BRUNNA CROCHES

PARTE 1

PYTHON

Guia iniciante: Python

ABOUT ME

Brunna Croches

Developer Full Stack

Brunna Croches é Dev FullStack, advogada e empreendedora.

Apaixonada por tech, vem adquirido vasto conhecimento na área.

Desenvolveu projetos ricos em diversidade, buscando captar as próximas tendências e necessidades do mercado.

Neste e-book você aprenderá ou recapitulará de forma simplificada e otimizados conceitos de programação feito por ela.

let's share

SUMMARY

1.1 Python tutorial: Tour pelalinguagem

HTML Content

Artigo

Python tutorial: Tour pela linguagem

Neste artigo apresentaremos as principais características da linguagem Python com exemplos de código e explicações.

198

198

Por que eu devo ler este artigo: **Python é fácil de aprender e cheio de recursos**. Para quem vem de outras linguagens baseadas em C, tais como o Java ou C#, alguns elementos do Python podem ser novidade. Aqui não há abertura e fechamento de chaves, é opcional o ponto e vírgula no final das expressões e a indentação é quem delimita os escopos como parte integrante da sintaxe. Este artigo tem por objetivo iluminar esses pontos para que a sua entrada no Python seja tranquila.

Ver mais +

Tire sua dúvida Marcado como lido Anotar

Artigos Python Python tutorial: Tour pela linguagem

A sintaxe de uma linguagem são as regras que nos dizem como escrever um código para que ele seja compilado sem erros. A seguir, veremos algumas dentre as **principais características da sintaxe do Python**, com exemplos de código e explicações. Uma vez que o Python é uma linguagem fácil de aprender, é possível que ao final desse artigo você já possa escrever os seus primeiros códigos com a mesma.

Aprenda como preparar o seu computador para programar em Python

Variável

Nas linguagens de programação, a variável é um local na memória que reservamos para armazenar dados. Para **criar uma variável no Python** basta informar nome e valor, como mostra o **Código 1**.

```
total_alunos = 10
print(total_alunos)
```

Código 1. Declaração de uma variável

Neste código declaramos a variável `total_alunos` na **Linha 1** e exibimos o seu valor na **Linha 2** com o comando print.
O nome de uma variável deve começar com uma letra ou sublinhado e pode ser seguido por letras, números e o caractere de sublinhado `_`. No **Código 1** temos um exemplo de variável que recebeu um nome composto por duas palavras separadas por um sublinhado. Outras formas de se escrever variáveis são erradas ou incomuns.
Vamos ver no **Código 2** um exemplo de código inválido na **linguagem Python**.

```
total_alunos = 10
_total_alunos = 10
1total_alunos = 10

print(total_alunos)
print(_total_alunos)
print(1total_alunos)
```

Código 2. Nomes de variáveis válidos e inválidos

Na **Linha 3**, temos uma variável que se inicia com um número, o que gera um erro de sintaxe. A **Linha 7** também contém um erro, pois está tentando exibir uma variável que tem um nome incorreto.
Python é uma linguagem case-sensitive, que faz diferenciação entre letras maiúsculas e minúsculas. Sendo assim, `total_alunos` e `Total_alunos` seriam nomes de variáveis diferentes. Vejamos um exemplo no **Código 3**.

```
total_aulas = 5
TOTAL_AULAS = 10

print(total_aulas)
print(TOTAL_AULAS)
```

Código 3. Nomes de variáveis diferenciados por letras maiúsculas e minúsculas

Nesse exemplo, `total_aulas` é uma variável diferente de `TOTAL_AULAS`.

Palavras-chave

Python, como toda linguagem, possui um conjunto de palavras-chave, que são termos que possuem significado especial para o compilador e que não podem utilizados como nomes. Abaixo temos uma lista destes termos:

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

O **Código 4** gera um erro de sintaxe, pois o nome da variável `for` está violando a regra de não usar palavras-chave.

```
for = 5 - 1
print(x)
```

Código 4. Nome de variável inválido porque é uma palavra-chave

Para resolver esse problema basta utilizar um nome que não seja uma palavra-reservada, como vemos no **Código 5**.

```
x = 5 - 1
print(x)
```

Código 5. Correção do código errado

Não se preocupe em decorar cada palavra-chave da linguagem agora. A maioria das IDEs com as quais você vai trabalhar te avisará quando você estiver utilizando alguma dessas palavras como nome de uma variável.

Não é necessário usar ponto vírgula no fim de um comando

Em boa parte das linguagens de programação as linhas de código devem terminar com um ponto e vírgula (`;`). No Python isso é opcional. Um caso onde o ponto e vírgula seria necessário é quando temos mais de um comando em uma mesma linha. O **Código 6** mostra como é isso na prática.

```
a = 1
b = 2
c = 3
print("o valor de a é ", a)
print("o valor de b é ", b); print("o valor de c é ", c)
```

Código 6. Mais de um comando em uma mesma linha separados por ponto e vírgula

Na **Linha 5**, usamos dois comandos numa mesma linha, e para isso utilizamos o ponto e vírgula. Acima podemos ver que no final do comando `print("o valor de b é ", b)` adicionamos um ponto e vírgula e no final do comando `print("o valor de c é ", c)` isso não foi necessário.

Indentação faz parte da sintaxe

Indentação é uma forma de arrumar o código, fazendo com que algumas linhas fiquem mais à direita que outras, à medida que adicionamos espaços em seu início. Para a maioria das linguagens a indentação não é obrigatória, mas no caso Python isso é diferente.
A indentação é uma característica importante no Python, pois além de promover a legibilidade é essencial para o bom funcionamento do código. Em outras palavras, se a indentação não estiver adequada o programa pode se comportar de forma inesperada ou até mesmo não compilar.
Por exemplo, no caso de um `if`, o que determina se um código está dentro da condicional é o fato dele ter sido indentado. O **Código 7** apresenta um exemplo onde o comando `print` só será executado se o valor da variável `x` for maior que 8.

```
x = 10

if x > 8:
   print("x é maior que 8")
```

Código 7. Comando print dentro de um if

Agora veja esse novo exemplo no qual o comando `print` será executado de qualquer forma por não estar indentado e, por isso, não estar contido no `if`.

```
x = 10

if x > 8:

print("x é maior que 8")
```

Código 8. Comando print fora de um if

No **Código 8** temos dois erros. O primeiro tem a ver com a forma como o código foi escrito, porque se se espera que o comando `print` seja executado apenas se `x` for maior que 8, ele deveria estar indentado.
Um outro erro está relacionado a estrutura de controle de fluxo `if`, que será vista em detalhes em um outro momento. Uma vez que o comando `print` não está indentado, o mesmo também não está contido no `if`, deixando-o vazio e isso vai causar um erro de sintaxe.

Parênteses são opcionais

Vale ressaltar que no Python não há a necessidade de fazer o uso de parêntesis para as estruturas de controle de fluxo como `if`, `for` e `while`. No PyCharm, isso será apontando como redundância, como podemos ver na **Figura 1**.

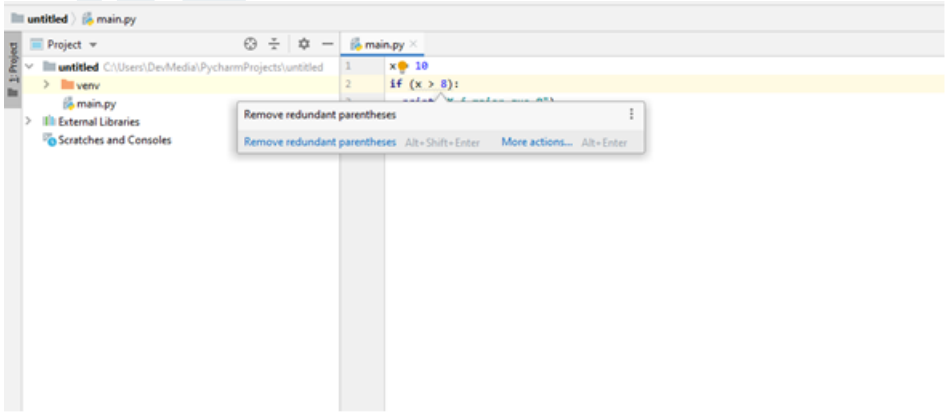


Figura 1. PyCharm alertando sobre o uso desnecessário de parênteses

Nos **Códigos 9 e 10** temos dois exemplos que mostram a diferença entre usar ou não os parênteses, mostrando que eles são opcionais.

```
x = 10
if (x > 8):
  print("x é maior que 8")
```

Código 9. Estrutura de controle de fluxo com parênteses

```
x = 10
if x > 8: # nessa linha removemos os parêntesis
  print("x é maior que 8")
```

Código 10. Estrutura de controle de fluxo sem parênteses

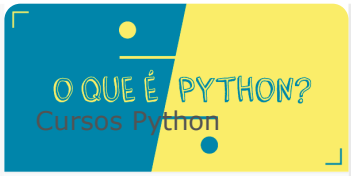
Ambos os códigos estão corretos. Dessa forma, o Python se diferencia da maior parte das linguagens, nas quais a sintaxe exige que os parêntesis sejam usados.

Conclusão

Conhecer a sintaxe do Python é muito importante para escrever programas que funcionem corretamente. Uma vez que essa é uma linguagem diferente das demais em muitos aspectos, muitos dos conceitos apresentados aqui podem não ser óbvios e representam armadilhas para os iniciantes. Com esse artigo em mãos, principalmente se você vem de uma linguagem baseada em C, o aprendizado do Python poderá ser mais agradável e ágil.

Confira também

Curso



Cursos

Curso

Tecnologias:

Python

Tire sua dúvida Marcado como lido Anotar

2.0 Python: Trabalhando com variáveis

HTML Content

Variáveis são um dos recursos mais básicos das linguagens de programação. Utilizadas para armazenar valores em memória, elas nos permitem gravar e ler esses dados com facilidade a partir de um nome definido por nós.
Neste documento aprenderemos a declarar e atribuir valores a **variáveis em Python**.

Tópicos

Declaração e atribuição

Assim como em outras linguagens, **o Python pode manipular variáveis básicas como strings** (palavras ou cadeias de caracteres), inteiros e reais (float). Para criá-las, basta utilizar um comando de atribuição, que define seu tipo e seu valor, conforme vemos no código abaixo:

```
mensagem = 'Exemplo de mensagem!'
n = 25
pi = 3.141592653589931
```

Nesse trecho foram feitas três atribuições. Na linha 1 foi atribuída uma string para uma nova variável chamada `mensagem`. Na linha 2 foi atribuído o valor inteiro 25 para `n` e na terceira linha foi atribuído um valor decimal para a variável `pi`.
Observe que não foi necessário fazer uma declaração explícita de cada variável, indicando o tipo ao qual ela pertence, pois isso é definido pelo valor que ela armazena, conforme vemos no código abaixo:

```
type (mensagem)
# <class 'str'>
type (n)
# <class 'int'>
type (pi)
# <class 'float'>
```

Nesse código, a linha 2 indica que a variável pertence à classe string. A linha 4 indica que a variável representa a classe de inteiros. Por sua vez, a sexta linha indica que a variável `pi` é do tipo float.

Para exibir o conteúdo dessas variáveis utilizamos o comando de impressão print, da seguinte forma:

```
print(mensagem)
# Exemplo de mensagem!
print (n)
# 25
print (pi)
# 3.141592653589931
```

Nomeando variáveis

As variáveis podem ser nomeadas conforme a vontade do programador, com nomes longos, contendo letras e números. No entanto, elas devem necessariamente começar com letras minúsculas.
Além dessa regra é importante também estar atento às palavras reservadas da linguagem (**Figura 1**), que não podem ser utilizadas para nomear variáveis.

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

Figura 1. Palavras reservadas da linguagem Python

2.1 Tipos de dados em Python: Numéricos

HTML Content

108

108

Por que eu devo ler este artigo: **Neste artigo começaremos a falar sobre tipos de dados na linguagem Python. Aqui abordaremos *booleanos*, *números* e *caracteres*.**

Tire sua dúvida Marcar como concluído Anotar

Artigos Python Tipos de dados em Python: Numéricos

Python é uma linguagem de tipagem dinâmica, em que o tipo da variável é definido de acordo com o valor que ela está recebendo.

Guia do artigo:

- **Numérico**
- **Complexos**
- **Booleanos (bool)**

Na **Listagem 1** vemos um exemplo de declaração de uma variável.

```
numero = 10
```

Listagem 1. Exemplo de declaração de variável

No código acima temos a variável `numero` recebendo o valor 10. Dado que esse valor é um inteiro, o tipo da variável será int.
O Python é uma linguagem fortemente tipada, entretanto permite que o tipo de uma variável seja alterado ao longo do código. Um exemplo disso pode ser visto na **Listagem 2**.

```
numero = "15"
numero = 15
```

Listagem 2. Exemplo de alteração de variável

Apesar dessa característica, o Python não converte automaticamente tipos de dados incompatíveis em operações. Por exemplo, em uma soma entre um número inteiro e uma string, o resultado será um erro, conforme demonstrado na **Listagem 3**.

```
numero_1 = 10
numero_2 = "15"

numero_3 = numero_1 + numero_2

print(numero_3)
```

Listagem 3. Exemplo de declaração de variável

Isso acontece porque o Python não vai converter a String "15" para que a mesma possa ser usada como um inteiro na operação de soma.
Já no exemplo abaixo, não teremos erro, pois o tipo Booleano é um subtipo do tipo numérico inteiro.

```
numero = True + 1
```

Listagem 4. Operação de Booleano com inteiro

Nota: Mais tarde veremos o tipo Booleano com mais detalhes nesse artigo

Para saber o tipo de uma variável usaremos a função `type()`, que retorna o tipo de qualquer variável que ela receba como parâmetro. Vamos usar o código anterior para ver na prática como isso funciona.

```
numero_1 = 10
numero_2 = "15"

print(type(numero_1))
print(type(numero_2))
```

Listagem 5. Exemplo de verificação do tipo das variáveis

O resultado do código da **Listagem 5** será Int para a variável numero_1 e String para a variável numero_2.

Tipo Numérico

Os tipos de dados usados para números se dividem em três conjuntos:

- Inteiros
- Números de ponto flutuante
- Complexos

Vejamos cada um deles.

Tipo Inteiro

Esse tipo representa os números inteiros positivos e negativos. Um exemplo desse tipo é mostrado no código da **Listagem 6**.

```
numero = 5 # Criação da variável numero

print(type(numero)) # Exibindo o tipo da variável
# que será int
```

Listagem 6. Exemplo de variável do tipo inteiro

O intervalo de valor desse tipo é ilimitado e está sujeito apenas à capacidade da memória.

Nota: Os números inteiros podem representados nos formatos hexadecimal, octal e binário. Abaixo um exemplo dessas representações:

- 0o1, 0o20, 0o377 # Representação Octal
- 0x01, 0x10, 0xFF # Representação Hexadecimal
- 0b10000, 0b11111111 # Representação binária

Ponto Flutuante (float)

Esses são números reais, que contém casas decimais. Por exemplo, a altura de uma pessoa ou o seu peso devem ser representadas usando números de ponto flutuante. Na **Listagem 7** temos um exemplo no qual criamos uma variável desse tipo.

```
altura = 1.79 # Declaração da variável altura

print(type(altura)) # Impressão do tipo da variável "altura"
```

Listagem 7. Exemplo de variável do tipo float

O código acima vai retornar o tipo float.
Outras linguagens possuem o tipo `double` para representar números de ponto flutuante. Em Python usamos float e, caso seja necessária uma precisão de casas decimais, podemos usar `Decimal`. A **Listagem 8** mostra a maneira de usar esse tipo.

```
from decimal import Decimal

numero = Decimal('0.1')
```

Listagem 8. Exemplo de precisão de casas decimas tipo float

Na **linha 1**, importamos o módulo `decimal`, para que esse tipo esteja disponível no código. E na **linha 3**, invocamos o construtor `Decimal()` para criar o objeto `numero`, contendo o número desejado. Note que o construtor de Decimal recebe uma string como parâmetro.

Complexos

Os números complexos são mais utilizados na engenharia e pesquisa. A parte imaginária do número recebe a letra `j` ou `J`. Na **Listagem 9** temos um exemplo de um número complexo.

```
numero = 1j * 1J

print(type(numero))
```

Listagem 9. Exemplo de variável do tipo complexos

A criação da variável fica na **linha 1**, enquanto a **linha 3** tem a impressão do tipo da variável. O resultado será o tipo `complex`.

Booleanos (bool)

O tipo Booleano é um subtipo Int e por isso pode ser representado pelos valores True e False. Quando uma variável é definida como `True`, seu valor é verdadeiro. E no caso de receber o valor `False`, seu valor é falso.
O exemplo da **Listagem 10** mostra a declaração de uma variável do tipo Booleano.

```
var1 = True
var2 = False

print(type(var1))
print(type(var2))
```

Listagem 10. Exemplo de variável do tipo booleano

As **linhas 1** e **2** criam variáveis do tipo Booleano, e nas **linhas 4** e **5**, o tipo delas é impresso.

Nota: Se uma variável for declarada como `True` ou `False`, entre aspas, ela não será do tipo Booleano, e sim do tipo String.

No Python, quando comparado com True o número 1 retorna `true` e todos os demais retornam `false`. Vejamos um exemplo em uma condicional na **Listagem 11**.

```
s = 1

if s == True:
    print("true")
else:
    print("false")
```

Listagem 11. Exemplo de condicional com do tipo booleano

O código acima terá como resultado true, pois o valor 1 vai funcionar de forma semelhante ao valor Booleano True. Caso a variável `s` tivesse o valor diferente de 1 o resultado seria false, como mostra a **Listagem 12**.

```
s = 2

if s == True:
    print("true")
else:
    print("false")
```

Listagem 12. Exemplo de variável do tipo inteiro

Apesar de ser possível utilizar números inteiros em expressões lógicas isso deve ser feito com muito cuidado.
Nesse artigo introduzimos os tipos de dados em Python, iniciando pelos numéricos. Vimos também que esse tipo é dividido em subtipos que podem ser usados para situações diferentes. Nos exemplos dados foi mostrado a forma de declará-los, o que nos permitirá fazer instruções com eles, como operações matemáticas.

2.3 Coleções no Python: Listas, Tuplas e Dicionários

HTML Content

Coleções no Python: Listas, Tuplas e Dicionários

Neste artigo abordaremos mais tipos de dados na linguagem Python. Aqui veremos os tipos que trabalham como coleções de dados, como as listas, tuplas e dicionários.

124

124

Por que eu devo ler este artigo: Aprenda neste artigo como usar e manipular coleções no Python, que são recursos para agrupar dados.

Tire sua dúvida Marcar como concluído Anotar

Artigos Python Coleções no Python: Listas, Tuplas e Dicionários

Guia do artigo:

As coleções permitem armazenar múltiplos itens dentro de uma única unidade, que funciona como um container. Neste artigo veremos três dentre as coleções mais utilizadas em Python, que são as listas, tuplas e dicionários.

Listas

Lista é uma coleção de valores indexada, em que cada valor é identificado por um índice. O primeiro item na lista está no índice 0, o segundo no índice 1 e assim por diante.
Para criar uma lista com elementos deve-se usar colchetes e adicionar os itens entre eles separados por vírgula, como mostra o **Código 1**.

```
programadores = ['Victor', 'Juliana', 'Samuel', 'Caio', 'Luana']
print(type(programadores)) # type 'list'
print(len(programadores)) # 5
print(programadores[4]) # Luana
```

Código 1. Lista no Python

Na linha 1 criamos uma variável do tipo lista chamada `programadores` contendo cinco nomes como os seus itens. Como já visto antes a função **type()** (Linha 2) traz o tipo de variável e **len()** (Linha 3) o tamanho do objeto. Observe que na linha 4 imprimimos um item da lista acessando o índice 4.
Outra característica das listas no Python é que elas são mutáveis, podendo ser alteradas depois de terem sido criadas. Em outras palavras, podemos adicionar, remover e até mesmo alterar os itens de uma lista.
No **Código 2** vemos um exemplo de como alterar uma lista.

```
programadores = ['Victor', 'Juliana', 'Samuel', 'Caio', 'Luana']
print(programadores) # ['Victor', 'Juliana', 'Samuel', 'Caio', 'Luana']

programadores[1] = 'Carolina'
print(programadores) # ['Victor', 'Carolina', 'Samuel', 'Caio', 'Luana']
```

Código 2. Alteração em lista

Primeiro, criamos uma lista contendo algumas strings e depois imprimimos o seu valor na linha 2. Após isso, acessamos um dos elementos dela e alteramos o valor dele para "Carolina" na linha 4.
Além de alterar elementos em listas, também é possível adicionar itens nelas, pois já vêm com uma coleção de métodos predefinidos que podem ser usados para manipular os objetos que ela contém. No caso de adicionar elementos, podemos usar o método `append()`, como veremos no **Código 3**
.

```
programadores = ['Victor', 'Juliana', 'Samuel', 'Caio', 'Luana']
print(programadores) # ['Victor', 'Juliana', 'Samuel', 'Caio', 'Luana
```

```
programadores.append('Renato')
print(programadores) # ['Victor', 'Juliana', 'Samuel', 'Caio', 'Luana', 'Renato']
```

Código 3. Adicionando item na lista

Na linha 1 criamos a lista e na linha 2 a imprimimos. Na linha 3 usamos o método `append()`, que adiciona elementos no final de uma lista. Quando imprimimos a lista na linha 4 vemos que ela exibirá o item adicionado na última posição. Há outra forma de adicionarmos itens na lista, que é através do método `insert()`. Ele usa dois parâmetros: o primeiro para indicar a posição da lista em que o elemento será inserido e o segundo para informar o item a ser adicionado na lista. O **Código 4** mostra como isso ocorre na prática.

```
programadores = ['Victor', 'Juliana', 'Samuel', 'Caio', 'Luana']
programadores.insert(1, 'Rafael')

print(programadores) # ['Victor', 'Rafael', 'Juliana', 'Samuel', 'Caio', 'Luana']
```

Código 4. Adicionando item na lista

No código acima adicionamos na posição 1 da lista a string 'Rafael'. O resultado que será gerado pela linha 4 será o seguinte:

```
['Victor', 'Rafael', 'Juliana', 'Samuel', 'Caio', 'Luana']
```

Assim como podemos adicionar itens em nossa lista, também podemos retirá-los. E para isso temos dois métodos: `remove()` para a remoção pelo valor informado no parâmetro, e `pop()` para remoção pelo índice do elemento na lista. Vejamos como isso funciona na lista de programadores que estamos usando como exemplo:

```
programadores = ['Victor', 'Juliana', 'Samuel', 'Caio', 'Luana']
print(programadores) # ['Victor', 'Juliana', 'Samuel', 'Caio', 'Luana']

programadores.remove('Victor')
print(programadores) # ['Juliana', 'Samuel', 'Caio', 'Luana']
```

Código 5. Removendo item da lista

A remoção é feita na linha 4 do **Código 5**, onde removemos um elemento pelo seu **valor**. Nesse caso, retiramos da lista `programadores` o item que tem a string 'Victor'. Isso fará com que a linha 5 imprima os seguintes valores:

```
['Juliana', 'Samuel', 'Caio', 'Luana']
```

Também é possível remover um item pelo seu índice, como podemos ver no **Código 6**.

```
programadores = ['Victor', 'Juliana', 'Samuel', 'Caio', 'Luana']
print(programadores) # ['Victor', 'Juliana', 'Samuel', 'Caio', 'Luana']

programadores.pop(0)
print(programadores) # ['Juliana', 'Samuel', 'Caio', 'Luana']
```

Código 6. Removendo item da lista

Usamos o método `pop()` para remover o primeiro item da lista `programadores`, gerando o seguinte resultado:

```
['Juliana', 'Samuel', 'Caio', 'Luana']
```

Note que nos códigos houve uma mudança na lista `programadores`.
No caso de tentarmos remover um item de uma posição inexistente, obtemos o erro `IndexError: pop index out of range` como veremos a seguir:

```
programadores = ['Victor', 'Juliana', 'Samuel', 'Caio', 'Luana']

programadores.pop(5)
```

Código 7. Erro ao remover item da lista

Na linha 3 do **Código 7**, o método `pop()` foi usado para remover um item da lista pelo seu índice. Entretanto, a posição usada como parâmetro não existe, visto que a última posição da lista é `4`, e foi usado o índice `5` para a remoção.
Também podemos obter um erro ao tentar remover um item com valor inexistente de uma lista, como é mostrado no **Código 8**.

```
programadores = ['Victor', 'Juliana', 'Samuel', 'Caio', 'Luana']

programadores.remove('Igor')
```

Código 8. Erro ao remover item da lista

Obtemos o erro `ValueError: list.remove(x): x not in list`, pois não existe na lista `programadores` um item com a string "Igor" que queremos excluir.
Outra característica das listas é que elas podem possuir diferentes tipos de elementos na sua composição. Isso quer dizer que podemos ter strings, booleanos, inteiros e outros tipos diferentes de objetos na mesma lista. O **Código 9** mostra como isso acontece na prática.

```
aluno = ['Murilo', 19, 1.79] # Nome, idade e altura

print(type(aluno)) # type 'list'
print(aluno) # ['Murilo', 19, 1.79]
```

Código 9. Diferentes tipos de objetos na lista

Essa característica heterogênea das listas é mostrada no exemplo acima, no qual criamos uma lista com variáveis dos tipos string, inteiro e float, que representam o nome, idade e altura, respectivamente.

Tuplas

Tupla é uma estrutura de dados semelhante a lista. Porém, ela tem a característica de ser imutável, ou seja, após uma tupla ser criada, ela não pode ser alterada. Vejamos o uso desse objeto no **Código 10**.

```
times_rj = ('Botafogo', 'Flamengo', 'Fluminense', 'Vasco')

print(type(times_rj)) # class='tuple'
print(times_rj) # ('Botafogo', 'Flamengo', 'Fluminense', 'Vasco')
```

Código 10. Tupla no Python

Acima vemos o uso de uma tupla no Python delimitada por parênteses na sua sintaxe. Na linha 1, a variável `times_rj` recebe quatro objetos do tipo string. Na linha 3, imprimimos o tipo da variável, que é uma tupla. E na linha 4, imprimimos também o conteúdo de `times_rj`, que dá o seguinte resultado:

```
('Botafogo', 'Flamengo', 'Fluminense', 'Vasco')
```

Assim como é feito nas listas, podemos acessar um determinado valor na tupla pelo seu índice, como no **Código 11**.

```
times_rj = ('Botafogo', 'Flamengo', 'Fluminense', 'Vasco')
```

```
print(times_rj[2]) # Fluminense
```

Código 11. Tupla no Python

Uma observação a ser feita no uso de uma tupla é que se ela tiver um único item, é necessário colocar uma vírgula depois dele, pois caso contrário, o objeto que vamos obter é uma string, porque o valor do item é do tipo string. No **Código 12**, vemos esse caso:

```
objeto_string = ('tesoura')
objeto_tupla = ('tesoura',)

print(type(objeto_string)) # class 'str'
print(type(objeto_tupla)) # class 'tuple'
```

Código 12. Tupla no Python

É possível também acessar os itens das tuplas por meio de seus índices, da mesma forma como é feito nas listas. No **Código 13** vemos como isso ocorre na prática.

```
vogais = ('a', 'e', 'i', 'o', 'u')

print(vogais[1]) # e
```

Código 13. Acesso de índice da tupla

O fato da tupla ser imutável faz com que os seus elementos não possam ser alterados depois dela já criada. Vamos usar a tupla `vogais` para mostrar um exemplo desse tipo. O **Código 14** exibirá o erro `TypeError: 'tuple' object does not support item assignment`.

```
vogais = ('a', 'e', 'i', 'o', 'u')

vogais[1] = 'E'
```

Código 14. Erro ao alterar valor da tupla

Veja que não é possível fazer alteração nas tuplas. Diferentemente do que acontece com as listas, não podemos trocar os elementos de um objeto do tipo tupla, pois se trata de uma sequência imutável.

Para fixar esse conceito, lembre-se que as tuplas e as strings são sequências imutáveis. Já as listas são sequências mutáveis. Isso está de acordo com a **documentação oficial** do Python.

As tuplas devem ser usadas em situações em que não haverá necessidade de adicionar, remover ou alterar elementos de um grupo de itens. Exemplos bons seriam os meses do ano, os dias da semana, as estações do ano etc. Nesses casos, não haverá mudança nesses itens (pelo menos isso é muito improvável).

Dicionários

Os dicionários representam coleções de dados que contém na sua estrutura um conjunto de `pares chave/valor`, nos quais cada chave individual tem um valor associado. Esse objeto representa a ideia de um mapa, que entendemos como uma coleção associativa desordenada. A associação nos dicionários é feita por meio de uma chave que faz referência a um valor. No **Código 15**, vemos a estrutura de um dicionário.

```
dados_cliente = {
    'Nome': 'Renan',
    'Endereco': 'Rua Cruzeiro do Sul',
    'Telefone': '982503645'
}

print(dados_cliente['Nome']) # Renan
```

Código 15. Dicionário no Python

A estrutura de um dicionário é delimitada por chaves, entre as quais ficam o conteúdo desse objeto. Veja que é criada a variável `dados_cliente`, à qual é atribuída uma coleção de dados que, nesse caso, trata-se de um dicionário. Na linha 7 do **Código 15**, imprimimos o conteúdo que é associado ao índice "Nome", trazendo o resultado Renan.

Nas listas e tuplas acessamos os dados por meio dos índices. Já nos dicionários, o acesso aos dados é feito por meio da chave associada a eles.

Para adicionar elementos num dicionário basta associar uma nova chave ao objeto e dar um valor a ser associado a ela. No **Código 16** vamos colocar a informação `Idade` em `dados_cliente`.

```
dados_cliente = {
    'Nome': 'Renan',
    'Endereco': 'Rua Cruzeiro do Sul',
    'Telefone': '982503645'
}

print(dados_cliente) # {'Nome': 'Renan', 'Endereco': 'Rua Cruzeiro do Sul',
```

```
    'Telefone': '982503645'}

dados_cliente['Idade'] = 40

print(dados_cliente) # {'Nome': 'Renan', 'Endereco': 'Rua Cruzeiro do Sul',
  'Telefone': '982503645', 'Idade': 40}
```

Código 16. Adicionando item no Dicionário

Para remover um item do dicionário, podemos usar o método `pop()`, como vemos no **Código 17**.

```
dados_cliente = {
    'Nome': 'Renan',
    'Endereco': 'Rua Cruzeiro do Sul',
    'Telefone': '982503645'
}

print(dados_cliente) # {'Nome': 'Renan', 'Endereco': 'Rua Cruzeiro do Sul',
  'Telefone': '982503645'}

dados_cliente.pop('Telefone', None)

print(dados_cliente) # {'Nome': 'Renan', 'Endereco': 'Rua Cruzeiro do Sul'}
```

Código 17. Removendo item no Dicionário

Na linha 9 do **Código 17** temos o uso do método `pop()`, usado para remover o item 'Telefone' do dicionário `dados_clientes`. Temos na chamada do método o parâmetro `None`, que é passado depois da chave a ser removida. O `None` serve para que a mensagem de erro `KeyError` não apareça devido a remoção de uma chave inexistente. Também poderíamos usar a palavra-chave `del`, que remove uma chave e o valor associado a ela no dicionário. Isso se faz por meio da passagem no parâmetro, como vemos no exemplo do **Código 18**.

```
dados_cliente = {
    'Nome': 'Renan',
    'Endereco': 'Rua Cruzeiro do Sul',
    'Telefone': '982503645'
}

print(dados_cliente) # {'Nome': 'Renan', 'Endereco': 'Rua Cruzeiro do Sul',
  'Telefone': '982503645'}

del dados_cliente['Telefone']

print(dados_cliente) # {'Nome': 'Renan', 'Endereco': 'Rua Cruzeiro do Sul'}
```

Código 18. Removendo item no Dicionário

Funções para coleções

O Python conta com funções úteis quando se trabalha com coleções. Vejamos algumas delas:

min() e max()

Veremos a seguir, as funções `min()` e `max()`. No **Código 19** temos um exemplo com elas.

```
numeros = [15, 5, 0, 20, 10]
nomes = ['Caio', 'Alex', 'Renata', 'Patrícia', 'Bruno']

print(min(numeros)) # 0
print(max(numeros)) # 20
print(min(nomes)) # Alex
print(max(nomes)) # Renata
```

Código 19. Funções min() e max()

Nesse código temos duas listas com os nomes `numeros` e `nomes`. A primeira lista trabalha com números, então a função `min()` retorna o menor valor dela, enquanto que a função `max()` retorna o maior valor. Já a segunda lista contém strings, o que faz com que as funções trabalhem com comparações alfabéticas. Portanto, nesse exemplo o menor valor é `Alex` e o maior `Renata`.

sum()

Para trabalhar com coleções na linguagem Python, temos também a função `sum()`, que é usada para retornar a soma de todos os elementos da coleção. No **Código 20**, temos um exemplo dela na prática:

```
numeros = [1, 3, 6]

print(sum(numeros)) # 10
```

Código 20. Função sum()

Como vemos no código acima, `sum()` retornou a soma dos itens da lista `numeros`. Essa função não trabalha com strings, pois não é um tipo suportado por ela. Caso fossem usadas strings, a mensagem de erro `TypeError: unsupported operand type(s) for +: 'int' and 'str'` seria exibida.

len()

A função `len()` é bastante usada em Python para retornar o tamanho de um objeto. Quando usada com coleções, retorna o total de itens que a coleção possui. Vemos isso no **Código 21**.

```
paises = ['Argentina', 'Brasil', 'Colômbia', 'Uruguai']

print(len(paises)) # 4
```

Código 21. Função len()

Essa função é de grande utilidade, pois pode ser usada em diversas situações, como nas estruturas condicionais e em laços de repetição por exemplo.

type()

Com a função `type()` podemos obter o tipo do objeto passado no parâmetro. No **Código 22** usamos essa função em algumas coleções.

```
professores = ['Carla', 'Daniel', 'Ingrid', 'Roberto']
estacoes = ('Primavera', 'Verão', 'Outono', 'Inverno')
cliente = {
    'Nome': 'Fábio Garcia',
    'email' : 'fabio_garcia_9@outlook.com'
}

print(type(professores)) # list
print(type(estacoes)) # tuple
print(type(cliente)) # dict
```

Código 22. Função type()

A execução do código acima nos traz uma lista, uma tupla e um dicionário, que são as coleções trabalhadas nesse artigo.

Conclusão

Neste artigo vimos como trabalhar com **coleções de dados no Python**, como as listas, tuplas e dicionários. Vimos suas características e alguns dos métodos que podemos usar para manipular esses tipos de objetos. Com eles podemos resolver uma ampla variedade de problemas em situações que precisamos estruturar os dados. Dependendo do contexto, uma determinada coleção é mais adequada para o uso.

CONTACT

Brunna Croches

Developer Full Stack

brunnacroches.dev

linkedin.com/brunnacroches

github.com/brunnacroches

@brunnacroches.dev

discord.com/brunnacroches

brunnacroches@gmail.com

let's share