PYTHON PARA DATA SCIENCE:
Primeiros Passos

alura | Escola de Data Science

SUMÁRIO

OLÁ, ESTUDANTE!

Essa é a nossa apostila do curso introdução ao Python voltado para a área de Ciência de Dados! Este material de estudo inclui os resumos dos conteúdos estudados nos cursos de **Python para Data Science**, além de alguns extras.

Como você sabe, Python é uma das linguagens mais populares e versáteis para Data Science e Análise de Dados. Nesta apostila, nós cobrimos os aspectos fundamentais desta linguagem abordados no curso, incluindo sua sintaxe, estruturas de controle e dados estruturados, funções, estruturas de dados e exceções. Cada tópico é acompanhado de exemplos claros e fáceis de seguir para que você possa desenvolver seus conhecimentos.

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

Boa leitura e bons estudos!

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 pessoas programadoras experientes. Também 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 adicionar 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.

Python também é uma linguagem multiplataforma, ou seja, o código escrito pode ser executado em diversos sistemas operacionais, incluindo Windows, Mac e Linux. Isso é uma vantagem para as pessoas desenvolvedoras, pois elas não precisam se preocupar com a compatibilidade de sistemas ao escrever seu código.

Algumas curiosidades sobre Python incluem:

- Foi criada por Guido van Rossum em 1989, mas seu uso só se tornou amplo a partir dos anos 2000.
- É uma linguagem dinâmica, pois permite a alteração dos tipos de variáveis durante a execução do código.
- É usada em uma ampla gama de aplicações, incluindo ciência de dados, inteligência artificial, desenvolvimento de jogos, automação de tarefas, entre outros.

1.2 Google Colaboratory

O Google Colaboratory (ou Colab) é uma plataforma poderosa e versátil que oferece às pessoas 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 vantagens para pessoas programadoras que desejam aprender Python.

Uma dessas principais vantagens é poder acessar o Google Colab de qualquer lugar que tenha acesso à Internet. Isso significa que você pode aproveitar seu tempo livre para estudar, mesmo quando estiver fora de casa ou do escritório. Como o Google Colab funciona diretamente no navegador, você não precisa se preocupar com a instalação de software adicional no seu computador.

Logo, o Google Colab é uma plataforma excelente para quem deseja aprender Python, oferecendo facilidade de acesso, colaboração em tempo real, armazenamento seguro e recursos avançados.

Para acessar e fazer os seus projetos, você pode acessar o link do Google Colab. Para 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 possibilidades de operações com valores numéricos e manipulações 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: de uma linha e 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 em uma linha de código ou um texto qualquer.

Nesta apostila, você encontrará vários comentários nos códigos, descrevendo-os ou exibindo 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, ou 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 componentes importantes 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. Por outro lado, também podemos trocar o valor de uma variável a qualquer momento.

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

- O nome da variável não pode começar com um número, mas sim com uma letra ou o caractere _. Logo, não podemos fazer: `10_notas`, `2_nomes_casa`
- Não podemos usar espaços em branco no nome da variável. Exemplos do que não fazer: `Nome escola`, `notas estudantes`, etc.
- Não é permitido utilizar 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.
- **Booleanos (`bool`):** valores lógicos de 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 utilizando os **operadores aritméticos**. Abaixo, temos 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 funções que podem ser aplicadas a valores numéricos para executar outros cálculos, tais como:

- **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 exponente, 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.

Como exemplo, 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: manooooo
```

Agora vamos para as 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 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, mas não necessitam dos parênteses `()`. 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: 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 como string. A sintaxe da função `input` é a seguinte:

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

A *string opcional* é exibida para a pessoa usuária na tela antes da entrada de dados. É uma boa prática incluir essa tela para orientar sobre o que deve ser digitado. Como exemplo, podemos coletar um dado de texto e mostrá-lo 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 deles 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, temos a formatação **f-string** (ou formatação de string), usando o operador de formatação `%`, ou com método format.

Para utilizar a formatação *f-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	%s
inteiro	%d
float	%f
caractere	%c

A sintaxe consiste na adição do 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 `%`. Nesse caso, os marcadores são apenas as chaves `{}`. 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 uma barra invertida única `\` 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 verificar se uma determinada condição é verdadeira ou falsa e, em seguida, executar 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, conforme o código abaixo:

```
# 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 o**peradores 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. 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 à 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 ao resultado; caso contrário, temos o valor de `'negativo'`.

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, ou seja, 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
`a = b`	Atribui o valor de `b` em `a`
`a += b`	Soma o valor de `b` na variável `a`
`a -= b`	Subtrai o valor de `b` na variável `a`
`a *= b`	Multiplica o valor de `b` por `a`
`a /= b`	Divide o valor de `b` por `a`
`a //= b`	Realiza divisão inteira da variável `b` por `a`

`a %= b`	O resto da divisão de `b` por `a` é atribuído à `a`

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, o `range()` gera uma sequência de números inteiros 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`.

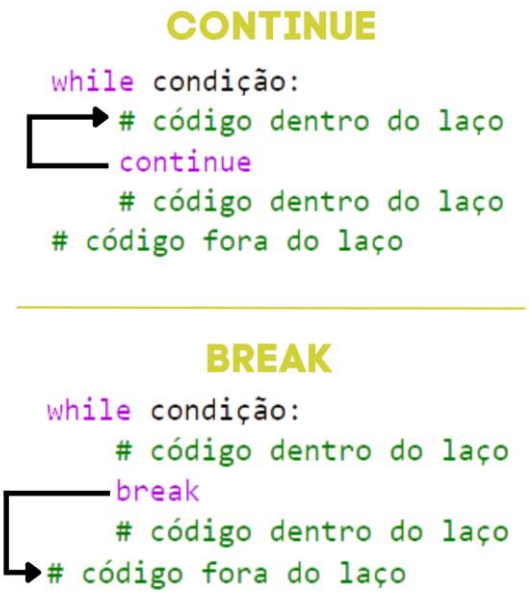
```
# Não é preciso inicializar uma
# variável
# o for percorre todo o conjunto
# de números 1 a 10
for contador in range(1, 11):
  print(contador)

## 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.



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 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
```

Podemos 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 `6`, 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:
# {'número_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 dicionários, 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
```

4.3 Tuplas

As tuplas são estruturas de dados imutáveis da linguagem Python que são utilizadas para armazenar conjuntos de múltiplos itens e frequentemente são aplicadas para agrupar dados que não devem ser modificados.

Assim, não é possível adicionar, alterar ou remover seus elementos depois de criadas. Elas são especialmente úteis em situações onde precisamos garantir que os dados não sejam alterados acidentalmente ou intencionalmente

Para criar uma tupla, basta separar seus elementos por vírgulas e envolvê-los entre parênteses.

Um exemplo de criação de tupla é a construção de um conjunto de dados com o registro de uma estudante:

```
cadastro = ("Ana", 23,
"Uberaba", "MG", "Python para DS
2")
```

Assim como na lista, cada elemento da tupla tem um índice que indica a sua posição. É possível acessar seus elementos colocando seu índice entre colchetes:

```
print(cadastro[0]) # imprime Ana
print(cadastro[-1]) # imprime
Python para DS 2
```

Por também ser um iterável, podemos desempacotar os dados de uma tupla passando cada valor para uma variável. Por exemplo:

```
nome, idade, cidade, estado,
turma = cadastro

print(f'A estudante {nome} tem
{idade} anos e mora em {cidade}-
{estado}. Ela está matriculada
na turma de {turma}.')

## Saída
# A estudante Ana tem 23 anos e
mora em Uberaba-MG. Ela está
matriculada na turma de Python
para DS 2.
```

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

```
cadastro = ("Ana", 23,
"Uberaba", "MG", "Python para DS
2")
for elemento in cadastro:
  print(elemento)

## Saída
# Ana
```

```
# 23
# Uberaba
# MG
# Python para DS 2
```

4.3.1 Métodos com tuplas

Diferentemente das listas e dicionários, as tuplas, por sua natureza imutável, oferecem poucos métodos úteis para manipular seus itens.

Vamos verificar esses métodos a partir da tupla `pares`:

```
pares = (2, 4, 8, 6, 2, 4, 2, 8, 12)
```

Esses são os métodos que conseguimos utilizar com qualquer variável do tipo `tuple` (tupla):

- **`count`:** retorna a quantidade de vezes que um dado valor se repete em uma tupla.
  - Exemplo: `pares.count(2)` retorna `3`.
- **`index`:** procura na tupla pelo valor especificado e devolve a posição deste valor, se encontrado.
  - Exemplo: `tupla.index(12)` retorna `8`.

Entretanto, podemos utilizar algumas funções que conseguem ler tuplas e passar informações interessantes. A função **`sorted()`** ordena os elementos da tupla em uma lista se os dados forem numéricos.

```
pares = (2, 4, 8, 6, 2, 4, 2, 8, 12)
sorted(pares)

## Saída
# [2, 2, 2, 4, 4, 6, 8, 8, 12]
```

Uma outra função bastante utilizada com tuplas é a **`len()`** que retorna a quantidade de elementos dentro delas:

```
pares = (2, 4, 8, 6, 2, 4, 2, 8, 12)
len(pares)

## Saída:
# 9
```

Assim como nas listas, podemos "fatiar" tuplas usando o conjunto de operadores de partição representado pelos colchetes e dois pontos (`[:]`) para acessar os elementos desejados.

```
pares = (2, 4, 8, 6, 2, 4, 2, 8, 12)
print(pares[2:4])

## Saída:
# (8, 6)
```

Com `pares[2:4]` são acessados os elementos da lista `pares` de índice `2` até o índice `4`, ou seja, os elementos `(8, 6)`.

5 BIBLIOTECAS

Na linguagem Python, utiliza-se bastante o conceito de bibliotecas como um conjunto de módulos pré-prontos que podem ser facilmente importados para o seu código, permitindo que você adicione funcionalidades avançadas sem precisar escrever tudo do zero.

Existe uma variedade de bibliotecas disponíveis para tarefas como ciência de dados, machine learning, web scraping, automação e muito mais.

5.1 Instalando bibliotecas

Para instalar ou atualizar uma biblioteca no Python, podemos recorrer ao `pip` que é um gerenciador de bibliotecas no Python.

O Google Colab já possui algumas bibliotecas pré-instaladas e é possível verificar isso usando o comando `!pip list` que lista todos os pacotes instalados com suas respectivas versões:

```
!pip list

## Saída
# Package      Version
# ---------------------------
# absl-py      1.4.0
# alabaster    0.7.13
# ...
```

Por exemplo, se quisermos instalar a biblioteca matplotlib, que é uma biblioteca para criação de gráficos e visualização de dados, podemos escrever da seguinte forma:

```
!pip install matplotlib
```

A exclamação é necessária para rodar o comando no prompt da máquina virtual do Colab. Na tela será exibido o log com o processo de instalação. Caso queiramos instalar uma versão específica, podemos prosseguir desta forma:

```
!pip install matplotlib==3.6.2
```

Nota: O uso de versão específica de uma biblioteca pode decorrer da necessidade do uso de recursos específicos para determinados projetos ou tarefas a serem executadas.

5.2 Importando bibliotecas

Para poder usar uma biblioteca precisamos importá-la em nosso ambiente. Para isso fazemos:

```
import matplotlib
```

Para testar a importação podemos ler a versão da biblioteca da seguinte forma:

```
matplotlib.__version__
```

Existem outras formas de importar uma biblioteca e uma delas é passando um apelido (`alias`).

É bem comum nas bibliotecas e pacotes de data science passarmos apelidos para elas como forma de simplificar a escrita dos códigos. Vamos testar essa forma em um pacote de visualização de dados: a `matplotlib.pyplot`.

```
import matplotlib.pyplot as plt
```

5.3 Utilizando pacotes/bibliotecas

Vamos iniciar a utilização dos métodos ou variáveis de uma biblioteca. É importante ter o costume de acessar a documentação das bibliotecas para aprender como utilizá-las.

5.3.1 Importando a biblioteca inteira

Neste exemplo vamos utilizar o pacote `matplotlib.pyplot` para gerar um gráfico de colunas com as médias de 4 estudantes de uma dada classe:

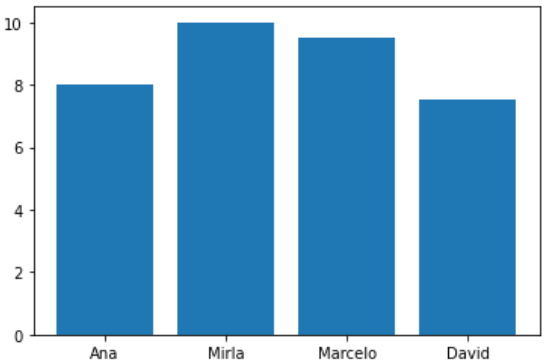
```
import matplotlib.pyplot as plt

estudantes = ["Ana", "Mirla",
"Marcelo", "David"]
notas = [8, 10, 9.5, 7.5]

plt.bar(estudantes, notas)
```

Perceba que primeiro passamos o nome (`alias`) que demos a biblioteca seguida do método que queremos utilizar. Essa é a forma básica de chamar um método de um pacote ou biblioteca.

Vamos executar o código para visualizar o gráfico que construímos:



5.3.2 Importando um método específico

Partindo agora para a importação de um método específico de uma biblioteca, vamos trabalhar com o pacote `random` que trabalha com seleções e gerações aleatórias de dados.

Neste exemplo, vamos selecionar aleatoriamente uma fruta dentro de uma lista. Para isso, utilizaremos o método choice, chamando-o da seguinte forma:

```
from random import choice

frutas = ["maçã", "uva",
"banana", "limão"]
choice(frutas)

## Saída:
# 'uva'
```

Note que usamos a palavra-chave `from` escolhendo a biblioteca desejada, seguido de `import` e o método requisitado. Observe que a função não precisa do nome da biblioteca atrelada a ela. Ela recebe a lista como parâmetro e devolve um dos itens da lista de maneira aleatória.

5.3.3 Importando mais de um método

Semelhante ao caso anterior, para importar mais de um método de uma mesma biblioteca, podemos passar a palavra-chave `from` seguido do nome da biblioteca, a palavra-chave `import` e os métodos desejados separados por vírgula. Por exemplo:

```
# Calculando o seno de 90° (π/2)
from math import pi, sin
```

```
sin(pi / 2)

## Saída
# 1.0
```

Note que não é necessário pedir a importação da mesma biblioteca mais de uma vez nem importá-la inteira no código.

5.3.4 Importando todos os métodos da biblioteca

Outra forma de importar mais métodos no Python é seguindo o mesmo caminho dos dois casos anteriores, mas após a palavra-chave `import` escrever apenas um asterisco, como podemos notar no exemplo abaixo:

```
# Calculando o desvio padrão de
uma amostra pela variância
from math import *

variancia = 25
desvio = sqrt(variancia)
desvio

## Saída
# 5.0
```

A diferença desta para o `import nome_biblioteca` é que, neste caso, não precisamos usar o nome da biblioteca para chamar um método. Podemos passar apenas o nome do método.

Atenção: A importação desta forma precisa de alguns cuidados:
- Podemos ter choque de nomes entre as variáveis, no caso de termos uma função chamada `sqrt` antes de importar a da biblioteca `math`, por exemplo.
- Podemos reduzir a eficiência da execução, se o número de funções importadas é grande.
- Não fica explícito de onde aquela variável, método ou classe veio.

6 FUNÇÕES

Na linguagem Python, as **funções** são sequências de instruções que executam tarefas específicas, podendo ser reutilizadas em diferentes partes do código. Elas podem receber parâmetros de entrada (que podemos chamar de *inputs*) e também retornar resultados.

6.1 Built-in functions

O interpretador do Python já possui uma série de funções embutidas que podem ser invocadas a qualquer momento. Inclusive ao longo de nossa jornada utilizamos uma série delas.

Elas fornecem uma série de funcionalidades básicas para a aplicação de tarefas e instruções na linguagem Python, sendo muito úteis em Data Science pela facilidade de uso e abstração em tarefas que podem ser complexas, como ordenações de lista ou manipulação de strings. Seguem alguns exemplos:

6.1.1 sum()

A função embutida `sum()` no Python é utilizada para somar os elementos de um iterável (lista, tuplas, conjuntos ou objeto que suporte iteração). Ela retorna a soma total dos elementos da sequência.

Por exemplo, se quisermos somar a população estimada em milhões dos Estados da região Sul do Brasil:

```
populacao = {"Paraná": 11.5,
"Santa Catarina": 7.7,"Rio
Grande do Sul": 11}

soma = sum(venda.values())

print(f'A população total
estimada da região Sul é de
{soma} milhões de habitantes')

## Saída
# A população total estimada da
região Sul é de 30.2 milhões de
habitantes
```

6.1.2 min() e max()

As funções embutidas `min()` e `max()` são usadas para encontrar o menor ou o maior valor de um conjunto de valores, respectivamente. Elas podem ser usadas em estrutura de dados como lista, tuplas e dicionários.

Por exemplo, podemos apresentar a menor e maior idade das clientes de uma empresa representadas por meio de uma lista:

```
idades = [24, 45, 67, 35, 71,
41, 76, 43, 62, 42]

print(f"A cliente com menor
idade possui {min(idades)} anos
e a com maior idade possui
{max(idades)} anos.")

## Saída
# A cliente com menor idade
possui 24 anos e a com maior
idade possui 76 anos.
```

6.1.3 zip()

A `zip()` é uma função embutida do Python que recebe um ou mais iteráveis e retorna-os como um iterador de tuplas no qual cada elemento dos iteráveis é pareado.

Ela é útil para fazer iterações simultâneas em várias listas. A função `zip()` pode ser usada em conjunto com

outras funções do Python, como `map()` e `filter()`, para criar soluções elegantes e concisas para certos problemas.

Por exemplo, se quisermos criar uma lista de tuplas pareando o id (código do produto) com o tipo de produto de uma loja de equipamentos de informática, podemos escrever o código da seguinte forma:

```
id = [1, 2, 3, 4, 5]
produtos = ["mouse", "teclado",
"headset", "webcam", "monitor"]

id_produtos = list(zip(id,
produtos))
id_produtos

## Saída:
# [(1, 'mouse'), (2, 'teclado'),
(3, 'headset'), (4, 'webcam'),
(5, 'monitor')]
```

Precisamos passar o objeto `zip` criado para uma lista para que possamos exibir o resultado.

Para fazer o processo contrário, de transformar uma tupla iterável em listas, basta passar o operador asterisco (*) ao lado esquerdo do nome da tupla iterável que quer extrair os dados, repassando cada tupla para uma variável.

```
produtos = [("banana", "fruta"),
("arroz", "mercearia"),
("frango", "carnes e
embutidos")]

produto, categoria =
zip(*produtos)

for i in range(len(produtos)):
  print(f'O produto {produto[i]}
está na categoria
{categoria[i]}.')

## Saída
# O produto banana está na
categoria fruta.
# O produto arroz está na
categoria mercearia.
# O produto frango está na
categoria carnes e embutidos.
```

6.1.4 enumerate()

A função embutida `enumerate()` permite iterar sobre uma sequência controlando o índice de cada elemento.

Para exemplificar, pense que você possui uma lista de frutas e quer imprimir cada elemento da lista junto com seu índice correspondente. Você pode usar a função enumerate() para fazer isso de forma simples:

```
frutas = ['maçã', 'banana',
'laranja', 'abacaxi']
for i, fruta in
enumerate(frutas):
    print(f'O id {i} pertence
à(ao) {fruta}')

## Saída
# O id 0 pertence à(ao) maçã
# O id 1 pertence à(ao) banana
# O id 2 pertence à(ao) laranja
# O id 3 pertence à(ao) abacaxi
```

Além disso, é possível definir um argumento opcional `start` na função `enumerate()` que indica o valor do índice inicial.

No exemplo a seguir, o código começa a partir do índice (ou id) 100:

```
frutas = ['maçã', 'banana',
'laranja', 'abacaxi']
for i, fruta in
enumerate(frutas, start=100 ):
    print(f'O id {i} pertence
```

```
à(ao) {fruta}')

## Saída
# O id 100 pertence à(ao) maçã
# O id 101 pertence à(ao) banana
# O id 102 pertence à(ao)
laranja
# O id 103 pertence à(ao)
abacaxi
```

6.1.5 map()

A função embutida `map()` é usada para aplicar uma função a cada elemento de uma sequência e retornar um novo iterável com os resultados. Sua forma básica é:

```
map(funcao, sequencia)
```

Por exemplo, caso seja necessário alterar uma lista de distâncias de pontos para que todos os valores sejam positivos, podemos fazer da seguinte maneira:

```
distancias = [8, -9.43, 5.65, -
7.15, 8.05]
dist_abs = list(map(abs, notas))
dist_abs

## Saída
# [8, 9.43, 5.65, 7.15, 8.05]
```

Ela é largamente utilizada por pessoas cientistas de dados junto a uma função lambda ou funções personalizadas, que veremos ainda neste capítulo.

6.1.6 filter()

A função embutida `filter()` permite filtrar elementos de uma sequência com base em uma função que retorna `True` ou `False`.

Ela retorna um objeto `filter` que contém apenas os elementos da lista que passam no teste da função. Para exibir os resultados, usamos a função `list()` para converter o objeto `filter` em uma lista.

Por exemplo, podemos filtrar uma lista com os volumes de um produto, deixando apenas aqueles que estão entre 350 a 360 ml:

```
volumes = [351, 352, 347, 335,
361]

def filtro(x):
  return x >= 350 and x <= 360

vol_aceito = list(filter(filtro,
volumes))
print(f'Volumes aceitos:
{vol_aceito}')

## Saída
# Volumes aceitos: [351, 352]
```

Neste caso, a função `filter()` aplica a função `filtro()` a cada número na lista `volumes: [351, 352]`.

6.2 Criando funções

As funções permitem que pessoas cientistas de dados definam blocos de código reutilizáveis, tornando o código mais legível, organizado e fácil de manter.

Podem ser utilizadas em uma ampla variedade de tarefas, desde cálculos matemáticos simples até a implementação de algoritmos complexos sendo essenciais para a criação de código eficiente e flexível.

6.2.1 Funções sem parâmetros

Para criar uma função, escrevemos: a palavra reservada `def`, que define uma função; o nome da função

seguido de parênteses; e dois pontos. Além disso, identamos as instruções que desejamos executar nesta função:

```
def nome_funcao():
  instrucoes
```

Como podemos observar, este tipo de função não recebe parâmetros de entrada (que ficam entre os parênteses).

Um exemplo simples pode ser a execução de uma saudação:

```
def saudacao():
  print("Olá, cientista!")
```

Após rodar o código de criação da função, podemos rodar esta função em outra célula, chamando-a pelo nome seguido dos parênteses:

```
saudacao()
## Saída
# Olá, cientista!
```

Nota: Após criarmos qualquer função, podemos utilizá-la em outras partes do código sempre que necessário. Lembrando que caso o seu Colab encerre a sessão é necessário rodar novamente a definição da função, pois a máquina virtual limpa todo o histórico de variáveis, funções e códigos que você utilizou.

6.2.2 Funções com parâmetros

Os parâmetros de entrada são os nomes dados aos atributos que uma função pode receber.

Quando queremos passar para uma função alguma variável, lista ou outra estrutura de dados, precisamos criar um parâmetro dentro dos parênteses da função e utilizá-la com o nome que definimos dentro do escopo da função.

```
def nome_func(p1, p2, ..., pn):
  instrucoes
```

Como exemplo, podemos criar uma função que calcule a média de 4 notas:

```
def media(n1, n2, n3, n4):
  res = (n1 + n2 + n3 + n4) / 4
  print(res)
```

Após definirmos a função, podemos testá-la passando 4 argumentos, um para cada parâmetro.

```
media(8, 7, 5, 6)

## Saída
# 6.5
```

Os nomes que damos às estruturas fora da função (argumentos) podem ser diferentes aos dos parâmetros. No entanto, os parâmetros de entrada ditam como elas vão ser exploradas dentro da função.

```
nota1 = 3
nota2 = 6
nota3 = 9
nota4 = 8

media(nota1, nota2, nota3,
nota4)

##Saída
# 6.5
```

6.2.3 Escopo da função

Quando utilizamos funções precisamos prestar atenção a uma propriedade chamada **escopo de uma função.**

Ela determina onde uma variável pode ser utilizada dentro do código. Como

exemplo, uma variável criada dentro de uma função existirá apenas dentro da função. Ou seja, encerrando a execução da função, a variável não estará disponível para a pessoa usuária no restante do código.

Para testar, vamos tentar acessar a variável `res` chamando-a em outra célula:

```
res
## Saída
# NameError: name 'res' is not defined
```

Recebemos um `NameError` na saída, alertando que não existe uma variável chamada `res`. Isso acontece, pois a variável existe apenas dentro da função `media()`.

Para corrigir esse erro, podemos tornar `res` uma variável global ou retornar seu valor na função e atribuí-lo a uma variável externa. Veremos o segundo caso no tópico a seguir.

6.2.4 Funções com retorno

Para gerar uma função que retorna valores para variáveis devemos utilizar ao final das instruções o comando `return` seguido do nome da variável ou expressão que deseja retornar:

```
def nome_func(p1, p2, ..., pn):
  instrucoes
  return resultado
```

Alterando o exemplo anterio da função `media()`, podemos pedir como parâmetro uma lista de notas e retornar o cálculo da média na variável `res`:

```
def media(lista):
  res = sum(lista)/len(lista)
  return res
```

Agora, podemos passar a lista de notas que queremos calcular a média e o retorno da função `media()` para a variável `resultado`:

```
notas = [8, 6.5, 7, 9]

resultado = media(notas)
print(f'Média: {resultado}')

## Saída
# Média: 7.625
```

Nota: O retorno de uma função pode ser uma variável ou estrutura de dados, como listas, tuplas e dicionários.

6.3 Funções lambda

As funções lambda no Python são chamadas de funções anônimas, pois elas não precisam ser definidas com um nome. Além disso, descrevem em uma única linha os comandos que desejamos aplicar.

Elas são criadas com a palavra-chave `lambda` e normalmente são usadas em conjunto com outras funções como `map()` e `filter()`.

A sintaxe básica para criar uma função lambda é a seguinte:

```
lambda variavel: expressao
```

Escrevemos a palavra reservada `lambda` que define uma função anônima, uma variável vinculada como parâmetro da função e a expressão ou corpo da função como retorno.

Podemos, por exemplo, utilizar uma função lambda para calcular o montante de um capital inicial de 100 reais a uma taxa `x`

de juros e um tempo `y` de aplicação levando em conta os juros compostos:

```
montante = lambda x, y: 100*(1+
x/100) ** y
```

Aplicando valores a função temos:

```
# taxa de 10% ao ano em 5 anos
taxa = 10
tempo = 5
res = montante(10,5)
print(f'O montante é de
{round(res,2)} reais')

## Saída
# O montante é de 161.05 reais
```

Entretanto, para uma pessoa cientista de dados a forma mais poderosa de se utilizar essa função é através do mapeamento de valores, combinando-a com `map()` ou filtrando valores por meio do `filter()`.

6.3.1 Mapeando valores

A combinação entre as funções lambda e a função embutida `map()` ajuda bastante na manipulação de estruturas de dados como listas e dicionários, mapeando e alterando os valores de acordo com um padrão ou regra desejada.

A sintaxe básica para mapear valores desta forma é:

```
map(lambda: val: expr, iterador)
```

Ou seja, dentro da função `map()` passamos dois argumentos: a função lambda que desejamos para mapear nossos dados e a estrutura de dados na forma de iterador.

Por exemplo, podemos utilizar essa estrutura para converter o valor das vendas de três produtos em dólares para reais e somar o valor total de vendas em reais:

```
# dicionário com as vendas por
prod
vendas = {"prod_1": 20,
"prod_2": 10, "prod_3": 30}
cotacao_usd = 5.20 # cot dolar
# função lambda de conversão
conversao = lambda x: x *
cotacao_usd


# lista dos valores mapeados de
dólar para real
vendas_brl = list(map(conversao,
vendas.values()))
# soma das vendas e exibição
venda_tot = sum(vendas_brl)
print(venda_tot)

## Saída
# 312.0
```

Neste exemplo, a função `map()` foi utilizada para transformar os valores de cada elemento do dicionário `vendas` de acordo com a função lambda que criamos.

6.3.2 Filtrando valores

A combinação entre as funções lambda e a função embutida `filter()` facilita na filtragem elementos de uma estrutura de dados com base em uma determinada condição.

A sintaxe básica para filtrar valores desta forma é:

```
filter(lambda: val: expr,
iterador)
```

Para ilustrar seu uso, vamos considerar um exemplo simples em que filtramos uma lista mantendo apenas as palavras que tem a letra `"a"`:

```
lista = ["python", "alura",
"dados", "funções"]

# Se a palavra tiver "a" retorna
True senão False
tem_a = lambda x: "a" in x
lista_a = list(filter(tem_a,
lista))
print(f'Apenas as palavras
{lista_a} tem a letra "a"')
## Saída
# Apenas as palavras ['alura',
'dados'] tem a letra "a"
```

6.4 Documentando funções

É importante deixar o nosso código ou análise dos dados o mais acessível possível para o público. E uma das formas de atingir esse propósito reside na documentação de funções.

Podemos auxiliar quem lê o nosso projeto ou utiliza as funções que desenvolvemos a entender quais tipos de variáveis podemos utilizar, se existem ou não valores padrões ou até descrever de maneira sucinta o que aquele pedaço de código faz.

6.4.1 Type hint

O **type hint** é uma sintaxe utilizada no Python para indicar o tipo de dado esperado de um parâmetro ou retorno de função, auxiliando na legibilidade e manutenção do código.

A sua sintaxe básica é:

```
def nome_func(p1: tipo_p1) ->
tipo_retorno:
  instrucoes
  return resultado
```

Trazendo para o nosso exemplo da função `media()`, podemos utilizar o type hint da seguinte forma:

```
def media(lista: list) -> float:
  res = sum(lista) / len(lista)
  return res
```

Aqui apontamos que a função recebe uma lista do tipo `list` e retorna uma variável do tipo `float`. Para verificar o type hint, podemos rodar a função `help()` para a função `media()`:

```
help(media)
## Saída
# Help on function media in
module __main__:
# media(lista: list) -> float
```

6.4.2 Default value

No Python, o **default value** é um valor padrão atribuído a um argumento de função, que é utilizado caso nenhum valor seja passado pelo usuário.

A sua sintaxe básica é:

```
def nome(p1: tipo_p1 = valor_p1)
```

Implementando ainda mais a função `media()`, podemos utilizar o default value da seguinte forma:

```
def media(lista: list = [0]) ->
float:
  res = sum(lista) / len(lista)
  return res
```

Aqui apontamos que a função recebe uma lista do tipo `list` e retorna uma variável do tipo `float`. Se não passarmos a lista, será passada uma lista com um único elemento sendo ele zero. Para verificar o default value, podemos rodar a função `media()` sem parâmetros:

```
media()
## Saída
# 0.0
```

6.4.3 Docstring

Por fim, temos o **Docstring** que é uma string literal usada para documentar um módulo, função, classe ou método em Python. Ela é colocada como o primeiro item de definição e pode ser acessada usando a função `help()`.

O Docstring deve descrever o propósito, parâmetros, tipo de retorno e exceções levantadas pela função.

É uma boa prática utilizar Docstrings em seu código para facilitar a leitura, manutenção e compartilhamento do código com outras pessoas desenvolvedoras.

A sua sintaxe básica é:

```
def nome_func(p1,p2, .. pn):
    ''' Texto documentando sua
função...
    '''
  instrucoes
  return resultado
```

Concluindo a implementação da função `media()`, podemos utilizar o docstring da seguinte forma:

```
def media(lista: list=[0]) ->
float:
  ''' Função para calcular a
média de notas passadas por uma
lista

  lista: list, default [0]
    Lista com as notas para
calcular a média
  return = res: float
    Média calculada
  '''
  res = sum(lista) / len(lista)
  return res
```

Se executarmos o código `help(media)` em uma outra célula, temos a seguinte saída:

```
## Saída:
'''
Help on function media in module
__main__:

media(lista: list = [0]) ->
float
    Função para calcular a média
de notas passadas por uma lista

    lista: list, default [0]
      Lista com as notas para
calcular a média
    return = res: float
      Média calculada
'''
```

7 ESTRUTURA DE DADOS COMPOSTAS

Aprendemos a manipular listas, tuplas e dicionários para trabalhar com uma sequência ou coleção de valores sejam numéricos, categóricos, etc.

Aqui, vamos abordar essas estruturas com recursos mais complexos, sendo através de estruturas aninhadas: listas de listas ou lista de tuplas, até a geração de listas e dicionários padronizados por meio do list e dict comprehension.

7.1 Estruturas aninhadas

Uma situação comum para a pessoa cientista de dados é trabalhar com algumas estruturas aninhadas, ou seja, quando temos, por exemplo, listas dentro de uma lista.

Estas estruturas ajudam bastante a organizar dados num formato ideal para construção de tabelas e matrizes, tendo a devida separação dos dados e a relação entre eles. Vamos focar em duas estruturas aninhadas: listas de listas e lista de tuplas.

7.1.1 Listas de listas

Sabemos que uma lista é uma coleção ordenada de valores separados por vírgulas e dentro de colchetes e que são utilizadas basicamente para armazenar diversos itens em uma única variável.

Quando falamos de diversos itens, podemos interpretar que é possível ter elementos de diferentes tipos, podendo ser números, strings, tuplas e, claro, listas.

Então, basicamente, uma lista de listas é uma estrutura de dados em que os itens de uma lista são listas com diferentes dados ou que separam diferentes observações destes dados.

A sua forma básica é a seguinte:

```
[[a1, ..., an], [b1, ..., bn],
..., [n1, ..., nn]]
```

Note que, da mesma forma que uma lista comum, a lista de listas possui colchetes e seus itens são separados por vírgulas. A diferença é que esses itens são também listas.

Por exemplo, se quisermos criar uma lista de listas podemos utilizar o método `append()` adicionando cada lista por vez ou passando a lista como uma variável de uma lista:

```
nomes = ["João", "Maria","José"]
notas = [7, 8, 9]
```

- Usando **append**:

```
lista_aninhada = []
lista_aninhada.append(nomes)
lista_aninhada.append(notas)
lista_aninhada
## Saída
# [['João', 'Maria', 'José'],
[7, 8, 9]]
```

- Usando as variáveis:

```
lista_aninhada_2 = [nomes,
notas]
lista_aninhada_2
## Saída
# [['João', 'Maria', 'José'],
[7, 8, 9]]
```

Já para acessar valores dentro da lista de listas, precisamos ter atenção ao nível que o elemento está. Da mesma forma de uma lista comum, a cada camada

utilizamos a posição do elemento desejado dentro dos colchetes. Isto se chama de multi indexação, pois utilizamos mais de um índice para acessar o dado desejado.

Por exemplo, podemos acessar a nota do 3º trimestre de um determinado estudante em uma lista de listas com as notas de 3 estudantes:

```
notas_turma = [[6, 7, 6], [9, 7, 10], [5, 4, 8]]
```

Como já aprendemos no tópico de listas, podemos acessar as notas do 2º estudante da seguinte forma:

```
notas_turma[1]
## Saída
# [9, 7, 10]
```

Já para extrair a sua nota do 3º trimestre, precisamos primeiro acessar todas as suas notas e depois a nota desejada. Para isso adicionamos mais um par de colchetes, agora com a posição da nota do 3º trimestre:

```
notas_turma[1][2]
## Saída
# 10
```

Atenção: A ordem de indexação importa, ou seja, `notas_turma[1][2] != notas_turma[2][1]`. Enquanto a primeira resulta na nota `10`, a segunda resulta em `4`.

As listas de listas também podem usar as estruturas de repetição para acessar e ler os valores. A forma mais comum de se fazer isso é utilizando o laço `for`:

```
notas_turma = [[6, 7, 6], [9, 7, 10], [5, 4, 8]]

for i, nota in enumerate(notas_turma):
  # pegando sempre a última nota de cada lista
  nota_3_tri = nota[-1]
  print(f'Estudante {i+1}: {nota_3_tri}')

## Saída
# Estudante 1: 6
# Estudante 2: 10
# Estudante 3: 8
```

7.1.2 Listas de tuplas

Outra estrutura de dados composta e bastante utilizada no contexto de ciência de dados são as **listas de tuplas**.

Diferente das listas de listas, elas são indicadas em situações nas quais precisamos garantir que os dados não sejam alterados acidentalmente ou intencionalmente, mas que possam ser acessados para agregar mais dados nas análises.

Uma lista de tuplas possui a seguinte forma:

```
[(a1, ..., an), (b1, ..., bn), ..., (n1, ..., nn)]
```

Note que as tuplas são separadas entre vírgulas e envoltas em uma lista.

Semelhante a uma lista de listas, uma lista de tuplas pode ser criada por meio do método `append()` adicionando cada tupla por vez ou passando as tuplas como variáveis de uma lista:

```
cadastro_1 = ("123", "João")
cadastro_2 = ("456", "Maria")

cad_1 = []
cad_1.append(cadastro_1)
cad_1.append(cadastro_2)
```

```
cad_2 = [cadastro_1, cadastro_2]

print("cad_1: ", cad_1)
print("cad_2: ", cad_2)

## Saída
# cad_1:  [('123', 'João'),
('456', 'Maria')]
# cad_2:  [('123', 'João'),
('456', 'Maria')]
```

O acesso de valores de listas de tuplas também é semelhante ao de listas de listas. Utilizamos a posição do elemento desejado dentro dos colchetes levando em conta a multi indexação.

Como exemplo, podemos acessar o estudante `"José"` em um lista de tuplas com os ids e nomes de 3 estudantes:

```
cadastro_turma = [("123",
"João"), ("456", "Maria"),
("789", "José")]
cadastro_turma[2][1]
## Saída
# 'José'
```

Aqui já realizamos o acesso a tupla que continha `"José"` e a posição que ele ocupa na tupla.

Atenção: A ordem de indexação de uma lista de tuplas também importa, ou seja, `cadastro_turma[2][1] != cadastro_turma[1][2]`. Enquanto a primeira resulta em `"José"`, a segunda resulta em `IndexError`.

As listas de tuplas também podem usar as estruturas de repetição para acessar e ler os valores. A forma mais comum de se fazer isso é utilizando o laço `for`:

```
cadastro_turma = [("123",
"João"), ("456", "Maria"),
("789", "José")]

estudantes = []
for cadastro in cadastro_turma:
  nome = cadastro[1]
  estudantes.append(nome)
estudantes
## Saída
# ['João', 'Maria', 'José']
```

7.2 List comprehension

É uma forma simples e concisa de criar uma lista. Podemos aplicar condicionais e laços para criar diversos tipos de listas a partir de padrões que desejamos para a nossa estrutura de dados.

A sua sintaxe mais simples é:

```
[expressao for item in iteravel]
```

Note que construímos o nosso código entre colchetes, pois ao final geramos uma lista.

Neste formato, primeiro passamos a expressão que queremos aplicar ou, em outras palavras, o retorno e um laço `for` em que uma variável/item será iterada em torno de uma lista.

Por exemplo, podemos criar uma lista para tratar os nomes de estudantes colocando em maiúsculo.

```
nomes = ['joão', 'maria',
'josé']
[nome.title() for nome in nomes]
## Saída
# ['João', 'Maria', 'José']
```

Perceba que geramos uma lista com os nomes tratados na estrutura do list comprehension em que:

- a expressão é o nome alterado com um método `str: title()`
- o item é cada nome passado pela lista `nomes`
- o iterável é a lista `nomes`

7.2.1 List comprehension com if

Podemos também utilizar expressões condicionais para criar ou modificar listas. Neste exemplo, adicionamos, à forma básica, a palavra reservada `if` seguida da condição que desejamos avaliar.

Desta forma, para utilizar list comprehension com condicionais escrevemos:

```
[exp for item in lista if cond]
```

Traduzindo essa estrutura para a linguagem escrita seria como:
"Aplique `exp` em cada `item` da `lista` caso a condição `cond` seja aceita".

Vamos testar em um exemplo de uma lista de tuplas com os nomes de estudantes e suas médias. Podemos selecionar quais estão com a média acima de 6 da seguinte forma:

```
medias_estudantes = [("João",
5), ("Maria", 8), ("José", 6)]

aprovados = [x[0] for x in
medias_estudantes if x[1] >= 6]
aprovados

## Saída
# ['Maria', 'José']
```

Passamos cada tupla para `x` e filtramos a lista de tuplas em uma lista com apenas os nomes dos estudantes acima da nota 6.

Isso é possível porque testamos na condição `x[1]`, que é a posição das notas de cada estudante, se a nota era `>= 6`. Caso seja satisfeita a condição, passamos para a nova lista o valor de `x[0]`, responsável por guardar o nome do(a) estudante que estamos avaliando a cada laço.

7.2.2 List comprehension com if-else

Existe uma outra forma de trabalhar com condicionais para criar ou modificar listas, que seria por meio da estrutura `if-else`.

Diferente da anterior, a forma de retorno da lista se a condição é verdadeira ou falsa possuem uma ordem diversa. Neste exemplo, a sintaxe da list comprehension é a seguinte:

```
[res_if if cond else res_else
for item in lista]
```

Traduzindo essa estrutura para a linguagem escrita seria como:
"Aplique `res_if` se a condição `cond` for aceita, caso contrário aplique `res_else` para cada `item` da `lista`".

Por exemplo, podemos ler uma lista da quantidade de vendas de um produto em diferentes filiais de uma empresa e definir qual bateu ou não a meta de 5000 produtos vendidos:

```
vendas = [5500, 4600, 8000,
5600, 3500, 7010, 2970, 6200]
meta = 5000

situacao = [True if venda >=
meta else False for venda in
```

```
vendas]
situacao

## Saída
# [True, False, True, True,
False, True, False, True]
```

Nesse ponto, ao invés de filtrarmos a lista de vendas, criamos uma nova lista com os retornos para cada situação. Esse tipo de estrutura é bem importante para analisarmos os dados e a relações entre eles de acordo com métricas ou valores de outras listas.

7.2.3 List comprehension aninhado

Outra forma bem interessante de trabalhar com list comprehension é por meio da criação de listas de listas ou lista de tuplas.

Existem diferentes formas de criar essas estruturas, aqui abaixo temos duas sintaxes prováveis:

```
 [[exp for item in lista] for
item in iteravel]
```

ou

```
[tuple(exp for item in tupla)
for item in iteravel]
```

A primeira forma uma lista de listas e a segunda uma lista de tuplas. Vamos explorar um exemplo para cada uma delas.

7.2.3.1 Lista de listas por list comprehension

Seguindo a forma apresentada logo acima, a criação de uma lista de listas pode ser realizada a partir da leitura de outra lista de listas utilizando um iterável que é apresentado na primeira camada do list comprehension. Esse iterável é responsável por passar a posição de cada elemento dentro das listas.

A segunda camada pode receber cada item desse iterável (posição do elemento) para realizar uma nova iteração, agora em cima de cada lista da lista de listas.

Pode parecer bem complexo, mas podemos compreender esse comportamento por meio de um exemplo.

Vamos supor que você recebeu a lista de listas abaixo e precisa gerar uma nova lista de listas que separe os valores do cabeçalho de uma coluna de dados e cada registro em uma lista, uma na sequência da outra:

```
cadastro = [['Nome', 'Márcio',
'Régis', 'Júlia'], ['Media',
'8', '5','7'], ['Status',
'Aprovado',
'Reprovado','Aprovado']]
```

Note que queremos separar cada elemento das listas organizando as linhas como colunas, bem semelhante a transposição de uma matriz. Geramos a nova lista de listas dessa forma:

```
qtd = len(cadastro[0]) # = 4

[[item[i] for item in cadastro]
for i in range(qtd)]

## Saída
# [['Nome', 'Media', 'Status'],
#  ['Márcio', '8', 'Aprovado'],
#  ['Régis', '5', 'Reprovado'],
#  ['Júlia', '7', 'Aprovado']]
```

Passamos para o iterável o *range* do comprimento da 1ª lista da lista de listas. Isso dita a quantidade de registros da nova lista de listas.

A *list comprehension* mais interna vai passar um dado de cada lista (`item`)

pelo índice (`i`) variando aqui de `0` até `3`. Assim, temos ao final 4 registros, em que o 1º é o cabeçalho dos dados e o restante os dados cadastrais de cada estudante.

Podemos então observar uma aparência semelhante a uma tabela com os dados de mesma natureza coluna a coluna.

7.2.3.2 Lista de tuplas por list comprehension

A criação de uma lista de tuplas por *list comprehension* é bem semelhante à anterior. Vamos resolver mais uma situação?

Pense que você está acessando um dataset no formato de lista de listas em que cada uma de suas listas possui o tipo de dado seguido dos registros daquele dado. Para organizar os dados por colunas (`produto` x `id`) você precisa criar uma lista de tuplas:

```
bd = [['Produto', 'A', 'B',
'C'], ['id', '10', '11', '12']]
```

Note que queremos separar cada elemento das listas organizando as linhas como colunas. Geramos a lista de tuplas dessa forma:

```
qtd = len(bd[0]) # = 4

[tuple(item[i] for item in bd)
for i in range(qtd)]

## Saída
# [('Produto', 'id'),
#  ('A', '10'),
#  ('B', '11'),
#  ('C', '12')]
```

Novamente passamos para o iterável o *range* do comprimento da 1ª lista da lista de listas. Isso dita a quantidade de registros da lista de tuplas.

Dentro da função embutida `tuple()`, montamos uma expressão semelhante a uma *list comprehension* que passará um dado de cada lista (`item`) pelo índice (`i`) variando aqui de `0` até `3`.

Assim, teremos 4 registros, em que o 1º é o cabeçalho dos dados e o restante o par `produto` e `id`.

7.3 Dict comprehension

É uma forma simples e concisa de criar um dicionário. Podemos então aplicar as mesmas situações que aprendemos em *list comprehension* para criar diversos tipos de dicionários a partir de padrões que desejamos para a nossa estrutura de dados.

A sua sintaxe mais simples é:

```
{chave: valor for item in
iteravel}
```

Note que construímos o nosso código entre chaves, pois ao final geramos um dicionário.

Neste formato, primeiro passamos a **chave** e o **valor** que queremos aplicar, seguido de um laço `for` em que uma variável/item será iterada em torno de um iterável ou lista de itens de um dicionário (`keys()` e `values()`).

Podemos criar, como exemplo, um dicionário a partir de uma lista de nomes de estudantes, com a **chave** sendo o primeiro nome de cada estudante e o **valor** seus nomes completos.

```
nomes = ['João Silva', 'Maria
Souza', 'José Tavares']
cadastro = {nome.split()[0]:
nome for nome in nomes}
cadastro
```

```
## Saída
# {'João': 'João Silva',
#  'Maria': 'Maria Souza',
#  'José': 'José Tavares'}
```

Perceba que geramos um dicionário em que:

- a **chave** é o primeiro nome usando um método `str`: `split()` e lendo a primeira posição que guarda o primeiro nome. O `split()` separa uma string por meio de um separador e por padrão o separador é o espaço.
- o **valor** é cada nome passado pela lista `nomes`
- o iterável é a lista `nomes`

7.3.1 Dict comprehension com if no iterável

Podemos também utilizar expressões condicionais para criar ou modificar dicionários. Neste exemplo, adicionamos, à forma básica, a palavra reservada `if`, seguida da condição que desejamos avaliar.

Desta forma, para utilizar dict comprehension com condicionais escrevemos:

```
{chave: valor for chave, valor
in iteravel if cond}
```

Traduzindo essa estrutura para a linguagem escrita seria como:

"Aplique `chave: valor` no par `chave, valor` do iterável `iteravel` caso a condição `cond` seja aceita".

Vamos testar em um exemplo com duas listas: uma com os nomes das filiais de uma empresa e a outra com a quantidade de vendas de um dado produto.

Devemos gerar um dicionário com as filiais que bateram a meta de 5000 produtos vendidos, junto da quantidade que venderam:

```
filiais = ["A", "B", "C", "D",
"E"]
vendas = [5500, 4600, 8000,
5600, 3500]
meta = 5000

meta_ok = {filial: venda for
filial, venda in zip(filiais,
vendas) if venda >= 5000}
meta_ok

## Saída
# {'A': 5500, 'C': 8000, 'D':
5600}
```

Pareamos as filiais com a quantidade de vendas correspondente por meio do `zip()` e passamos o objeto `zip` para um par chave e valor só quando a condição de venda de mais de `5000` produtos foi atendida.

7.3.2 Dict comprehension com if na chave

Existe uma outra forma de trabalhar com condicionais para criar ou modificar dicionários, que seria por meio da estrutura `if-else` aplicada na chave.

Diferente da anterior, aqui a condição vai ser testada na chave de um dicionário, retornando valores diferentes de acordo com o resultado da condicional. Neste exemplo, a sintaxe da dict comprehension é a seguinte:

```
{chave_if if cond else
chave_else: valor for chave,
valor in iteravel}
```

Traduzindo essa estrutura para a linguagem escrita seria como:
"Aplique `chave_if` se a condição `cond` for aceita, caso contrário aplique `chave_else` para cada `item` do `iteravel`".

Por exemplo, podemos alterar a chave de um dicionário com o nome das amostras e os valores de volume de um produto. Caso este produto tenha entre 190 ml e 210 ml ele estará aprovado:

```
p = {"A": 200, "B": 180, "C":
240, "D": 210}

res = {f'{am}-ok'
if (vol >= 190 and vol <= 210)
else f'{am}-rep': vol
for am, vol in p.items()}

res

## Saída
# {'A-ok': 200, 'B-rep': 180,
#  'C-rep': 240, 'D-ok': 210}
```

Alteramos o nome das chaves nos dicionários de acordo com a condição. Esse tipo de estrutura pode ser importante para modificar as chaves de dicionário de maneira mais simples de acordo com métricas ou valores padrões que definimos.

7.3.3 Dict comprehension com if no valor

Outra forma de trabalhar com condicionais para criar ou modificar dicionários é por meio da estrutura `if-else` aplicada ao valor.

É bem semelhante ao caso anterior, apenas trocando o local que deve ser alterado de acordo com a condição a ser testada. Neste novo exemplo, a sintaxe da dict comprehension é a seguinte:

```
{chave: valor_if if cond else
valor_else for chave, valor in
iteravel}
```

Por exemplo, podemos gerar um dicionário por meio de uma lista de números, em que as chaves serão cada um dos números e os valores retornam se esses números são múltiplos de 4:

```
numeros = [12, 31, 74, 92, 60]

mult_4 = {num: True if num % 4
== 0 else False for num in
numeros}
mult_4

## Saída
# {12: True, 31: False,
#  74: False, 92: True,
#  60: True}
```

Note que por meio de uma lista geramos um dicionário. Os itens se tornaram chaves e as verificações de múltiplo de 4 se tornaram o valores.

Então, não só utilizamos esse caso para alterar dicionários, mas para criá-los também. Isso tudo de acordo com condições nos valores ou chaves dos dicionários a depender da posição da condicional.

8 LIDANDO COM EXCEÇÕES

Ao longo da nossa trajetória neste curso, observamos alguns erros durante a execução dos nossos códigos.

Muitos desses erros são conhecidos no Python como exceções que são erros detectados durante a execução e, se não tratadas, interrompem o fluxo do programa, encerrando-o.

8.1 Tipos de Exceções

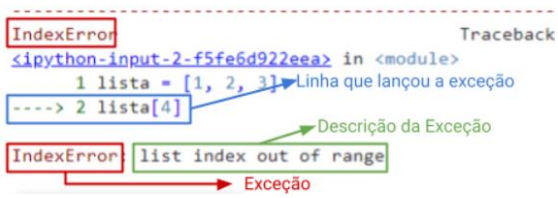
No Python existem basicamente duas formas distintas de erros: os **erros de sintaxe** e as **exceções**. As exceções são uma forma de lidar com erros e situações inesperadas no código, garantindo um fluxo de execução mais controlado.

Como uma pessoa cientista de dados, você precisará ter atenção a situações como esta para evitar bugs ou problemas em seus códigos e análises que possam afetar a experiência tanto da pessoa usuária quanto a eficiência da sua análise.

8.1.1 Traceback

Traceback é uma mensagem de erro que o Python exibe quando ocorre uma exceção. Ele mostra o fluxo do código da linha que ocorreu o erro até a exceção que foi lançada, ajudando a identificar e corrigir o problema.

Uma exceção aparece em nossa saída mais ou menos como podemos notar na imagem a seguir:



Na parte superior da imagem, temos o **tipo da exceção** (*IndexError*), seguido dos rastros que chamaram a exceção até a **linha exata que lançou a exceção** `lista[4]`. Logo abaixo, temos novamente o **tipo da exceção** e uma **breve descrição** sobre ela (*list index out of range*).

Vamos agora observar alguns dos tipos mais comuns de exceções que podem ser lançadas no Python.

8.1.2 SyntaxError

Ocorre quando é detectado pelo ***parser*** (analisador) um erro na descrição do código. Normalmente uma seta aponta para a parte do código que gerou o erro, como uma espécie de dica onde o erro possa ter ocorrido.

```
print(10 / 2
```

Saída:

```
  File "<ipython-input-16-
2db3afa07d68>", line 1
    print(10/2
                ^
SyntaxError: unexpected EOF
while parsing
```

Perceba que esquecemos de fechar o parênteses na linha de código do *print* e por isso foi apresentado um erro de sintaxe, ou seja, de escrita de código.

8.1.3 NameError

Exceção lançada quando tentamos utilizar um nome de algum elemento que não está presente em nosso código.

```
raiz = sqrt(100)
```

Saída:

```
---------------------------------
NameError                 Traceback
(most recent call last)
<ipython-input-17-2e14e900fb9f>
in <module>
----> 1 raiz = sqrt(100)

NameError: name 'sqrt' is not
defined
```

Neste caso, o interpretador não consegue aplicar o método da raiz quadrada por não ter sido importado junto a biblioteca `math`.

8.1.4 IndexError

Exceção lançada quando tentamos indexar alguma estrutura de dados como lista, tupla ou até string além de seus limites.

```
lista = [1, 2, 3]
lista[4]
```

Saída:

```
---------------------------------
IndexError                Traceback
(most recent call last)
<ipython-input-18-f5fe6d922eea>
in <module>
      1 lista = [1, 2, 3]
----> 2 lista[4]

IndexError: list index out of
range
```

Para esta situação temos apenas 3 elementos na lista e tentamos ler o elemento da posição 4, que não existe. Recebendo a mensagem de que o `index` está fora da faixa.

8.1.5 TypeError

Exceção lançada quando um operador ou função são aplicados a um objeto cujo tipo é inapropriado.

```
"1" + 1
```

Saída:

```
---------------------------------
TypeError                 Traceback
(most recent call last)
<ipython-input-20-ec358fc6499a>
in <module>
----> 1 "1" + 1

TypeError: can only concatenate
str (not "int") to str
```

Tentamos “somar” uma string com um número inteiro que gerou uma exceção em nosso código. Isso ocorreu por 2 razões: o operador de soma foi considerado de concatenação, por iniciarmos utilizando uma string (nesse caso, o sinal de adição serve para

concatenar strings). Além disso, um valor do tipo inteiro não consegue ser concatenado nesse tipo de operação.

8.1.6 KeyError

Exceção lançada quando tentamos acessar uma chave que não está no dicionário presente em nosso código.

```
estados = {'Bahia': 1, 'São Paulo': 2, 'Goiás': 3}
estados["Amazonas"]
```

Saída:

```
---------------------------------
KeyError                  Traceback (most recent call last)
<ipython-input-22-45729db26889> in <module>
      1 estados = {'Bahia': 1, 'São Paulo': 2, 'Goiás': 3}
----> 2 estados["Amazonas"]

KeyError: 'Amazonas'
```

Tentamos acessar os dados do Estado `"Amazonas"` que não está presente no dicionário, lançando assim a exceção.

8.1.7 ValueError

Exceção lançada quando um argumento passado para uma função ou método não é do tipo esperado ou tem um valor inválido.

```
val = "dez"
int(val)
```

Saída:

```
---------------------------------
ValueError                Traceback (most recent call last)
<ipython-input-18-483bc4303492> in <module>
      1 val = "dez"
----> 2 int(val)

ValueError: invalid literal for int() with base 10: 'dez'
```

Tentamos converter uma string que não representa um número em um tipo numérico, como `int`. O Python reconhece um número escrito por extenso apenas como string.

8.1.8 Warning

Exceção lançada em situações que precisamos alertar a pessoa usuária sobre algumas condições do código.

Essas condições não necessariamente interrompem a execução do programa, mas podem lançar avisos sobre uso de módulos obsoletos, ou que possam ser depreciados em atualizações futuras ou também para alterações que podem reverberar sobre alguma parte do código.

Lembrando que, no caso dos ***Warnings*** eles podem ser ignorados ou tratados como exceções.

```
import numpy as np
a = np.arange(5)
a / a  # apresenta um warning
```

Saída:

```
<ipython-input-23-93a37b275923>:4: RuntimeWarning: invalid value encountered in true_divide
  a / a  # apresenta um warning
array([nan,  1.,  1.,  1.,  1.])
```

Tentamos fazer a divisão de zero por zero. Em um **array Numpy**, que é essa estrutura na saída do console, esse resultado gera um valor `nan` (Not a Number).

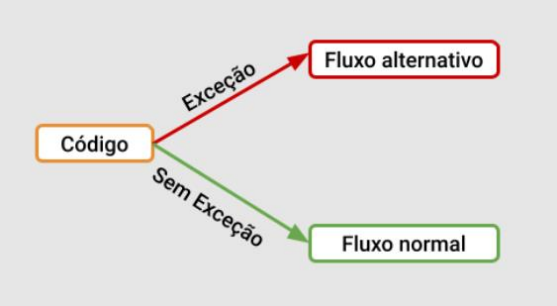
É possível seguir a execução do programa, mas é provável que precise tratar os dados para poder utilizar esse array em alguma operação mais a frente.

8.2 Tratando exceções

Se o seu código não lida com alguns dos tipos de exceções existentes no Python, ele irá parar de funcionar.

É nesse ponto que entra o **tratamento com as exceções**. Por meio dele, identificamos os erros ocorridos no código fornecendo mais detalhes sobre aquela exceção e contribuindo no estabelecimento de um fluxo alternativo para a execução do código evitando a interrupção dos processos inesperadamente.

Quando trabalhamos com exceções passamos para o nosso programa o que fazer se uma exceção for lançada:



Na imagem acima, o código Python apresenta uma rota alternativa de execução a depender do lançamento ou não de uma exceção.

É bem interessante navegar na documentação do Python para visualizar algumas das exceções e suas hierarquias.

Vamos aprender a identificar e tratar algumas das exceções.

8.2.1 try … except

Tratamos exceções em Python por meio de cláusulas `try`, `except`, `else` e `finally`.

O tratamento de exceções é feito em pelo menos duas etapas: a captura da exceção e o tratamento da mesma por meio de uma ação que minimize ou resolva o problema ocorrido.

O caso mais básico de tratamento da exceção é por meio do uso das cláusulas `try` e `except`:

```
try:
  # código a ser executado normalmente
except:
  # código se uma exceção for lançada no try
```

O primeiro bloco é o `try`. Nele adicionamos o código que deverá ser executado e pode ser levantado uma exceção (código crítico). O bloco `except` é responsável por rodar um fluxo alternativo do código para tratar a exceção.

Como exemplo para este capítulo, vamos trabalhar com o seguinte dicionário que possui a quantidade de vendas de um produto nos quatro trimestres de 2022 de filiais de uma empresa de tecnologia separadas por listas:

```
vendas = {
 'A': [6180, 8067, 7294, 3025],
 'B': [6078, 8381, 2800, 4471],
 'C': [2621, 3135, 3504, 2237],
 'D': [4423, 7921, 6347, 5914]}
```

Para acessar a lista das vendas trimestrais de uma filial devemos passar o índice correspondente à filial que queremos avaliar.

Temos um ponto de falha aqui: acessar uma filial inexistente. Para tratar

esse caso podemos utilizar uma exceção genérica (por meio do `Exception`) ou uma específica (`KeyError`):

- Capturando exceções genéricas

```
try:
  filial = input("Digite a
filial: ").upper()
  vend_tri = vendas[filial]
except Exception as e:
  print(type(e), f'Erro: {e}')
```

Se digitarmos, por exemplo, uma filial inexistente `"E"` recebemos a saída:

```
## Entrada
Digite a filial: E
## Saída
# <class 'KeyError'> Erro: 'E'
```

- Capturando exceções específicas

```
try:
  filial = input("Digite a
filial: ").upper()
  vend_tri = vendas[filial]
except KeyError as e:
  print(f'{e} não é uma filial
aceita')
```

Se digitarmos novamente a filial inexistente `"E"` recebemos a saída:

```
## Entrada
Digite a filial: E
## Saída
# 'E' não é uma filial aceita
```

É bem importante que saibamos de antemão qual o tipo de erro que pode ocorrer para dar o devido tratamento àquele tipo específico de erro.

8.2.2 Cláusula else

A cláusula `else` permite executar um bloco de código, caso o código que rodamos dentro do bloco `try` tenha sido executado sem erros (ou seja, sem lançar exceção).

A sintaxe nesse tratamento é:

```
try:
  # código a ser executado
except:
  # código se houver exceção no
try
else:
  # código se não houver exceção
```

Neste caso, adicionamos o bloco `else` que será executado apenas se uma exceção **não** for lançada no `try`.

Seguindo o exemplo das vendas, podemos colocar a exibição das vendas trimestrais do produto nesta cláusula.

```
try:
  filial = input("Digite a
filial: ").upper()
  vend_tri = vendas[filial]
except KeyError as e:
  print(f'{e} não é uma filial
aceita')
else:
  for i in range(len(vend_tri)):
    print(f'{i+1}º Tri =
{vend_tri[i]} unidades')
```

Agora, vamos realizar dois testes:

- Sem exceção:

```
## Entrada
Digite a filial: A
## Saída
# 1º Tri = 6180 unidades
# 2º Tri = 8067 unidades
# 3º Tri = 7294 unidades
# 4º Tri = 3025 unidades
```

- Com exceção:

```
## Entrada
Digite a filial: E
## Saída
# 'E' não é uma filial aceita
```

Criamos dois fluxos para o nosso código: lançando ou não a exceção. Perceba que o código no bloco `else` não representa um código crítico, pois já tratamos o problema que poderia ocorrer antes desse ponto.

8.2.3 Cláusula finally

A cláusula `finally` permite que todo código inserido em seu bloco seja executado independentemente se houve ou não alguma exceção lançada no bloco `try`.

É usual utilizar este bloco para liberar recursos externos como conexões a bancos de dados, acesso a arquivos ou outros recursos.

A sintaxe nesse tratamento é:

```
try:
  # código a ser executado
except:
  # código se houver exceção no try
else:
  # código se não houver exceção
finally:
  # sempre rode essa parte
```

Neste caso, adicionamos o bloco `finally` que sempre será executado ao final do fluxo do código, tendo ou não uma exceção.

Seguindo o exemplo das vendas, podemos exibir uma mensagem validando o tipo de entrada no final da execução.

```
try:
  filial = input("Digite a filial: ").upper()
  vend_tri = vendas[filial]
except KeyError as e:
  print(f'{e} não é uma filial aceita')
else:
  for i in range(len(vend_tri)):
    print(f'{i+1}º Tri = {vend_tri[i]} unidades')
finally:
  if filial in vendas.keys():
    print("Consulta encerrada com sucesso")
  else:
    print("Consulta mal-sucedida")
```

Agora, vamos realizar dois testes:

- Sem exceção:

```
## Entrada
Digite a filial: C
## Saída
# 1º Tri = 2621 unidades
# 2º Tri = 3135 unidades
# 3º Tri = 3504 unidades
# 4º Tri = 2237 unidades
# Consulta encerrada com sucesso
```

- Com exceção:

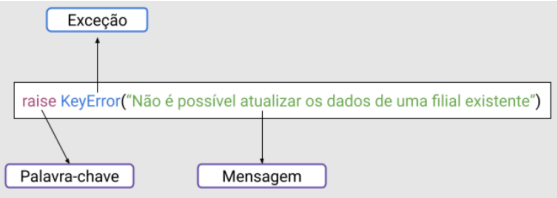
```
## Entrada
Digite a filial: P
## Saída
# 'P' não é uma filial aceita
# Consulta mal-sucedida
```

Note que independente de lançar uma exceção ou não o bloco de código criado no `finally` foi executado. Utilizamos o mesmo bloco para reforçar se a consulta foi bem sucedida ou não.

8.3 Raise

Uma outra forma de trabalhar com as exceções em seu código, é criar as suas próprias exceções para determinados comportamentos que deseja em seu código.

Para isso, utilizamos a palavra-chave `raise` junto ao tipo de exceção que desejamos lançar e a mensagem a ser exibida.



Podemos utilizar vários tipos de exceção para testar o comportamento do `raise`, mas aqui vamos focar em utilizá-lo com a exceção KeyError.

A sintaxe para lançar uma exceção própria é:

```
raise NomeDoErro("mensagem")
```

Ainda seguindo o exemplo das vendas, podemos modificá-lo para testar esse comportamento. Suponha que queremos adicionar uma filial que não está no dicionário `vendas` e passar pelo código a quantidade de vendas do produto em questão em cada trimestre.

Relembrando o dicionário do exemplo:

```
vendas = {
 'A': [6180, 8067, 7294, 3025],
 'B': [6078, 8381, 2800, 4471],
 'C': [2621, 3135, 3504, 2237],
 'D': [4423, 7921, 6347, 5914]}
```

Nesse problema há a possibilidade de pontos críticos no código. Vamos avaliar a entrada da filial pela pessoa usuária e lançaremos uma exceção caso seja repetido alguma delas no console. Além disso, vamos tratar de um erro, caso a pessoa usuária digite um valor que não seja um número inteiro na quantidade de produtos vendidos em cada trimestre.

```
try:
  vend_tri = []
  filial = input("Digite a
filial: ").upper()
  if filial in vendas.keys():
    raise KeyError("Não é
possível atualizar os dados de
uma filial existente")

  for i in range(4):
    tri = input(f"Quantidade
vendida no {i+1}º Tri: ")
    tri = int(tri)
    vend_tri.append(tri)
except KeyError as e:
  print(e)
except ValueError as e:
  print(f"Não foi possível
cadastrar o valor '{tri}' como
quantidade de listas. Só são
aceitos números inteiros!")
else:
  vendas[filial] = vend_tri
finally:
  if (len(vend_tri) == 4):
    print("Dados cadastrados com
sucesso!")
    print(vendas)
  else:
    print("Não foi possível
realizar o cadastro!")
```

Agora, vamos testar tanto o lançamento da exceção quanto a execução do código em fluxo normal:

- Lançando a exceção pelo `raise`:

```
## Entrada
Digite a filial: B
## Saída
# 'Não é possível atualizar os dados de uma filial existente'
# Não foi possível realizar o cadastro!
```

- Lançando a exceção `ValueError`:

```
## Entrada
Digite a filial: E
Quantidade vendida: 2500
Quantidade vendida: 100.5
## Saída
# Não foi possível cadastrar o valor '100.5' como quantidade de listas. Só são aceitos números inteiros!
# Não foi possível realizar o cadastro!
```

- Sem exceção:

```
## Entrada
Digite a filial: E
Quantidade vendida: 2500
Quantidade vendida: 4600
Quantidade vendida: 3800
Quantidade vendida: 4900
## Saída
# Dados cadastrados com sucesso!
# {
  'A': [6180, 8067, 7294, 3025],
  'B': [6078, 8381, 2800, 4471],
  'C': [2621, 3135, 3504, 2237],
  'D': [4423, 7921, 6347, 5914],
  'E': [2500, 4600, 3800, 4900]}
```

Trabalhamos com 2 tipos de exceção: uma que criamos por meio do uso do `raise` apresentando um comportamento indesejado no código e outro gerado por meio do uso de um número não inteiro ou string para a quantidade de produto.

Utilizamos também um bloco `finally` para reforçar se o cadastro foi bem sucedido ou não.

CHEGAMOS AO FIM

Esperamos que você tenha um bom proveito das informações colocadas aqui e que elas possam contribuir com seus estudos em Python e Data Science!

O resumo contém as informações já estudadas em aula. Para fortalecer o aprendizado em Python, é necessário praticar bastante. Continue sempre estudando e quaisquer dúvidas você pode utilizar o fórum e o Discord da Alura.

Bons estudos!

CRÉDITOS

Conteúdo
Afonso Rios
Mirla Borges

Designer gráfico
Alysson Manso

Acompanhamento técnico
Rodrigo Dias

Acompanhamento didático
Morgana Gomes
Thais de Faria

Apoio
Rômulo Henrique

alura | Escola de Data Science