

FORMAÇÃO EXPERT EM PYTHON

DEGRAU 1

APOSTILA COMPLETA

Primeiros passos na linguagem Python

Os principais conceitos abordados no **primeiro degrau** da Formação Expert em Python



Seja bem-vindo(a) à Formação Expert em Python da Empowerdata.

O objetivo dessa apostila é servir como um guia, auxiliando nos **temas mais importantes** para facilitar ainda mais o seu aprendizado.

Existem diversas linguagens de programação disponíveis, mas a que se destaca – *justamente pela facilidade* – é a Python.

Entrar nesse mundo do Python, é **fazer parte de uma comunidade altamente engajada**, que disponibiliza uma grande quantidade de bibliotecas com muitas aplicações.

De forma geral, a linguagem Python é utilizada para analisar dados, automatizar tarefas, desenvolver programas, dentre outras aplicações muito poderosas.

E a boa notícia é que **você vai aprender tudo isso** ao longo da sua jornada aqui conosco.

Então, vamos começar a colocar a mão na massa e criar projetos incríveis com o Python.

Função print()

A função **print()** é uma das funções mais utilizadas em python, ela serve para mostrar algo na tela. Pode ser uma mensagem, o valor de uma variável ou até uma combinação de texto com variável.

Sua estrutura é: **print(*var*)**

Exemplo

ENTRADA

```
print("Olá mundo!")
print(2+3)
```

SAÍDA

```
Olá mundo!
5
```

Exemplo

ENTRADA

```
nome = "Vinicius"
print(nome)
```

SAÍDA

```
Vinicius
```



Link dos códigos no Google Colab:

[Clique aqui para acessar](#)

Função input()

A função **input** serve para que o usuário envie dados para o programa enquanto ele está sendo executado.

Sua estrutura é: **variavel = input(*var*)**

Exemplo

ENTRADA

```
input("Digite o seu nome: ")
```

SAÍDA

```
Digite o seu nome: Vinicius  
'Vinicius'
```

Exemplo

ENTRADA

```
valor_a = input("Digite o valor de a: ")  
valor_b = input("Digite o valor de b: ")  
# como o input retorna uma string (variável do tipo texto), é necessário  
# converter esse texto para número  
print(float(valor_a)+float(valor_b))
```

SAÍDA

```
Digite o valor de a: 2  
Digite o valor de b: 3  
5
```



Link dos códigos no Google Colab:

[Clique aqui para acessar](#)

Operadores aritméticos

A Python utiliza a **mesma sintaxe da matemática** para operações aritméticas.

- Use **+** para realizar a **soma**
- Use **-** para realizar a **subtração**
- Use ***** para realizar a **multiplicação**
- Use **/** para realizar a **divisão**
- Use **//** para realizar a **divisão**, e só retornar a **parte inteira**
- Use **%** para obter o **módulo/resto da divisão**
- Use ****** para **exponenciação**

A ordem das operações também seguem as regras da matemática:

- 1 Parêntesis
- 2 Expoentes
- 3 Multiplicações e divisões
- 4 Somas e subtrações

Exemplo

ENTRADA

```
print('Soma 3+2:')
print(3+2)
print('Subtração 3-2:')
print(3-2)
print('Multiplicação 3*2:')
print(3*2)
print('Divisão 3/2:')
print(3/2)
print('Divisão, pegando só o valor inteiro 3//2:')
print(3//2)
print('Resto da divisão 3%2:')
print(3%2)
print('Exponenciação 3**2:')
print(3**2)
```

SAÍDA

```
Soma 3+2:
5
Subtração 3-2:
1
Multiplicação 3*2:
6
Divisão 3/2:
1.5
Divisão, pegando só o valor inteiro 3//2:
1
Resto da divisão 3%2:
1
Exponenciação 3**2:
9
```



[Link dos códigos no Google Colab:](#)

[Clique aqui para acessar](#)

Variáveis

Variável pode ser pensada como um **espaço na memória**, com objetivo de armazenar alguma informação. Esse espaço, a variável, tem um apelido/nome criado pela pessoa que está programando.

Para definir o apelido da variável, e criar a mesma, algumas **regras** devem ser seguidas:

Regras para nomear uma variável

- › Precisa **começar** com uma **letra** ou **underscore** ('_')
- › **Não** pode começar com um **número**
- › **Não** pode conter **caracteres especiais** (ex.: *, \$, %, &, @, etc)
- › Nomes de variáveis são case sensitive ('nome' é **diferente** de 'Nome')
- › Não pode ser uma **palavra reservada** da linguagem (ex.: False, True, None, for, etc)

As variáveis podem ser definidas pelos seus **tipos**:

Tipos de dados

- › **int** Tipo para **números inteiros**, como: 1 2 -5 0
- › **float** Tipo para **números decimais**, como: 1.5 9.873
- › **string** Tipo para **texto**, como: 'Vinicius' 'maçã' '1234'
- › **booleanos** Tipo **lógico** onde a variável é: **True** (verdadeiro) ou **False** (falso)

Você pode utilizar a **função type(*var*)** para avaliar o tipo de uma variável.



Link dos códigos no Google Colab:

[Clique aqui para acessar](#)

Exemplo

ENTRADA

```
nome = 'Vinicius'
print(nome)
print(type(nome))
```

SAÍDA

```
Vinicius
<class 'str'>
```

Exemplo

ENTRADA

```
valor = 1
print(valor)
print(type(valor))
```

SAÍDA

```
1
<class 'int'>
```

Operadores relacionais e lógicos

Operadores **relacionais** são operadores que permitem a comparação entre objetos, e **retornam os valores de True ou False**. Já os **operadores lógicos** permitem operações entre o True e False.

Os **operadores relacionais** são:

>	Maior	>	EXEMPLOS	3>5 (False)	5>3 (True)
>	Menor	<	EXEMPLOS	3<5 (True)	5<3 (False)
>	Igual	==	EXEMPLOS	3==5 (False)	5==5 (True)
>	Maior igual	>=	EXEMPLOS	3>=5 (False)	5>=5 (True)
>	Menor igual	<=	EXEMPLOS	3<=5 (True)	5<=3 (False)
>	Diferente	!=	EXEMPLOS	3!=5 (True)	5!=3 (False)

Os **operadores lógicos** são: **AND**, **OR** e **NOT**.

> Operador AND (E)

A	B	A E B
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

> Operador OR (OU)

A	B	A OU B
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

Exemplo

ENTRADA	print(9<6)
SAÍDA	False

Exemplo

ENTRADA	print(6<9)
SAÍDA	True

Exemplo

ENTRADA

```
print((9<6) or (5<9))
# o primeiro parênteses é False, e o segundo é True, como a operação é or,
# então False or True é True
```

SAÍDA

True



Link dos códigos no Google Colab:

[Clique aqui para acessar](#)

Tipagem dinâmica

Na linguagem Python, **não é necessário definir o tipo da variável** quando ela for criada, como é obrigatório em outras linguagens.

A Python é uma linguagem inteligente e **define o tipo da variável** automaticamente quando a criamos.

Exemplo

ENTRADA

```
print("A variavel inicia como str, com valor 'Vinicius'")
nome = 'Vinicius'
print(type(nome))
print('Agora a variavel eh int com valor 2')
nome = 2
print(type(nome))
```

SAÍDA

```
A variavel inicia como str, com valor 'Vinicius'
<class 'str'>
Agora a variavel eh int com valor 2
<class 'int'>
```



Link dos códigos no Google Colab:

[Clique aqui para acessar](#)

Strings

O que são strings?

Strings são um tipo de variável para armazenar **informações de texto**, como nomes, endereços, produtos, CPF. Para criar uma string em Python, é só utilizar **aspas simples** ou **aspas duplas**.

Strings são case sensitive: '**Vinicius**' é diferente de '**vinicius**'.

Podemos **concatenar** strings com o operador aritmético de **adição**. Além disso, strings são iteráveis - você pode acessar cada caractere da string separadamente, pela sua posição.

Exemplo

ENTRADA

```
print('Exemplos de strings:')
print('Vinicius', 'vinicius', 'Avenida Paulista', '123.123.123-12', '123')

print('Não são exemplos de strings: ')
print(2)
print(True)
```

SAÍDA

```
Exemplos de strings:
Vinicius vinicius Avenida Paulista 123.123.123-12 123
Não são exemplos de strings:
2
True
```

Exemplo

ENTRADA

```
# concatenando strings
print('Somando duas strings: ')
nome = 'Vinicius'
sobrenome = 'Rocha'
print(nome+sobrenome)

print(nome+' '+sobrenome)
```

SAÍDA

```
Somando duas strings:
ViniciusRocha
Vinicius Rocha
```

Exemplo

ENTRADA

```
# iterando strings
var = 'abcdef'

print('string: '+var)
print('Valor na posicao 0: '+var[0])
print('Valor na posicao 1: '+var[1])
print('Valor na posicao -1: '+var[-1])
```

SAÍDA

```
string: abcdef
Valor na posicao 0: a
Valor na posicao 1: b
Valor na posicao -1: f
```



[Link dos códigos no Google Colab:](#)

[Clique aqui para acessar](#)

Utilizando f-strings

Strings literais formadas, ou f-strings, são strings com letra f no início, e chaves {} para realizar a **concatenação de variáveis ou expressões**.

Para utilizar f-strings: `f'texto{expressão}'`

Exemplo

ENTRADA

```
nome = 'Ana'
texto = f'Olá {nome}!'
# a string acima é estática com 'Olá ', mas varia o nome da pessoa de acordo com
# o valor da variável nome
print(texto)
```

SAÍDA

```
Olá Ana!
```



[Link dos códigos no Google Colab:](#)

[Clique aqui para acessar](#)

Métodos para manipulação de strings

split()

A função **split** **divide uma string com base em algum parâmetro**. Se o parâmetro não for definido, será utilizado o **espaço**.

```
str.split(<parâmetro>)
```

Exemplo

ENTRADA

```
frase = 'Meu nome é Vinicius. Eu estou estudando a linguagem Python'
print(frase.split())
```

SAÍDA

```
['Meu', 'nome', 'é', 'Vinicius.', 'Eu', 'estou', 'estudando', 'a',
 'linguagem', 'Python']
```

strip()

A função **strip()** **remove os espaços em branco** no **início** ou no **final** de uma string.

```
str.strip()
```

Exemplo

ENTRADA

```
frase = '      Olá Mundo      '
print(f'Sem strip() >{ frase }<')
print(f'Com strip() >{ frase.strip() }<')
```

SAÍDA

```
Sem strip() >      Olá Mundo      <
Com strip() >Olá Mundo<
```



Link dos códigos no Google Colab:

Clique aqui para acessar

capitalize()

A função capitalize() retorna uma string com a **primeira letra maiúscula**, e todas as demais minúsculas.

```
str.capitalize()
```

Exemplo

ENTRADA

```
frase = 'Meu nome é Vinicius, e GoSto mUiTo de estudar Python'
print(f'Frase sem capitalize() >{ frase }<')
print(f'Frase com capitalize() >{ frase.capitalize() }<')
```

SAÍDA

```
Frase sem capitalize()>Meu nome é Vinicius, e GoSto mUiTo de estudar Python<
Frase com capitalize()>Meu nome é vinicius, e gosto muito de estudar python<
```

lower()

Retorna uma string com **todas as letras minúsculas**.

```
str.lower()
```

Exemplo

ENTRADA

```
frase = 'Meu nome é Vinicius, e GoSto mUiTo de estudar Python'
print(f'Frase sem lower() >{ frase }<')
print(f'Frase com lower() >{ frase.lower() }<')
```

SAÍDA

```
Frase sem lower() >Meu nome é Vinicius, e GoSto mUiTo de estudar Python<
Frase com lower() >meu nome é vinicius, e gosto muito de estudar python<
```

upper()

Retorna uma string com todas as **letras maiúsculas**.

```
str.upper()
```

Exemplo

ENTRADA

```
frase = 'Meu nome é Vinicius, e GoSto mUiTo de estudar Python'
print(f'Frase sem upper() >{ frase }<')
print(f'Frase com upper() >{ frase.upper() }<')
```

SAÍDA

```
Frase sem upper() >Meu nome é Vinicius, e GoSto mUiTo de estudar Python<
Frase com upper() >MEU NOME É VINICIUS, E GOSTO MUITO DE ESTUDAR PYTHON<
```

title()

Retorna uma string com a **primeira letra de cada palavra** maiúscula, e as outras minúsculas.

```
str.title()
```

Exemplo

ENTRADA

```
frase = 'Meu nome é Vinicius, e GoSto mUiTo de estudar Python'
print(f'Frase sem title() >{ frase }<')
print(f'Frase com title() >{ frase.title() }<')
```

SAÍDA

```
Frase sem title() >Meu nome é Vinicius, e GoSto mUiTo de estudar Python<
Frase com title() >Meu Nome É Vinicius, E Gosto Muito De Estudar Python<
```

Utilizando a técnica de slicing (fatiamento)

Com python, é possível acessar **partes da string com base no seu índice**. Para utilizar essa técnica, é necessário informar o **primeiro índice**, utilizar os **dois pontos “:”** e o número do **último índice** somado em 1.

Exemplo: string[1:4]

vai retornar uma string com **3 caracteres**, das posições 1/2/3.

Utilizando o **-1 depois dos dois pontos**, será retornado a string até a penúltima posição.

Para retornar até a **última posição**, é só deixar em branco após “:”.

Exemplo: string[4:]

Também é possível definir os passos que o índice deve pular para retornar para a tela. Ou seja, se você quer pegar uma string entre o **índice 4:10**, mas pulando **2 caracteres**, fica [4:10:2], e ele retorna os índice 4/6/8.

Exemplo

ENTRADA

```
frase = 'Meu nome é Vinicius, e gosto muito de estudar Python'
print(f'Frase completa >{ frase }<')
print(f'Frase fatiada >{ frase[3:7] }<')
# o índice 3 é referente ao espaço entre meu e nome, e o último caractere mostrado
# está no índice 6, letra m de nome.
```

SAÍDA

```
Frase completa >Meu nome é Vinicius, e gosto muito de estudar Python<
Frase fatiada > nome<
```

Exemplo

ENTRADA

```
frase = 'Meu nome é Vinicius, e gosto muito de estudar Python'
print(f'Frase completa >{ frase }<')
print(f'Frase fatiada >{ frase[3:] }<')
```

SAÍDA

```
Frase completa >Meu nome é Vinicius, e gosto muito de estudar Python<
Frase fatiada > nome é Vinicius, e gosto muito de estudar Python<
```

Exemplo

ENTRADA

```
frase = 'Meu nome é Vinicius, e gosto muito de estudar Python'
print(f'Frase completa >{ frase }<')
print(f'Frase fatiada >{ frase[4:10:2] }<')
```

SAÍDA

```
Frase completa >Meu nome é Vinicius, e gosto muito de estudar Python<
Frase fatiada >nm <
```



Link dos códigos no Google Colab:

[Clique aqui para acessar](#)

Estrutura de dados

Quando você precisa criar **20 variáveis similares** como nome1, nome2, nome3... Você precisa criar todas as 20 separadamente?

➤ **A resposta é NÃO, você pode utilizar a estrutura de dados.**

Criar variáveis utilizando estrutura de dados deixa seu código mais limpo, mais rápido e mais fácil de entender e realizar manutenção.

As estruturas de dados são principalmente:

- **Listas**
- **Dicionários**
- **Tuplas**
- **Conjuntos**

Todas essas estruturas são nativas do Python, então **não é necessário instalar** nada adicional.

Listas

Introdução

Uma lista é uma estrutura de dados composta por **itens organizados de forma linear**, na qual cada item pode ser acessado a partir de um **índice**.

Para criar uma lista, é utilizado os **colchetes**. Uma lista vazia fica:

```
lista = []
```

Uma lista com duas strings fica:

```
lista = ['Ana', 'Vinicius']
```

Listas, como strings, começam a **contagem do índice a partir do número 0**. Então o valor '**Ana**', está no **índice 0** dessa lista, e '**Vinicius**' está na **posição 1**. A estrutura para utilizar índice e slicing é igual ao da string.

Exemplo

ENTRADA

```
lista = ['Ana', 'Vinicius']
print(lista)
type(lista)
```

SAÍDA

```
['Ana', 'Vinicius']
list
```

Exemplo

ENTRADA

```
print(f'Está no índice 0 da lista >{lista[0]}<')
print(f'Está no índice 1 da lista >{lista[1]}<')
```

SAÍDA

```
Está no índice 0 da lista >Ana<
Está no índice 1 da lista >Vinicius<
```

Lista dentro de lista

Listas armazenam **qualquer tipo de variável**. Assim, também é possível armazenar uma lista dentro de outra.

```
lista = ['Ana', 'Python', 1, ['a', 'b', 'c']]
```

Como é possível acessar o valor 'b', da lista que está dentro da lista maior?

lista[3][1]

Primeiro, você precisa acessar a lista ['a', 'b', 'c'] no **índice 3**, para depois acessar o valor de 'b' no **índice 1**. Isso resultará na string 'b'.

Exemplo

ENTRADA

```
lista = ['Ana', 'Python', 1, ['a', 'b', 'c']]
print(lista)
```

SAÍDA

```
['Ana', 'Python', 1, ['a', 'b', 'c']]
```

Exemplo

ENTRADA

```
print(lista[3])
print(lista[3][1])
```

SAÍDA

```
['a', 'b', 'c']
c
```

Slicing - Fatiamento de listas

Igual ao visto nas string, é possível realizar o **slicing** (fatiamento) nas listas.

Lembrando que o **primeiro índice é inclusivo** (aparece no resultado), e o **último índice é exclusivo** (não aparece no resultado).

Exemplo

ENTRADA

```
lista = ['Ana', 'Python', 1, ['a', 'b', 'c']]
print(lista)
```

SAÍDA

```
['Ana', 'Python', 1, ['a', 'b', 'c']]
```

Exemplo

ENTRADA

```
print(lista[1:3])
```

SAÍDA

```
['Python', 1]
```



Link dos códigos no Google Colab:

[Clique aqui para acessar](#)

Operadores **in** ou **not in**

É possível utilizar os operadores **in** ou **not in** para **verificar se algum valor está dentro da lista**. O resultado será sempre um booleano (**True/False**).

Exemplo

ENTRADA

```
lista = ['Ana', 'Python', 1, ['a', 'b', 'c']]  
'Ana' in lista
```

SAÍDA

```
True
```

Manipulando elementos de uma lista

append()

O método **append()** **insere um elemento no final** da lista. A estrutura é:

```
lista.append(*valor_elemento*)
```

Exemplo

ENTRADA

```
lista = ['Ana', 'Python', 1, ['a', 'b', 'c']]  
print(lista)  
lista.append('oi')  
print(lista)
```

SAÍDA

```
['Ana', 'Python', 1, ['a', 'b', 'c']]  
['Ana', 'Python', 1, ['a', 'b', 'c'], 'oi']
```

insert()

O método **insert()** recebe **dois** argumentos: **o índice** onde o elemento vai ser inserido, e o **valor do elemento**.

➤ **Todos os valores da lista posteriores ao índice inserido são deslocados.**

```
lista.insert(*num_indice*, *valor_elemento*)
```

Exemplo

ENTRADA

```
lista = ['Ana', 'Python', 1, ['a', 'b', 'c']]
print(lista)
lista.insert(2, 'oi')
print(lista)
```

SAÍDA

```
['Ana', 'Python', 1, ['a', 'b', 'c']]
['Ana', 'Python', 'oi', 1, ['a', 'b', 'c']]
```

pop()

O método `pop()` **remove o último elemento** da lista:

`lista.pop()`

Exemplo

ENTRADA

```
lista = ['Ana', 'Python', 1, ['a', 'b', 'c']]
print(lista)
lista.pop()
print(lista)
```

SAÍDA

```
['Ana', 'Python', 1, ['a', 'b', 'c']]
['Ana', 'Python', 1]
```

remove()

O método `remove()` **remove um item** da lista, com **base no valor** do elemento.

`lista.remove(*valor_elemento*)`

Exemplo

ENTRADA

```
lista = ['Ana', 'Python', 1, ['a', 'b', 'c']]
print(lista)
lista.remove(1)
print(lista)
```

SAÍDA

```
[ 'Ana', 'Python', 1, ['a', 'b', 'c']]
[ 'Ana', 'Python', ['a', 'b', 'c']]
```

Exemplo**ENTRADA**

```
lista = [ 'Ana', 'Python', 1, ['a', 'b', 'c']]
print(lista)
lista.remove('Ana')
print(lista)
```

SAÍDA

```
[ 'Ana', 'Python', 1, ['a', 'b', 'c']]
['Python', 1, ['a', 'b', 'c']]
```

del

Outro método para **remover itens da lista** é pelo comando del. Ele não é um método das listas, mas é um método **nativo** da linguagem python.

```
del lista[*índice*]
```

Exemplo**ENTRADA**

```
lista = [ 'Ana', 'Python', 1, ['a', 'b', 'c']]
print(lista)
del lista[3]
print(lista)
```

SAÍDA

```
[ 'Ana', 'Python', 1, ['a', 'b', 'c']]
[ 'Ana', 'Python', 1]
```

Alterando elemento da lista

Para alterar um elemento da lista, só é preciso **referenciar a posição da lista**, e igualar ao novo valor.

```
lista[*índice*] = *novo_valor*
```

Exemplo

ENTRADA

```
lista = ['Ana', 'Python', 1, ['a', 'b', 'c']]
print(lista)
lista[0] = 'Vinicius'
print(lista)
```

SAÍDA

```
['Ana', 'Python', 1, ['a', 'b', 'c']]
['Vinicius', 'Python', 1, ['a', 'b', 'c']]
```

Dicionários

Introdução

Dicionários são **estruturas organizadas em pares** para armazenar informações. As informações são divididas em **chaves e valores**.

```
variavel = {} # dicionário vazio
```

```
variavel = {chave: valor} # dicionário com um par
```

São **mutáveis**, mas as **chaves precisam ser únicas**. Além disso, dicionários podem ser construídos com **diferentes tipos de dados** e com diferentes estruturas de dados.

Uma **chave** pode ser **string**, outra pode ser **float**, outra pode ser uma **lista**, etc.

Exemplo

ENTRADA

```
dados = {
    'nome': 'Vinicius',
    'estado': 'Rio de Janeiro',
    'linguagem': 'Python',
    12: 'Valor'}
```

SAÍDA

```
{'nome': 'Vinicius', 'estado': 'Rio de Janeiro', 'linguagem': 'Python',
 12: 'Valor'}
```



Link dos códigos no Google Colab:

[Clique aqui para acessar](#)

keys()

Método para **obter as chaves** de um dicionário:

dicionario.keys()

Exemplo

ENTRADA

```
dados = {  
    'nome': 'Vinicius',  
    'estado': 'Rio de Janeiro',  
    'linguagem': 'Python',  
    12: 'Valor'  
}  
chaves = dados.keys()  
print(chaves)
```

SAÍDA

dict_keys(['nome', 'estado', 'linguagem', 12])

values()

Método para **obter os valores** de um dicionário.

dicionario.values()

Exemplo

ENTRADA

```
dados = {  
    'nome': 'Vinicius',  
    'estado': 'Rio de Janeiro',  
    'linguagem': 'Python',  
    12: 'Valor'  
}  
valores = dados.values()  
print(valores)
```

SAÍDA

dict_values(['Vinicius', 'Rio de Janeiro', 'Python', 'Valor'])

items()

Método para **retornar uma lista** com a **combinação das chaves** com seus respectivos **valores**.

```
dicionario.items()
```

Exemplo

ENTRADA

```
dados = {
    'nome': 'Vinicius',
    'estado': 'Rio de Janeiro',
    'linguagem': 'Python',
    12: 'Valor'
}
itens = dados.items()
print(itens)
```

SAÍDA

```
dict_items([('nome', 'Vinicius'), ('estado', 'Rio de Janeiro'), ('linguagem', 'Python'), (12, 'Valor')])
```

Acessando os elementos de um dicionário

Para **acessar um elemento** de um dicionário, diferente da lista que precisa do número do índice, o dicionário **precisa da chave**.

Exemplo

ENTRADA

```
dados = {
    'nome': 'Vinicius',
    'estado': 'Rio de Janeiro',
    'linguagem': 'Python',
    12: 'Valor'
}
print(dados['estado'])
```

SAÍDA

```
Rio de Janeiro
```

Manipulando elementos de um dicionário

Inserindo elementos

Para adicionar um elemento no dicionário, só é necessário **informar a nova chave** e atribuir um **valor para essa chave**.

```
dicionario['nova_chave'] = 'valor'
```

Exemplo

ENTRADA

```
dados = {  
    'nome': 'Vinicius',  
    'estado': 'Rio de Janeiro',  
    'linguagem': 'Python',  
    12: 'Valor'  
}  
print(dados)  
dados['idade'] = 30  
print(dados)
```

SAÍDA

```
{'nome': 'Vinicius', 'estado': 'Rio de Janeiro', 'linguagem': 'Python', 12: 'Valor'}  
{'nome': 'Vinicius', 'estado': 'Rio de Janeiro', 'linguagem': 'Python', 12: 'Valor', 'idade': 30}
```

Atualizar os elementos

Para **atualizar um elemento** de um dicionário, é **igual a inserir um novo** elemento, mas com uma chave já existente.

Exemplo

ENTRADA

```
dados = {  
    'nome': 'Vinicius',  
    'estado': 'Rio de Janeiro',  
    'linguagem': 'Python',  
    12: 'Valor'  
}  
print(dados)  
dados['nome'] = 'Ana'  
print(dados)
```

SAÍDA

```
{'nome': 'Vinicius', 'estado': 'Rio de Janeiro', 'linguagem': 'Python', 12: 'Valor'}
{'nome': 'Ana', 'estado': 'Rio de Janeiro', 'linguagem': 'Python', 12: 'Valor'}
```

pop()

Método para **apagar a chave especificada**: dicionario.pop('chave_deletada')

Exemplo**ENTRADA**

```
dados = {
    'nome': 'Vinicius',
    'estado': 'Rio de Janeiro',
    'linguagem': 'Python',
    12: 'Valor'
}
print(dados)
dados.pop('linguagem')
print(dados)
```

SAÍDA

```
{'nome': 'Vinicius', 'estado': 'Rio de Janeiro', 'linguagem': 'Python', 12: 'Valor'}
{'nome': 'Vinicius', 'estado': 'Rio de Janeiro', 12: 'Valor'}
```

popitem()

Método para apagar a **última chave adicionada** no dicionario: dicionario.popitem()

Exemplo**ENTRADA**

```
dados = {
    'nome': 'Vinicius',
    'estado': 'Rio de Janeiro',
    'linguagem': 'Python',
    12: 'Valor'
}
print(dados)
dados.popitem()
print(dados)
```

SAÍDA

```
{'nome': 'Vinicius', 'estado': 'Rio de Janeiro', 'linguagem': 'Python', 12: 'Valor'}
{'nome': 'Vinicius', 'estado': 'Rio de Janeiro', 'linguagem': 'Python'}
```

del

Semelhante ao mostrado nas listas:

```
del dicionario['chave']
```

Exemplo

ENTRADA

```
dados = {
    'nome': 'Vinicius',
    'estado': 'Rio de Janeiro',
    'linguagem': 'Python',
    12: 'Valor'
}
print(dados)
del dados[12]
print(dados)
```

SAÍDA

```
{'nome': 'Vinicius', 'estado': 'Rio de Janeiro', 'linguagem': 'Python', 12: 'Valor'}
{'nome': 'Vinicius', 'estado': 'Rio de Janeiro', 'linguagem': 'Python'}
```

clear()

Método para **apagar todos os elementos** de um dicionário:

```
dicionario.clear()
```

Exemplo

ENTRADA

```
dados = {
    'nome': 'Vinicius',
    'estado': 'Rio de Janeiro',
    'linguagem': 'Python',
    12: 'Valor'
}
print(dados)
dados.clear()
print(dados)
```

SAÍDA

```
{'nome': 'Vinicius', 'estado': 'Rio de Janeiro', 'linguagem': 'Python', 12: 'Valor'}
{}
```

Tuplas

Introdução

Tuplas são uma estrutura de dados do Python, cujo objetivo (assim como das listas), é armazenar **diversos dados**. Algumas características das tuplas são:

- › É uma **sequência ordenada** de elementos
- › Seus elementos possuem **índice**
- › Podem ser **heterogêneas** (possuírem tipos diferentes de dados)
- › É **imutável**

```
tupla = (elemento1, elemento2, elemento3...)
```



IMPORTANTE

Uma das principais vantagens das tuplas é sobre a **alocação de memória**. Como a tupla é um **objeto imutável**, o Python pode **reduzir a quantidade de memória disponível** para essa estrutura de dados, em relação às listas ou dicionários.

Exemplo

ENTRADA

```
tupla = (1, 2, 'a', True, [1,2,3])
print(tupla)
type(tupla)
```

SAÍDA

```
(1, 2, 'a', True, [1, 2, 3])
tuple
```

Exemplo

ENTRADA

```
tupla = (1, 2, 'a', True, [1,2,3])
print(tupla[1])
print(tupla[-1])
```

SAÍDA

2

[1, 2, 3]

**Link dos códigos no Google Colab:****Clique aqui para acessar**

Sets - conjuntos

Introdução

Sets são estruturas de dados destinadas a armazenar um **conjunto de dados distintos**. Suas principais características são:

- › Elementos são **desordenados**
- › Elementos **não possuem índices**
- › **Não** podem conter **elementos repetidos**
- › Permitem **operações** de conjuntos
- › São **mutáveis**

```
sets = {elemento1, elemento2, elemento3}
```

**Link dos códigos no Google Colab:****Clique aqui para acessar**

Exemplo

ENTRADA

```
conjunto = { 1, 2, 3, 2, 3, 6, 4}
print(conjunto)
```

SAÍDA

{1, 2, 3, 4, 6}

Exemplo

ENTRADA

```
conjunto = { 1, '2', False, 'Vinicius'}
type(conjunto)
```

SAÍDA

```
set
```

Manipulando elementos de um conjunto

add()

Método para adicionar um elemento ao sets:

```
sets.add(elemento)
```

Exemplo

ENTRADA

```
conjunto = { 1, '2', False, 'Vinicius'}
conjunto.add(123)
print(conjunto)
```

SAÍDA

```
{False, 1, 'Vinicius', '2', 123}
```

remove()

```
sets.remove(elemento)
```

Método para **remover um elemento** do sets, com base no **valor do elemento**, e não do índice.

Exemplo

ENTRADA

```
conjunto = { 1, '2', False, 'Vinicius'}
conjunto.remove('2')
print(conjunto)
```

SAÍDA

```
{1, 'Vinicius', False}
```

Algumas operações com conjuntos

union()

Dois conjuntos podem ser **unidos** por meio do método `union()`.

```
conjunto1.union(conjunto2)
```

Exemplo

ENTRADA

```
conjunto1 = { 1, 2, 3, 4}
conjunto2 = {'a', 3, 7, 8}
conjunto_unido = conjunto1.union(conjunto2)
print(conjunto_unido)
```

SAÍDA

```
{1, 2, 3, 4, 7, 8, 'a'}
```

intersection()

Método para realizar a **intersecção entre dois conjuntos**. Ou seja, vai retornar somente os valores que estão presentes simultaneamente nos dois conjuntos.

```
conjunto1.intersection(conjunto2)
```

Exemplo

ENTRADA

```
conjunto1 = { 1, 2, 3, 4}
conjunto2 = {'a', 3, 7, 8, 2}
conjunto_intersection = conjunto1.intersection(conjunto2)
print(conjunto_intersection)
```

SAÍDA

```
{2, 3}
```

Estruturas de repetição

Estruturas de repetição são úteis para que determinado **bloco de comandos sejam executados por diversas vezes**, repetidamente. As principais estruturas de repetição são: **for e while**.

Essas duas estruturas precisam que uma **condição lógica seja verdadeira** para que o loop seja executado, e será executado **até que a condição vire falsa**.

Estrutura for

O **comando for** é a estrutura de repetição mais comum e utilizada em python.

A forma mais comum de utilizar o **comando for** é por meio da **navegação de uma variável em uma lista**. Para isso, é utilizado o comando **in** em conjunto com o **for**:

```
for *variavel* in *lista*:
```

➤ **O bloco de repetição está delimitado por meio da identação.**

Então, se você quer definir algo dentro do loop, **é necessário dar um tab** antes do comando. Quando um comando não tiver um tab, o loop estará finalizado.

Exemplo

ENTRADA

```
lista = ['python', 'Vinicius', 'Ana', False, 12]
for i in lista:
    print(i)
print(i)
# veja que a linha acima está sem tab e fora do loop, então ele repetirá
o último valor
```

SAÍDA

```
python
Vinicius
Ana
False
12
12
```



Link dos códigos no Google Colab:

[Clique aqui para acessar](#)

Em conjunto com o **for** e o **in**, é muito utilizado também o comando **range()**. Ele serve para **criar uma lista** de **0 até o número** passado **menos 1**.

Exemplo

ENTRADA

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

for i in lista:
    print(i)
```

SAÍDA

```
1
2
3
4
5
6
7
8
9
10
```

Exemplo

ENTRADA

```
for i in range(11):
    print(i)
```

SAÍDA

```
0
1
2
3
4
5
6
7
8
9
10
```

Lista de 0 até o número passado (11) menos 1, ou seja, de 0 a 10.

Exemplo

ENTRADA

```
frase = 'Meu nome é Vinicius'  
for i in frase:  
    print(i)
```

SAÍDA

M
e
u

n
o
m
e

é

V
i
n
i
c
i
u
s

Estrutura while

Enquanto na estrutura for nós temos a **quantidade de repetições já definida**, no while elas acontecem **até que uma condição de paradas seja atingida**.

```
while (condicao):
```

ATENÇÃO: Muito cuidado com loopings infinitos!



Link dos códigos no Google Colab:

Clique aqui para acessar

Exemplo

ENTRADA

```
contador = 0
while contador < 10:
    print(contador)
    contador = contador + 1
```

SAÍDA

```
1
2
3
4
5
6
7
8
9
```

Estruturas de condicionais

if else

Tem como objetivo **realizar um teste lógico** e permitir que um certo bloco de comandos seja executado ou não.

```
if condicao:
    *bloco de codigo*
else:
    *bloco se condicao for falso*
# if significa 'se'
# else significa 'senão'.
```

Esse comando pode ser pensado exatamente como comandos da vida real: se estiver sol, vou à praia, senão, vou assistir ao curso da empowerdata.

```
se estiver sol:
    vou à praia
senão:
    vou assistir ao curso da empowerdata
```



Link dos códigos no Google Colab:

[Clique aqui para acessar](#)

Exemplo

ENTRADA

```
idade = 18

if idade < 18:
    print('Menor de idade!')
else:
    print('Maior de idade!')
```

SAÍDA

Maior de idade!

Funções

Introdução

Funções são blocos de código que realizam uma certa tarefa, e podem ser reutilizáveis.

Evitam a repetição de código, deixam o código mais limpo e organizado, e facilitam a manutenção do programa. Podem ser:

- › Definidas pelo programador
- › Funções **internas do python** (ex: `print()`, `input()`)
- › Funções de **terceiros** (importado por **bibliotecas** externas)

Para criar uma funções, é necessário utilizar a palavra **def**, definir o nome da função e informar os **parâmetros** passados para a função (**se existirem**).

Se a função deve retornar algum valor, deve-se utilizar o **return** e a **variável/valor**.

```
def exemplo_funcao (parametro):
    *bloco de código*
    return valor
```

Todas as variáveis criadas dentro de uma função, só existem dentro do bloco da função.



BOAS PRÁTICAS PARA UM CÓDIGO LIMPO

- › Utilizar nomes que expliquem a **funcionalidade** da função;
- › **Não** criar funções que realizem **muitas tarefas**;
- › Caso a função for realizar várias tarefas distintas, procure fragmentar essas tarefas em funções menores.

Exemplo

ENTRADA

```
def mostrar_numero(num):
    print(f'Seu numero é: {num}')

def mostrar_intro():
    print('Apendendo funções!!!')

# para chamar a função do bloco acima, é só utilizar o nome da
# função e ()
mostrar_intro()
```

SAÍDA

Apendendo funções!!!

ENTRADA

mostrar_numero(2)

SAÍDA

Seu numero é: 2

Exemplo

ENTRADA

```
def mostrar_maior(num1, num2):
    if num1 > num2: # se o numero 1 for maior que o 2, a função já retorna o num1
        return num1
    else: # se o numero 2 for >=, a função retorna o num2
        return num2

maior_numero = mostrar_maior(3, 4)
print(f'O maior número é: {maior_numero}')
```

SAÍDA

O maior número é: 4



Link dos códigos no Google Colab:

Clique aqui para acessar

Exemplo

ENTRADA

```
def calculos(num1, num2):
    soma = num1 + num2
    subt = num1 - num2
    mult = num1 * num2
    divi = num1 / num2
    return soma, subt, mult, divi

resultados = calculos(3,5)
print(f'Soma: {resultados[0]}')
print(f'Subtração: {resultados[1]}')
print(f'Multiplicação: {resultados[2]}')
print(f'Divisão: {resultados[3]}')
```

SAÍDA

```
Soma: 8
Subtração: -2
Multiplicação: 15
Divisão: 0.6
```

FORMAÇÃO EXPERT EM PYTHON

ACOMPANHE MAIS CONTEÚDOS EM

 **Instagram @empowerdata**

 **Instagram @empowerpython**

 **Canal no Youtube**

