Curso de Python

Básico e Análise Exploratória de Dados

Breno Cauã Rodrigues da Silva

Índice

Capítulo 1

Introdução

Bem-vindo ao **Curso de Python para Análise de Dados**! Este curso foi cuidadosamente desenvolvido para introduzir você às ferramentas e técnicas fundamentais necessárias para trabalhar com dados de forma eficiente e profissional. Python é uma das linguagens mais utilizadas no mundo da ciência de dados, graças à sua simplicidade, flexibilidade e poderosas bibliotecas.

1.1 Objetivo do Curso

O objetivo principal deste curso é capacitar você a realizar análises de dados desde a coleta, manipulação e visualização até a exportação de relatórios. Ao final do curso, você será capaz de: - Manipular grandes conjuntos de dados com eficiência. - Criar visualizações impactantes que auxiliam na tomada de decisões. - Aplicar técnicas exploratórias para compreender padrões e tendências nos dados.

1.2 Público-Alvo

Este curso é voltado para: - Iniciantes do curso que desejam iniciar seus estudos em Python para Análise de Dados - Estudantes que procuram uma introdução sólida à ciência de dados.

1.3 Estrutura do Curso

O curso está dividido em módulos que cobrem tópicos essenciais de forma progressiva:

1. **Python Básico** Aprenda os fundamentos da linguagem Python, incluindo sintaxe, estruturas de controle, manipulação de variáveis e funções.

2. **Numpy**
   Descubra como manipular arrays numéricos e realizar cálculos matemáticos com eficiência.

3. **Pandas** Explore como carregar, manipular e analisar dados tabulares usando DataFrames e Series.
4. **Matplotlib e Seaborn** Domine as ferramentas de visualização de dados para criar gráficos informativos e visuais.
5. **Relatório de Análise de Dados** Combine todas as ferramentas aprendidas para realizar análises exploratórias completas e gerar relatórios que apresentam insights valiosos.

1.4 Metodologia

O curso combina teoria com prática, oferecendo: - **Aulas explicativas**: Conceitos apresentados de forma clara e objetiva. - **Exemplos práticos**: Demonstração de como aplicar as ferramentas e técnicas no mundo real. - **Exercícios e projetos**: Oportunidade de praticar e consolidar o aprendizado com desafios variados.

1.5 Pré-requisitos

- Conhecimento básico em informática.
- Nenhuma experiência prévia em programação é necessária, embora seja desejável.

1.6 Ferramentas Necessárias

- Python 3.7 ou superior.
- Jupyter Notebook (recomendado), Programiz, Google Colaboratory, Jupyter Notebook Online ou qualquer editor de código, como VSCode.
- Instalação das bibliotecas: `numpy`, `pandas`, `matplotlib`, `seaborn`.

Esperamos que este curso seja uma jornada enriquecedora, ajudando você a adquirir habilidades práticas e teóricas para trabalhar com dados de forma confiante. Vamos começar!

Capítulo 2

1º Módulo: Python Básico

2.1 O que é Python?

Python é uma *linguagem de programação*. Isso significa basicamente duas coisas:

1. Existem regras que determinam como as palavras são dispostas, já que é uma linguagem;
2. O texto descreve instruções para o computador realizar tarefas.

Ou seja, podemos escrever um documento - que chamamos de código fonte - em Python para o computador ler e realizar nossos desejos e tarefas. **Mas porque usar Python?** Python tem algumas características interessantes:

- É interpretada, ou seja, o interpretador do Python executa o código fonte diretamente, traduzindo cada trecho para instruções de máquina;
- É de alto nível, ou seja, o interpretador se vira com detalhes técnicos do computador. Assim, desenvolver um código é mais simples do que em linguagens de baixo nível, nas quais o programador deve se preocupar com detalhes da máquina;
- É de propósito geral, ou seja, podemos usar Python para desenvolver programas em diversas áreas. Ao contrário de linguagens de domínio específico, que são especializadas e atendem somente a uma aplicação específica;
- Têm tipos dinâmicos, ou seja, o interpretador faz a magia de descobrir o que é cada variável.

Por essas e várias outras características, Python se torna uma linguagem simples, bela, legível e amigável. É uma linguagem utilizada por diversas empresas, como Wikipedia, Google, Yahoo!, CERN, NASA, Facebook, Amazon, Instagram, Spotify.

2.2 Instalação (Anaconda) e primeiro contato com o Jupyter Notebook

Acesse o link para realizar o download da Distribuição Anaconda: https://www.anaconda.com/download

Como material de auxílio para instalação, veja o video do YouTube: vídeo de suporte

Qualquer dúvida, entrem em contato comigo:

- Email: breno.silva@icen.ufpa.br
- Telefone: (91) 9 8120-3378

2.3 Sintaxe Básica

2.3.1 Função `print()`

Vamos começar com o famoso e clássico, *Hello World*, em Python.

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

```
Hello, World!
```

Em programação, é muito comum utilizar a palavra *imprimir* (ou *print*, em inglês) como sinônimo de mostrar algo na tela.

`print()` é uma função nativa do Python. Basta colocar algo dentro dos parênteses que o Python se encarrega de fazer a magia de escrever na tela! **:)**

2.3.1.1 Erros Comuns:

- **Usar a letra P maiúscula ao invés de minúscula:**

```
[Input]  : Print("Hello World!")
[Output] : NameError: name 'Print' is not defined
```

- **Esquecer de abrir e fechar aspas no texto que é passado para a função `print()`:**

```
[Input]  : print(Hello, World!)
[Output] : SyntaxError: invalid syntax
```

- **Esquecer de abrir ou fechar as aspas:**

```
[Input]  : print("Hello, World!)
[Output] : SyntaxError: unterminated string literal (detected at line 1)
```

- **Começar com aspas simples e terminar com aspas duplas ou vice-versa:**

```
[Input]  : print('Hello, World!")
[Output] : SyntaxError: unterminated string literal (detected at line 1)
```

Mas, e se eu precisar usar aspas dentro do texto a ser mostrado na tela? Bem, caso queira imprimir aspas duplas ou apas simples, siga os dois exemplos abaixo:

```
print('Python é legal! Mas não o "legal" como dizem pra outras coisas')
```

```
Python é legal! Mas não o "legal" como dizem pra outras coisas
```

```
print("Python é legal! Mas não o 'legal' como dizem pra outras coisas")
```

```
Python é legal! Mas não o 'legal' como dizem pra outras coisas
```

E como faz para imprimir um texto em várias linhas? Bom, para isso precisamos lembrar de um caractere especial, a quebra de linha: `\n`. Esse n é um caractere especial que significa aqui acaba a linha, o que vier depois deve ficar na linha de baixo. Por exemplo:

```
print("Veja esse texto. \n Aspas duplas: \" \n Aspas simples: '")
```

```
Veja esse texto.
 Aspas duplas: "
 Aspas simples: '
```

2.3.2 Operadores matemáticos

A linguagem Python possui operadores que utilizam símbolos especiais para representar operações de cálculos, assim como na matemática:

- Operação de Soma/Adição (+) & Operação de Subtração (−):

```
2 + 3
```

```
5
```

```
1.77 + 4.95
```

```
6.720000000000001
```

```
6 - 4
```

```
2
```

```
7 - 8
```

```
-1
```

- Operação de Multiplicação/Produto (∗) & Operação de Divisão (/):

```
7 * 8
```

```
56
```

```
7.5 * 8.9
```

```
66.75
```

```
2 * 2 * 2
```

```
8
```

```
10 / 3
```

```
3.3333333333333335
```

```
666 / 137
```

```
4.861313868613139
```

```
50 / 0.75
```

```
66.66666666666667
```

E se fizermos uma divisão por zero?

```
[Input]  : 1 / 0
[Output] : ZeroDivisionError: division by zero
```

- Outras formas de dividir:

1. Divisão inteira (//):

```
10 // 3
```

```
3
```

```
666 // 137
```

```
4
```

```
50 // 0.75
```

```
66.0
```

2. Resto da Divisão (%):

```
1 % 2
```

```
1
```

```
4 % 2
```

```
0
```

```
9 % 3
```

```
0
```

Agora que aprendemos os operadores aritméticos básicos podemos seguir adiante. Como podemos calcular 210 ? O jeito mais óbvio seria multiplicar o número dois dez vezes:

```
2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2
```

```
1024
```

Porém, isso não é muito prático, pois há um operador específico para isso, chamado de Potenciação/Exponenciação: (**):

```
2 ** 10
```

```
1024
```

```
10 ** 3
```

```
1000
```

```
9 ** 0
```

```
1
```

E a *raiz quadrada*? Lembrando que $\sqrt{x} = x^{\frac{1}{2}}$, então podemos calcular a raiz quadrada do seguinte modo:

```
81 ** 0.5
```

```
9.0
```

Porém, a maneira mais recomendada para fazer isso é usar a função `sqrt()` da biblioteca **math**:

```
import math
math.sqrt(81)
```

```
9.0
```

Na primeira linha do exemplo importamos, da biblioteca padrão do Python, o módulo **math** e então usamos a sua função `sqrt()` para calcular $\sqrt{81}$. Não esqueça que é preciso ter executado `import math` antes de usar as funções e constantes dessa biblioteca.

2.3.3 Expressões Numéricas

Agora que já aprendemos diversos operadores, podemos combiná-los e resolver problemas mais complexos:

```
3 + 4 * 2
```

```
11
```

```
7 + 3 * 6 - 4 ** 2
```

```
9
```

```
(3 + 4) * 2
```

```
14
```

```
1 / (8 / 4) ** (5 - 2)
```

```
0.125
```

Quando mais de um operador aparece em uma expressão, a ordem de avaliação depende das regras de precedência. O Python segue as mesmas regras de precedência da matemática. O acrônimo **PEMDAS** ajuda a lembrar essa ordem:

1. **P**arênteses
2. **E**xponenciação
3. **M**ultiplicação e **D**ivisão (mesma precedência)
4. **A**dição e **S**ubtração (mesma precedência)

2.3.3.1 Nota:

Você observou um comportamento inesperado ao trabalhar com números decimais em Python, mas isso não é um erro no seu código ou na linguagem. Na verdade, essa é uma característica inerente à forma como os computadores representam números de ponto flutuante.

Por que isso acontece?

A maioria dos computadores utiliza a representação binária (base 2) para armazenar números. O problema é que muitas frações decimais (base 10), como 0.1, não podem ser representadas exatamente como frações binárias. Isso ocorre porque a representação binária tem um número finito de dígitos, assim como a representação decimal.

Imagine tentar representar 1/3 em decimal: 0.33333... A sequência de 3's se repete infinitamente. Da mesma forma, ao converter 0.1 para binário, obtemos uma fração infinita.

IEEE 754 e a precisão limitada:

A maioria dos computadores modernos utiliza o padrão IEEE 754 para representar números de ponto flutuante. Esse padrão define como os números são armazenados em memória, incluindo a precisão. Em Python, os números de ponto flutuante geralmente correspondem à precisão dupla do IEEE 754, que oferece 53 bits de precisão.

Quando você digita 0.1 em Python, o computador tenta encontrar a fração binária mais próxima que possa ser representada com esses 53 bits. O resultado é um valor muito próximo de 0.1, mas não exatamente igual.

Implicações práticas:

Essa limitação na representação de números em ponto flutuante pode levar a pequenos erros de arredondamento em cálculos. Por exemplo, 0.1 + 0.2 pode não resultar em exatamente 0.3.

2.3.4 Comparações Lógicas

A Tabela 2.1 mostra operadores de comparação em Python, veja:

Tabela 2.1: Tabela de Operações e seus Significados

Operação	Significado
<	menor que
>	maior que
<=	menor ou igual que
>=	maior ou igual que
==	igual
!=	diferente

Veja alguns exemplos:

```
2 < 10
```

```
True
```

```
2 > 10
```

```
False
```

```
print('Comparação Lógica - 10 menor que 10:', 10 < 10)
```

```
Comparação Lógica - 10 menor que 10: False
```

```
print('Comparação Lógica - 10 maior que 10:', 10 > 10)
```

```
Comparação Lógica - 10 maior que 10: False
```

```
print('Comparação Lógica - 10 menor ou igual a 10:', 10 <= 10)
```

```
Comparação Lógica - 10 menor ou igual a 10: True
```

```
print('Comparação Lógica - 10 maior ou igual a 10:', 10 >= 10)
```

```
Comparação Lógica - 10 maior ou igual a 10: True
```

```
print('Comparação Lógica - 10 igual a 10:', 10 == 10)
```

```
Comparação Lógica - 10 igual a 10: True
```

```
print('Comparação Lógica - 10 diferente de 10:', 10 != 10)
```

```
Comparação Lógica - 10 diferente de 10: False
```

A Tabela 2.2 mostra conectores lógicos em Python, veja:

Tabela 2.2: Tabela de Operações e seus Significados

Operação	Significado
and	1ª condição E 2ª condição
or	1ª condição OU 2ª condição

Tabela 2.2: Tabela de Operações e seus Significados

Operação	Significado
not	Negação (Não)
in	Está contido em

Veja alguns exemplos:

```
print((1 and 4) < 3)
```

```
False
```

```
print((1 or 4) < 3)
```

```
True
```

```
print((1 and 2 and 2.99) < 3)
```

```
True
```

```
print((1 or 2 or 2.99) > 3)
```

```
False
```

```
print((5 >= 4.99) and (10 <= 10.01))
```

```
True
```

```
print((5 >= 4.99) and (10 == 10.01))
```

```
False
```

```
print((5 >= 4.99) or (10 <= 10.01))
```

```
True
```

```
print((5 >= 4.99) or (10 == 10.01))
```

```
True
```

```
print(1 == 1)
```

```
True
```

```
print(not 1 == 1)
```

```
False
```

```
print(not not 1 == 1)
```

```
True
```

```
print(not not not 1 == 1)
```

```
False
```

> **Nota:** Assim como os operadores aritméticos, os operadores boleanos também possuem uma ordem de prioridade: `not` tem maior prioridade que `and` que tem maior prioridade que `or`:

```
not False and True or False
```

```
True
```

2.3.5 Variáveis e Atribuições

Variável é nome que se refere a um valor. Atribuição é processo de criar uma nova variável e dar um novo valor a ela.

Exemplos:

```
numero = 2 * 3
numero
```

```
6
```

```
frase = "Me dá um copo d'água."
frase
```

```
"Me dá um copo d'água."
```

```
pi = 3.141592
print(pi)
```

```
3.141592
```

```
pi = math.pi
print(pi)
```

```
3.141592653589793
```

No exemplo anterior realizamos três atribuições. No primeiro atribuímos um número inteiro à variável de nome `numero`; no segundo uma frase à variável `frase`; no último um número de ponto flutuante à `pi`.

2.3.5.1 Nome para Variáveis

Bons programadores escolhem nomes significativos para as suas variáveis - eles documentam o propósito da variável.

Nomes de variáveis podem ter o tamanho que você achar necessário e podem conter tanto letras como números, porém não podem começar com números. É possível usar letras maiúsculas, porém a convenção é utilizar somente letras minúsculas para nomes de variáveis.

```
crieiumavariavelcomnomegiganteeestoucompreguiçadeescrevertudodenovo = 10
crieiumavariavelcomnomegiganteeestoucompreguiçadeescrevertudodenovo
```

```
10
```

Tentar dar um nome ilegal a uma variável ocasionará erro de sintaxe:

```
[Input]  : 123voa = 10
[Output] : SyntaxError: invalid decimal literal
```

```
[Input]  : ol@ = "oi"
[Output] : SyntaxError: invalid syntax
```

```
[Input]  : def = 2.7
[Output] : SyntaxError: invalid syntax
```

`123voa` é ilegal pois começa com um número. `ol@` é ilegal pois contém um caractere inválido (@), mas o que há de errado com def? A questão é que `def` é uma ***palavra-chave da linguagem***. O Python possui diversas palavras-chave que são utilizadas na estrutura dos programas, por isso não podem ser utilizadas como nomes de variáveis.

Outro ponto importante é que não é possível acessar variáveis que ainda não foram definidas.

```
[Input]  : nao_definida
[Output] : NameError: name 'nao_definida' is not defined
```

Também podemos atribuir expressões a uma variável:

```
x = 3 * 5 - 2
print(x)
```

```
13
```

```
y = 3 * x + 10
print(y)
```

```
49
```

```
z = x + y
print(z)
```

```
62
```

```
n = 10
n + 2 # 10 + 2
```

```
12
```

```
9 - n
```

```
-1
```

Outra forma de somar/multiplicar na variável:

```
num = 4
num += 3
print(num)
```

7

```
num = 2
num *= 3
print(num)
```

6

2.3.5.2 Atribuição Múltipla

Uma funcionalidade interessante do Python é que ele permite atribuição múltipla. Isso é muito útil para trocar o valor de duas variáveis:

```
a = 1
b = 200
```

Para fazer essa troca em outras linguagens é necessário utilizar uma variável auxiliar para não perdemos um dos valores que queremos trocar. Vamos começar da maneira mais simples:

```
a = b # Perde-se o valor original de `a` (1)
a
```

200

```
b = a # Como perdeu-se `a`, `b` vai continuar com seu valor original (200)
b
```

200

A troca é bem sucedida se usamos uma variável auxiliar:

```
a = 1
b = 200
print(a, b)
```

1 200

```
aux = a
a = b
b = aux
print(a, b)
```

200 1

A atribuição múltipla também pode ser utilizada para simplificar a atribuição de variáveis, por exemplo:

```
a, b = 1, 200
print(a, b)
```

```
1 200
```

```
a, b, c, d = 1, 2, 3, 4
print(a, b, c, d)
```

```
1 2 3 4
```

```
a, b, c, d = d, c, b, a
print(a, b, c, d)
```

```
4 3 2 1
```

2.3.6 Tipos de Objetos

Criamos muitas variáveis até agora. Você lembra o tipo de cada uma? Para saber o tipo de um objeto ou variável, usamos a função `type()`:

```
x = 1
print(type(x))
```

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

```
y = 2.3
print(type(y))
```

```
<class 'float'>
```

```
palavra = "Python"
print(type(palavra))
```

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

```
logit = True
print(type(logit))
```

```
<class 'bool'>
```

Python vem com alguns tipos básicos de objetos, dentre eles:

- `bool`: Verdadeiro ou Falso (True or False);
- `int`: Números Inteiros;
- `float`: Números Reais;
- `complex`: Números Complexos;
- `str`: Textos ou conjunto de caracteres (strings);
- `list`: listas;
- `dict`: Dicionários.

Os demais tipos de objetos serão vistos mais a frente com mais afinco.

2.3.7 Buscando ajuda

Está com dúvida em alguma coisa? Use a função `help()` e depois digite o que você busca.

```
help()
```

```
Welcome to Python 3.8's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at https://docs.python.org/3.8/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help>
You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)".  Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
```

E para buscar ajuda em uma coisa específica?

```
help(print)
```

```
Help on built-in function print in module builtins:

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

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

```
help(math.sqrt)
```

```
Help on built-in function sqrt in module math:

sqrt(x, /)
```

```
    Return the square root of x.
```

2.3.8 Autoconhecimento do código

Em algum momento durante o seu código você pode querer saber quais variáveis já foram declaradas, ou até mesmo o valor atual delas. Podemos listar todas as variáveis declaradas no código usando o comando `dir()`. Veja um exemplo:

```
a = 1
b = 2
dir()
```

```
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__',
```

Veja que nossas variáveis declaradas aparecem proxímo do final do resultado de `dir()`. Não se assuste com os outros elementos que aparecem nesse resultado. Essas variáveis são criadas e usadas pelo próprio Python, e não são importantes nesse momento.

Outra opção para visualizar as variáveis declaradas são os comandos `globals()` e `locals()`. Ambas mostram não só as variáveis declaradas, mas também seu valor atual. A diferença entre ambas está no escopo em que atuam, mas veja que seus resultados são semelhantes:

```
globals()
```

```
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '_fr
```

```
locals()
```

```
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '_fr
```

Caso você esteja usando o IPython, os comandos mágicos `%who` e `%whos` são ótimas alternativas ao que já vimos anteriormente, pois retiram do resultado as variáveis declaradas pelo próprio Python, permitindo uma melhor visualização das que você mesmo declarou. Olhe como o IPython pode simplificar nossa vida nesse caso:

```
[Input]  : %who
```

```
[Input]  : %whos
```

2.3.9 Strings

Tamém chamada de sequência de caracteres, textos ou dados alfanuméricos. *Strings* são tipos que armazenam uma *sequência de caracteres*:

```
"Texto com acentos de cedilhas: hoje é dia de caça!"
```

```
'Texto com acentos de cedilhas: hoje é dia de caça!'
```

```
# As strings aceitam aspas simples também
nome = 'Silvio Santos'
nome
```

```
'Silvio Santos'
```

Também é possível fazer algumas operações com strings:

```
nome * 3
```

```
'Silvio SantosSilvio SantosSilvio Santos'
```

```
[Input]:  nome * 3.14
[Output]: TypeError: can't multiply sequence by non-int of type 'float'
```

```
canto1 = 'vem aí, '
canto2 = 'lá '
nome + ' ' + canto1 + canto2 * 6 + '!!'
```

```
'Silvio Santos vem aí, lá lá lá lá lá lá !!'
```

```
# Para strings em várias linhas, utilize 3 aspas
str_grande = ''' Aqui consigo inserir um textão com várias linhas, posso iniciar em uma
... e posso continuar em outra ...
... e em outra ...
... e mais uma ...
... e acabou.'''
```

```
str_grande
```

```
' Aqui consigo inserir um textão com várias linhas, posso iniciar em uma ...\n... e poss
```

```
print(str_grande)
```

```
 Aqui consigo inserir um textão com várias linhas, posso iniciar em uma ...
... e posso continuar em outra ...
... e em outra ...
... e mais uma ...
... e acabou.
```

Caso queira um texto que dentro tem aspas, como Me dá um copo d'água, é necessário utilizar aspas duplas para formar a string:

```
agua = "Me dá um copo d'água"
agua
```

```
"Me dá um copo d'água"
```

E também é possível utilizar aspas simples, duplas e triplas ao mesmo tempo! Olha só:

```
todas_as_aspas = """Essa é uma string que tem:
                    - aspas 'simples'
                    - aspas "duplas"
                    - aspas '''triplas'''
                    Legal né?"""
```

```
todas_as_aspas
```

```
'Essa é uma string que tem:\n                    - aspas \'simples\'\n
```

```
print(todas_as_aspas)
```

```
Essa é uma string que tem:
                    - aspas 'simples'
                    - aspas "duplas"
                    - aspas '''triplas'''
                    Legal né?
```

2.3.9.1 Tamanho

A função embutida `len()` nos permite, entre outras coisas, saber o tamanho de uma *string*:

```
len('Abracadabra')
```

```
11
```

```
palavras = 'Faz um pull request lá'
len(palavras)
```

```
22
```

Resaltando que a função `len()` conta o número de **caracteres** e não somente o número de letras, que contém uma `string`.

2.3.9.2 Índices

Como visto anteriormente, a função `len()` pode ser utilizado para obter o tamanho de estruturas, sejam elas strings, listas, etc. Esse tamanho representa a quantidade de elementos na estrutura.

Para obter somente um caractere de dentro dessas estruturas, deve-se utilizar o acesso por índices, no qual o índice entre colchetes `[]` representa a posição do elemento que se deseja acessar.

> **Nota:** Os índices começam em zero.

P	y	t	h	o	n
0	1	2	3	4	5

```
palavra = "Python"
```

```
palavra[0] # primeiro caractere
```

```
'P'
```

```
palavra[5] # último caractere
```

```
'n'
```

Índices negativos correspondem à percorrer a estrutura na ordem reversa:

```
palavra[-1] # último caractere
```

```
'n'
```

```
palavra[-3] # terceira de trás para frente
```

```
'h'
```

2.3.9.3 Fatiamento

Se, ao invés de obter apenas um elemento de uma estrutura, deseja-se obter múltiplos elementos, deve-se utilizar slicing (fatiamento). No lugar de colocar o índice do elemento entre chaves, deve-se colocar o índice do primeiro elemento, dois pontos (`:`) e o próximo índice do último elemento desejado, tudo entre colchetes. Por exemplo:

```
frase = "Aprender Python é muito divertido!"
```

```
frase[0:8] # Slicing do primeiro até o (8 - 1) caractere
```

```
'Aprender'
```

```
frase[9:] # Omitir o segundo índice significa 'obter até o final'
```

```
'Python é muito divertido!'
```

```
frase[:8] # Omitir o primeiro indice, significa 'obter desde o começo'
```

```
'Aprender'
```

```
frase[:8] # Omitir o primeiro indice, significa 'obter desde o começo'
```

```
'Aprender'
```

```
frase[:] # Toda string!
```

```
'Aprender Python é muito divertido!'
```

Também funciona com índices negativos.

```
frase[::1] # Do começo ao fim, de 1 em 1 caractere. Logo, não faz nenhuma diferença
```

```
'Aprender Python é muito divertido!'
```

```
frase[::2] # Do começo ao fim,de 2 em 2
```

```
'Arne yhnémiodvrio'
```

```
frase[2:-2:2] # Do terceiro até o ante penúltimo, de 2 em 2
```

```
'rne yhnémiodvri'
```

É possível controlar o passo que a fatia usa. Para isso, coloca-se mais um dois pontos (`:`) depois do segundo índice e o tamanho do passo:

```
frase[:8] # Omitir o primeiro indice, significa 'obter desde o começo'
```

```
'Aprender'
```

```
frase[:8] # Omitir o primeiro indice, significa 'obter desde o começo'
```

```
'Aprender'
```

```
frase[:] # Toda string!
```

```
'Aprender Python é muito divertido!'
```

Resumindo: para fazer uma fatia de nossa string, precisamos saber de onde começa, até onde vai e o tamanho do passo.

```
fatiável[começo : fim : passo]
```

> **Nota:** As fatias incluem o índice do primeiro elemento e não incluem o elemento do índice final. Por isso que `frase[0:-1]` perde o último elemento.

Caso o **fim** da fatia seja antes do começo, obtemos um resultado vazio:

```
[Input]  : frase[15:2]
[Output] : ''
```

E se quisermos uma fatia fora da string?

```
[Input]  : frase[123:345]
[Output] : ''
```

Mas e se o **fim** da fatia for maior que o tamanho da string? Não tem problemas, o Python vai até o onde der:

```
[Input]  : frase[8:123456789]
[Output] : ' Python é muito divertido!'
```

```
[Input]  : frase[8:]
[Output] : ' Python é muito divertido!'
```

Surge problemas quando não existe o valor do índice passado dentro do colchetes.

```
[Input]  : frase[123456789]
[Output] : IndexError: string index out of range
```

Tamanhos negativos de passo também funcionam. Passos positivos significam para frente e passos negativos significam para trás:

```
[Input]  : "Python"[::-1]
[Output] : 'nohtyP'
```

Quando usamos passos negativos, a fatia começa no **fim** e termina no **começo** e é percorrida ao contrário. Ou seja, invertemos a ordem. Mas tome cuidado:

```
[Input]  : "Python"[2:6]
[Output] : 'thon'
```

```
[Input]  : "Python"[2:6:-1]
[Output] : ''
```

```
[Input]  : "Python"[6:2]
[Output] : ''
```

```
[Input]  : "Python"[6:2:-1]
[Output] : 'noh'
```

No caso de `"Python"[6:2]`, o começo é depois do fim. Por isso a string fica vazia.

No caso de `"Python"[2:6:-1]`, o começo é o índice 6, o fim é o índice 2, percorrida ao contrário. Ou seja, temos uma string vazia ao contrário, que continua vazia.

Quando fazemos `"Python"[6:2:-1]`, o começo é o índice 2, o fim é o índice 6, percorrida ao contrário. Lembre que o índice final nunca é incluído. Ou seja, temos a *string* `hon` a ser invertida. O que resulta em `noh`.

2.3.9.4 Separação de strings

Usando a função `split()`:

```
frase
```

```
'Aprender Python é muito divertido!'
```

```
frase.split()
```

```
['Aprender', 'Python', 'é', 'muito', 'divertido!']
```

```
todas_as_aspas.split('\n')
```

```
['Essa é uma string que tem:', "                    - aspas 'simples'", '
```

2.3.10 Entrada de dados

Em Python também é possível ler do teclado as informações digitadas pelo usuário. E isso é feito por meio da função embutida `input()` da seguinte forma:

```
[Input]  : valor_lido = input("Digite um valor: ")
           Digite um valor: 10
```

```
[Input]  : type(valor_lido) # deve-se notar que o valor lido é SEMPRE do tipo string
[Output] : str
```

Nota: A função `input()` «termina» de ser executada quando pressionamos *enter*.
O valor lido é sempre do tipo string.

Mas, como realizar operações com os valores lidos?

```
[Input]  : valor_lido = int(input("Digite um valor: "))
           print(f"O quadrado do número fornecido na entreda é: {valor_lido ** 2}")

[Output] : Digite um valor: 10
           O quadrado do número fornecido na entrada é 100
```

Tudo o que for digitado no teclado, até pressionar a tecla enter, será capturado pela função `input()`. Isso significa que podemos ler palavras separadas por um espaço, ou seja, uma frase inteira:

```
[Input]  : frase = input()

[Output] : Amanhã nos veremos!
```

2.3.10.1 Formatação

```
[Input]  : nome = input("Digite seu nome: ")

[Output] : Digite seu nome: Breno Cauã

[Input]  : frase = "Olá, {}".format(nome)
           frase

[Output] : Olá, Breno Cauã
```

Vale lembrar que as chaves `{}` só são trocadas pelo valor após a chamada do método `str.format()`:

```
[Input]  : string_a_ser_formatada = '{} me formate!'
          string_a_ser_formatada

[Output] : '{} me formate!'

[Input]  : string_a_ser_formatada.format("Não")

[Output] : 'Não me formate!'
```

A string a ser formatada não é alterada nesse processo, já que não foi feita nenhuma atribuição:

```
[Input]  : string_a_ser_formatada

[Output] : '{} me formate!'
```

2.3.10.1.1 Alternativa ao `format()`

Uma maneira mais recente de formatar strings foi introduzida a partir da versão 3.6 do Python: PEP 498 – Literal String Interpolation, carinhosamente conhecida como fstrings e funciona da seguinte forma:

```
nome = "Breno"
f"Olá, {nome}."
```

```
'Olá, Breno.'
```

```
num = 12
print(
    f"""
    Número de meses em um ano: {num}.
    Número de meses em um semestre: {num // 2}.
    Número de meses em um trimestre: {num // 4}.
    Número de meses em um quadrimestre: {num // 3}.
    Número de meses em um bimestre: {num // 6}
    """
)
```

```
    Número de meses em um ano: 12.
    Número de meses em um semestre: 6.
    Número de meses em um trimestre: 3.
    Número de meses em um quadrimestre: 4.
    Número de meses em um bimestre: 2
```

2.3.11 Listas

Listas são estruturas de dados capazes de armazenar múltiplos elementos.

2.3.11.1 Declaração

Para a criação de uma lista, basta colocar os elementos separados por vírgulas dentro de colchetes `[]`, como no exemplo abaixo:

```
nomes_frutas = ["maça", "banana", "abacaxi"]
nomes_frutas
```

```
['maça', 'banana', 'abacaxi']
```

```
numeros = [2, 13, 17, 47]
numeros
```

```
[2, 13, 17, 47]
```

Uma lista também pode ser contida por diferentes tipos de elementos, por exemplo:

```
['lorem ipsum', 150, 1.3, [-1, -2]]
```

```
['lorem ipsum', 150, 1.3, [-1, -2]]
```

Uma lista também pode ser vazia, algo que futaremente veremos que pode ser muito útil, por exemplo:

```
vazia = []
vazia
```

```
[]
```

2.3.11.2 Índices

Assim como nas strings, é possível acessar separadamente cada item de uma lista a partir de seu índice:

```
lista = [100, 200, 300, 400, 500]
lista[0] # Primeiro elemento
```

```
100
```

```
lista[2] # Terceiro elemento
```

```
300
```

```
lista[4] # Último elemento
```

```
500
```

```
lista[-1] # Outra maneira de acessar o último elemento
```

```
500
```

Tentar acessar uma posição inválida de uma lista causa um erro:

```
[Input]  : lista[10]
```

```
[Output] : IndexError: list index out of range
```

2.3.11.3 Slincing (Fatiamento)

Semelhante ao processo de fatiamento de strings.

```
lista[0:1] # Do começo até o primeiro elemento
```

```
[100]
```

```
lista[0:2] # Do começo até o segundo elemento (que está na posição de índice 1)
```

```
[100, 200]
```

```
lista[::2] # Do começo ao fim, de 2 em 2 elementos
```

```
[100, 300, 500]
```

```
lista[::-2] # Do fim ao começo, de 2 em 2 elementos
```

```
[500, 300, 100]
```

2.3.11.4 Verificações

Verificar se um elemento está contido em uma lista. Utilizando o conector lógico `in`.

```
lista_estranha = ['duas palavras', 42, True, ['batman', 'robin'], -0.84, 'hipófise']
42 in lista_estranha
```

```
True
```

```
'duas palavras' in lista_estranha
```

```
True
```

```
'batman' in lista_estranha
```

```
False
```

```
'batman' in lista_estranha[3] # Note que o elemento com índice 3 também é uma lista
```

```
True
```

Verificando o tamnaho de uma lista. É possível obter o tamanho da lista utilizando função `len()`:

```
len(lista)
```

```
5
```

```
len(lista_estranha)
```

```
6
```

```
len(lista_estranha[3])
```

```
2
```

2.3.11.5 Removendo itens da lista

Devido à lista ser uma estrutura mutável, é possível remover seus elementos utilizando o comando `del`:

```
lista_estranha
```

```
['duas palavras', 42, True, ['batman', 'robin'], -0.84, 'hipófise']
```

```
del lista_estranha[2]
lista_estranha
```

```
['duas palavras', 42, ['batman', 'robin'], -0.84, 'hipófise']
```

```
del lista_estranha[-1] # Remove o último elemento da lista
lista_estranha
```

```
['duas palavras', 42, ['batman', 'robin'], -0.84]
```

2.3.11.6 Trabalhando com listas

O operador + concatena listas:

```
a = [1, 2, 3]
b = [4, 5, 6]
c = a + b
c
```

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

O operador ∗ repete a lista dado um número de vezes:

```
[0] * 3
```

```
[0, 0, 0]
```

```
[1, 2, 3] * 2
```

```
[1, 2, 3, 1, 2, 3]
```

```
[1, 2, 3] * 2
```

```
[1, 2, 3, 1, 2, 3]
```

O método `append()` adiciona um elemento ao final da lista:

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

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

```
lista.append('e')
lista
```

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

Temos também o `insert()`, que insere um elemento na posição especificada e move os demais elementos para direita:

```
lista.insert(3, 'd') # Insere 'd' na posição 3
lista
```

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

> **Aviso:** Cuidado com `lista.insert(-1, algo)`! Nesse caso, inserimos algo na posição -1 e o elemento que estava previamente na posição `-1` é movido para a direita:

```
lista.insert(-1, 'ç')
lista
```

```
['a', 'b', 'c', 'd', 'ç', 'e']
```

> Use `append()` caso queira algo adicionado ao final da lista.

`extend()` recebe uma lista como argumento e adiciona todos seus elementos a outra:

```
lista1 = ['a', 'b', 'c']
lista2 = ['d', 'e']
```

```
lista1.extend(lista2)
lista1
```

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

```
lista2 # `lista2` não é modificado
```

```
['d', 'e']
```

O método `sort()` ordena os elementos da lista em ordem ascendente:

```
lista_desordenada = ['b', 'z', 'k', 'a', 'h']
lista_desordenada
```

```
['b', 'z', 'k', 'a', 'h']
```

```
lista_desordenada.sort()
lista_desordenada # Agora está ordenada!
```

```
['a', 'b', 'h', 'k', 'z']
```

```
lista2_desordenada = [5, 6.4, 1.2, 34, 2.1]
lista2_desordenada
```

```
[5, 6.4, 1.2, 34, 2.1]
```

```
lista2_desordenada.sort()
lista2_desordenada # Agora está ordenada!
```

```
[1.2, 2.1, 5, 6.4, 34]
```

Para fazer uma cópia de uma lista, devemos usar o método `copy()`:

```
lista1 = ['a', 'b', 'c']
lista2 = lista1.copy()
```

```
print(lista1)
```

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

```
print(lista2)
```

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

```
lista2.append('d')
lista2
```

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

```
lista1
```

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

Se não usarmos o `copy()`, acontece algo bem estranho:

```
lista1 = ['a', 'b', 'c']
lista2 = lista1
```

```
print(lista1)
```

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

```
print(lista2)
```

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

```
lista2.append('d')
lista2
```

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

```
lista1
```

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

Para alterar um valor em específico de uma lista basta sabermos em que índice tal valor está.

```
# Criando uma lista que contém as cinco primeiras letras do alfabelto
alfabeto = ['A', 'B', 'C', 'Z', 'E']

# Visualizando antes da mudança
print(f"Lista Atual: {alfabeto}")
```

```
Lista Atual: ['A', 'B', 'C', 'Z', 'E']
```

```
# Verificando em qual posição está o elemneto que deve ser alterado
id = alfabeto.index('Z')

# Fazendo a alteração na lista original
```

```
alfabeto[id] = 'D'

# Visualizando lista alterada
print(f"Lista Alterada: {alfabeto}")
```

```
Lista Alterada: ['A', 'B', 'C', 'D', 'E']
```

2.3.12 Função `range()`

Assim com as funções `print()` e `len()`, a função `range()` é do Python básico.

Aprendemos a adicionar itens a uma lista mas, e se fosse necessário produzir uma lista com os números de 1 até 200?

`[Input]: lista_grande = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, ..., 200]`

Em Python existe a função embutida `range()`, com ela é possível produzir uma lista extensa de uma maneira bem simples:

```
list(range(1, 200))
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
```

```
list(range(1, 200))
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
```

Além disso, o `range()` também oferece algumas coisas interessantes. Por exemplo, imprimir os números espaçados de 5 em 5, entre 0 e 30:

```
list(range(0, 30, 5))
```

```
[0, 5, 10, 15, 20, 25]
```

Repare que os argumentos passados são da forma: `range(start, stop, step)`.

Onde **start:** É o início. **stop:** É o fim $(n-1)$. **step:** É o espaço entre o valores contidos entre $[start; stop)$

Mas por que precisamos transformar o `range()` em `list`? O que acontece se não fizermos isso?

```
print(range(200))
```

```
range(0, 200)
```

```
print(type(range(200)))
```

```
<class 'range'>
```

A função `range()` retorna algo do tipo range, por isso precisamos transformar em uma lista para vermos (imprimir) todos os números no `print()`!

```
range_lista = list(range(200))
range_lista
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 2
```

2.3.13 Dicionários

Dicionário é uma coleção de itens (chamados chaves) e seus respectivos significados (chamados de valores): `{chave: valor}`.

Cada chave do dicionário deve ser única! Ao contrário de listas, dicionários, não podem ter chaves repetidas.

> **Nota:** As chaves devem ser únicas.

2.3.13.1 Declaração

Declaramos um dicionário colocando entre chaves `{}` cada chave e o seu respectivo valor, da seguinte forma:

```
telefones = {"ana": 123456, "yudi": 40028922, "julia": 4124492}
telefones
```

```
{'ana': 123456, 'yudi': 40028922, 'julia': 4124492}
```

No caso acima, a chave `"ana"`, por exemplo, está relacionada ao valor `123456`. Cada par chave-valor é separado por uma vírgula.

2.3.13.2 Função `dict()`

A função `dict()` constrói um dicionário. Existem algumas formas de usá-la:

- Com uma lista de listas:

```
# Definindo três listas diferentes
lista1 = ["brigadeiro", "leite condesado, achocolatado"]
lista2 = ["omelete", "ovos, azeite, condimentos a gosto"]
lista3 = ["ovo frito", "ovo, óleo, condimentos a gosto"]

# Criando uma lista de listas
lista_receitas = [lista1, lista2, lista3]

# Visualizando resultado
lista_receitas
```

```
[['brigadeiro', 'leite condesado, achocolatado'], ['omelete', 'ovos, azeite, condimentos
```

```
# Transformando lista de listas em um dicionário
receitas = dict(lista_receitas)
```

```
# Visualizando o resultado
receitas
```

```
{'brigadeiro': 'leite condesado, achocolatado', 'omelete': 'ovos, azeite, condimentos a
```

- Atribuindo os valores diretamente:

```
constantes = dict(pi=3.14, e=2.7, alpha=1/137)
constantes
```

```
{'pi': 3.14, 'e': 2.7, 'alpha': 0.0072992700729927005}
```

2.3.13.3 Chaves

Acessamos um determinado valor do dicionário através de sua chave:

```
# Definindo um dicionário para capitais de estados brasileiros
capitais = {"SP": "São Paulo", "AC": "Rio Branco", "TO": "Palmas",
            "RJ": "Rio de Janeiro", "SE": "Aracaju", "MG": "Belo Horizonte"}

# Acessando o valor correspondente a chave "MG"
capitais["MG"]
```

```
'Belo Horizonte'
```

Até o momento, usamos apenas strings, mas podemos colocar todo tipo de coisa dentro dos dicionários, incluindo listas e até mesmo outros dicionários:

```
# Dicionários para tipos de número
numeros = {"primos": [2, 3, 5], "pares": [0, 2, 4], "ímpares": [1, 3, 5]}

# Acessando o valor correspondente a chave "ímpares"
numeros["ímpares"]
```

```
[1, 3, 5]
```

Mesmo que os pares chave-valor estejam organizados na ordem que foram colocados, não podemos acessá-los por índices como faríamos em listas:

```
[Input]  : numeros[2]
[Output] : KeyError: 2
```

O mesmo erro ocorre se tentarmos colocar uma chave que não pertence ao dicionário:

```
[Input]  : numeros["negativos"]
[Output] : KeyError: 'negativos'
```

Assim como os valores não precisam ser do tipo string, o mesmo vale para as chaves:

```
numeros_por_extenso = {2: "dois", 1: "um", 3: "três", 0: "zero"}
numeros_por_extenso[0]
```

```
'zero'
```

```
numeros_por_extenso[2]
```

```
'dois'
```

> **Nota:** Listas e outros dicionários não podem ser usados como chaves por serem de tipos mutáveis.

2.3.13.4 Adicionando e removendo elementos

Podemos alterar o valor relacionado a uma chave da seguinte forma:

```
# Informações de Cleiton
pessoa = {"nome": "Cleiton", "idade": 34, "família": {"mãe": "Maria", "pai": "Enzo"}}

# Visualizando
pessoa
```

```
{'nome': 'Cleiton', 'idade': 34, 'família': {'mãe': 'Maria', 'pai': 'Enzo'}}
```

```
pessoa["idade"] # Acessando a informação "idade" de Cleiton
```

```
34
```

```
# Alterando a informação da "idade" de Cleiton
pessoa["idade"] = 35

# Acessando a informação nova da "idade" de Cleiton
pessoa["idade"]
```

```
35
```

Para adicionar um elemento novo à um dicionário, podemos simplesmente fazer o seguinte:

```
# Dicionário de meses do ano
meses = {1: "Janeiro", 2: "Fevereiro", 3: "Março"}

# Adicionando o mês de "Abril" na chave 4
meses[4] = "Abril"

# Visualizando
meses
```

```
{1: 'Janeiro', 2: 'Fevereiro', 3: 'Março', 4: 'Abril'}
```

Aqui nos referimos a uma chave que não está no dicionário e associamos um valor a ela. Desta forma, adicionando esse conjunto chave-valor ao dicionário. Removemos um conjunto chave-elemento de um dicionário com o comando `del()`:

```
# Excluíndo o mês de "Abril"
del(meses[4])

# Visualizando
meses
```

```
{1: 'Janeiro', 2: 'Fevereiro', 3: 'Março'}
```

Para alterar um valor em específico de uma lista que está dentro de um dicionário, podemos usar o seguinte código.

```
# Novo dicionário
lixo = {"plástico": ["garrafa", "copinho", "canudo"], "papel": ["folha amassada", "guard

# Visualizando
lixo
```

```
{'plástico': ['garrafa', 'copinho', 'canudo'], 'papel': ['folha amassada', 'guardanapo']
```

```
# Obtendo lista de interesse para alteração
lista_de_interesse = lixo['plástico']

# Verificando em qual posição está o elemneto que deve ser alterado
id = lista_de_interesse.index('garrafa')

# Fazendo a alteração diretamente no dicionário
lixo['plástico'][id] = 'sacola'

# Visualizando dicionário alterado
lixo
```

```
{'plástico': ['sacola', 'copinho', 'canudo'], 'papel': ['folha amassada', 'guardanapo'],
```

Para apagar todos os elementos de um dicionário, usamos o método `clear`:

```
# Apagando todos os elementos do dicionário
lixo.clear()

# Visualizando
lixo
```

```
{}
```

2.3.13.5 Função `list()`

A função `list()` recebe um conjunto de objetos e retorna uma lista. Ao passar um dicionário, ela retorna uma lista contendo todas as suas chaves:

```
institutos_uspsc = {"IFSC": "Instituto de Física de São Carlos", "ICMC": "Instituto de C
                    "EESC": "Escola de Engenharia de São Carlos", "IAU": "Instituto de A
institutos_uspsc
```

```
{'IFSC': 'Instituto de Física de São Carlos', 'ICMC': 'Instituto de Ciências Matemáticas
```

```
list(institutos_uspsc)
```

```
['IFSC', 'ICMC', 'EESC', 'IAU', 'IQSC']
```

2.3.13.6 Função `len()`

A função `len()` retorna o número de elementos (tamanho) do objeto passado para ela. No caso de uma lista, fala quantos elementos há. No caso de dicionários, retorna o número de chaves contidas nele:

```
institutos_uspsc
```

```
{'IFSC': 'Instituto de Física de São Carlos', 'ICMC': 'Instituto de Ciências Matemáticas
```

```
len(institutos_uspsc)
```

```
5
```

Você pode contar o número de elementos na lista gerada pela função `list()` para conferir:

```
len(list(institutos_uspsc))
```

```
5
```

2.3.13.7 Método get()

O método `get(chave, valor)` pode ser usado para retornar o valor associado à respectiva chave! O segundo parâmetro `<valor>` é opcional e indica o que será retornado caso a chave desejada não esteja no dicionário:

```
institutos_uspsc.get("IFSC")
```

```
'Instituto de Física de São Carlos'
```

Dá para ver que ele é muito parecido com fazer assim:

```
institutos_uspsc["IFSC"]
```

```
'Instituto de Física de São Carlos'
```

Mas ao colocarmos uma chave que não está no dicionário:

```
institutos_uspsc.get("Poli")
```

```
institutos_uspsc.get("Poli", "Não tem!")
```

```
'Não tem!'

[Input]  : institutos_uspsc["Poli"]
[Output] : KeyError: 'Poli'
```

2.3.13.8 Alguns, outros, métodos

Vamos criar um dicionário para exemplo.

```
pessoa = {"nome": "Enzo", "RA": 242334, "curso": "fiscomp"}
pessoa.items()
```

```
dict_items([('nome', 'Enzo'), ('RA', 242334), ('curso', 'fiscomp')])
```

Usando a função `list()` nesse resultado, obtemos:

```
itens = list(pessoa.items())
itens
```

```
[('nome', 'Enzo'), ('RA', 242334), ('curso', 'fiscomp')]
```

Experimente usar a função `dict()` na lista `itens`!

```
dict(itens)
```

```
{'nome': 'Enzo', 'RA': 242334, 'curso': 'fiscomp'}
```

O método `values()` nos retorna os valores do dicionário:

```
pessoa.values()
```

```
dict_values(['Enzo', 242334, 'fiscomp'])
```

```
valores = list(pessoa.values())
valores
```

```
['Enzo', 242334, 'fiscomp']
```

O método `keys()` nos retorna as chaves do dicionário:

```
pessoa.keys()
```

```
dict_keys(['nome', 'RA', 'curso'])
```

Repare que nesse último obtemos o mesmo que se tivéssemos usado a função `list()` diretamente no objeto dicionário:

```
list(pessoa)
```

```
['nome', 'RA', 'curso']
```

2.3.13.9 Ordem dos elementos

Dicionários não tem sequência dos seus elementos. As listas têm. Dicionários mapeiam um valor a uma chave. Veja este exemplo:

```
numerinhos = {"um": 1, "dois": 2, "três": 3}
numeritos = {"três": 3, "dois": 2, "um": 1}
numerinhos == numeritos
```

```
True
```

```
numeritos
```

```
{'três': 3, 'dois': 2, 'um': 1}
```

```
numerinhos
```

```
{'um': 1, 'dois': 2, 'três': 3}
```

Vemos que `numerinhos` e `numeritos` têm as mesmas chaves com os mesmos valores e por isso são iguais. Mas quando imprimimos cada um, a ordem que aparece é a que os itens foram inseridos.

2.3.13.10 Está no dicionário?

Podemos checar se uma chave está ou não em um dicionário utilizando o comando `in`. Voltando para o dicionário que contem os institutos da USP São Carlos:

```
institutos_uspsc
```

```
{'IFSC': 'Instituto de Física de São Carlos', 'ICMC': 'Instituto de Ciências Matemáticas
```

```
"IFSC" in institutos_uspsc
```

```
True
```

```
"ESALQ" in institutos_uspsc
```

```
False
```

E checamos se uma chave ***não está*** no dicionário com o comando `not in`:

```
"IFSC" not in institutos_uspsc
```

```
False
```

```
"ESALQ" not in institutos_uspsc
```

```
True
```

2.4 Controle de Fluxo

2.4.1 Estruturas de Decisão

As estruturas de controle servem para decidir quais blocos de código serão executados.

Exemplo:

```
Se estiver nublado:
    Levarei guarda-chuva
Senão:
    Não levarei
```

> **Nota:** Na linguagem Python, a indentação (espaço dado antes de uma linha) é utilizada para demarcar os blocos de código, e são obrigatórios quando se usa estruturas de controle.

```
# Definindo uma variável qualquer
a = 7

if a > 3: # Se a for maior que 3
  print("estou no if")
else: # Senão
  print("cai no else")
```

```
estou no if
```

Também é possível checar mais de uma condição com o `elif`. É a abreviatura para `else-if`. Ou seja, se o `if` for falso, testa outra condição antes do `else`:

```
valor_entrada = 10

if valor_entrada == 1:
  print("a entrada era 1")
elif valor_entrada == 2:
  print("a entrada era 2")
elif valor_entrada == 3:
  print("a entrada era 3")
elif valor_entrada == 4:
  print("a entrada era 4")
else:
  print("o valor de entrada não era esperado em nenhum if")
```

```
o valor de entrada não era esperado em nenhum if
```

Note que quando uma condição for verdadeira, aquele bloco de código é executado e as demais condições (`elif` e `else`) são puladas:

```
a = 1
```

```
if a == 1:
  print("é 1")
elif a >= 1:
  print("é maior ou igual a 1")
else:
  print("é qualquer outra coisa")
```

```
é 1
```

Desta forma, se não optarmos usar o `elif`, mas sim `if` seguido de `if`, veja o que pode acontecer:

```
a = 1

if a == 1:
  print("Caiu no 1º `if`")
```

```
Caiu no 1º `if`
```

```
if a >= 1:
  print("Caiu no 2º `if`")

else:
  print("Caiu no `else`")
```

```
Caiu no 2º `if`
```

2.4.2 Estruturas de Repetição

As estruturas de repetição são utilizadas quando queremos que um bloco de código seja executado várias vezes.

Em Python existem duas formas de criar uma estrutura de repetição: * O `for` é usado quando se quer iterar sobre um bloco de código um determinado número de vezes. * O `while` é usando quando queremos que o bloco de código seja repetido até que uma condição seja satisfeita.

Ou seja, é necessário que uma expressão boleana dada seja verdadeira. Assim que ela se tornar falsa, o `while` para.

> **Nota:** Na linguagem Python a indentação é obrigatória. assim como nas estruturas de decisão, as estruturas de repetição também precisam.

```
# Interação usando `for`
for n in range(0, 3): # `para n em [0;3) faça`
  print(n)
```

```
0
1
```

2

```
# Iniciando em n em zero
n = 0

while n < 3: # `enquanto n menor que três faça`
  print(n)
  n += 1
```

```
0
1
2
```

O loop `for` em Python itera sobre os itens de um conjunto, sendo assim, o `range(0, 3)` precisa ser um conjunto de elementos. E na verdade ele é:

```
list(range(0, 3))
```

```
[0, 1, 2]
```

Para iterar sobre uma lista usando `for`:

```
lista = [1, 2, 3, 4, 10]
for numero in lista:
  print(numero ** 2)
```

```
1
4
9
16
100
```

Em dicionários podemos fazer assim:

```
# Define um dicionário chamado 'gatinhos' que armazena a tradução da palavra "gato" em d
# As chaves do dicionário são os idiomas e os valores são as traduções correspondentes.
gatinhos = {"Português": "gato", "Inglês": "cat", "Francês": "chat", "Finlandês": "Kissa

# Itera pelos pares chave-valor do dicionário 'gatinhos' usando o método .items().
# Para cada par chave-valor, a chave é atribuída à variável 'chave' e o valor à variável
for chave, valor in gatinhos.items():
  # Imprime a chave (idioma) e o valor (tradução) separados por "->".
  print(chave, "->", valor)
```

```
Português -> gato
Inglês -> cat
Francês -> chat
Finlandês -> Kissa
```

Para auxiliar as estruturas de repetição, existem dois comandos:

- `break`: É usado para sair de um loop, não importando o estado em que se encontra.
- `continue`: Funciona de maneira parecida com a do break, porém no lugar de encerrar o loop, ele faz com que todo o código que esteja abaixo (porém ainda dentro do loop) seja ignorado e avança para a próxima iteração.

Veja a seguir um exemplo de um código que ilustra o uso desses comandos. Note que há uma string de documentação no começo que explica a funcionalidade. O primeiro bloco de código, mostrando a seguir, é o bloco de entrada (que deve ser compilado).

```
"""
Esse código deve rodar até que a palavra "sair" seja digitada.
- Caso uma palavra com 2 ou menos caracteres seja digitada, um aviso
deve ser exibido e o loop será executado do início (devido ao continue),
pedindo uma nova palavra ao usuário.
- Caso qualquer outra palavra diferente de "sair" seja digitada, um aviso
deve ser exibido.
- Por fim, caso a palavra seja "sair", uma mensagem deve ser exibida e o
loop deve ser encerrado (break).
"""

# Este é um loop infinito que continuará até que o usuário digite "sair".
while True:
    # Solicita ao usuário que digite uma palavra.
    string_digitada = input("Digite uma palavra: ")
    # Verifica se a string digitada é igual a "sair", ignorando maiúsculas e minúsculas.
    if string_digitada.lower() == "sair":
        # Se for "sair", imprime "Fim!" e encerra o loop.
        print("Fim!")
        break

    # Verifica se o comprimento da string digitada é menor ou igual a 2.
    if len(string_digitada) <= 2:
        # Se for muito pequena, imprime "String muito pequena" e continua à próxima iter
        print("String muito pequena")
        continue

    # Verifica se a string digitada é diferente de "sair", ignorando maiúsculas e minúsc
    if string_digitada.lower() != "sair":
        # Se for diferente de "sair", imprime "Mais uma vez:" e continua para a próxima
        print("Mais uma vez:")

    # Imprime "Tente digitar \"sair\"" se a string digitada não for "sair".
    print("Tente digitar \"sair\"")
```

Agora, veja uma das possíveis saídas para o código acima.

```
Digite uma palavra: Aí
```

```
String muito pequena
Digite uma palavra: Saída
Mais uma vez:
Tente digitar "sair"
Digite uma palavra: Será que eu consigo digitar "sair"?
Mais uma vez:
Tente digitar "sair"
Digite uma palavra: Sa
String muito pequena
Digite uma palavra: Agora foi quase!
Mais uma vez:
Tente digitar "sair"
Digite uma palavra: Sair
Fim!
```

```
# Loop externo: itera sobre os números de 2 a 8 (n).
for n in range(2, 9):
  # Loop interno: itera sobre os números de 2 a n-1 (x).
  for x in range(2, n):
    # Verifica se n é divisível por x (resto da divisão igual a 0).
    if n % x == 0:
      # Se for divisível, imprime a mensagem indicando que n não é primo e o resultado d
      print(n, 'é igual a ', x, '*', n//x)
      # Sai do loop interno (break).
      break
    else:
      # Se não for divisível por x, imprime a mensagem indicando que n é primo.
      print(n, 'é um número primo')
      # Sai do loop interno (break).
      break
```

```
3 é um número primo
4 é igual a  2 * 2
5 é um número primo
6 é igual a  2 * 3
7 é um número primo
8 é igual a  2 * 4
```

```
# Itera pelos números no intervalo de 2 a 9 (inclusive).
for num in range(2, 10):
  # Verifica se o número atual (num) é par, ou seja, se o resto da divisão por 2 é 0.
  if num % 2 == 0:
    # Se o número for par, imprime a mensagem "Número par:" seguido do número.
    print("Número par:", num)
    # Pula para a próxima iteração do loop, ignorando o restante do código dentro do loo
    continue
```

```
  # Se o número não for par (ou seja, for ímpar), imprime a mensagem "Número ímpar:" seg
  print("Número ímpar:", num)
```

```
Número par: 2
Número ímpar: 3
Número par: 4
Número ímpar: 5
Número par: 6
Número ímpar: 7
Número par: 8
Número ímpar: 9
```

2.4.2.1 List Comprehension

List Comprehension em Python: Criando listas de forma concisa e elegante List Comprehension (compreensão de listas) é uma forma concisa e poderosa de criar listas em Python. Ela permite que você construa novas listas a partir de listas existentes, aplicando expressões e condições de forma direta e intuitiva.

```
[Input]: nova_lista = [expressão for item in iteravel if condicao]
```

Exemplos:

- Dada a lista: `numeros = [1, 2, 3, 4, 5]`, crie uma nova lista que contém o quadrado desses números.

```
# Lista inicial
numeros = [1, 2, 3, 4, 5]

# Lista que contém o quadrado dos num da lista inicial
quadrados = [x**2 for x in numeros]
quadrados
```

```
[1, 4, 9, 16, 25]
```

- Dada uma lista de formada pelos números de 0 a 9. Capture os números pares em uma nova lista

```
# Números de 0 a 9
numeros = range(10)

# Lista contendo somente num pares
pares = [x for x in numeros if x % 2 == 0]
pares
```

```
[0, 2, 4, 6, 8]
```

- Pensando em uma resolução binária.

```
binario = [1 if x % 2 == 0 else 0 for x in numeros]
binario
```

```
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
```

- Dada uma lista de palavras, crie uma nova lista que contenha o tamanho das strings.

```
# Lista de palavras
palavras = ["Python", "é", "legal", "e", "poderoso"]

# Lista com o tamanho de cada palavra
tamanhos = [len(palavra) for palavra in palavras]
tamanhos
```

```
[6, 1, 5, 1, 8]
```

- Dada uma lista de com os graus de celsius, crie uma lista com a conversão da medida de celsius para fahrenheit.

```
celsius = [0, 10, 20, 30]
fahrenheit = [(temp * 9/5) + 32 for temp in celsius]
fahrenheit
```

```
[32.0, 50.0, 68.0, 86.0]
```

- Dada uma lista de números positivos e negativos, filtre para uma nova lista somente os números positivos.

```
# Lista de positivos e negativos
numeros = [-5, -2, 0, 3, 7, -3.5, 4.89, 1.01]

# Lista de positivos
positivos = [x for x in numeros if x > 0]
positivos
```

```
[3, 7, 4.89, 1.01]
```

2.5 Funções

Uma função é uma sequência de instruções que executa uma operação específica de computação. Ao definir uma função, você especifica um nome e a sequência de instruções que serão executadas quando a função for chamada pelo nome.

A ideia é semelhante às funções matemáticas, mas, em linguagens de programação, as funções não se restringem a realizar apenas cálculos. Elas podem realizar uma ampla variedade de tarefas, como manipulação de dados, operações de entrada e saída, entre outras.

Em Python, funções são blocos de código reutilizáveis que ajudam a organizar e modularizar um programa. Elas recebem dados de entrada, chamados de argumentos, aplicam uma

sequência de operações sobre esses dados e, opcionalmente, retornam um resultado, conhecido como valor de retorno. Essa relação é análoga à definição de função na matemática, onde cada elemento de um conjunto (domínio) é associado a um único elemento de outro conjunto (imagem).

Vimos o `type()`, um exemplo de função:

```
[Input]  : type(23)
[Output] : int
```

```
[Input]  : type('textinho')
[Output] : str
```

Defini-se função de forma que:

```
[Input]  : def NOME_DA_FUNÇÃO(parâmetro_1, parâmetro_2, ..., parâmetro_n):
               <1º comando>
               <2º comando>
               ...
               <n-ésimo comando>

               print(var_1, var_2, ..., var_n) ou retun var_1, var_2, ..., var_n
```

> **Nota:** Assim como nas estruturas de decisão e loops, as funções em Python é necessário utilizar os dois pontos (:) seguidos de uma indentação para indicar que um bloco de código pertence a essas estruturas.

Veja exemplos:

```
def soma():
  print(1 + 1)
```

```
soma()
```

2

```
def soma():
  return 1 + 1
```

```
soma()
```

2

Qual a diferença entre utilizar `print()` e `return` aqui em cima?!?

```
# Definindo a função `soma`
def soma():
  print(1 + 1)

# Executando a função
soma()
```

2

```
# Atribuindo o resultado de `soma` a uma variável `a`
a = soma()
```

2

```
# Chamando o valor de `a`
a # Note que a = 2, porém
```

```
# Definindo a função `soma`
def soma():
  return 1 + 1

# Executando a função
soma()
```

2

```
# Atribuindo o resultado de `soma` a uma variável `b`
b = soma()

# Chamando o valor de `b`
b
```

2

Tal diferença surge porque a função `print()` é usado somente para imprimir as informações/resultados na tela. Com isso, não é possível atribuir uma impressão à uma variável.

Por isso, quando desejarmos guardar (atribuir) os valores resultantes de uma função devemos usar `return`.

2.5.1 Funções com Argumentos

Queremos multiplicar um número qualquer, x, por 2 e somar com 3, assim, a função em Python pode ser escrita dessa forma:

```
def linear(x):
  return 2 * x + 3
```

```
linear(1)
```

5

```
linear(2)
```

7

Como posso calcular a tabuada de um número onde tal número é o dado de entrada? Veja:

```
def tabuada_num(num):
  for n in range(1, 11):
    print(f'{num} x {n} = {num * n}')
```

```
tabuada_num(7)
```

```
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
```

```
tabuada_num(13)
```

```
13 x 1 = 13
13 x 2 = 26
13 x 3 = 39
13 x 4 = 52
13 x 5 = 65
13 x 6 = 78
13 x 7 = 91
13 x 8 = 104
13 x 9 = 117
13 x 10 = 130
```

```
tabuada_num(1.5)
```

```
1.5 x 1 = 1.5
1.5 x 2 = 3.0
1.5 x 3 = 4.5
1.5 x 4 = 6.0
1.5 x 5 = 7.5
1.5 x 6 = 9.0
1.5 x 7 = 10.5
1.5 x 8 = 12.0
1.5 x 9 = 13.5
1.5 x 10 = 15.0
```

Capítulo 3

Exercicíos

1. Aulas Faltadas:

Davinir não gosta de ir às aulas, mas ele precisa comparecer a pelo menos 75% delas. Sabendo que há duas aulas por semana durante quatro meses, ajude Davinir a calcular:

- a) Quantas aulas ele pode faltar.
- b) Quantas aulas ele deve assistir para não ser reprovado.

Nota: Um mês tem quatro semanas.

2. Área de um Círculo:

Calcule a área de um círculo de raio `R = 2`. Crie uma função chamada `calcula_area` que receba o valor do raio e retorne a área.

- a) Teste a função com diferentes valores de raio, como `3.5` e `7`.

Lembrete: a área de um círculo é dada por:

$$A = \pi R^2$$

3. Conversão de Tempo:

Escreva uma função chamada `converte_tempo` que converta uma quantidade de tempo dada em horas, minutos e segundos para apenas segundos. Teste a função com os seguintes valores:

- a) 3 horas, 23 minutos e 17 segundos.
- b) 2 horas, 45 minutos e 50 segundos.
- c) 0 horas, 30 minutos e 15 segundos.

4. Expressão Matemática:

Resolva as expressões abaixo usando o Python:

- a)

$$\frac{100 - 413 \cdot (20 - 5 \times 4)}{5}$$

- b)

$$\frac{\left[(3^4 + \sqrt{144})(100 - 95,5)\right] + 6}{-80 + 2^4}$$

- c)

$$3,9 \cdot 10^{-2} + 5,2 \cdot 10^{-3}$$

5. Média Ponderada:

Escreva um script para calcular a média ponderada de 4 notas. Considere pesos 0.1, 0.2, 0.3 e 0.4 para cada avaliação. Use variáveis e `print()` para exibir o resultado.

- a) Modifique o script para permitir a entrada das notas e dos pesos pelo usuário.
- b) Garanta que os pesos somem 1.0; caso contrário, exiba uma mensagem de erro.

6. Divisão de Contas:

Você e seus amigos foram ao supermercado e compraram:

- 75 latas de cerveja: R$ 2,20 cada (da ruim ainda, pra fazer o dinheiro render);
- 2 pacotes de macarrão: R$ 8,73 cada;
- 1 pacote de molho de tomate: R$ 3,45;
- 420g de cebola: R$ 5,40/kg;
- 250g de alho: R$ 30/kg;
- 450g de pães franceses: R$ 25/kg;

Calcule: - a) O valor total da compra. - b) Quanto cada um deve pagar, considerando que são 4 pessoas. - c) O valor da compra se houvesse um desconto de 5% nas latas de cerveja.

7. Investimento:

Suponha que você tenha R$ 100,00 para investir, com um retorno de 10% ao ano. Após 7 anos, quanto dinheiro você terá?

- a) Implemente uma função que calcule esse valor para qualquer número de anos e taxa de retorno.
- b) Simule o investimento para retornos de 5%, 10% e 15% ao ano.

8. Conversão de Moeda:

Com a cotação do dólar a R$ 3,25, quanto você teria ao cambiar R$ 65,00? Escreva um script que permita ao usuário inserir o valor em reais e a cotação para calcular o valor em dólares.

9. Média Aritmética, Geométrica e Harmônica:

Abelindo precisa decidir como calcular a média final de Rondinelly, que obteve as seguintes notas: `8.66`, `5.35`, `5` e `1`.

- a) Calcule a média aritmética (M.A.), geométrica (M.G.) e harmônica (M.H.) dessas notas.
- b) Qual dessas médias dá a maior nota para Rondinelly?

Média Aritimética:

$$MA = \frac{\sum_{i=1}^{n} x_i}{n}$$

Média Geométrica:

$$MG = \sqrt[n]{\Pi_{i=1}^{n} x_i}$$

Média Harmônica:

$$MH = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$$

10. Compras Internacionais:

Josefson deseja comprar na China um celular de USD 299,99, uma chaleira de USD 23,87, um gnomo de jardim de USD 66,66 e 6 adesivos de unicórnio de USD 1,42 cada. O frete para Rolândia, no Paraná, é de USD 12,34.

- a) Calcule o valor total da compra em dólares.
- b) Usando o valor do dólar do exercício anterior, calcule o preço final em reais.
- c) Calcule quanto Josefson pagou apenas de IOF (6,38%).

11. Manipulação de Strings:

Dada a frase "**Python é muito legal**", use fatiamento para:

- a) Criar uma variável contendo cada palavra.
- b) Calcular o tamanho da frase e de cada palavra.
- c) Use slicing para inverter a string "Python".

12. Listas e Listas Aninhadas:

-

3.1 Crie três listas:

a) `Frutas`

- b) Docinhos de festa (inclua brigadeiros)
- c) Ingredientes de feijoada

•

3.2 Crie uma lista de listas chamada `listona` e execute os seguintes passos:

d) `Acesse o elemento "brigadeiro".`

- e) Adicione mais brigadeiros à lista de docinhos de festa. O que aconteceu com a lista original?
- f) Adicione bebidas ao final da `listona`, sem criar uma nova lista.

13. Manipulação de Listas:

Usando a `listona` do exercício anterior:

- a) Remova todos os elementos usando `del` até que a lista fique vazia.
- b) Verifique se a lista está vazia usando uma estrutura de controle.

14. Dicionários:

- a) Crie um dicionário com as 5 pessoas mais próximas de você, usando o nome como chave e a cor da camisa como valor.
- b) Crie um dicionário `semana = {}` e complete-o com os dias da semana como chave e uma lista de aulas como valor.
- c) Crie um dicionário `filmes = {}` com 5 filmes como chave e, como valor, outro dicionário contendo vilão e ano de lançamento.

15. Doação de Sangue:

Crie um programa que verifique se uma pessoa pode doar sangue com base nos critérios:

- a) Ter entre 16 e 69 anos.
- b) Pesar mais de 50 kg.
- c) Ter dormido pelo menos 6 horas nas últimas 24 horas.

16. Equação do Segundo Grau:

Crie uma função que receba os coeficientes `a`, `b` e `c` de uma equação do segundo grau e determine se a equação possui duas raízes reais, uma, ou nenhuma.

- a) Calcule e imprima as raízes, se existirem.

17. Média com Conceito:

Melhore o código de cálculo da média ponderada do Exercício 5 de um(a) aluno(a), incluindo um conceito final com base na média:

- a) 9.00 - 10.00: Excelente
- b) 7.00 - 8.99: Bom
- c) 5.00 - 6.99: Regular
- d) 0.00 - 4.99: Insuficiente

18. Estatísticas de Grupo:

Leia do teclado a idade e o sexo de 10 pessoas e calcule:

- a) Idade média das mulheres.
- b) Idade média dos homens.
- c) Idade média do grupo.

19. Somatório:

Calcule e imprima o somatório dos números de 1 a 100.

20. Sequência de Fibonacci:

Escreva um código que gere a sequência de Fibonacci n termos definido pelo usuário.

Lembrando que:

$$F_n = \begin{cases} 0 \ , \text{ se } n = 1 \\ 1 \ , \text{ se } n = 2 \\ F_{n-2} + F_{n-1} \ , \text{ para os demais casos} \end{cases}$$

21. Fatorial:

Desenvolva uma função que retorne o valor do fatorial de um número inteiro fornecido pelo usuário.

22. Listas:

Crie uma lista contendo o quadrado de todos os números ímpares entre 1 e 20.

21. Lista de Tuplas:

Crie uma lista de tuplas onde cada tupla contenha o número e seu cubo, para números de 1 a 10. Exemplo: `[(1, 1), (2, 8), (3, 27), ...]`

EXTRA. Sistema de Controle de Estoque e Vendas de uma Loja

Uma loja de conveniência deseja criar um sistema simples para gerenciar o estoque e calcular o valor total das vendas diárias. O sistema deve ser capaz de:

1. **Cadastrar Produtos**: Permitir a inserção de novos produtos no estoque. Cada produto deve ter as seguintes informações:
    - Nome do produto (string)
    - Preço unitário (float)
    - Quantidade em estoque (inteiro)
2. **Atualizar Estoque**: Aumentar ou diminuir a quantidade de um produto específico.
3. **Realizar Venda**:
    - Perguntar ao usuário quais produtos ele deseja comprar e a quantidade de cada um.
    - Verificar se a quantidade em estoque é suficiente para a venda.
    - Caso seja suficiente, atualizar o estoque e calcular o valor total da venda.
    - Caso contrário, exibir uma mensagem informando que a quantidade em estoque é insuficiente.
4. **Relatório de Vendas**: Ao final do dia, o sistema deve gerar um relatório contendo:
    - Produtos vendidos e quantidade vendida de cada um.
    - Valor total arrecadado.

Desafio Extra:

- Implemente uma função que calcule um desconto progressivo para as vendas:
    - 5% para compras acima de R$ 100,00.
    - 10% para compras acima de R$ 200,00.
    - 15% para compras acima de R$ 500,00.
- Adicione a opção de reabastecimento automático: se a quantidade de um produto no estoque estiver abaixo de um determinado valor, reabasteça automaticamente para a quantidade inicial.

Dicas:

- Use um dicionário para armazenar os produtos e suas informações.
- Crie funções para cada uma das funcionalidades do sistema (cadastrar produtos, atualizar estoque, realizar venda, gerar relatório).
- Utilize loops e condições para controlar o fluxo do programa.

EXTRA. Sistema de Cadastro de Alunos e Notas

Crie um programa para gerenciar o cadastro de alunos e suas notas em uma escola. O programa deve:

1. **Cadastrar Alunos**:
   - O usuário deve ser capaz de cadastrar novos alunos, informando o nome e uma lista de notas (mínimo de 3 e máximo de 5 notas).
2. **Consultar Alunos**:
   - O usuário deve ser capaz de consultar um aluno específico e visualizar suas notas e média.
3. **Calcular Média e Conceito**:
   - A média deve ser calculada e um conceito deve ser atribuído ao aluno de acordo com a média:
     - A: Média 9.0
     - B: 7.0 Média $<$ 9.0
     - C: 5.0 Média $<$ 7.0
     - D: 3.0 Média $<$ 5.0
     - E: Média $<$ 3.0
4. **Alterar Notas**:
   - O usuário deve ser capaz de alterar as notas de um aluno específico.
5. **Gerar Relatório Geral**:
   - Exibir um relatório contendo todos os alunos, suas médias e seus conceitos.
6. **Desafios Extras**:
   - Calcular a média da turma e o número de alunos em cada conceito.
   - Encontrar o aluno com a maior média e o aluno com a menor média.

   Dicas:
   - Utilize dicionários para armazenar as informações dos alunos e suas notas.
   - Crie funções para cada uma das funcionalidades do sistema (cadastrar aluno, consultar aluno, calcular média, alterar notas, gerar relatório).
   - Use loops e estruturas de controle para gerenciar as operações.

- Explore a manipulação de strings para melhorar a visualização dos dados no relatório.