



PYTHON PARA DATA SCIENCE:

Primeiros Passos

SUMÁRIO

OLÁ, ESTUDANTE!	03
1 INTRODUÇÃO	04
1.1 Python	04
1.2 Google Colaboratoy	04
2 COMANDOS BÁSICOS	05
2.1 Comentários	05
2.2 print()	05
2.3 As variáveis	06
2.3.1 A criação	06
2.3.2 Tipos de variáveis	06
2.3.3 Operações com valores numéricos	07
2.4 Manipulação de strings	07
2.5 input()	09
2.5.1 Formatando a saída	10
2.5.1.1 Casas decimais	11
2.5.1.2 Caracteres especiais	11
3 ESTRUTURAS DE CONTROLE	13
3.1 Estruturas condicionais	13
3.1.1 Operadores em condicionais	14
3.1.2 Operador Ternário	14
3.2 Estruturas de repetição	15
3.2.1 Comandos relacionados aos laços	16
4 ESTRUTURA DE DADOS	17
4.1 Listas	17
4.1.1 Métodos com listas	18
4.2 Dicionários	19
4.2.1 Métodos com dicionários	20
CHEGAMOS AO FIM	22

OLÁ, ESTUDANTE!

Essa é a nossa apostila do curso introdução ao Python! Estamos empolgados em apresentar este material de estudo, que inclui os resumos dos conteúdos ensinados no nosso curso Python para Data Science, além de algumas novidades extras.

Como você já sabe, Python é uma das linguagens mais populares e versáteis para a área de Data Science e análise de dados. Nesta apostila, nós cobrimos todos os aspectos fundamentais desta linguagem abordados no curso, incluindo sua sintaxe, estruturas de controle e dados estruturados.

Fizemos um esforço para que você tenha tudo o que aprendeu em um único documento - e adicionamos um pouco mais. Cada tópico é acompanhado de exemplos claros e fáceis de seguir para que você possa consolidar seus conhecimentos.

O nosso objetivo é fornecer uma base sólida para você se aprofundar no mundo da programação com Python e se tornar especialista em Data Science. Nós acreditamos que esta apostila será uma ferramenta valiosa em sua jornada de aprendizado e esperamos que você aproveite ao máximo esta oportunidade.

Boa leitura!

1 INTRODUÇÃO

1.1 Python

Python é uma linguagem de programação altamente versátil e acessível, tornando-a uma das escolhas mais populares para iniciantes e programadores experientes. É uma linguagem de programação de alto nível, o que significa que ela permite que você se concentre na solução do problema ao invés de se preocupar com detalhes técnicos de baixo nível. Além disso, o uso de sintaxe clara e intuitiva, a semântica simples e a facilidade de leitura do código fazem com que Python seja fácil de aprender e de usar.

Outra vantagem de Python é a quantidade de recursos e bibliotecas disponíveis. Existem inúmeras bibliotecas e pacotes prontos para uso, que permitem que você adicione recursos avançados em seus projetos sem precisar escrever o código do zero. As bibliotecas mais populares incluem NumPy para cálculo científico, Pandas para análise de dados, Matplotlib para visualização de dados, entre outras.

Além disso, Python é uma linguagem multiplataforma, o que significa que o código escrito em Python pode ser executado em diversos sistemas operacionais, incluindo Windows, Mac e Linux. Isso é uma vantagem para os desenvolvedores, pois eles não precisam se preocupar com a compatibilidade de sistemas ao escrever seu código.

Algumas curiosidades sobre Python incluem:

- Python foi criada por Guido van Rossum em 1989, mas seu uso só se tornou amplo a partir dos anos 2000.
- Python é uma linguagem dinâmica, o que significa que ela permite a alteração do tipo de variáveis durante a execução do código.

- Python é usada em uma ampla gama de aplicações, incluindo ciência de dados, inteligência artificial, desenvolvimento de jogos, automação de tarefas, entre outros.

Em resumo, Python é uma linguagem de programação que oferece muitas vantagens para os programadores, incluindo a facilidade de aprendizado, a versatilidade, a disponibilidade de recursos e bibliotecas, e a opção de ser multiplataforma. Se você está procurando por uma linguagem de programação para aprender, Python é uma excelente escolha.

1.2 Google Colaboratoy

O Google Colab é uma plataforma poderosa e versátil que oferece aos usuários uma maneira fácil e eficiente de aprender e experimentar com Python. Além de ser uma ferramenta gratuita e fácil de usar, o Google Colab também oferece muitas vantagens para pessoas programadoras que desejam aprender Python.

Uma das principais vantagens de usar o Google Colab para aprender Python é que você pode acessá-lo de qualquer lugar, desde que você tenha acesso à Internet. Isso significa que você pode aproveitar seu tempo livre para estudar, mesmo quando estiver fora de casa ou do escritório. Além disso, como o Google Colab funciona diretamente no navegador, você não precisa se preocupar com a instalação de software adicional no seu computador.

Em resumo, o Google Colab é uma plataforma excelente para aqueles que desejam aprender Python, oferecendo facilidade de acesso, colaboração em tempo real, armazenamento seguro e recursos avançados.

Para acessar o Google Colab e fazer os seus projetos, você pode acessar esse [link](#). Para que você consiga usá-lo é necessário ter uma conta Gmail, pois todo notebook ficará armazenado no Google Drive.

2 COMANDOS BÁSICOS

Os comandos básicos em Python variam de acordo com o tipo de variável manipulada. Existem operações possíveis com valores numéricos e também manipulações possíveis para strings (valores textuais). Dentre os comandos básicos gerais podemos citar o `print()` e o `input()`, que conseguimos utilizar com as variáveis.

2.1 Comentários

Comentários são úteis quando precisamos descrever alguma etapa, função ou estrutura dentro do próprio código. Essa descrição precisa ser dada como uma anotação e, por isso, não pode ser considerada um código para ser interpretada dentro do ambiente.

Temos dois tipos de comentários em Python: comentários de uma linha e comentários de várias linhas.

Comentários de uma linha são feitos adicionando um símbolo de hashtag (#) no início de uma linha de código. Tudo o que vier depois do símbolo # em uma linha será considerado um comentário:

```
# Esse é um comentário de uma
linha
print(10) # Podemos colocar
outro comentário em uma linha
após um código
```

Já os comentários de várias linhas são feitos usando um conjunto de aspas triplas: `'''` ou `"""`. Tudo o que estiver entre as aspas triplas será considerado um comentário, mesmo que seja em várias linhas. Exemplo:

```
'''
Esse é um comentário
```

```
de várias linhas.
'''
```

Enquanto o texto estiver dentro das aspas, ele será ignorado durante a execução do código, seja ele uma linha de código ou um texto qualquer.

Durante esse resumo, você verá vários comentários nos códigos, descrevendo o código ou mostrando a saída de uma execução.

2.2 print()

A função `print()`, imprimir em inglês, tem por finalidade mostrar uma frase ou dados definidos por quem constrói o código. Sua sintaxe é simples e fácil de entender.

```
print(argumentos)
```

Os argumentos são os valores que desejamos imprimir na saída. Pode ser um texto, um número, entre outros valores. Os textos podem ser escritos usando aspas simples (') ou duplas ("), como mostrado abaixo:

```
# usando aspas simples
print('Olá mundo!')

# usando aspas duplas
print("Olá mundo!")
```

Com isso, conseguimos imprimir um texto ou um dado numérico através dessa função.

Podemos imprimir também vários tipos de valores no `print`, necessitando apenas separar os dados com vírgulas:

```
print('Estamos', 'no', 'capítulo',
2)
## Saída: Estamos no capítulo 2
```

2.3 As variáveis

2.3.1 A criação

As variáveis são um componente importante de qualquer linguagem de programação, pois permitem armazenar e manipular dados. Em Python, não é necessário definir o tipo de uma variável antes de atribuir um valor a ela, pois o tipo da variável é determinado automaticamente pelo valor atribuído. Isso é conhecido como tipagem dinâmica.

Para criar uma variável precisamos atribuir um valor à ela. Para isso, precisamos dar nome à variável, o operador de atribuição (=) e por fim, o valor que desejamos atribuir como mostrado na sintaxe abaixo:

```
nome_da_variável = valor
```

Assim, conseguimos definir quaisquer valores a variáveis. Além disso, também podemos trocar o valor de uma variável a qualquer momento, por outro dado.

Existem algumas regras que devem ser seguidas na criação do nome de uma variável.

- O nome da variável não pode começar com um número. Deve começar com uma letra ou o caractere `_`. Exemplos do que não fazer: `10_notas`, `2_nomes_casa`, etc.;
- Não pode ser usado espaços em branco no nome da variável. Exemplos do que não fazer: `Nome escola`, `notas estudantes`, etc.
- Não é permitido serem usados nomes de funções ou palavras-chave do Python. Exemplos do que não fazer: `print`, `type`, `True`, etc.
- Não podemos usar caracteres especiais, exceto o subtraço ("`-`"). Exemplos do que não fazer: `nota-1`, `nota+usada`, `contagem&soma`.

Outras especificações como a descrição da lista de funções e palavras-chave das regras de criação de nomes

para variáveis podem ser encontradas na [documentação](#).

Além disso, é recomendável que os nomes de variáveis sejam escritos com letras minúsculas e separados pelo caractere `_` para facilitar a leitura e manutenção do código.

2.3.2 Tipos de variáveis

Em Python, existem vários tipos de variáveis, incluindo inteiros, pontos flutuantes, strings e booleanos:

- **Inteiros (int):** números inteiros, como `-1`, `0`, `1`, `203`, etc;
- **Ponto flutuante (float):** números de ponto flutuante, como `10.0`, `0.5`, `-2.45`, etc;
- **Strings (str):** sequências imutáveis de caracteres, como `"olá mundo"`. As strings são denotadas por aspas simples ou duplas; e
- **Booleanos (bool):** valores lógicos verdadeiro ou falso, representam o `True` ou `False`.

Cada tipo de variável tem seus próprios métodos e propriedades específicas que podem ser usados para manipular e trabalhar com seus valores.

Podemos criar uma variável de cada tipo, seguindo a regra de atribuição:

```
# inteiro
inteiro = 10

# ponto flutuante (float)
ponto_flutuante = 35.82

# String
string = 'Brasil'

# Booleano
booleano = False
```

Podemos identificar o tipo de uma variável utilizando a função `type()`, seguindo a sintaxe:

```
type(variavel)
```

Como exemplo, é possível encontrar a definição de todos os tipos de variáveis que criamos.

```
print(type(inteiro))
print(type(ponto_flutuante))
print(type(string))
print(type(booleano))
''' Saída:
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
'''
```

2.3.3 Operações com valores numéricos

Com os valores numéricos em Python podemos realizar operações aritméticas, mas para isso é necessário fazer o uso dos operadores aritméticos. Abaixo, uma tabela descrevendo a função dos operadores e sua sintaxe de uso, sendo a e b variáveis numéricas:

Descrição	Operação
Soma	a+b
Subtração	a-b
Multiplicação	a*b
Divisão com resultado real	a/b
Divisão com resultado inteiro	a//b
Potência	a**b
Resto de divisão (Módulo)	a%b

Além deles, existem também funções que podem ser aplicadas a valores numéricos para executar outros cálculos:

- **Função abs:** retorna o valor absoluto da variável, ou seja, seu valor positivo.
 - Sintaxe: `abs(variavel)`
 - Exemplo:

```
abs(-13)
## Saída
# 13
```

- **Função round:** retorna o número arredondado com uma precisão definida *n* casas decimais após o ponto decimal. Se não especificarmos as casas, será retornado o inteiro mais próximo do ponto flutuante.

- Sintaxe: `round(variavel, numero_de_casas)`
- Exemplo:

```
round(14.3213,2)
## Saída
# 14.32
```

- **Função pow:** retorna a potenciação de uma base por seu expoente, funciona do mesmo modo que o operador `**`.

- Sintaxe: `pow(base, expoente)`
- Exemplo:

```
pow(3, 2)
## Saída
# 9
```

2.4 Manipulação de strings

As strings são usadas para armazenar valores de texto e podem ser criadas colocando aspas simples (') ou aspas duplas ("). Essas variáveis são dadas por uma sequência de caracteres podendo ser números, letras e até símbolos. Como no exemplo:

```
string_1 = 'isso é uma string'
string_2 = "isso também é uma string"
```

Assim como nas variáveis numéricas, é possível manipular as strings a partir de operações, funções e até métodos. Por serem imutáveis, sua manipulação resulta em cópias, ou seja, são criadas novas strings a partir de uma original que foi manipulada.

Começando por operações, é possível utilizar os operadores de adição (+) e multiplicação (*) para trabalhar e criar

novas strings. O operador de soma, permite unir duas ou mais strings e gerar uma string única. Exemplo:

```
ola = 'Olá '  
mundo = 'mundo!!'  
frase = ola+mundo  
print(frase)  
  
## Saída: Olá mundo!!
```

Já o operador de multiplicação vai repetir a string em uma quantidade de vezes igual à especificada. Para usar esse operador colocamos a string, depois o operador * e o número de vezes que desejamos a repetição. Vamos escrever a palavra “mano” repetindo a letra “o” por 5 vezes.

```
parte_1 = 'man'  
parte_2 = 'o' * 5  
palavra = parte_1 + parte_2  
print(palavra)  
  
## Saída: manoooo
```

Agora vamos falar de funções que podem ser úteis na manipulação de strings, como a `len()` e `str()`.

A função `len()` retorna o tamanho da string, ou seja, a quantidade de caracteres que ela tem.

```
frase = 'o rato roeu a roupa do  
rei de Roma'  
print(len(frase))  
  
## Saída: 34
```

Já a função `str()` retorna a representação de uma string para uma entrada.

```
ano = str(2023)  
ano  
  
## Saída: '2023'
```

É possível trabalhar com diversos **métodos** em strings. Métodos são funções que são associadas a objetos em Python. Eles são usados para, de maneira fácil e consistente, realizar ações ou operações em um objeto e para obter informações sobre o objeto. Por essa razão, os métodos são uma parte importante da programação em Python.

Métodos podem ser executados ao definirmos um objeto seguindo a seguinte estrutura:

```
objeto.metodo()
```

Existem também os *atributos* que são declarados da mesma forma que os métodos, no entanto, não necessitam dos `()`. É preciso verificar a documentação de cada caso.

Agora vamos verificar alguns métodos com strings. Considerando que “string” é o seguinte texto:

```
string = 'o rato roeu a roupa do  
rei de Roma'
```

Vejamos alguns métodos que conseguimos utilizar com qualquer variável do tipo `str`:

- **`string.upper()`**: converte uma string para maiúsculas.
 - Saída: 'O RATO ROEU A ROUPA DO REI DE ROMA'
- **`string.lower()`**: converte uma string para minúsculas.
 - Saída: 'o rato roeu a roupa do rei de roma'
- **`string.capitalize()`**: coloca a primeira letra de uma string em maiúscula e as restantes em minúsculas.
 - Saída: 'O rato roeu a roupa do rei de roma'
- **`string.replace(antigo_valor, novo_valor)`**: retorna uma cópia da string com a substituição das ocorrências. Exemplo: `string.replace('r', 'T')`.

- Saída: 'o Tato Toeu a Toupa do Tei de Roma'
- **string.find(dado):** retorna o índice da primeira ocorrência de um texto em na string. Exemplo, vamos encontrar o local da primeira aparição de 't' com, `string.find('t')`.
 - Saída: 4
- **string.strip():** retorna uma cópia da string original sem espaços desnecessários no início e no final. Com o texto, ' Olá! ', podemos aplicar o strip e obteremos a seguinte saída:
 - Saída: 'Olá!'
- **string.title():** retorna uma cópia da string original com a primeira letra de cada palavra em maiúsculas.
 - Saída: 'O Rato Roeu A Roupa Do Rei De Roma'
- **string.count(string):** retorna o número de vezes que um determinado valor aparece na string original. Exemplo: `string.count('r')`.
 - Saída: 4
- **string.isupper():** retorna True se todas as letras na string original estiverem em maiúsculas.
 - Saída: False
- **string.islower():** retorna True se todas as letras na string original estiverem em minúsculas.
 - Saída: False

Lembrando que todos esses métodos retornam novos valores, não alteram a string original. Para ser feita a alteração é preciso atribuir o resultado do método na mesma string. Por exemplo:

```
string = 'o rato roeu a roupa do rei de Roma'
print(string)
## 1º Saída: o rato roeu a roupa do rei de Roma

string = string.capitalize()
print(string)
## 2º Saída: O rato roeu a roupa do rei de roma
```

2.5 input()

A função `input()` permite a quem programa receber dados da pessoa usuária. É usado para ler e retornar uma entrada digitada pelo usuário como string. A sintaxe da função `input` é a seguinte:

```
input('string opcional')
```

A *string opcional* é exibida para o usuário na tela antes da entrada de dados. É uma boa prática incluir ela para orientar o usuário sobre o que ele deve digitar. Como exemplo, podemos coletar um dado de texto e mostrá-lo ao usuário com `print`.

```
nome = input('Digite seu nome: ')
print('Seu nome é:', nome)
```

```
## Saída:
#Digite seu nome: Mirla
#Seu nome é: Mirla
```

A variável `nome` é uma string, pois a função `input` apenas retorna strings. Para receber outros valores é necessário fazer a conversão de valores com as funções de conversão:

- **int(dado):** converte o dado para o tipo inteiro.
- **float(dado):** converte o dado para o tipo ponto flutuante (float).
- **str(dado):** converte o dado para o tipo string.
- **bool(dado):** converte o dado para o tipo booleano.

Desse modo, podemos receber os dados em strings e transformá-los para o tipo de dado que precisamos. Como exemplo, vamos construir um algoritmo somador:

```
# Nesse código vamos somar dois números inteiros
# A função int vai converter a saída de input para um valor inteiro
```

```
num_1 = int(input('Digite o  
primeiro número: '))  
num_2 = int(input('Digite o  
segundo número: '))  
soma = num_1 + num_2  
  
print('Resultado da soma:',soma)
```

```
## Saída:  
# Digite o primeiro número: 2  
# Digite o segundo número: 3  
# Resultado da soma: 5
```

O mesmo conseguimos fazer para as demais funções de conversão.

2.5.1 Formatando a saída

Conseguimos visualizar o resultado de variáveis dentro de strings, bem como imprimir o texto final em um print. Existem várias maneiras de formatar os dados mostrados dentro de um `print`, entre elas a formatação f-string, usando o operador de formatação `%`, ou com método `format`.

Para utilizar a formatação **f-string** (ou formatação de string), colocamos um `f` antes da criação da string e as variáveis entre chaves `{}`. Exemplo:

```
nome = "Ana Maria"  
idade = 17  
print(f"O nome da aluna é {nome}  
e sua idade é {idade} anos.")  
## Saída: O nome da aluna é Ana  
Maria e sua idade é 17 anos.
```

O **operador de formatação** permite a inserção de variáveis em pontos específicos na string com o operador `%`. Ele precisa ser acompanhado de uma palavra-chave para cada tipo de variável que se deseja adicionar. Seguindo a tabela abaixo:

Tipo de variável	Palavra-chave
string	<code>%s</code>
inteiro	<code>%d</code>
float	<code>%f</code>

caractere	<code>%c</code>
-----------	-----------------

A sintaxe consiste na adição no operador ao ponto desejado do texto. Finalizada a escrita do texto que se deseja exibir, o símbolo `%` é adicionado, com a especificação da variável entre parênteses. Exemplo:

```
nome_aluno = 'Fabricio Daniel'  
print('Nome do aluno: %s'  
%(nome_aluno))  
## Saída: Nome do aluno:  
Fabricio Daniel
```

Caso tenha mais de uma variável, devemos ordená-las conforme o surgimento delas no texto e separá-las por vírgula.

```
nome_aluno = 'Fabricio Daniel'  
idade_aluno = 15  
media_aluno = 8.45  
print('Nome do aluno é %s, ele  
tem %d anos e sua média é %f.'  
%(nome_aluno, idade_aluno,  
media_aluno))  
## Saída: Nome do aluno é  
Fabricio Daniel, ele tem 15 anos  
e sua média é 8.450000.
```

Os operadores de formatação de strings com `%` não funcionam diretamente com valores booleanos. Uma maneira de lidar com isso é convertendo o valor booleano para uma string antes de usá-lo na formatação com a função `str()`.

É possível também usar o método `format()` para fazer a formatação de strings. Ele é mais flexível e permite passar as variáveis diretamente dentro da string, sem a necessidade do operador `%`. Pelo contrário, os marcadores são apenas as `{}`. Exemplo:

```
nome_aluno = 'Fabricio Daniel'  
idade_aluno = 15
```

```
media_aluno = 8.45
print('Nome do aluno é {}, ele
tem {} anos e sua média é {}'.
.format(nome_aluno, idade_aluno,
media_aluno))
```

```
## Saída: Nome do aluno é
Fabricio Daniel, ele tem 15 anos
e sua média é 8.45.
```

2.5.1.1 Casas decimais

Quando trabalhamos com pontos flutuantes (float), podemos determinar a quantidade de casas decimais após a vírgula em todas as formatações de saída de texto.

Com a formatação f-string, usamos a sintaxe `:.xf` após especificar a variável, sendo `x` o número de casas decimais desejadas:

```
ponto_flutuante = 23.458012
print(f'Limitando as casas
decimais {ponto_flutuante:.2f}')
```

```
## Saída:
# Limitando as casas decimais
23.46
```

Já com a formação do operador `%`, a sintaxe é `%.xf`:

```
ponto_flutuante = 23.458012
print('Limitando as casas
decimais
%.3f'%(ponto_flutuante))
```

```
## Saída:
# Limitando as casas decimais
23.458
```

Por fim, com o método `format()`, a sintaxe para limitar as casas é `{:.xf}`:

```
ponto_flutuante = 23.458012
print('Limitando as casas
```

```
decimais
{:.2f}'.format(ponto_flutuante))
```

```
## Saída:
# Limitando as casas decimais
23.46
```

2.5.1.2 Caracteres especiais

Caracteres especiais são usados para representar ações especiais ou caracteres que não podem ser digitados diretamente, como o Enter e a tabulação.

- `'\n'` é o caractere de nova linha, usado para pular uma linha no texto (função do Enter). Exemplo:

```
print("Precisamos de dedicação
e paciência,\nPara ver o fruto
amadurecer.")
```

```
## Saída:
# Precisamos de dedicação e
paciência,
# Para ver o fruto amadurecer.
```

- `'\t'` é o caractere de tabulação, usado para adicionar um espaço de tabulação no texto. Exemplo:

```
print('Quantidade\tQualidade\n
5 amostras\tAlta\n3
amostras\tBaixa')
```

```
## Saída:
# Quantidade    Qualidade
# 5 amostras    Alta
# 3 amostras    Baixa
```

- `'\\'` é usado para imprimir uma única barra invertida. Caso não seja usada a dupla barra invertida, o código poderá resultar em erro ou em um resultado inesperado, pois o Python considera a `\` um chamado para um caractere especial. Para garantir que não ocorram erros, usamos esta sintaxe. Exemplo:

```
print("Caminho do arquivo:  
C:\\arquivos\\documento.csv")
```

```
## Saída:  
# Caminho do arquivo:  
C:\arquivos\documento.csv
```

- `"\""` é usado para imprimir aspas duplas quando estamos trabalhando com uma string criada a partir de aspas duplas `"`. Caso seja uma string criada por aspas simples `'`, isso não é necessário. Um exemplo:

```
print("Ouvi uma vez \"Os  
frutos do conhecimento são os  
mais doces e duradouros de  
todos.\"")
```

```
## Saída:  
# Ouvi uma vez "Os frutos do  
conhecimento são os mais doces  
e duradouros de todos."
```

- `'\"'` é usado para imprimir aspas simples quando estamos trabalhando com uma string criada a partir de aspas simples `'`. Caso seja uma string criada por aspas duplas `"`, isso não é necessário. Exemplo:

```
print('Minha professora uma  
vez disse: \'Estudar é a chave  
do sucesso.\\' ')
```

```
## Saída:  
# Minha professora uma vez  
disse: 'Estudar é a chave do  
sucesso.'
```

3 ESTRUTURAS DE CONTROLE

Entre as estruturas de controle estão as estruturas condicionais e as estruturas de repetição.

3.1 Estruturas condicionais

Nas estruturas condicionais temos o `if`, o `else` e o `elif`.

O `if` é uma palavra-chave em Python que significa “se”. Ele é usado para formar uma estrutura condicional, que permite que você verifique se uma determinada condição é verdadeira ou falsa e, em seguida, execute um bloco de código específico, se a verificação for verdadeira. A sintaxe para usar o `if` é:

```
if condição:
    # faça algo
```

Podemos montar um exemplo que identifica se um dado número é maior que 5:

```
num = int(input('Digite um
número: '))
if num>5:
    print('O número é maior que
5')

## Saída:
# Digite um número: 8
# O número é maior que 5
```

Já o `else` é uma estrutura opcional usada em conjunto com o `if` para formar uma estrutura condicional. O `else` é uma palavra-chave para “senão” e é executado quando a condição especificada na estrutura condicional anterior não for

verdadeira (`False`). A sintaxe para usar o `else` é:

```
if condição:
    # código caso seja verdade
else:
    # código caso seja falso
```

Podemos montar um exemplo que identifica se um dado número é maior ou menor que 5:

```
num = int(input('Digite um
número: '))
if num>5:
    print('O número é maior que
5')
else:
    print('O número é menor que
5')

## Saída:
# Digite um número: 3
# O número é menor que 5
```

Por fim, temos o `elif`, uma palavra-chave em Python que significa “senão, se” e pode ser considerado uma união do `else` com o `if`. Ela é usada em conjunto com a palavra-chave `if` para formar uma estrutura condicional encadeada. Sua sintaxe é dada pela seguinte estrutura:

```
if condição1:
    # faça algo
elif condição2:
    # faça outra coisa
elif condição3:
    # faça mais alguma coisa
```

O `elif` permite encadear condições. Se a primeira condição for avaliada como `False`, o interpretador Python avaliará a próxima condição no `elif`, e assim por diante. Isso continuará até que uma condição seja avaliada como `True`, ou nenhuma das condições sejam verdadeiras e sejam ignorados os blocos de código.

Um exemplo de código com `elif` é a estrutura para verificar, dados dois números, qual é o maior entre eles ou se ambos são iguais:

```
# Coletar os números
num1 = float(input('Digite o 1º
número: '))
num2 = float(input('Digite o 2º
número: '))

# Comparamos ambos os números e
descobrimos qual é o maior
if num1 > num2:
    print(f'O primeiro número é
maior: {num1}')
elif num2 > num1:
    print(f'O segundo número é
maior: {num2}')
else: # Caso os números sejam
iguais
    print('Os dois números são
iguais.')
```

3.1.1 Operadores em condicionais

Para formar uma expressão lógica podemos fazer o uso de operadores relacionais e operadores lógicos.

Os **operadores relacionais** são símbolos utilizados com objetivo de comparar valores ou expressões e verificar a relação entre eles. Vejamos alguns deles:

- **Maior que (>):** verifica se a primeira expressão é maior que a segunda.
- **Menor que (<):** verifica se a primeira expressão é menor que a segunda.
- **Maior ou igual a (>=):** verifica se a primeira expressão é maior ou igual à segunda.
- **Menor ou igual a (<=):** verifica se a primeira expressão é menor ou igual à segunda.
- **Igual a (==):** verifica se duas expressões são iguais.

- **Diferente de (!=):** verifica se duas expressões são diferentes.

Estes operadores retornam um valor booleano (**True** ou **False**) baseado na comparação entre os valores ou expressões.

Os **operadores lógicos** são símbolos utilizados para realizar operações lógicas, entre valores, podendo retornar **True** ou **False**. Os operadores lógicos são: **and**, **or** e **not**. De modo que,

- **and** retorna verdadeiro se ambas as expressões lógicas forem verdadeiras;
- **or** retorna verdadeiro se pelo menos uma das expressões lógicas for verdadeira; e
- **not** inverte o valor lógico da expressão, ou seja, se a expressão era **True**, ele retorna **False**, e vice-versa.

3.1.2 Operador Ternário

Podemos compactar o resultado de uma condição *if-else* em uma única linha. É uma alternativa útil para codificação mais simples e clara, especialmente quando se trata de uma condição simples. A sintaxe do operador ternário é a seguinte:

```
valor_caso_verdade if condição
else valor_caso_falso
```

Seu funcionamento se dá de modo que a condição é avaliada e, se for verdadeira, o `valor_caso_verdade` é retornado, caso contrário, o `valor_caso_falso` é retornado. Em outras palavras, é uma forma de atribuir valores a variáveis com base em uma condição.

Um exemplo simples é a verificação se um número é positivo ou negativo.

```
num = -5
resultado = 'positivo' if num >=
0 else 'negativo'
print(resultado)
```



```
## Saída:  
# negativo
```

Entendendo o código: o operador ternário testa a condição `num >= 0`, se for verdadeira, o valor de `'positivo'` é atribuído a resultado, caso contrário, o valor de `'negativo'` é atribuído a resultado.

3.2 Estruturas de repetição

Nas estruturas de repetição temos o `while` e o `for`.

O `while` é uma palavra-chave em Python que significa “*enquanto*”. Ela permite executar um bloco de código repetidamente enquanto uma determinada condição é verdadeira. Sua sintaxe é dada da seguinte forma:

```
while condição:  
    # bloco de código
```

O bloco de código dentro do laço `while` será executado repetidamente enquanto a condição for avaliada como `True`. Quando a condição for avaliada como `False`, o laço será interrompido e o programa continuará a executar o código depois do laço.

Um exemplo muito comum de uso do `while` é construindo um contador, um projeto que permite uma variável numérica ser incrementada 1 a 1, como uma contagem. Vamos fazer o contador de 1 a 10:

```
# inicializamos a variável  
contadora em 1  
# essa variável será  
incrementada  
contadora = 0  
# o while irá repetir enquanto a  
# contadora não for maior que 10  
while contadora < 10:  
    # incrementamos seu valor em 1
```

```
# a cada iteração  
contadora += 1  
print(contadora)
```

```
## Saída:  
# 1  
# 2  
# 3  
# 4  
# 5  
# 6  
# 7  
# 8  
# 9  
# 10
```

O incremento de `contadora` é dado por um operador de atribuição `+=`. Em Python temos vários tipos de operadores de atribuição como pode ser observado na tabela abaixo:

Sintaxe do operador	Descrição
<code>a = b</code>	Atribui o valor de <code>b</code> em <code>a</code>
<code>a += b</code>	Soma o valor de <code>b</code> na variável <code>a</code>
<code>a -= b</code>	Subtrai o valor de <code>b</code> na variável <code>a</code>
<code>a *= b</code>	Multiplica o valor de <code>b</code> por <code>a</code>
<code>a /= b</code>	Divide o valor de <code>b</code> por <code>a</code>
<code>a //= b</code>	Realiza divisão inteira da variável <code>b</code> por <code>a</code>
<code>a %= b</code>	O resto da divisão de <code>b</code> por <code>a</code> é atribuído à <code>a</code>

Enquanto isso, o laço `for` permite iterar sobre um conjunto de elementos. A sua sintaxe é dada da seguinte forma:

```
for elemento in conjunto:  
    # código a ser executado para  
    cada elemento
```

O `for` é uma palavra chave para “*por*” e podemos entender sua estrutura como “*por cada elemento em um dado conjunto faça...*”. Isso porque ele itera sobre cada elemento do conjunto

especificado e executa o bloco de código dentro do laço para cada elemento.

Quando o laço chega ao final do conjunto, ele é interrompido e o programa continua a execução após o laço.

Podemos também construir uma contadora com `for`, mas para isso precisamos de um conjunto iterável que possa ser utilizado nele. Pensando nisso, utilizaremos a função `range`.

A função `range` é capaz de gerar uma sequência de números inteiros. A sua sintaxe é dada por:

```
range(inicio, fim, passo)
```

Segundo a [documentação](#), `range()` gera uma sequência de números inteiros a partir do valor do parâmetro `inicio` até o valor do parâmetro `fim`, de acordo com o valor do parâmetro `passo`. Se `inicio` não for especificado, o valor padrão é `0`. Se `passo` não for especificado, o valor padrão é `1`.

```
# Não é preciso inicializar uma  
# variável  
# o for percorre todo o conjunto  
# de números 1 a 10  
for contador in range(1, 11):  
    print(contador)
```

```
## Saída:  
# 1  
# 2  
# 3  
# 4  
# 5  
# 6  
# 7  
# 8  
# 9  
# 10
```


3.2.1 Comandos relacionados aos laços

Conseguimos controlar o fluxo de execuções dentro do bloco de código dos laços `while` e `for` através dos comandos de controle `continue` e `break`.

O `continue` interrompe a iteração atual do laço e salta para a próxima iteração, ou seja, retorna ao início do código. Já o `break` vai interromper a execução do laço completamente, saindo do bloco de código.


CONTINUE

```
while condição:  
    # código dentro do laço  
    continue  
    # código dentro do laço  
# código fora do laço
```



BREAK

```
while condição:  
    # código dentro do laço  
    break  
    # código dentro do laço  
# código fora do laço
```



4 ESTRUTURA DE DADOS

4.1 Listas

Listas são sequências mutáveis, normalmente usadas para armazenar coleções de itens.

Podemos agrupar um conjunto de dados com as listas de maneira ordenada. Para criar uma lista com valores, são colocados os dados entre colchetes ([]) separados por vírgulas.

Elas também podem armazenar qualquer tipo de item, incluindo números, strings, objetos, outras listas e também outras estruturas de dados. Todos esses tipos de dados podem ser armazenados juntos em uma mesma lista, pois ela não se limita a armazenar um único tipo de dado.

Um exemplo de criação de lista é construção de um conjunto de dados cadastrais:

```
# lista com nome, salário, idade
# e situação de empregabilidade
lista = ['Daniel', 2310.30,
24, True]
print(lista)
```

```
## Saída:
# ['Daniel', 2310.3, 24, True]
```

Cada elemento da lista tem um índice que indica sua posição na lista. Os índices começam em 0 e vão até o tamanho da lista menos 1.

No exemplo anterior, temos 4 elementos com índices variando de 0 a 3, ordenadamente. Ou ordenados de -4 a -1, visto que os índices podem ser negativos também.

É possível selecionar separadamente os elementos de uma lista colocando seus índices.

```
lista = ['Daniel', 2310.30,
24, True]
# com índices positivos
print(lista[0], lista[1], lista[2],
lista[3])
# com índices negativos
print(lista[-4], lista[-3], lista[-2], lista[-1])
```

```
## Saída:
# Daniel 2310.3 24 True
# Daniel 2310.3 24 True
```

É possível usar as estruturas de repetição para acessar e ler os valores de uma lista. A forma mais comum de se fazer isso é utilizando o laço **for**:

```
lista = ['Daniel', 2310.30,
24, True]
for elemento in lista:
    print(elemento)
```

```
## Saída:
# Daniel
# 2310.3
# 24
# True
```

Podemos também usar o laço **while**, através de um contador:

```
lista = ['Daniel', 2310.30,
24, True]
contadora = 0
while contadora < 4:
    print(lista[contadora])
    contadora += 1
```

```
## Saída:
# Daniel
# 2310.3
# 24
```

```
# True
```

É possível observar que o bloco de código para ler uma lista com `while` é muito maior que com `for`. Por isso, quando trabalhamos com estruturas de dados, é preferível utilizar o laço `for`.

Além disso, podemos usar as mesmas funções e métodos utilizados em variáveis únicas como strings, inteiros, floats, etc nos elementos da lista. Mas é preciso que o elemento manipulado seja do tipo de variável que permite a aplicação das funções e métodos.

Por exemplo, é possível deixar o elemento `'Daniel'`, uma string, em maiúsculas com o método `upper()`:

```
lista = ['Daniel', 2310.30, 24, True]
```

```
print(lista[0].upper())
```

```
## Saída:  
# DANIEL
```

Conseguimos também substituir elementos, além de manipulá-los. Para isso, especificamos o índice do elemento a ser substituído e atribuímos o novo dado com o operador de atribuição. Podemos então substituir a string `'Daniel'` por sua correspondente em maiúscula (`'DANIEL'`):

```
lista = ['Daniel', 2310.30, 24, True]
```

```
print(lista)
```

```
lista[0] = lista[0].upper()  
print(lista)
```

```
## Saída:  
# ['Daniel', 2310.3, 24, True]  
# ['DANIEL', 2310.3, 24, True]
```

4.1.1 Métodos com listas

Listas oferecem muitas funções e métodos úteis para manipular os itens armazenados, como adicionar, remover, classificar e pesquisar elementos.

Então vamos verificar alguns métodos com listas. Considerando que “lista” é o seguinte conjunto de dados:

```
lista = ['Osman Presley', 13, 8.5, 8.5]
```

Vejamos alguns métodos que conseguimos utilizar com qualquer variável do tipo `list` (lista):

- **append:** adiciona um elemento ao final da lista.
 - Exemplo: `lista.append(4.5)`, a lista agora é `['Osman Presley', 13, 8.5, 8.5, 4.5]`.
- **clear:** remove todos os elementos da lista.
 - Exemplo: `lista.clear()`, a lista agora é `[]`.
- **copy:** faz uma cópia da lista.
 - Exemplo: `nova_lista = lista.copy()`, `nova_lista` agora é `['Osman Presley', 13, 8.5, 8.5, 4.5]` e é uma cópia independente de lista.
- **count:** conta quantas vezes um elemento aparece na lista.
 - Exemplo: `lista.count(8.5)`, retorna `2`.
- **extend:** adiciona os elementos de uma lista ao final da lista atual.
 - Exemplo: `lista.extend(['manhã', 'tarde'])`, a lista agora é `['Osman Presley', 13, 8.5, 8.5, 'manhã', 'tarde']`.
- **index:** retorna o índice do primeiro elemento com o valor especificado.
 - Exemplo: `lista.index(13)`, retorna `1`.
- **insert:** adiciona um elemento em uma posição específica da lista.
 - Exemplo: `lista.insert(2, 4.5)`, a lista agora é `['Osman`

```
Presley', 13, 4.5, 8.5,
8.5].
```

- **pop:** remove e retorna o elemento na posição especificada (ou na última posição se não for fornecida uma posição).
 - Exemplo: `lista.pop(1)`, retorna `13` e a lista agora é `['Osman Presley', 8.5, 8.5]`.
- **remove:** remove o primeiro elemento com o valor especificado.
 - Exemplo: `lista.remove(8.5)`, a lista agora é `['Osman Presley', 13, 8.5]`.
- **reverse:** inverte a ordem dos elementos na lista.
 - Exemplo: `lista.reverse()`, a lista agora é `[8.5, 8.5, 13, 'Osman Presley']`.

Há também o método `sort` que ordena os elementos na lista. Mas como nossa lista não é unicamente numérica, não é possível utilizar esse método, pois ele ordenará apenas valores numéricos ou apenas valores textuais.

Uma função bastante utilizada com listas é a `len()`, que retorna a quantidade de elementos dentro delas:

```
idades = [23,15,37,43,90,53]
print(len(idades))
```

```
## Saída:
# 8
```

Além dela também existe a estrutura de partição: uma maneira de acessar um subconjunto de elementos de uma lista. Podemos fazer isso através da estrutura de partição por índices.

O operador de partição de lista é representado pelos colchetes (`[]`) e permite que você especifique um intervalo de índices separados por dois pontos (`:`) para acessar os elementos desejados.

```
letras =
['A','B','C','D','E','F','G']
print(letras[2:6])
```

```
## Saída:
# ['C', 'D', 'E', 'F']
```

Com `letras[2:6]`, é acessado os elementos da lista `letras` de índice `2` até o índice `5`, ou seja, os elementos `['C', 'D', 'E', 'F']`.

4.2 Dicionários

Os dicionários são um tipo de estrutura de dados que armazenam pares de *chave-valor*. Eles são delimitados por chaves `{}` e os pares *chave-valor* são separados por vírgulas, como mostra a sintaxe:

```
dicionario = {chave: valor}
```

A chave é um elemento único que identifica um valor no dicionário, enquanto o valor é o item que é armazenado para a chave. As chaves e os valores podem ser de qualquer tipo de dado.

Os dicionários são úteis para armazenar e acessar dados de maneira organizada e rápida, além de permitirem uma organização mais dinâmica.

Um exemplo é construir uma ficha de cadastro de uma pessoa funcionária em uma empresa.

```
cadastro = {'numero_cadastro':
130089,
'dia_cadastro': 3,
'mes_cadastro': 2,
'funcao': 'limpeza'}
```

```
cadastro
## Saída:
# {'numero_cadastro': 130089,
# 'dia_cadastro': 3,
# 'mes_cadastro': 2,
# 'funcao': 'limpeza'}
```

Acessamos os valores de cada elemento especificando sua chave correspondente:

```
cadastro = {'numero_cadastro':  
130089,  
            'dia_cadastro': 3,  
            'mes_cadastro': 2,  
            'funcao': 'limpeza'}  
print(cadastro['numero_cadastro'],  
cadastro['dia_cadastro'], cadas  
tro['mes_cadastro'], cadastro['fu  
ncao'])  
  
## Saída:  
# 130089 3 2 limpeza
```

Podemos alterar o valor de cada chave especificando a chave do elemento que será modificado e atribuindo o novo dado com o operador de atribuição. Vamos trocar a função de limpeza para manutenção:

```
cadastro = {'numero_cadastro':  
130089,  
            'dia_cadastro': 3,  
            'mes_cadastro': 2,  
            'funcao': 'limpeza'}  
print(cadastro)  
cadastro['funcao'] =  
'manutenção'  
print(cadastro)  
  
## Saída:  
# {'numero_cadastro': 130089,  
'dia_cadastro': 3,  
'mes_cadastro': 2, 'funcao':  
'limpeza'}  
# {'numero_cadastro': 130089,  
'dia_cadastro': 3,  
'mes_cadastro': 2, 'funcao':  
'manutenção'}
```

4.2.1 Métodos com dicionários

Assim como as listas, os dicionários também oferecem muitas funções e métodos úteis para manipular seus itens.

Então, vamos verificar alguns métodos com listas, considerando que “dici” é o seguinte conjunto de dados:

```
dici = {'nome': 'Osman',  
        'idade': 13,  
        'nota_1': 8.5,  
        'nota_2': 8.5}
```

Vejamos alguns métodos que conseguimos utilizar com qualquer variável do tipo dict (dicionário):

- **clear:** remove todos os itens de um dicionário.
 - Exemplo: `dici.clear()` a lista agora é {}.
- **copy:** retorna uma cópia do dicionário.
 - Exemplo: `novo_dicionario = dici.copy()`, `novo_dicionario` agora é {'nome': 'Osman', 'idade': 13, 'nota_1': 8.5, 'nota_2': 8.5} e é uma cópia independente de `dici`.
- **dict.fromkeys:** cria um novo dicionário com chaves fornecidas por um iterável e todos os valores definidos como o valor padrão fornecido.
 - Exemplo: `novas_notas = dict.fromkeys(['nota_1', 'nota_2'], 8.0)`, `novas_notas` é {'nota_1': 8.0, 'nota_2': 8.0}.
- **get:** retorna o valor associado a uma chave específica no dicionário.
 - Exemplo: `dici.get('idade')`, retorna 13.
- **items:** retorna uma lista de [tuplas](#) que representam os itens do dicionário (chave e valor).
 - Exemplo: `dici.items()`, retorna [('nome', 'Osman'), ('idade', 13), ('nota_1', 8.5), ('nota_2', 8.5)].

- **keys:** retorna uma lista de todas as chaves do dicionário.
 - Exemplo: `dici.keys()`, retorna `['nome', 'idade', 'nota_1', 'nota_2']`.
- **pop:** remove e retorna o valor associado a uma chave específica no dicionário.
 - Exemplo: `dici.pop('nome')`, retorna `'Osman'` e o dicionário sem a chave `'nome'`.
- **popitem:** remove e retorna um item aleatório do dicionário.
 - Exemplo: `dici.popitem()`
- **setdefault:** retorna o valor associado a uma chave específica no dicionário. Se a chave não existir, ela é adicionada ao dicionário com o valor padrão fornecido.
 - Exemplo: `dici.setdefault('nome')`, retorna `'Osman'`; `dici.setdefault('nota_3', 4.5)` o dicionário será `{'nome': 'Osman', 'idade': 13, 'nota_1': 8.5, 'nota_2': 8.5, 'nota_3': 4.5}`.
- **update:** adiciona os itens de um outro dicionário para o dicionário atual e atualiza pares chave-valor existentes.
 - Exemplo: `dici.update({'nota_3': 4.5, 'sobrenome': 'Presley'})`, o dicionário agora é `{'nome': 'Osman', 'idade': 13, 'nota_1': 8.5, 'nota_2': 8.5, 'nota_3': 4.5, 'sobrenome': 'Presley'}`.
- **values:** retorna uma lista de todos os valores do dicionário.
 - Exemplo: `dici.values()`, retorna `['Osman', 13, 8.5, 8.5]`.

Os métodos `items`, `keys` e `values` permitem a leitura dos dados do dicionário através de laços `for`. Vamos fazer três diferentes leituras com `for`, utilizando os três métodos que aprendemos:

```
dici = {'nome': 'Osman',
        'idade': 13,
        'nota_1': 8.5,
```

```
        'nota_2': 8.5}
```

```
# Leitura com keys
for chaves in dici.keys():
    print(dici[chaves])
## Saída
# Osman
# 13
# 8.5
# 8.5

# Leitura com values
for valores in dici.values():
    print(valores)
## Saída
# Osman
# 13
# 8.5
# 8.5

# Leitura com items
for chaves, valores in
dici.items():
    print(chaves, valores)
## Saída
# nome Osman
# idade 13
# nota_1 8.5
# nota_2 8.5
```

CHEGAMOS AO FIM

Espero que você consiga ter um bom proveito das informações colocadas aqui e que elas consigam te ajudar em seus estudos em Python e Data Science!

O resumo contém as informações já transmitidas em aula, mas para solidificar o aprendizado em Python é necessário sempre praticar bastante. No curso, você tem acesso a vários desafios que você consegue solucionar com os estudos na Alura!

Continue sempre estudando e quaisquer dúvidas você pode utilizar o fórum e o discord da Alura.

Bons estudos!