamamos") esta função, um texto é exibido no output. Para chamarmos qualquer função, precisamos digitar seu nome, e em seguida abrir e fechar parênteses.

Algumas funções aceitam **argumentos**. No caso da função `print()`, ela simplesmente pega todos os argumentos entregues a ela e os exibe no output.

Valores que não forem "printados" não aparecem no output. Modifique o código para o código abaixo e execute novamente (o Mu salva as alterações automaticamente ao executar):

```
print('Hello World!')
"Este é o meu primeiro script"
print("Estou aprendendo Python!")
```

A segunda linha não aparece no output, porque não usamos a função `print()`!

Erros em Python

Modifique o código para o código abaixo e rode novamente:

```
print('Hello World!")
```

Note que colocamos aspas simples na esquerda, e aspas duplas na direita. Isso causa um **erro** em Python. No seu output deve ter algo como:

```
File [ ... ], line 1

print("Hello World!')
^
SyntaxError: EOL while scanning string literal
```

Sempre que você encontrar um erro em Python (também conhecido como `Exception` ou **Exceção**), não se assuste! Erros são extremamente comuns em programação. Lembre-se sempre de **ler a mensagem de erro** e tentar entender o que ela está tentando lhe dizer, para então modificar o seu código e solucioná-lo.

Em último caso, copie e cole a mensagem de erro no Google. As chances de alguém já ter encontrado o mesmo erro que você são muito altas, especialmente quando você está começando a aprender a programar. Sem brincadeiras: **saber ler mensagens de erro e procurar por respostas rapidamente no Google são algumas das principais habilidades que bons programadores desenvolvem com o tempo**.

Resumo

Escreva código e salve em um script para executá-lo no Mu.
Use a função `print()` para exibir algum texto ou valor no console.
Leia as mensagens de erro e busque por soluções na internet.

03. Script, console e debugger

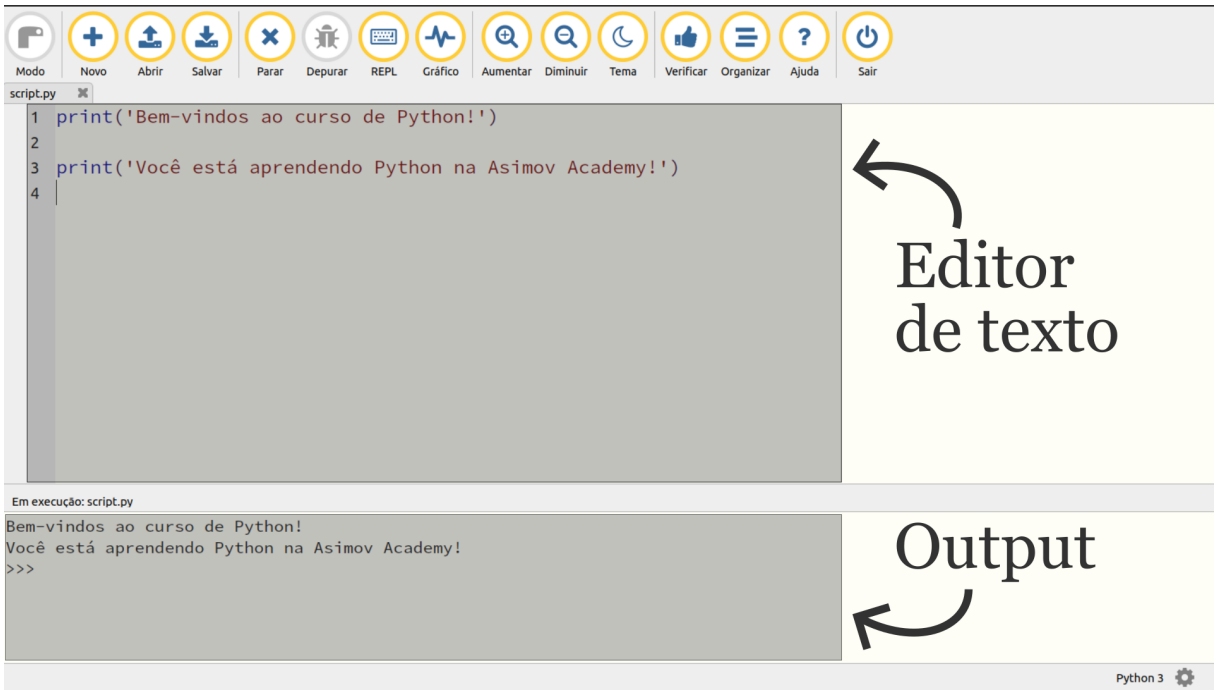
Podemos rodar código em Python de duas formas: estruturando uma série de comandos em um **script de Python**, ou escrevendo e executando cada linha de código de forma interativa, através de um **console de Python**.

Scripts

Scripts são usados para criar um arquivo contendo código Python. Arquivos Python possuem a extensão `.py`. Este arquivo fica salvo dentro de alguma pasta no seu computador, pronto para ser executado.

Programas capazes de editar um script de Python são chamados de **editores de texto**. O exemplo mais básico de editor de texto é o bloco de notas do Windows, porém há opções melhores para escrever código Python. A janela principal do Mu é um editor de texto.

Uma vez criado um script, é preciso executá-lo a partir de um **Interpretador de Python**. O Mu inclui um interpretador de Python na sua instalação. Portanto, basta abrir o script no programa e clicar no botão `Executar` na barra de ferramentas. O resultado do código aparecerá na seção de **output**.



Console

O console é um ambiente que permite a execução interativa de código. O resultado de cada linha digitada é exibido imediatamente abaixo do comando, sem que precisemos utilizar a função `print()`. Dessa forma, é possível testar seu código e inspecionar valores de forma dinâmica.

De forma geral, nada do que é escrito no console fica salvo no seu computador. Por isso, é importante salvar as linhas de interesse em algum script, para não perdê-las ao fechar o console. É possível acessar o **histórico de comandos** de um console com a tecla de "seta pra cima" do seu teclado, mas esta não é uma forma segura de armazenar código.

Dentro do Mu, podemos abrir/fechar o console através do botão REPL. Esta sigla vem do inglês `read-evaluate-print loop`, ou "loop de leitura-avaliação-exibição", o que descreve bem o comportamento de um console.

Qual utilizar?

Programadores experientes são proficientes no uso tanto de scripts quanto do console. Utilize o **console** para testar ideias e inspecionar valores. Em seguida, passe o código para um **script** organizado, de forma a salvá-lo no seu computador.

Nessa apostila, usaremos a seguinte notação para denotar um código a ser desenvolvido em um script, ou digitado em um console:

Código em script

Scripts serão indicados por diversos comandos um abaixo do outro, formando um script completo.

```
print('Olá Mundo!')
print('Vou aprender Python!')
```

Código no console

Exemplos de código que devem ser reproduzidos no console aparecem com os caracteres de prefixo >>> (este é o prompt padrão de um console de Python). O output esperado aparece na linha diretamente abaixo:

```
>>> 2 + 3
5

>>> 'Hello' + 'World'
'HelloWorld'
```

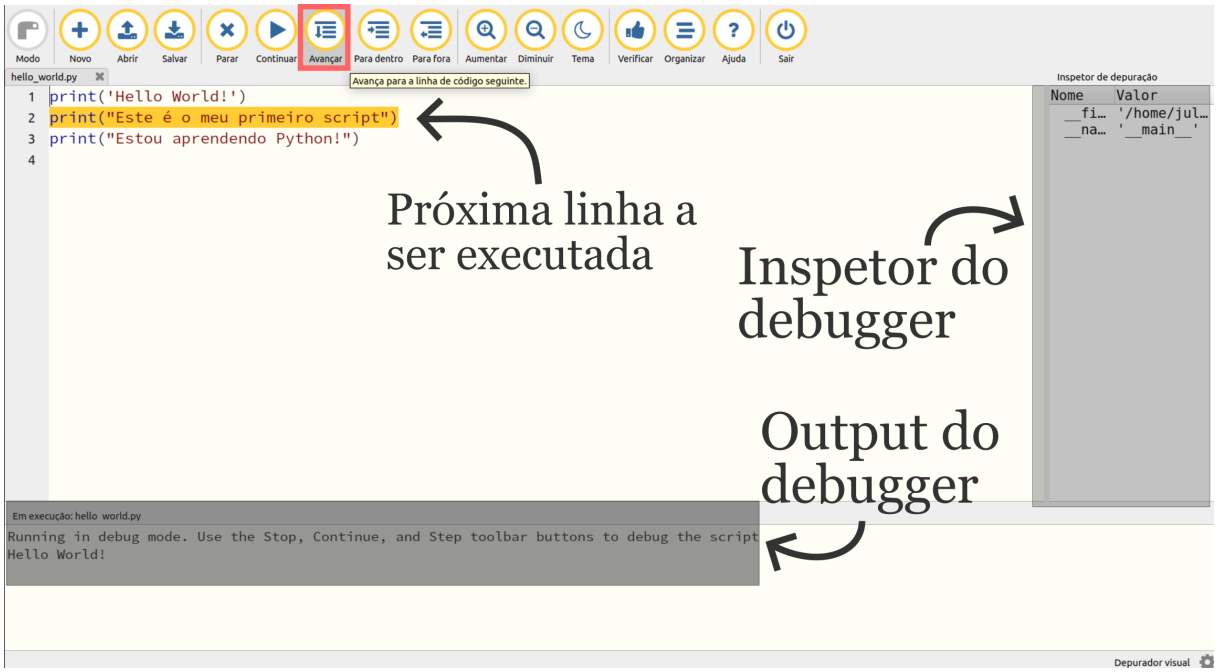
Lembre-se de **não incluir** o prompt >>> na hora de rodar o comando no seu console!

Depurador (debugger)

Um script de Python executa sempre linha a linha. Se usarmos o depurador do Mu (mais conhecido pela palavra em inglês *debugger*), veremos a execução de cada linha. Modifique o código para o código abaixo e clique em `Depurar`:

```
print('Hello World!')
print("Este é o meu primeiro script")
print("Estou aprendendo Python!")
```

No modo de depuração, a linha em laranja representa a próxima linha a ser executada. Se clicarmos no botão `Avançar` (destaque em vermelho na imagem abaixo), a linha em laranja é executada. Se a linha possuir uma chamada para a função `print()`, então o texto é exibido na janela de output. O código segue então para a próxima linha. Este processo se repete até que o código chegue ao fim.



O debugger possui esse nome justamente porque permite passarmos pelo código linha a linha. Isso nos ajuda a encontrar e solucionar erros (*bugs*) do nosso programa!

O debugger também possui uma janela chamada **inspetor**, que nos permite acompanhar o valor de variáveis durante a execução do programa. Vamos falar mais de variáveis mais pra frente!

Resumo

> Código dentro de um script fica salvo no seu computador, podendo ser executado novamente no futuro.
> O console é usado para testar ideias e trechos de código.
> O debugger é usado para acompanhar a execução do código e detectar erros.

04. Números em Python: `int` e `float`

Existem dois tipos de dados numéricos em Python: `int` e `float`.

`int`

Um `int` representa um número inteiro, como `1`, `50` e `-2`.

```
>>> 1 + 2
3
```

`float`

Um `float` representa um número inteiro, como `1.5`, `50.25`, ou `-2.86`.

```
>>> 1.5 + 3.4
4.9
```

A função `type`

A função `type` retorna o *tipo de dado* do objeto que é passado para ela.

Podemos conferir o tipo de dado de números dessa forma:

```
>>> type(1.5)
float

>>> type(1)
int

>>> type(1.0)
float
```

Note que, ao escrevermos `1.0`, representa um `float`, por mais que matematicamente seu valor seja "inteiro".

Matemática básica com Python

Podemos usar tanto ints quanto floats para fazer operações matemáticas:

```
>>> 2 + 2.5  # Soma
4.5

>>> 5 - 2.5  # Subtração
2.5
```

```
>>> 5 * 3  # Multiplicação
15

>>> 30 / 3  # Divisão
10.0

>>> 3 ** 2  # Exponenciação (potência)
9
```

Note que é possível misturar ints e floats sem nenhum problema.

Regras básicas de matemática

A ordem de operação é dada pelas regras básicas de aritmética. Se quisermos priorizar alguma operação, por exemplo, temos que usar parênteses:

```
>>> 2 * 4 + 6  # Primeiro multiplicação, depois soma
14

>>> 2 * (4 + 6)  # Primeiro soma, depois multiplicação
20
```

Além disso, as regras básicas da matemática continuam sendo verdadeiras. Por exemplo, é impossível dividir algum número por zero:

```
>>> 5 / 0
ZeroDivisionError: division by zero
```

Neste caso, o Python gerou um erro com a mensagem `ZeroDivisionError: division by zero`. O erro está evidente: não é possível dividir números por zero!

Resumo

Dois tipos de dados numéricos: `int` (inteiros) e `float` (números “quebrados”).
Função `type()` inspeciona o tipo de dado de um valor qualquer.
Operações matemáticas com `int` e `float`:

Operação	Sintaxe
Soma	`a + b`
Subtração	`a - b`
Multiplicação	`a * b`
Divisão	`a / b`
Exponenciação	`a ** b`

05. Texto em Python: `str`

Texto em Python é representado pelo tipo de dado *string* (comumente abreviado para as letras `str`). O nome representa a ideia de um "fio", "corda" ou sequência de caracteres que representam conjuntamente um bloco de texto.

Para criar um string, precisamos usar aspas simples (`'abc'`) ou duplas (`"abc"`). Não há diferença entre as formas, mas é preciso lembrar de abrir e fechar o string com o mesmo caractere:

```
>>> "Olá Mundo!"
'Olá Mundo!'

>>> 'Estou aprendendo Python. Python é uma linguagem de programação.'
'Estou aprendendo Python. Python é uma linguagem de programação.'
```

Texto e números

Um string *sempre* representa texto, ainda que contenha apenas números. Sendo assim, é impossível somar um `str` e um `int`, por exemplo:

```
>>> '50' + 50
TypeError: can only concatenate str (not "int") to str
```

Para realizarmos a operação acima, é preciso converter o texto em um número, usando a função `int()` ou `float()`:

```
>>> int('50') + 50
100

>>> float('50') + 50
100.0
```

Da mesma forma, se quisermos converter um valor numérico em texto, podemos usar a função `str()`:

```
>>> str(50)
'50'

>>> str(2.5)
'2.5'
```

Operações com strings

Apesar de representarem texto, strings aceitam alguns operadores. Para concatenar texto (isto é, "colar" strings diferentes um atrás do outro), usamos o operador de soma (+):

```
>>> "Hello" + " " + "World"
'Hello World'

>>> "50" + "10"
'5010'
```

Alguns detalhes:

- O Note que foi necessário adicionar um caractere de *espaço em branco* entre as palavras `"Hello"` e `"World"` para que elas não aparecessem grudadas no texto final. Espaço em branco também é um caractere como qualquer outro!
- Strings numéricos não são somados, mas sim concatenados (mais uma vez: para fazer a operação aritmética, é necessário converter os strings para números primeiro).

Strings também aceitam outros operadores, como multiplicação por um inteiro para repetir a palavra:

```
>>> "Python" * 4
"PythonPythonPythonPython"
```

Mas não aceitam *todos* os operadores. Por exemplo, não é possível fazer uma "subtração" de strings:

```
>>> 'abc' - 'c'
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

É o próprio *tipo de dado* (`str`, `int`, `float`) que determina quais operadores funcionam ou não para ele!

Tamanho de um string

A função `len()` retorna o "tamanho" de um objeto. No caso de um string, o tamanho é representado pelo número de caracteres (incluindo pontuação e espaços em branco):

```
>>> len('Python')
6

>>> len('Conferido o tamanho desse string...')
35
```

Funções e tipos de dados

Algumas funções só funcionam com certos tipos de dados. Por exemplo, para os criadores de Python, não faz sentido perguntar pelo "tamanho de um número":

```
>>> len(3)
TypeError: object of type 'int' has no len()
```

Já outras funções, como `print()`, funcionam com virtualmente qualquer tipo de dado.

Para sabermos exatamente o que uma função ou objeto aceitam, podemos usar outra função, `help()`, para ler sua documentação:

```
>>> help(print)
print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
# ... etc
```

Dito isso, na prática é muito mais comum procurarmos pela documentação de Python online (faça uma busca por "Python print function documentation", por exemplo), ou em sites contendo exemplos práticos.

Resumo

> `str`: texto em Python. Sempre entre aspas simples ou duplas.
> Aceita operações como "soma" de strings (concatenação) ou multiplicação por inteiro (repetição). Outras operações são proibidas (ex: somar string com número).
> Função `len()` retorna o tamanho de um objeto (número de caracteres, no caso de `str`).
> Função `help()` retorna documentação de um objeto ou tipo de dado.

06. Variáveis

Em muitas partes do código, será útil dar nomes para valores, para facilitar o entendimento e desenvolvimento do nosso próprio código. Por exemplo, se quisermos calcular a área A de um círculo, a fórmula é:

$$A = \pi r^2$$

Onde π é a constante pi, e r é o raio do círculo.

Se nosso círculo possuir raio 5, e aproximando pi com o valor `3.14`, podemos representar essa fórmula em Python como:

```
print(3.14 * 5 ** 2)
```

Este código funcionará sem problemas. Contudo, não é fácil entender exatamente o que ele faz. Mesmo para o criador do código, pode ser difícil lembrar o que os números representam depois de alguns dias sem voltar a este problema.

Podemos melhorar este código com o uso de **variáveis**. Variáveis são formas de darmos **nomes** a um certo **valor**.

O código abaixo, por exemplo, assinala o valor 4 para dentro da variável x e depois inspeciona a variável para obter seu valor:

```
>>> x = 4
>>> x
x
```

Reescrevendo o exemplo do círculo com variáveis, temos:

```
pi = 3.14
raio = 5
raio_ao_quadrado = raio ** 2

print(pi * raio_ao_quadrado)
```

A intenção do código fica muito mais evidente!

Regras para nome de variáveis

Apenas letras, números e underscore:

```
>>> x = 4  # OK
>>> meu_nome = "Juliano"  # OK
>>> meu   nome = "Juliano"  # Errado, contém espaço!
SyntaxError: invalid syntax
```

Python difere letras maiúsculas, minúsculas e acentos:

```
>>> a = 5

>>> a  # OK
5

>>> A  # Variável "A" maiúsculo não foi criada!
NameError: name 'A' is not defined

>>> á  # Variável "a" com acento não foi criada!
NameError: name 'á' is not defined
```

Não pode começar com número:

```
>>> var1 = 10  # OK
>>> 1var = 10  # Errado, começa com número!
SyntaxError: invalid syntax
```

Não pode ser uma palavra com significado especial em Python, como `if` e `for`:

```
>>> if = 2  # Errado, palavra especial!
SyntaxError: invalid syntax
```

Cuidado: sobrescrevendo nomes já existentes

Se uma variável for redefinida, apenas o último valor é mantido em memória:

```
>>> x = 2
>>> x = 10
>>> x
10
```

Cuidado: se uma variável sobrescrever uma função do Python, como `print()`, a função está "perdida" até o final da execução do código:

```
print("Olá, mundo!")

print = 2

print("Olá, mundo!")
TypeError: 'int' object is not callable
```

Se isso acontecer, é só rodar o script novamente - as variáveis são sempre zeradas quando o script reinicia.

Por outro lado, se eu quiser que alguma variável persista entre execuções diferentes, então vou precisar de **algum arquivo** no qual escrevê-la. Pode ser um arquivo de texto simples (`.txt`), um arquivo Excel, um banco de dados, não importa: precisa ficar registrado em algum lugar! Este curso não se aprofunda na escrita de arquivos, mas é importante saber que esta é a forma de "salvar" uma variável fora do código.

Resumo

Definir variáveis com operador = (exemplo: `var = 5`).
Regras para nome de variáveis:

- Apenas letras, números e underscore
- Não pode começar com números
- Não pode ser palavra reservada do Python

Cuidado para não sobrescrever outras variáveis ou funções!

07. Inserindo e formatando texto

A função `input()`

A função `input()` é usada para **pegar valores** inseridos pelo usuário.

Ela não precisa de nenhum argumento, mas ainda precisamos abrir e fechar os parênteses para simbolizar que queremos executá-la:

```
x = input()
print(x)
```

Na realidade, a função `input()` tem um argumento *opcional*. Ele representa um prompt explicativo, que fica imediatamente antes da posição onde o usuário insere o texto:

```
nome = input("Digite seu nome: ")
print(nome)
```

Note que o valor sempre é retornado como um string. Então se quisermos trabalhar com números, é preciso converter o tipo de dado:

```
num = input('Digite um número: ')
resultado = int(num) + 10

print(resultado)
```

Inserindo variáveis no texto com f-strings

Os f-strings são strings que facilitam a inserção de variáveis no texto.

Um string qualquer pode se tornar um f-string apenas adicionando o caractere `f` imediatamente antes do seu começo (à esquerda das aspas). As variáveis são passadas diretamente dentro de chaves `{ }` no meio do string.

Exemplo comparando uso de f-strings:

```
# Sem f-strings
nome = input('Qual o seu nome? ')
idade = input('Qual a sua idade? ')

n_letras = len(nome)
n_letras_str = str(n_letras)

idade_futuro = int(idade) + 5
idade_futuro_str = str(idade_futuro)

print('Olá, ' + nome + '!')
print('Seu nome tem ' + n_letras_str + ' letras.')
print('Daqui 5 anos, você terá ' + idade_futuro_str + ' anos.')
```

```
# Com f-strings
nome = input('Qual o seu nome? ')
idade = input('Qual a sua idade? ')

print(f'Olá, {nome}!')
print(f'Seu nome tem {len(nome)} letras.')
print(f'Daqui 5 anos, você terá {int(idade) + 5} anos.')
```

Como visto acima, quando usamos f-strings podemos passar até mesmo números ou outros tipos de dado dentro das chaves, que a conversão para string é feita automaticamente. É até possível realizar operações simples, como chamar a função `len()`, diretamente de um f-string!

Quebras de linha

Uma quebra de linha (também chamada de caractere **nova linha** em Python) é representado em Python pelo caractere `\n`. Ao ser printado, o caractere não é exibido em tela, e no seu lugar surge uma quebra de linha:

```
s = 'Primeira linha\nSegunda linha\nTerceira linha'
print(s)
```

O output do código acima é:

```
Primeira linha
Segunda linha
Terceira linha
```

Strings brutos

A sequência `\n` representa uma *sequência de controle*, isto é, uma combinação especial de caracteres com funcionalidade específica dentro de um string.

Se quisermos ignorar todas as sequências de controle de um string, e escrever literalmente os caracteres `\` e `n` um seguidos do outro, podemos usar um string bruto. De forma similar a um f-string, um string bruto é identificado pelo prefixo `r` (do inglês *raw*):

```
s = r'Primeira linha\nSegunda linha\nTerceira linha'
print(s)
```

Neste caso, o output é:

```
Primeira linha\nSegunda linha\nTerceira linha
```

Resumo

Função `input("prompt opcional")` para pegar valor do usuário (sempre lido como um `str`).
f-strings ajudam a inserir variáveis no texto. Exemplo:

```
nome = 'Juliano'
print(f'Olá, {nome}!')
```

Quebra de linha representado pelo caractere especial `"\n"`. Usar string bruto (com prefixo `r`) para ignorar caracteres especiais.

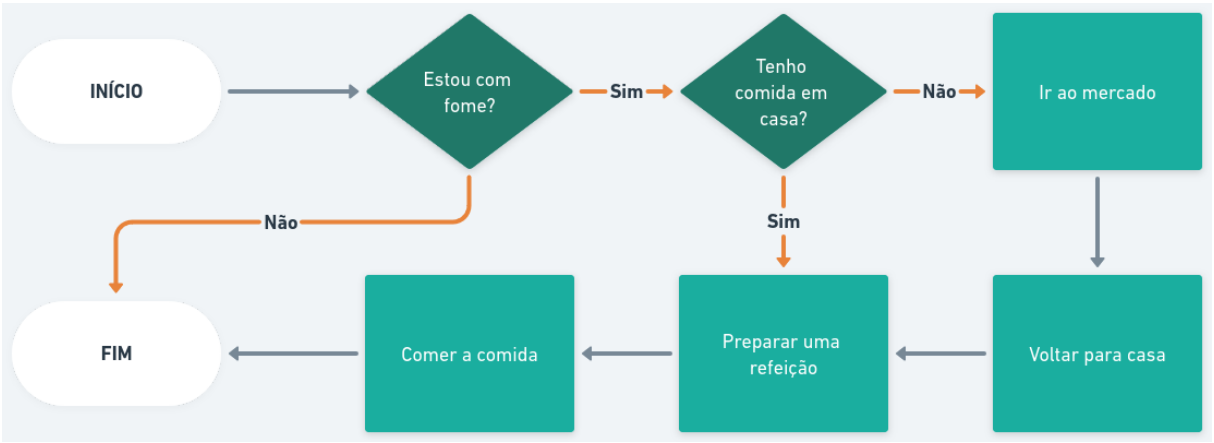
08. Controle de fluxo e operadores

Controle de fluxo na vida real

No nosso dia a dia, há diversos momentos em que nos deparamos com uma **estrutura de controle de fluxo**:

- Se isso for verdadeiro, então faço aquilo.
- Caso contrário, faço essa outra coisa.

Exemplo: algoritmo para decidir se eu devo comer uma comida:



Essa estrutura pode ser representada através de código Python!

Controle de fluxo em Python: `if` e `else`

O Python utiliza as palavras `if` e `else` para controle de fluxo, junto de algum tipo de **comparador**:

```
idade = int(input('Digite sua idade: '))

if idade < 18:
    print('Você é menor de idade')
    print('Você não pode dirigir um carro')
else:
    print('Você é maior de idade')
```

O código irá exibir valores diferentes, de acordo com a resposta que o usuário passar no `input()`!

- O bloco `if` executa **apenas se a condição for verdadeira**. Neste exemplo, isso significa ter menos de 18 anos.
- O bloco `else` executa **apenas se as condições anteriores forem falsas**. Neste exemplo, isso significa ter 18 anos ou mais. Podemos pensar no `else` como um "caso contrário": se nada for verdadeiro, então o código no `else` é executado.

Sintaxe de controle de fluxo

Note que há uma sintaxe específica nestes blocos de `if` e `else`:

- Uso de dois pontos (`:`) depois da comparação `if idade < 18`.
- O bloco imediatamente abaixo da comparação possui **indentação**, isto é, espaço em branco à esquerda. Isto não é meramente uma questão de "estilo" de código. **A indentação é essencial em Python**, pois indica onde começa e termina o bloco de código dentro da condicional.
- O tamanho padrão da indentação é de 4 espaços, mas Python aceita outros valores, desde que seja consistente.
- O bloco termina quando a indentação retorna ao valor original.

A palavra-chave `elif`

Para testar mais de duas condições, podemos usar `elif` (que é uma combinação de `else` e `if`):

```
idade = int(input('Digite sua idade: '))

if idade < 18:
    print('Você tem menos de 18 anos.')
elif idade == 18:
    print('Você tem exatamente 18 anos!')
else:
    print('Você tem mais de 18 anos.')
```

Se o `if` não passar, o código testa o `elif`. Se o `elif` também não passar, então o código testa o `else`.

Posso incluir mais de um `elif`. Na realidade posso incluir quantos `elif` eu quiser, de acordo com o que fizer sentido!

Operadores de comparação

São usados nas comparações dos blocos `if` e `elif`. Retornam sempre um valor `True` ou `False` (verdadeiro ou falso, respectivamente).

Note que `True`/`False` são valores próprios em Python, e pertencem ao um tipo de dado específico (booleano). São escritos com letra inicial maiúscula.

Já utilizamos dois operadores de comparação até aqui: menor que(<) e igual a(==). Aqui está a lista completa de comparadores:

- Igual a: ==
- Diferente de: !=

- Maior que: >
- Menor que: <
- Maior ou igual a: >=
- Menor ou igual a: <=

E alguns exemplos práticos:

```
>>> 4 == 4.0  # Igual a
True

>>> 4 != "4"  # Diferente de
True

>>> 5 > 10  # Maior que
False

>>> 5 < 10  # Menor que
True

>>> 10 >= 10  # Maior ou igual a
True

>>> 11 <= 10  # Menor ou igual a
False
```

Operadores booleanos

Usados para combinar 2 valores `True`/`False` de formas específicas:

- Operador `and`: retorna `True` apenas se ambos os valores forem `True`
- Operador `or`: retorna `True` apenas se pelo menos um dos valores for `True`
- Operador `not`: inverte o valor (`True` vira `False`, `False` vira `True`)

Exemplos:

```
>>> True and True
True

>>> True and False
False

>>> True or True
True

>>> True or False
True

>>> False or False
False
```

```
>>> not True
False

>>> not False
True
```

Recriando o controle de fluxo da imagem

Usando operadores de comparação:

```
print('--- INÍCIO ---')

resposta1 = input('Estou com fome? (Digite s para sim)')
if resposta1 == 's':

    resposta2 = input('Tenho comida em casa? (Digite s para sim)')
    if resposta2 != 's':

        print('Ir ao mercado')
        print('Voltar para casa')

    print('Preparar uma refeição')
    print('Comer a comida')

print('--- FIM ---')
```

Usando operadores booleanos:

```
print('--- INÍCIO ---')

estou_com_fome = input('Estou com fome? (Digite s para sim)') == 's'
tenho_comida = input('Tenho comida em casa? (Digite s para sim)') == 's'

if estou_com_fome and not tenho_comida:
    print('Ir ao mercado')
    print('Voltar para casa')

if estou_com_fome:
    print('Preparar uma refeição')
    print('Comer a comida')

print('--- FIM ---')
```

Resumo

Estrutura de controle de fluxo:

```
if condicao_01:
    # Código que roda quando condição 01 é verdadeira
elif condicao_02:
    # Código que roda quando condição 01 é falsa
    # e condição 02 é verdadeira
elif condicao_03:
    # ...
else:
    # Código que roda quando nenhuma condição é verdadeira
```

Operadores de comparação

Tipo de comparação	Operador
Igual a	`a == b`
Diferente de	`a != b`
Maior que	`a > b`
Maior ou igual a	`a >= b`
Menor que	`a < b`
Menor ou igual a	`a <= b`

Operadores booleanos

Operador	Descrição
`a and b`	`True` se ambos a e b forem `True`
`a or b`	`True` se um valor dentre a ou b for `True`
`not a`	Inverte o valor de a (`True` vira `False` e vice-versa)

09. Listas e tuplas

Listas

Listas são um tipo de dado diferente dos que já vimos até aqui. Elas são **sequências ordenadas de elementos**, sendo que seus elementos podem ser de qualquer tipo (`str`, `int`, `float`, `bool`, até mesmo outras listas).

Listas são definidas com colchetes `[ ]` e seus elementos são separados uns dos outros com vírgulas `,`.

Exemplos de listas:

```
minha_lista = [1, 2, 3]

outra_lista = ['hello', 'olá', 'bom dia']

lista_misturada = [0, 1.1, 'PYTHON', True, [1, 2]]
```

Conseguimos usar listas para representar dados da vida real:

```
alunos = ['Ana', 'Bruno', 'Carlos']
vendas_por_dia = [50, 30, 35, 48, 70, 45, 50]
```

Indexação

Podemos utilizar índices entre colchetes para pegar os elementos da lista. **Importante: em Python, a indexação começa sempre de zero!**

```
>>> lista_misturada = [0, 1.1, 'PYTHON', True, [1, 2]]
>>> lista_misturada[0]  # Primeiro elemento (índice 0)
0

>>> lista_misturada[1]  # Segundo elemento (índice 1)
1.1

>>> lista_misturada[2]  # Terceiro elemento (índice 2)
'PYTHON'
```

Uma consequência da indexação começar em `0` é que o último elemento está no índice `len(lista_misturada)-1`:

```
>>> lista_misturada[len(lista_misturada)-1]
[1, 2]
```

Para não termos que escrever isso toda vez, Python permite que usemos índices negativos para pegar elementos de trás pra frente:

```
>>> lista_misturada[-1]  # Último elemento
[1, 2]

>>> lista_misturada[-2]  # Penúltimo elemento
True
```

Se um elemento da lista for ele mesmo outra lista, podemos usar índices encadeados:

```
>>> lista_misturada[-1]
[1, 2]

>>> lista_misturada[-1][0]
1

>>> lista_misturada[-1][1]
2
```

Elementos fora do alcance do índice geram um `IndexError`:

```
>>> lista_misturada[1000]
IndexError: list index out of range

>>> lista_misturada[-1000]
IndexError: list index out of range
```

Modificando listas

Listas são um tipo especial de dado em Python, porque podem ser **modificadas** diretamente:

```
>>> alunos = ['Ana', 'Bruno', 'Carlos']

>>> alunos[0] = 'Marcos'
>>> alunos[1] = 0.0
>>> alunos[2] = ['XXX', 'YYY']

>>> alunos
['Marcos', 0.0, ['XXX', 'YYY']]
```

Podemos também usar a palavra especial `del` para remover algum elemento da lista, a partir do seu índice:

```
>>> alunos = ['Ana', 'Bruno', 'Carlos']

>>> del alunos[0]

>>> alunos
['Bruno', 'Carlos']
```

Isso é diferente de todos os dados com que já trabalhamos até agora! Com dados imutáveis, temos certeza que a variável sempre se refere ao mesmo valor (a não ser que ela seja redefinida). Já com dados mutáveis como listas, o seu conteúdo pode ser modificado dinamicamente.

Mais pra frente, quando falarmos de métodos de listas, vamos voltar a outras formas mais eficazes de modificá-las.

Tuplas

Tuplas são muito parecidas com listas, mas com as seguintes diferenças:

- São escritas com parênteses ( ) no lugar dos colchetes.
- São **imutáveis**.

Acompanhe o exemplo:

```
>>> alunos = ('Ana', 'Bruno', 'Carlos')

>>> alunos[0] = 'Marcos'
TypeError: 'tuple' object does not support item assignment
```

Você pode se perguntar: qual o sentido de uma estrutura imutável, se temos as listas?

- Um objeto não mutável está "protegido" de programadores inadvertidos que tentam modificar algum valor crucial no programa.
- Ela indica a quem ler o código que aqueles valores são constantes.
- Pode ser usado pra representar alguma estrutura, como por exemplo uma tupla de 3 elementos: nome, endereço, CPF.

Em termos práticos, e especialmente nos códigos simples com que iremos trabalhar aqui no curso, as listas nos bastam. Mas é importante saber que as tuplas existem!

Resumo

Listas são **sequências** com elementos de qualquer tipo.
Acesso, modifico e deleto um elemento da lista com seu **índice**:

```
li = [1, 2, 3]

primeiro_elemento = li[0]
li[1] = 100
del li[2]
```

Índice negativo = pegar elementos a partir do final da lista.
Tuplas são como listas, mas não podem ser modificadas (objeto **imutável**).

10. Sequências e slicing

Sequências

Listas e tuplas permitem indexação porque são **sequências**. Strings também funcionam como sequências (imutáveis), e possuem as mesmas regras de indexação:

```
>>> nome = "Juliano"

>>> nome[0]
'J'

>>> nome[-1]
'o'
```

Sequências em Python podem ser vazias - nesse caso, possuem tamanho zero e qualquer indexação resultará em erro:

```
>>> s = ""  # String vazio
>>> len(s)
0

>>> s[0]
IndexError: string index out of range

>>> li = []  # Lista vazia
>>> len(li)
0
>>> li[0]
IndexError: list index out of range

>>> tup = ()  # Tupla vazia
>>> len(tup)
0
>>> tup[0]
IndexError: tuple index out of range
```

Sequências verdadeiras e falsas

O que significa uma sequência ser “falsa” ou “verdadeira”? Em Python, isso é equivalente a ela estar vazia ou não. Se ela tiver algum elemento, ela será considerada verdadeira (ainda que os elementos em si sejam “falsos”).

Podemos checar esse comportamento passando uma sequência à função `bool`.

```
>>> bool(str())  # String vazio
False

>>> bool('    ')
```

```
True

>>> bool('Olá!')
True

>>> bool(list())  # Lista vazia
False

>>> bool([0, 0, 0])
True

>>> bool(tuple())  # Tupla vazia
False

>>> bool((1, 2, 3))
True
```

Em geral, qualquer tipo de dado pode ser considerado `True` ou `False`. As regras dependem do tipo de dado:

- Strings vazios são `False`, qualquer outro string é `True`
- Listas e tuplas vazias são `False`, qualquer outra lista ou tupla é `True`
- Número zero (seja um `int` ou `float`) são `False`, qualquer outro número é `True`

A forma tradicional de testar se uma sequência `seq` está vazia ou não é simplesmente usar a construção `if seq`:

```
seq = []

if seq:
    print('Sequência não é vazia')
else:
    print('Sequência é vazia')
```

Note que o código acima funcionaria para qualquer sequência!

Em outras linguagens de programação, que não aceitam a construção acima, precisaríamos checar se o tamanho da sequência é zero com uma função equivalente a `len()`. Python permite essa expressividade concisa!

Slices

Além de índices únicos, podemos passar *slices* a uma sequência para obter uma "fatia" dos seus elementos:

```
>>> pessoas = ['João', 'Paulo', 'Clara', 'Maria']

>>> pessoas[1]
```

```
'Paulo'

>>> pessoas[1:3]
['Paulo', 'Clara']

>>> pessoas[3:4]
['Maria']
```

Nestes exemplos, os valores `1:3` e `3:4` são os slices.

Notas sobre os slices:

- Incluem o elemento no primeiro índice.
- Pegam todos os elementos até o elemento no segundo índice, **sem incluí-lo**.
- Retornam uma nova lista, mesmo se ela possuir apenas 1 elemento.

Esse comportamento pode parecer confuso, mas dessa forma, significa que se formos do índice zero até o tamanho da sequência, pegamos todos seus elementos:

```
>>> pessoas[0:4]
['João', 'Paulo', 'Clara', 'Maria']

>>> pessoas[0:len(pessoas)]
['João', 'Paulo', 'Clara', 'Maria']
```

Na realidade, é tão comum começar de zero e ir até o final, que Python nos permite omitir esse valores:

```
>>> pessoas[:2]  # Do primeiro até o elemento de índice 2
['João', 'Paulo']

>>> pessoas[:-1]  # Excluir o último elemento apenas
['João', 'Paulo', 'Clara']

>>> pessoas[2:]  # Do elemento de índice 2 até o final
['Clara', 'Maria']

>>> pessoas[1:]  # Excluir o primeiro elemento apenas
['Paulo', 'Clara', 'Maria']
```

Usamos listas nestes exemplos, mas note que tudo isso funciona para qualquer outra sequência, como strings:

```
>>> nome = "Juliano"

>>> nome[1:]  # Exclui a primeira letra
'uliano'

>>> nome[:-1]  # Exclui a última letra
'Julian'

>>> nome[2:5]  # "lia"
'lia'
```

O valor de pulo no slice

Por fim, podemos passar um terceiro valor a um slice, que corresponde ao pulo.

Por padrão, esse valor é 1. Mas se quisermos pegar um elementos a cada dois ou três elementos, podemos modificá-lo:

```
>>> numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> numeros[0:len(numeros):2]  # Apenas os ímpares
[1, 3, 5, 7, 9]

>>> numeros[::2]
[1, 3, 5, 7, 9]

>>> numeros[1:len(numeros):2]  # Apenas os pares
[2, 4, 6, 8]

>>> numeros[1::2]
[2, 4, 6, 8]
```

O pulo também aceita valores negativos. Neste caso, os valores são percorridos de trás pra frente!

```
>>> numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> numeros[::-1]  # Do final pro começo
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

O importante aqui não é decorar todas essas regrinhas de slicing, mas saber que existem toda essa funcionalidade em qualquer sequência de Python. Assim, na hora em que vocês se depararem com um problema prático, vocês lembrarão que isso existe, e irão atrás da solução!

Resumo

Strings também são sequências. Sequência "falsa" = sequência vazia.
Slicing retorna "fatia" de sequência, com sintaxe `[inicio:fim:pulo]`.
O slicing inclui elemento no índice `inicio` e exclui o elemento no índice `fim`.
Por padrão, `inicio` vale `0` e `fim` é igual ao tamanho da sequência.
`pulo` controla de quantos em quantos elementos pegar. O valor padrão é `1`.

11. Função `range` e for loops

A função `range`

A função `range()` cria uma sequência de números em memória:

```
>>> range(10)
range(0, 10)
```

Mas para ver a sequência, é preciso usar a função `list()` ou `tuple()`. Isso porque ela não carrega os números em memória, apenas os deixa preparados para serem percorridos:

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

>>> tuple(range(20))
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19)
```

Na realidade, um range não é uma lista ou tupla, mas um objeto próprio em Python:

```
>>> type(range(10))
range
```

Função `range` vs slicing

Note que o último elemento não apareceu nas sequências acima.

Na realidade, a função `range()` aceita argumentos análogos ao slicing que fizemos anteriormente:

```
>>> numeros = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> numeros[1:5]
[1, 2, 3, 4]
>>> list(range(1, 5))
[1, 2, 3, 4]

>>> numeros[0:10:2]
[0, 2, 4, 6, 8]
>>> list(range(0, 10, 2))
[0, 2, 4, 6, 8]

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

As diferenças são apenas a sintaxe (dois pontos no slicing `:`, vírgulas no range `,`) e o fato de que para usar o slicing, a lista precisa estar previamente definida.

Parâmetros de um `range`: Start, stop, step

Dependendo do número de argumentos do range, os valores padrão mudam:

- 1 número = stop (começa de zero e pula de 1 em 1)
- 2 números = start, stop (pula de 1 em 1)
- 3 números = start, stop, step

Os exemplos abaixo são equivalentes:

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

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

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

É muito comum usarmos range para definirmos uma lista grande de valores de uma só vez!

```
# Números de 1 a 1000
>>> list(range(1, 1001))  # Números de 1 a 1000
[1, 2, 3, ..., 999, 1000]

# Números pares de 1 a 300
>>> list(range(0, 301, 2))
[0, 2, 4, ..., 298, 300]  # etc
```

For loops

Em muitos momentos em programação, vamos querer repetir uma ação N vezes. Podemos fazer isso com um **for loop**.

Para criarmos um for loop, usamos a mesma estrutura que já aprendemos para condicionais, mas utilizamos a palavra-chave `for`:

```
for n in range(10):
    print(f'O valor de n é: {n}')

print('O loop acabou')
```

Esse código exibe a mensagem `O valor de n é: 0`, `O valor de n é: 1`, e assim por diante, até percorrer todos os valores do `range()`!

Note que a variável n não precisou ser definida antes do loop! Ela é criada automaticamente pelo for loop, e representa cada elemento da sequência sendo iterada. Posso chamá-la do que quiser.

Sugiro acompanhar a iteração pelo debugger, para ver como a variável n altera seu valor a cada iteração.

Repetindo ações múltiplas vezes

Podemos usar um for loop para repetir ações um certo número de vezes. Neste exemplo, a mensagem `Olá!` é exibida 3 vezes:

```
for n in range(3):
    print('Olá!')

print('O loop acabou')
```

Note que, neste caso específicos, nem utilizamos a variável `n`. Ela corresponde ao número gerado pelo `range()`, mas em alguns casos esse número não é necessário pra nada.

Se a variável não for usada para nada, existe a convenção de nomeá-la como `_` :

```
for _ in range(3):
    print('Olá!')

print('O loop acabou')
```

Resumo

> Função `range(inicio, fim, pulo)` prepara uma sequência de números. É preciso entregá-la para uma lista ou percorrê-la com um for loop para pegar seus números.
> For loop pode ser usado com `range(n)` para repetir uma ação n vezes.
> Sintaxe de um for loop:
>
> ```
> for numero in range(n):
>     # Repete a ação n vezes
>
> # Final do loop
> ```

12. Iterando sobre sequências

Vimos como iterar sobre os números de um `range` usando um for loop. Na realidade, podemos iterar sobre qualquer sequência de Python!

Ao usar um for loop com uma lista, itero sobre seus valores:

```
valores = [10, 20, 30]
for valor in valores:
    print(f'O valor é: {valor}')
```

Ao usar um for loop com um string, itero sobre seus caracteres:

```
nome = "Juliano"
for caractere in nome:
    print(f"O caractere é: {caractere}")
```

Posso até iterar sobre estruturas mais complexas, como listas de tuplas!

```
clientes = [('Ana', 'xxx', 'xxx@gmail.com'), ('Eduardo', 'yyy', 'yyy@gmail.com')]

for cliente in clientes:
    nome = cliente[0]
    cpf = cliente[1]
    email = cliente[2]
    print(f'cliente: {nome}\nCPF: {cpf}\nemail: {email}\n---')
```

Este tipo de iteração funciona com qualquer tipo de dado que seja uma **sequência**.

Lembre-se de dar um nome útil para a variável do for loop!

Formatando sequências

Valores dentro de listas (ou qualquer outra estrutura) podem ser reformatados com espaço em branco, sem que isso cause um erro no Python. Isso nos ajuda a ler e editar código de forma mais clara:

```
clientes = [
    ('Ana',     'xxx', 'xxx@gmail.com'),
    ('Eduardo', 'yyy', 'yyy@gmail.com'),
]
```

Desempacotamento de sequências

Se sei o tamanho de uma sequência, posso "desempacotá-la" em variáveis em uma única linha. Tomando como exemplo o código dos clientes:

```
clientes = [
    ('Ana',     'xxx', 'xxx@gmail.com'),
    ('Eduardo', 'yyy', 'yyy@gmail.com'),
]

for cliente in clientes:
    nome, cpf, email = cliente
    print(f'cliente: {nome}\nCPF: {cpf}\nemail: {email}\n---')
```

Isso é chamado de **desempacotamento de sequências**, e posso usar com qualquer tipo de sequência:

```
>>> x, y = (10, 20)

>>> x
10

>>> y
20

>>> letra1, letra2, letra3 = 'ABC'

>>>letra1
'A'

>>> letra2
'B'

>>> letra3
'C'
```

O único detalhe é que preciso saber o número de elementos de antemão, pois não podem sobrar/faltar variáveis:

```
>>> x, y, z = (10, 20)
ValueError: not enough values to unpack (expected 3, got 2)
```

Posso até mesmo desempacotar uma sequência na chamada do for loop!

```
clientes = [
    ('Ana',     'xxx', 'xxx@gmail.com'),
    ('Eduardo', 'yyy', 'yyy@gmail.com'),
]

for nome, cpf, email in clientes:
    print(f'cliente: {nome}\nCPF: {cpf}\nemail: {email}\n---')
```

Este também é um exemplo de aplicação de tuplas - como elas são imutáveis, sei que sempre terão 3 elementos dentro delas.

Resumo

Posso usar for loops com qualquer sequência.
Posso adicionar espaço em branco e novas linhas dentro de listas e tuplas, de modo a formatar melhor o seu conteúdo.
Desempacotamento de sequências = desmembrar cada elemento de uma sequência em uma variável:

```
x, y, z = (10, 20, 30)

print(x)
print(y)
print(z)
```

13. While Loops, `break` e `continue`

While loops

Um while loop é utilizado para repetir uma ação até que uma condição deixe de ser verdadeira. Sua sintaxe é muito parecida com um for loop, mas precisamos de uma **condição de parada** que impeça nosso código de executar infinitamente!

Exemplo:

```
n = 0

while n < 3:
    print(f'O valor de n é: {n}')
    n = n + 1

print('O loop acabou')
```

Neste código, a linha `n = n + 1` serve de condição de parada. Como o valor de n está sempre sendo incrementado, eventualmente a condição `n < 3` deixará de ser verdadeira (neste caso, no momento em que n vale 3). Quando isso acontecer, o loop terminará e o código seguirá para os comandos abaixo dele.

Loop infinito

Se retirarmos a condição de parada, então o código não para de executar nunca!

```
n = 0

while n < 3:
print(f'O valor de n é: {n}')
    # n += 1

print('O loop acabou')
```

Para forçar a parada do código, podemos usar as combinação `Ctrl + C` (de **cancelar** a operação). Em IDEs como o Mu, geralmente há também algum botão de forçar a parada na interface.

Neste exemplo, o "código infinito" não produz nenhum problema, além de ficar emitindo a mesma mensagem para o terminal indefinidamente. Mas se o código ficasse adicionando dados em um arquivo, ou ficasse criando listas de números em memória, poderia acabar com o espaço em disco e/ou memória!

Palavra-chave `break`

Podemos usar a palavra-chave `break` para forçar a saída de um loop em certo ponto do código (isso serve tanto para while loops quanto for loops). O código abaixo executaria até n = 9, mas com o `break`, ele acaba precocemente:

```
n = 0

while n < 10:
    print(f'O valor de n é: {n}')
    n += 1
    if n == 5:
        break

print('O loop acabou')
```

Palavra-chave `continue`

Podemos usar a palavra-chave `continue` para seguir imediatamente para a próxima iteração do loop, funcionando assim como um "curto circuito".

No exemplo abaixo, iteramos sobre uma sequência de números e para cada número n calculamos o valor de 1 dividido por n. Usamos o `continue` para pular esta operação na iteração em que n vale 0, de forma a evitar o erro de `ZeroDivisionError`:

```
for n in range(-5, 6):
    if n == 0:
        continue  # Pula para evitar divisão por zero!
    resultado = 1 / n
    print(f'1 dividido por {n} = {resultado:.2f}')
```

Uso para `while loop`: pedir por um input específico

Um loop muito comum de ser usado com while é o `while True`. Como o valor `True` nunca deixará de ser verdadeiro, este loop realiza uma ação infinitamente. Contudo, ainda podemos usar a palavra-chave `break` para finalizá-lo.

Esta construção é muito usada para rodar um programa até que o usuário decida fechá-lo. No exemplo abaixo, o código finaliza apenas quando o usuário passar o valor `"q"` para o `input()`:

```
while True:  # Infinito!
    entrada = input('Digite qualquer coisa ("q" para sair): ')
    if entrada == 'q':
        break
    print(f'O valor digitado foi: {entrada}')

print('O código acabou')
```

Resumo

Sintaxe de um while loop:

```
while condicao:
    # Repete a ação até condicao se tornar falsa

# Final do loop
```

Palavra-chave `break`: finaliza o loop atual instantaneamente.
Palavra-chave `continue`: segue para a próxima iteração do loop instantaneamente.
Usar loop `while True` para repetir ação indefinidamente, até o código encontrar um `break`!

14. Dicionários e o operador `in`

Dicionários

Dicionários são um dos principais tipos de dados em Python. Com eles, podemos "mapear" ou associar valores entre si, de acordo com alguma lógica.

Se pararmos para pensar, veremos que existem muitos casos de associação de valores na vida real:

- Cada país está associado à sua capital
- Cada produto de um mercado está associado ao seu preço
- Cada pessoa está associada a uma lista dos seus animais de estimação
- Cada CPF está associado a um nome

Todas essas associações podem ser representadas em Python através de **dicionários**.

Sintaxe de um dicionário

Um dicionário é composto por **chaves** associadas a **valores**. Cada associação é chamada de um **par chave-valor**.

No código, representamos uma associação de uma chave com seu valor com dois pontos (`:`), e separamos cada par chave-valor por vírgulas (`,`). O dicionário em si é definido com chaves (`{ }`).

Exemplo:

```
>>> capitais = {'Brasil': 'Brasília', 'França': 'Paris', 'Japão', 'Tóquio'}

>>> capitais
{'Brasil': 'Brasília', 'França': 'Paris', 'Japão', 'Tóquio'}
```

Da mesma forma como nas listas, podemos reestruturar o dicionário em linhas para facilitar a digitação e compreensão:

```
>>> capitais = {
    'Brasil': 'Brasília',
    'França': 'Paris',
    'Japão': 'Tóquio',
}
>>> capitais
{'Brasil': 'Brasília', 'França': 'Paris', 'Japão', 'Tóquio'}
```

Usando dicionários

De forma semelhante a listas, usamos colchetes para pegar valores do dicionário. A diferença é que, ao invés de usarmos índices, usamos a chave para pegar o valor correspondente:

```
>>> capitais['Brasil']
'Brasília'

>>> capitais['França']
'Paris'

>>> capitais['Japão']
'Tóquio'
```

Também de forma parecida com listas, tentar pegar uma chave que não está criada no dicionário resulta em um erro específico (`KeyError` neste caso):

```
>>> capitais['Inglaterra']
KeyError: 'Inglaterra'
```

Dicionários também são dados **mutáveis**. É possível adicionar criar novas associações de chave-valor usando o operador = (o mesmo que usamos pra criar variáveis):

```
>>> capitais['Inglaterra'] = 'Londres'
>>> capitais
{'Brasil': 'Brasília',
 'França': 'Paris',
 'Japão': 'Tóquio',
 'Inglaterra': 'Londres'}

>>> capitais['Inglaterra']
'Londres'
```

- Importante: **um dicionário possui chaves únicas, ou seja, não pode ter chaves repetidas**. Se eu passar um novo valor para uma mesma chave, ela é sobrescrita!

```
>>> capitais['Inglaterra'] = '????'
>>> capitais
{'Brasil': 'Brasília',
 'França': 'Paris',
 'Japão': 'Tóquio',
 'Inglaterra': '????'}

>>> capitais['Inglaterra']
'????'
```

Além disso, embora os dicionários sejam mutáveis, suas chaves não podem ser dados mutáveis:

```
>>> capitais[['esta', 'chave', 'é', 'uma', 'lista']] = 'xxx'
TypeError: unhashable type: 'list'
```

Isso faz sentido se pensarmos na restrição de chaves repetidas: eu poderia começar com duas listas distintas como chaves do dicionário, e modificá-las até que se tornem idênticas. Isso iria contra a regra de um dicionário possuir chaves únicas. Para evitar esse problema, o Python proíbe dados mutáveis como chaves de dicionários.

Também é possível deletar algum par chave-valor usando a palavra-chave `del`:

```
>>> del capitais['Inglaterra']
>>> capitais
{'Brasil': 'Brasília', 'França': 'Paris', 'Japão', 'Tóquio'}

>>> capitais['Inglaterra']
KeyError: 'Inglaterra'
```

Iterando sobre dicionários

Se iterarmos sobre um dicionário, vamos iterar sobre suas *chaves*. A partir delas, é possível pegar os *valores*:

```
for pais in capitais:
    capital = capitais[pais]
    print(f'A capital de {pais} é {capital}')
```

Ordem de iteração: os dicionários preservam sua *ordem de inserção*. Isto significa que, ao iterarmos sobre ele, as chaves são devolvidas na ordem em que foram criadas:

```
dic = {}  # Criando um dicionário vazio
dic = dict()  # Forma alternativa para criar dicionário vazio

dic[10] = 'abc'
dic[3.14] = True
dic['CHAVE'] = 5
dic[False] = ''

print(dic)

for k in dic:
    v = dic[k]
    print(f'Chave: {k}  ->  Valor: {v}')
```

Por favor, não criem dicionários confusos como este! É apenas um exemplo para ilustrar a iteração. Normalmente vamos querer trabalhar com dicionários que representem uma *associação existente na vida real*.

O operador `in` com dicionários

Podemos usar o operador `in` para checar se a chave existe no dicionário, antes de acessá-la:

```
>>> capitais = {
    'Brasil': 'Brasília',
    'França': 'Paris',
    'Japão': 'Tóquio',
}
>>> 'Brasil' in capitais
True
```

```
>>> 'Inglaterra' in capitais
False
```

Dessa forma, é possível criar um script que checa se a chave existe ou não, e exibe uma mensagem de acordo com o resultado (evitando assim o `KeyError`):

```
capitais = {
    'Brasil': 'Brasília',
    'França': 'Paris',
    'Japão': 'Tóquio',
}

pais = 'Inglaterra'

if pais in capitais:
    print(f'A capital do país {pais} é {capitais[pais]}')
else:
    print(f'Não há capital registrada para o país {pais}')
```

Operador `in` em sequências

O operador `in` não é exclusivo de dicionários. Podemos utilizá-lo também com *qualquer sequência* para checar se ela contém algum valor específico:

```
# Listas
>>> valores = [1, 2, 3]
>>> 4 in valores
False

>>> 3 in valores
True

# Tuplas
>>> nomes = ('Ana', 'Carlos', 'Eduardo')
>>> 'Bruno' in valores
False

>>> 'Ana' in valores
True

# Strings
>>> texto = 'Eu estou estudando Python na Asimov Academy!'

>>> 'Java' in texto
False

>>> 'Python' in texto
True
```

Note que, no caso de strings, podemos usar o operador `in` para checar se qualquer palavra ou "substring" está dentro de um string!

Resumo

Dicionários são associações entre valores, com cada **chave** associada a um **valor**.
Dicionários são **mutáveis**. Suas chaves são únicas e não podem ser dados mutáveis.

```
>>> capitais = {
    'Brasil': 'Brasília',
    'França': 'Paris',
    'Japão': 'Tóquio',
 }

>>> capitais['Inglaterra'] = 'Londres'
>>> capitais['Brasil'] = '!'
>>> del capitais['Japão']
>>> capitais
{'Brasil': '!', 'França': 'Paris', 'Inglaterra': 'Londres'}
```

O operador `in` é usado para checar se um valor está dentro de uma sequência.
No caso de dicionários, `x in d` retorna se a chave `x` existe no dicionário `d`.
No caso de strings, `s in texto` retorna se o "substring" `s` aparece dentro do string `texto`.

15. Métodos

O que é um método?

Em Python, cada tipo de dado diferente (strings, ints, listas, dicionários) é considerado como sendo um **objeto** próprio.

Existem operações que vamos querer realizar em um determinado tipo de dado, por exemplo:

- Strings: trocar letras maiúsculas por minúsculas
- Listas: adicionar um novo elemento ao final da lista
- Dicionários: combinar valores de 2 dicionários diferentes

Algumas dessas operações são tão comuns e úteis que já são definidas para cada um dos objetos. Chamamos estas operações de **método** (de string, de lista, de dicionário…).

Outra forma de pensarmos em métodos é que são **funções vinculadas a um objeto específico**. Este conceito de método não é exclusivo de Python: outras linguagens de programação usam a mesma ideia.

Como usar um método

Os métodos são acessados usando um **ponto final** entre o objeto e o nome do método. Como são similares a funções, métodos precisam ser chamados com parênteses `( )` para executar.

Exemplo: limpar todos os elementos de um dicionário com o método `dict.clear()`.

```
produtos = {
    'banana': 3.60,
    'leite': 4.90,
    'carne': 15.50,
    'pão': 9.00,
}

print(produtos)

produtos.clear()

print(produtos)
```

Quais métodos existem? Que argumentos aceitam? Podemos descobrir isso da seguinte forma:

- `dir(produtos)`: a função `dir()` retorna uma lista dos métodos existentes para o dicionário `produtos`. (Os métodos que começam com dois underscore, como `__str__`, são internos do Python e podem ser ignorados por enquanto).

- `help(produtos.nome_do_metodo)`: a função `help()` exibe a documentação de um método
- "Vida real": procuramos na documentação online e tutoriais com exemplos.

Como vamos aprender métodos

Vamos passar pelos principais tipos de dados que aprendemos neste curso, para entender que métodos possuem e como podemos utilizá-los. Conforme formos passando pelos métodos, pense em como você poderia aplicar cada um deles a algum problema, seja um problema que você encontra no seu dia a dia, ou algum exercício anterior que poderia ser simplificado.

Métodos de dicionários

`dict.clear()`

Limpa todos os valores de um dicionário.

```
>>> produtos = {
    'banana': 3.60,
    'leite': 4.90,
    'carne': 15.50,
    'pão': 9.00,
}
>>> produtos.clear()
>>> produtos
{}
```

`dict.get()`

Retorna o valor associado a uma chave, ou retorna um valor substituo caso a chave não exista (valor-padrão: None)

```
>>> produtos = {
    'banana': 3.60,
    'leite': 4.90,
    'carne': 15.50,
    'pão': 9.00,
}

>>> produtos.get('banana')
3.6

>>> produtos.get('pão')
9.0
```

```
>>> produtos.get('arroz')
```

Caso a chave não exista, o valor retornado é None, que representa um valor "nulo". Como o console omite valores None por padrão, precisamos pegar o valor para exibi-lo em tela:

```
>>> resultado = produtos.get('arroz')
>>> print(resultado)
None
```

Também podemos passar um segundo argumento para ser o valor padrão, ao invés de None:

```
>>> resultado = produtos.get('arroz', 'não cadastrado')
>>> print(resultado)
não cadastrado
```

`dict.setdefault()`

Faz o mesmo que `dict.get()`, mas cria a associação chave-valor caso não exista:

```
produtos = {
    'banana': 3.60,
    'leite': 4.90,
    'carne': 15.50,
    'pão': 9.00,
}

print(produtos)  # Dicionário original

preco = produtos.setdefault('banana', 100.0)

print(produtos)  # chave existe -> dicionário não modifica
print(preco)  # preço é o valor antigo da chave 'banana'

preco = produtos.setdefault('arroz', 100.0)

print(produtos)  # chave não existe -> associação nova criada
print(preco)  # preço é o valor novo adicionado
```

`dict.keys()`, `dict.values()`, `dict.items()`

Estes métodos retornam sequências contendo as chaves, valores, ou pares chave-valor de um dicionário, respectivamente. Muito utilizado para iterar sobre o dicionário, principalmente para iterar sobre cada par com `dict.items()`:

```
produtos = {
    'banana': 3.60,
    'leite': 4.90,
    'carne': 15.50,
```

```
    'pão': 9.00,
}

for chave in produtos.keys():
    print(chave)


for preco in produtos.values():
    print(preco)


for par in produtos.items():
    print(par)


for chave, preco in produtos.items():
    print(f'{chave} -> R$ {preco:.02f}')
```

dict.update()

Atualiza um dicionário a partir de outro. Atualizar significa que chaves novas são inseridas, e chaves existentes são atualizadas:

```
produtos = {
    'banana': 3.60,
    'leite': 4.90,
    'carne': 15.50,
    'pão': 9.00,
}

novos_produtos = {
    'massa': 5.70,
    'banana': 4.40,
}

print(produtos)

produtos.update(novos_produtos)

print(produtos)
```

dict.copy()

Cria uma cópia independente de um dicionário:

```
produtos = {
    'banana': 3.60,
    'leite': 4.90,
    'carne': 15.50,
    'pão': 9.00,
```

```
}

produtos_copia = produtos.copy()
produtos_copia['morango'] = 3.30

print(produtos)
print(produtos_copia)
```

Métodos de números

`float.as_integer_ratio()`

Mostra dois inteiros que, quando divididos, geram (ou se aproximam) do valor do número:

```
>>> x = 4.5
>>> x.as_integer_ratio()
(9, 2)  # 9 dividido por 2 é 4.5

>>> x = 38.125
>>> x.as_integer_ratio()
(305, 8)  # 305 dividido por 8 é 38.125
```

`float.is_integer()`

Retorna `True` se o `float` representar um número inteiro (isto é, porção decimal com valor `.0`), caso contrário retorna `False`:

```
>>> x = 4.5
>>> x.is_integer()
False

>>> x = 40.0
>>> x.is_integer()
True
```

Métodos de strings

`str.upper()` e `str.lower()`

Converte palavras para letras maiúsculas / minúsculas, respectivamente:

```
>>> palavra = 'Olá MUnDo!'
>>> palavra.uppercase()
'OLÁ MUNDO!'

>>> palavra.lowercase()
'olá mundo!'
```

`str.startswith()` e `str.endswith()`

Checa se o string começa ou termina com certa parte de texto:

```
>>> arquivo = '2023_01_01_NotaFiscal.pdf'
>>> arquivo.startswith('2023_01_01')
True

>>> arquivo.startswith('2023_02_03')
False

>>> arquivo.endswith('.docx')
False

>>> arquivo.endswith('.pdf')
True
```

Podemos usar para encontrar um arquivo específico!

```
arquivo = '2023_01_01_NotaFiscal.pdf'

if arquivo.startswith('2023_01_01') and arquivo.endswith('.pdf'):
    print('Encontrado arquivo! Enviando por email ...')
```

`str.count()`

Conta o número de ocorrências de um caractere ou substring:

```
>>> texto = 'Hoje em dia todo dia é um novo dia. Mais um dia chega. Dia!'
>>> texto.count('a')
7

>>> texto.count('dia')
4
```

A distinção entre maiúsculas e minúsculas faz com que o último `"Dia"` não seja contado. Podemos corrigir usando **métodos encadeados**. Primeiro, vamos transformar o texto em letras minúsculas, e em seguida contar a ocorrência do string `"dia"`:

```
>>> texto = 'Hoje em dia todo dia é um novo dia. Mais um dia chega. Dia!'
>>> texto.lower().count('dia')
5
```

`str.find()` e `str.index()`

Retorna o primeiro índice onde há um caractere/substring no string:

```
>>> seq = 'aaaaabaaaaaabaaaaaa'
>>> seq.find('b')
5

>>> seq.index('b')
5
```

Funcionam da mesma forma quando encontram a diferença é quando o substring não está presente: o método `str.find()` retorna o valor `-1`, enquanto o método `str.index()` causa um erro:

```
>>> seq = 'aaaaabaaaaaabaaaaaa'
>>> seq.find('c')
-1

>>> seq.index('c')
ValueError: substring not found
```

`str.isdigit()` e `str.isalpha()`

Retorna se o string é composto apenas de algarismos numéricos ou apenas de letras, respectivamente:

```
>>> s1 = '202363'
>>> s1.isdigit()
True

>>> s2 = 'mOinoiSAUInaSCiouNACS'
>>> s2.isalpha()
True

>>> s3 = 'Olá 2023 Python!'
>>> s3.isdigit()
False

>>> s3.isalpha()
False
```

`str.replace()`

Substitui um caractere/substring por outro:

```
>>> frase = 'Estou estudando Javascript!'
>>> frase.replace('!', '?')
'Estou estudando Javascript?'

>>> frase.replace('Javascript', 'Python')
'Estou estudando Python!'
```

Muito utilizado para remover espaços em branco e quebras de linha!

```
>>> frase = 'Esta é uma frase comprida e bem estruturada.\nEsta frase marca o começo de um novo
↪  parágrafo.'
>>> print(frase)
Esta é uma frase comprida e bem estruturada.
Esta frase marca o começo de um novo parágrafo.

>>> nova_frase = frase.replace('\n', ' ').replace(' ', '')
>>> print(nova_frase)
Estaéumafrasecompridaebemestruturada.Estafrasemarcaocomeçodeumnovoparágrafo.
```

`str.split()` e `str.join()`

`str.split()` separa um string em um certo caractere, gerando uma lista. Por padrão divide nos espaços, mas podemos passar um outro caractere para ser o delimitador:

```
>>> linha = 'Item1    Item2      Item3'
>>> linha.split()
['Item1', 'Item2', 'Item3']

>>> linha = 'Item1;Item2;Item3'
>>> linha.split(';')
['Item1', 'Item2', 'Item3']
```

`str.join()` faz o contrário: junta uma lista a partir de um caractere intermediário:

```
>>> nomes = ['Joana', 'Marcelo', 'Paulo']
>>> ' - '.join(nomes)
'Joana - Marcelo - Paulo'
```

Métodos de tuplas

Os métodos de tuplas também existem nas listas, e são os mesmos que já vimos para strings:

`tuple.count()`

Conta elementos

```
>>> tup = (0, 0, 0, 1, 0, 1, 0)
>>> tup.count(1)
2
```

`tuple.index()`

Retorna o índice do primeiro elemento igual ao argumento. Se não existir, `IndexError`:

```
>>> tup = (0, 0, 0, 1, 0, 1, 0)
>>> tup.index(1)
3

>>> tup.index(2)
ValueError: tuple.index(x): x not in tuple
```

Métodos de listas

As listas possuem os métodos `list.clear()` e `list.copy()`, que já vimos nos dicionários, e `list.count()` e `list.index()` que acabamos de ver nas tuplas.

```
l1 = [0, 0, 0, 1, 0, 1, 0]

l2 = l1.copy()

l1[0] = 'x'
l2.clear()

print(l1)
print(l2)
```

`list.append()`

Adiciona um elemento ao final da lista. é a principal forma de adicionar novos elementos em uma lista!

```
numeros = []

for n in range(5):
    numeros.append(n * 2)

print(numeros)  # [0, 2, 4, 6, 8]
```

Podemos **filtrar valores de uma lista** se criarmos uma lista vazia, iterarmos sobre a lista original, e usarmos o método `list.append()` apenas quando o elemento passar pelo filtro:

```
valores = [10, 30, -1, 0, 90, -100]

valores_positivos = []

for valor in valores:
    if valor > 0:
        valores_positivos.append(valor)

print(valores_positivos)
```

`list.extend()`

Se usarmos `list.append()` com uma lista, a lista inteira entra como um único elemento:

```
>>> numeros = [1, 2, 3]
>>> numeros.append([4, 5, 6])
>>> numeros
[1, 2, 3, [4, 5, 6]]
```

Neste caso, vamos querer usar `list.extend()` para inserir *cada elemento* da lista dentro da primeira lista:

```
>>> numeros = [1, 2, 3]
>>> numeros.extend([4, 5, 6])
>>> numeros
[1, 2, 3, 4, 5, 6]
```

Atenção: **listas são mutáveis**, e esta operação modifica a lista original! Se quisermos criar uma nova lista, podemos fazer a mesma operação usando +:

```
>>> numeros = [1, 2, 3]
>>> novos_numeros = numeros + [4, 5, 6]
>>> numeros
[1, 2, 3]

>>> novos_numeros
[1, 2, 3, 4, 5, 6]
```

`list.insert()`

Parecido com `list.append()`, mas requer posição de inserção ao invés de adicionar ao final da lista.

```
>>> vogais = ['a', 'i', 'o', 'u']
>>> vogais.insert(1, 'e')
>>> vogais
['a', 'e', 'i', 'o', 'u']
```

- Não dá `IndexError` (no pior caso, insere no começo ou final):

```
>>> vogais = ['a', 'i', 'o', 'u']
>>> vogais.insert(100, 'e')
>>> vogais
['a', 'i', 'o', 'u', 'e']
```

Mesmo assim, requer que saibamos a posição para inserir de antemão.

`list.pop()`

Remove um elemento da lista e o retorna:

```
>>> valores = [150, 30, 50, 75, 45, 90]
>>> valor_removido = valores.pop()

>>> valor_removido
90

>>> valores
[150, 30, 50, 75, 45]
```

Por padrão pega o último elemento (fazendo assim a operação contrária ao `list.append()`), mas aceita um índice para escolhermos o valor a remover:

```
>>> valores = [150, 30, 50, 75, 45, 90]
>>> valor_removido = valores.pop(0)  # Remove o primeiro da lista
>>> valor_removido
150

>>> valores
[30, 50, 75, 45, 90]
```

Muito útil para processamento em sequência!

```
clientes = ['xxxx', 'yyyy', 'zzzz']

while clientes:  # Enquanto lista não está vazia
    cliente = clientes.pop()
    print(f'Processando pedido do cliente {cliente}...')

print('Todos os pedidos processados!')
print(clientes)
```

`list.reverse()` e `list.sort()`

Como os nomes sugerem, são usados para inverter e ordenar os elementos:

```
>>> valores = [150, 30, 50, 75, 45, 90]
>>> valores.reverse()
>>> valores
[90, 45, 75, 50, 30, 150]

>>> valores.sort()
>>> valores
[30, 45, 50, 75, 90, 150]
```

Para ordenarmos os elementos do menor pro maior, podemos primeiro fazer chamar `list.sort()`, e em seguida `list.reverse()`.

Para ordenar uma lista, é preciso que os elementos sejam comparáveis entre si! Não é possível ordenar uma lista contendo strings e ints, porque Python não sabe comparar números e texto:

```
>>> valores = [150, 30, 'Python']
>>> valores.sort()
TypeError: '<' not supported between instances of 'str' and 'int'
```

Resumo

Métodos são funções associadas a um objeto. Em geral, apresentam funcionalidade muito útil para aquele tipo de dado

Chamamos métodos com a sintaxe `objeto.nome_do_metodo()`, (possivelmente passando argumentos dentro dos parênteses).

Cada objeto possui seus métodos específicos, de acordo com operações comuns que gostaríamos de realizar com ele.

Utilize a lista de métodos acima para rever a utilidade de cada método!

16. Funções

Criando funções

Até aqui, usamos algumas funções pré-fabricadas pelo Python. Oficialmente, são funções embutidas (*built-in functions*) da linguagem. Mas podemos criar nossas próprias funções para o nosso código!

Sintaxe de criação de funções

- `def` para definir uma função
- Abrir e fechar parênteses para definir os parâmetros
- Dois pontos para iniciar o corpo da função
- Corpo da função com indentação
- Ao final da função, algum valor pode ser **retornado** dela usando a palavra-chave `return`

Exemplo: função que soma o valor 2 ao número que recebe como argumento:

```
def somar_dois(n):
    return n + 2


# Usando a função
print(somar_dois(10))

print(somar_dois(0))

print(somar_dois(-3.15))
```

O importante aqui é entender que o valor `10`, que é passado com argumento da função, ocupa o espaço do parâmetro `n`, que definimos ao criar a função!

Tipo de dado e nomenclatura

Os tipos de dados que as funções aceitam não precisam ser declarados. Na função acima, em nenhum lugar informamos que `n` é um `int`. Na realidade, a função funciona também com `float`, como pudemos ver.

Python nunca vai impor uma declaração do tipo de dado para determinada função. Enquanto o programa não der bug, ele segue adiante!

Isso é bom e ruim ao mesmo tempo: bom porque nos permite ser flexíveis, e ruim porque podemos introduzir bugs em algum ponto do código sem perceber.

Escolhendo nomes

Quando crio uma função, é importante escolher bem tanto o **nome da função** quanto o **nome dos parâmetros** (se houver). Ambos são formas de entendermos o que exatamente a função faz!

A função abaixo serve para concatenar texto. Podemos chamá-la simplesmente de `c`:

```
def c(s1, s2):
    return s1 + s2


print(c('xxx', 'yyy'))

print(c('Python', 'Básico'))

print(c('Meu nome é ', 'Juliano'))
```

Mas faz mais sentido chamá-la de algo descritivo, como `concatenar_texto`:

```
def concatenar_texto(texto1, texto2):
    return texto1 + texto2


print(concatenar_texto('xxx', 'yyy'))

print(concatenar_texto('Python', 'Básico'))

print(concatenar_texto('Meu nome é ', 'Juliano'))
```

E lembre-se: não importa o nome da função, nada impede que eu passe outro tipo de dado para ela. Enquanto ela não gerar um erro, vai executar, mesmo que não faça exatamente o que eu gostaria.

A função `concatenar_texto`, por mais que tenha sido criada apenas para juntar dois strings, também serve para somar 2 valores. Isso é consequência da flexibilidade de Python.

```
def concatenar_texto(texto1, texto2):
    return texto1 + texto2


print(concatenar_texto(2, 3))
```

Para quê usar funções?

As funções que vimos aqui são simples, mas pensem em qualquer bloco de código que criamos nos desafios, que pudessem ser representadas por uma ação única:

- Validação de input do usuário
- Perguntar por um chute (desafio de adivinhe o número)

- ...

Exemplo: transformando validação de input em um código próprio:

```
def pegar_input_validado():
    while True:  # Infinito!
        opt = input('Escolha uma opção (1, 2) | "q" para sair: ')
        if opt == 'q':
            break
        elif opt not in ('1', '2'):
            print('Opção inválida! Digite 1 ou 2.')
            continue
        # else:
        #     print(f'Opção selecionada: {opt}')
        return opt


for n in range(3):
    opcao = pegar_input_validado()
    print(f'Opção selecionada: {opcao}')
```

Funções nos ajudam a **organizar nosso próprio código**, dando um nome a blocos lógicos que fazem uma ação específica. Além disso, se eu descobrir que há um bug no código que pega um input validado, já sei onde procurar: na sua função!

Parâmetros e argumentos

Formalmente, as variáveis na definição da função são chamados de **parâmetros**. Eles são substituídos por **argumentos** na hora de chamar a função.

No exemplo abaixo, n é o parâmetro, e x é o argumento:

```
def somar_dois(n):
    return n + 2

x = 10

resultado = somar_dois(x)
print(resultado)
```

No dia a dia, os conceitos acabam sendo intercambiáveis, mas é importante ter claro que são coisas distintas.

Parâmetros com valor padrão

Podemos criar valores-padrão para parâmetros:

```
>>> def adicionar_final(texto, final='!!!'):
...     return texto + final


>>> adicionar_final('Olá')
'Olá!!!'

>>> adicionar_final('Olá', '???')
'Olá???'
```

Quando não passamos um dos argumentos, o valor padrão do parâmetro é utilizado.

O único detalhe é que *parâmetros com valor padrão devem obrigatoriamente vir após os demais*. O Python tem essa obrigação para evitar confusão na hora de chamar a função:

```
>>> def adicionar_final(texto='Olá', final):
...     return texto + final
SyntaxError: non-default argument follows default argument
```

Se eu conseguisse definir a função da forma acima, e a chamasse com `adicional_final('XXXX')`, ficaria ambíguo se o valor `'XXXX'` deveria substituir o parâmetro `texto` (já que há apenas um argumento na chamada da função) ou o parâmetro `final` (já que o parâmetro `texto` possui um valor padrão).

Passando argumentos através de palavras-chave

Se soubermos o nome dos parâmetros, podemos passá-los explicitamente, mesmo que estejam fora de ordem. Observe:

```
def dividir(a, b):
    if b == 0:
        return 'Impossível dividir!'
    else:
        return a / b


print(dividir(10, 5))

print(dividir(a=10, b=5))

print(dividir(a=10, b=0))

print(dividir(b=10, a=0))
```

Muitos programas como o Mu ou outras IDEs exibem o nome dos parâmetros quando estamos escrevendo código. Isso nos ajuda a chamarmos as funções passando parâmetros de forma explícita, e ajuda na leitura e compreensão do nosso próprio código.

Em códigos mais avançados, é comum termos funções com muitos parâmetros, sendo que apenas poucos argumentos são passados diretamente para a função. Os valores-padrão são utilizados para todos os outros parâmetros!

```
def funcao_complexa(
    param_1=0,
    param_2=0,
    param_3=0,
    param_4=0,
):
    return param_1 + param_2 + param_3 + param_4


funcao_complexa(param_3=10)
```

Funções sem parâmetros

É possível criar funções sem parâmetros:

```
def retornar_lista():
    return [1, 2, 3]

print(retornar_lista())
```

É mais comum pensarmos em funções como "caixas" para as quais entregamos algum input e recebemos seu output. Contudo, em alguns casos podemos querer ter funções que não recebam parâmetros e nem retornem nada, como no caso de uma função que exibe algum texto na tela:

```
def dizer_ola():
    print('Olá!')

dizer_ola()
dizer_ola()
dizer_ola()
```

Dito isso, tome cuidado para não deixar seu código complexo. Funções sem parâmetros são indicativos de um fluxo confuso no seu código!

O valor None

Além de não ter parâmetros, a função `dizer_ola()` que definimos acima também não retorna nada... Ou pelo menos *parece* não retornar nada.

Na realidade, a função retorna o valor `None`, que representa a "ausência" de valor. Este valor é um nome próprio em Python, escrito com letra N maiúscula.

É bastante utilizado como "valor sentinela" para indicar que uma operação falhou, como por exemplo no método `dict.get()`:

```
produtos = {
    'banana': 3.60,
    'leite': 4.90,
    'carne': 15.50,
    'pão': 9.00,
}

print(produtos.get('banana'))  # Output: 3.6

print(produtos.get('pão'))  # Output: 9.0

print(produtos.get('arroz'))  # Output: None
```

Note que o valor `None` não é o número zero, pois não posso fazer contas matemáticas com ele. Em outras linguagens de programação, esse valor é comumente chamado de "null", "NA", entre outros nomes.

Retornando None

Qualquer função que não possua um `return` explícito vai retornar `None`:

```
def dizer_ola():
    print('Olá!')


retorno = dizer_ola()
print(retorno)
```

Caso eu prefira, posso ser explícito e retorná-lo diretamente:

```
def dizer_ola():
    print('Olá!')
    return None


retorno = dizer_ola()
print(retorno)


def dizer_ola():
    print('Olá!')
    return  # também funciona assim


retorno = dizer_ola()
print(retorno)
```

Funções sem retorno

A própria função `print` é um exemplo de função que não retorna nada:

```
retorno = print('Olá!')
print(retorno)
```

Alguns métodos também não retornam valores, principalmente aqueles que modificam dados mutáveis:

```
lista = [1, 2, 3]
lista.append(4)

print(lista)

retorno = lista.append(5)

print(lista)
print(retorno)
```

Muitos iniciantes na linguagem esperam que a linha `retorno = lista.append(5)` devolva uma nova lista, e se surpreendem que a variável `retorno` esteja vazia. Na realidade, o método `list.append()` modifica a lista que chama o método diretamente, sem que seja necessário retornar nada. Isso é uma decorrência direta da mutabilidade de listas!

Em Python, se diz que o método `list.append()` modifica os dados *in-place*, ou seja, mantendo-os na mesma variável.

Por outro lado, métodos que trabalham com dados imutáveis vão sempre retornar um novo objeto (já que não podem modificar o objeto que chama o método):

```
texto = 'Python'

print(texto)

retorno = texto.upper()

print(texto)
print(retorno)
```

Confusão com console de Python

O console de Python sempre exibe valores retornados, **exceto quando** o valor é None. Essa omissão é útil quando a conhecemos e estamos testando código, mas esse comportamento pode confundir iniciantes da linguagem!

```
>>> lista = [1, 2, 3]
>>> lista.append(4)  # Aparentemente sem retorno no console!
```

```
>>> texto = 'Python'
>>> texto.upper()  # Com retorno no console!
"PYTHON"
```

O mais confuso é a função `print()`. A função por si só retorna None, que é omitido do console. Mas o texto aparece no console de qualquer forma simplesmente porque este é o comportamento da função `print()`!

```
>>> print('Olá')  # None retornado, mas texto aparece no console
Olá
```

Estes conceitos são detalhes em Python, mas é bom entender para não se confundir nos seus códigos.

Se em algum momento você estiver trabalhando com uma variável que contém o valor None, provavelmente você esperava obter algum valor de retorno de alguma função / método, mas este retornou None. Fique de olho!

Resumo

Criamos nossas próprias funções com a sintaxe:

```
def nome_funcao(parametro1, parametro2='valor padrão vai aqui'):
    # corpo da função vai aqui
    return algum_valor
```

E depois chamamos as funções como qualquer outra função:

```
>>> nome_funcao('XXX')
# ...

>>> nome_funcao(parametro1='YYY', parametro2='ZZZ')
# ...
```

A lógica do corpo da função é inteiramente definida por nós mesmos, de acordo com o que queremos que ela faça.
Funções que não retornem nenhum valor acabam por retornar um valor nulo, chamado None.
O valor None também é retornado pela função `print()`, e por métodos que atuem diretamente em objetos mutáveis, como `list.append()` ou `dict.update()`.

17. Módulos de Python e a biblioteca padrão

Importação

Não estamos limitados apenas ao código que escrevemos e às funções *built-in* de Python. Podemos também importar conteúdo de outros scripts através de módulos! Até aqui, estive trabalhando sempre no mesmo script, o `hello_world.py`.

Vamos criar o seguinte script chamado `meu_modulo.py`, que será importado a partir de outro script:

```
def minha_funcao():
    print('Rodando "minha_funcao" do módulo "meu_modulo"!')
    return 10


x = 30
```

Rode o script abaixo da mesma pasta em que está o script `meu_modulo.py`:

```
import meu_modulo

retorno = meu_modulo.minha_funcao()

print(retorno)

print(meu_modulo.x)
```

Como pudemos ver, nosso script tem acesso a todas as variáveis e funções que estão vinculadas ao script `meu_modulo.py`!

A palavra-chave `import`

Usamos a palavra-chave `import` para importar algum módulo. Por convenção, sempre importamos tudo no topo do script. Posso importar um módulo de formas diferentes.

1) Importando o módulo em si: variáveis ficam vinculadas ao nome do módulo

```
import meu_modulo

retorno = meu_modulo.minha_funcao()

print(retorno)

print(meu_modulo.x)
```

2) Importando cada variável individualmente: acesso apenas com o nome da variável.

```
from meu_modulo import minha_funcao, x

retorno = minha_funcao()

print(retorno)

print(x)
```

3) Importando todas as variáveis (com *): importo todos os valores do módulo.

```
from meu_modulo import *

retorno = minha_funcao()

print(retorno)

print(x)
```

A opção 3) é prática, porém esconde o nome das variáveis. Pode ser que alguma variável minha seja sobrescrita por um nome que é importado!

Alias

Podemos mudar o nome do módulo ao importar. Isso nos ajuda a abreviar nomes compridos, ou a dar nomes mais significativos a algum módulo:

```
import meu_modulo as mm

retorno = mm.minha_funcao()

print(retorno)

print(mm.x)
```

Também posso fazer isso com variáveis individuais de um módulo:

```
from meu_modulo import minha_funcao as mf, x as y

retorno = mf()

print(retorno)

print(y)
```

Pra quê módulos?

Da mesma forma como funções nos ajudam a organizar blocos de código dentro de um script, módulos nos ajudam a organizar código em pedaços que façam sentido. Por exemplo, posso querer

organizar meu código que acessa um banco de dados fica separadamente do código que manipula e analisa os dados, que por sua vez está separado do código que exibe os dados em um dashboard.

Se você conseguir organizar tudo em um script, ótimo. Mas é comum em projetos maiores ou mais complexos dezenas de scripts de Python, cada um fazendo uma ação específica.

Essa divisão ajuda também na hora de organizar trabalho em equipe: cada pode focar em um ponto diferente do programa (isto é, em arquivos distintos).

A biblioteca padrão

Não precisamos fazer tudo do zero! Python inclui uma **biblioteca padrão** que contém inúmeras funcionalidades (inúmeras **mesmo**), prontas para utilizarmos. Estão disponíveis em qualquer instalação padrão de Python. Portanto, temos garantia de que vamos conseguir importá-las sem precisar instalar nada a mais!

O site principal da biblioteca padrão está neste link.

Tour por alguns módulos da biblioteca padrão

`math.py`: contas e definições matemáticas avançadas.

```
import math

print(math.pi)

print(math.log(16, 2))
```

`datetime.py`: objetos que representam datas e horas.

```
import datetime

print(datetime.datetime.now())

agora = datetime.datetime.now()
ano_2000 = datetime.datetime(2000, 1, 1)

print(agora - ano_2000)
```

`random.py`: módulo para sorteios aleatórios de números e valores.

```
import random

for _ in range(5):
    n = random.randint(1, 10)
    print(f'Número escolhido: {n}')
```

```
nomes = ['Juliano', 'Marcos', 'Pedro']

for _ in range(5):
    nome = random.choice(nomes)
    print(f'Nome escolhido: {nome}')
```

`os.py`: módulo de interação com meu sistema operacional e seus arquivos.

```
import os

print(os.getcwd())

print(os.listdir())
```

`time.py`: módulo para medição de tempo (ex: medir quanto tempo o programa leva para executar).

```
import time

inicio = time.time()

print('Primeira linha')
time.sleep(2)
print('Segunda linha')

final = time.time()

tempo_execucao = final - inicio

print(f'Script rodou em {tempo_execucao:.3f}')
```

Utilize a biblioteca padrão!

Programadores experientes já passaram **muitas** vezes pela seguinte situação: gastar tempo escrevendo código para algo que já existia na biblioteca padrão.

Às vezes, esse processo de "reinventar a roda" é bom para o aprendizado. Mas pensando em produção de código, além de ser uma "perda de tempo", provavelmente o código final será pior que o código na biblioteca padrão. Afinal de contas, o código da biblioteca padrão já foi testado por milhares ou milhões de desenvolvedores que o utilizam todos os dias!

Resumo

Importe scripts com a palavra-chave `import`.
É possível importar de diferentes formas:

```
import modulo

import modulo as M

from modulo import X, Y, Z

from modulo import X as xxx

from modulo import *
```

Utilize funcionalidade da biblioteca padrão sempre que possível!

18. Aula extra – Compreensão de lista

Quando queremos filtrar valores em Python, é muito comum desenvolvermos um código como este:

```
valores = list(range(10))

maiores_que_cinco = []
for valor in valores:
    if valor > 5:
        maiores_que_cinco.append(valor)

print(maiores_que_cinco)
# output: [6, 7, 8, 9]
```

Podemos recriar esta lógica em uma única linha, usando uma **compreensão de lista**!

A estrutura de uma compreensão de lista

A estrutura de uma compreensão de lista tem o seguinte formato:

```
NOVA_LISTA = [RESULTADO para cada ELEMENTO em SEQUÊNCIA se CONDIÇÃO]
```

Pode parecer confuso. Mas pensarmos no código anterior, já tínhamos esta estrutura básica distribuída no bloco central:

```
NOVA_LISTA = []
para cada ELEMENTO em SEQUÊNCIA:
    se CONDIÇÃO:
        RESULTADO entra em NOVA_LISTA
```

Criando a compreensão de lista

Se reestruturarmos o código acima para uma compreensão de lista, temos:

```
valores = list(range(10))

maiores_que_cinco = [valor for valor in valores if valor > 5]

print(maiores_que_cinco)
# output: [6, 7, 8, 9]
```

Para deixar ainda mais claro, podemos até reordenar cada bloco dentro da lista:

```
valores = list(range(10))

maiores_que_cinco = [
    valor  # RESULTADO
    for valor in valores # para cada ELEMENTO em SEQUÊNCIA
    if valor > 5  # se CONDIÇÃO
]

print(maiores_que_cinco)
# output: [6, 7, 8, 9]
```

Também é possível modificar os valores na lista original de acordo com uma lógica qualquer:

```
valores = list(range(10))

resultado = [
    valor + 5  # RESULTADO
    for valor in valores # para cada ELEMENTO em SEQUÊNCIA
    if valor > 5  # se CONDIÇÃO
]

print(resultado)
# output: [11, 12, 13, 14]
```

Por que usar compreensão de lista?

Pode parecer apenas um detalhe, mas usar compreensão de lista facilita bastante a escrita de código. Ela é uma forma mais enxuta de criar listas a partir de outras listas, especialmente quando nos acostumamos com sua sintaxe. Ao invés de precisarmos "ocupar" o código com um for loop de diversas linhas, usamos a compreensão de lista para fazer a mesma tarefa em uma única linha.