# PYTHON

## Do Básico ao Funcional: construa ferramentas e soluções com Python.


Emanuela da Silva Silveira
Marta Adriana Machado da Silva
Eliane Pozzebon

UFSC

LabTeC
Laboratório de Tecnologias Computacionais

# PYTHON

## Do Básico ao Funcional: construa ferramentas e soluções com Python.

Realização:

**LabTeC - UFSC** | Laboratório de Tecnologias Computacionais

Área do Conhecimento:

**Linguagem Python**

**1ª Edição**
Araranguá,
2025

**Emanuela da Silva Silveira**
**Marta Adriana Machado da Silva**
**Eliane Pozzebon**

# PYTHON

## Do Básico ao Funcional: construa ferramentas e soluções com Python.


**Idealização e Conteúdo**
Emanuela da Silva Silveira
Marta Adriana Machado da Silva
Eliane Pozzebon

**Projeto Gráfico e Experiência Visual**
Emanuela da Silva Silveira
Ananda Muxfeldt Palma

**Revisão Técnica**
Anderson Luiz Fernandes Perez

**Revisão Gramatical e Ortográfica**
Karla Goularte da Silva Gründler

**Apoio técnico**
Débora Maria Russiano Pereira



Silveira, Emanuela da Silva.
Python do básico ao funcional: construa ferramentas e soluções com Python / Emanuela da Silva Silveira, Marta Adriana Machado da Silva, Eliane Pozzebon. - Araranguá : LabTeC; UFSC, 2025.
137 p. : il.

ISBN: 978-65-01-85945-3

1. Python. 2. Linguagem de programação. 3. Programação básica. I. LabTeC. II UFSC. III. Silva, Marta Adriana Machado da. IV. Pozzebon, Eliane. V. Título

Ficha catalográfica elaborada por: Debora Maria Russiano Pereira, CRB-14/1125

# Prefácio

**Prezados professores e estudantes,**

Aprender a programar é muito mais do que escrever códigos: é desenvolver uma nova forma de pensar, criar e resolver problemas. Este e-book foi criado para convidar você a dar o primeiro passo nesse universo, mostrando que programar em Python pode ser simples, acessível e até divertido.

Ao longo das páginas, cada conceito é apresentado de forma clara e gradual, começando do básico e avançando aos poucos. Não é preciso ter experiência prévia — apenas curiosidade e vontade de aprender. A cada capítulo, você ganhará mais confiança para testar ideias, experimentar soluções e acompanhar sua própria evolução.

Sempre que possível, utilize o Google Colab, uma plataforma online e gratuita que permite executar os códigos sem complicações. Ver o código funcionando na prática torna o aprendizado mais envolvente. E, caso não seja possível praticar no momento da leitura, os exemplos ilustrados ajudam a manter o entendimento fluido.

Este livro não é apenas um material de estudo, mas um convite à descoberta e à prática. Leia no seu ritmo, explore os exemplos, experimente sem medo e aproveite cada conquista ao longo do caminho.

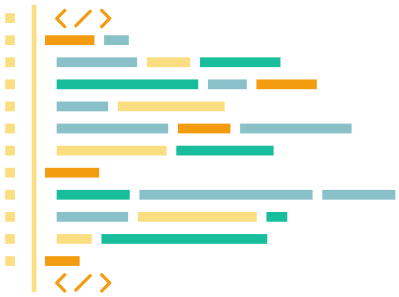
**Agora é sua vez: vire a página, explore o livro e dê início à sua jornada com Python.**

**As autoras.**

# Sumário





CAPÍTULO 11

## Funções



CAPÍTULO 12

## Bibliotecas




CAPÍTULO 13

## Programação Orientada a Objeto (POO)



CAPÍTULO 14

## Fazendo uma calculadora em Python

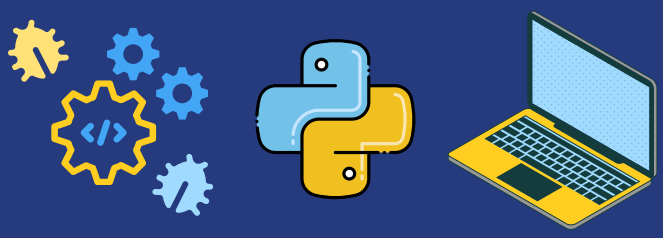


CAPÍTULO 15

## Desafios Finais


# Capítulo 1:

# Apresentando a linguagem



# Objetivo

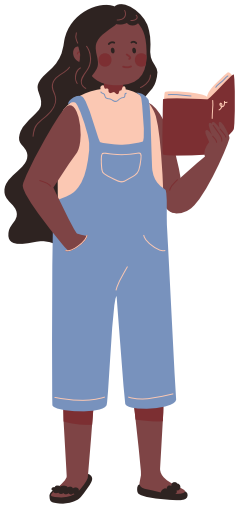
Antes de iniciarmos propriamente o conteúdo, vamos apresentar a linguagem Python a vocês. Vamos fazer uma pequena viagem no tempo para entender como ela surgiu, por que usar e o que é um algoritmo, preparando o terreno para os próximos capítulos.

# Você vai aprender

- Um resumo da história do Python e de quem o criou.
- Por que Python é tão usado hoje (simplicidade, código aberto, bibliotecas e onde é aplicado).
- O que é um algoritmo e como ele aparece no nosso dia a dia.

# 1 História da linguagem

1.1.



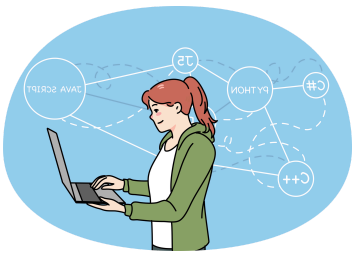
Vamos viajar até o ano de 1991, ano no qual **Guido Van Rossum** criou a linguagem conhecida como **Python**.

Desenvolvida para ser de **fácil aprendizado, intuitiva e de código aberto**, foi inspirada e influenciada pela linguagem ABC e, com o tempo, foi aprimorada e ampliada para que combinasse **simplicidade e eficiência**, tornando-a uma linguagem de alto desempenho e amplamente utilizada nas mais diversas áreas.


## Curiosidade

Embora associemos a palavra Python com a grande cobra Píton, seu nome foi uma referência divertida ao programa de TV que Van Rossum era fã, chamado Fly Circus de Monty Python.

Embora a ideia inicial do Python tenha surgido de Guido Van Rossum, seu crescimento é fruto do esforço coletivo de uma comunidade global. Milhares de programadores, testadores e entusiastas contribuem para a evolução da linguagem.



Atualmente, a linguagem é mantida pela **Python Software Foundation (PSF)**, uma organização sem fins lucrativos dedicada a impulsionar o desenvolvimento e a popularização do Python.

# 1

1.2.

# Por que usar Python

Devido a sua sintaxe simples é comum que a linguagem Python seja uma das primeiras opções ao se iniciar os estudos em linguagens de programação.

Por mais simples que sua codificação seja, grandes projetos podem ser feitos utilizando Python, alguns exemplos são grandes empresas como Google, Meta, Amazon e até a NASA que utilizam esta linguagem em diversas aplicações.

A facilidade do Python se manifesta tanto no aprendizado quanto na implementação, por ser uma linguagem de alto nível com sintaxe intuitiva, semelhante à linguagem falada, e compatível com praticamente todos os sistemas (Linux, Windows, etc.).

Além disso, Python é uma linguagem completa, contando com vastas bibliotecas que expandem suas capacidades para tarefas como acesso a banco de dados, construção de interfaces gráficas e processamento de arquivos, o que ajuda consideravelmente na produção de soluções.


# O que é um algoritmo

Um algoritmo é um conjunto de regras e/ou instruções escritas ao decorrer do código para criar ou solucionar algum sistema. Ele define e sequencia os passos para a realização de comandos em softwares e aplicativos, funcionam através de etapas bem definidas cooperando para um objetivo final.

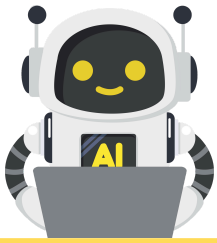
**Exemplos de algoritmos no dia a dia:** receitas culinárias, instruções de montagem e sistemas de navegação GPS.

# Erros Comuns

**Confundir algoritmo com código pronto** - algoritmo são os passos; o código é uma forma de escrever esses passos para o computador.

**Achar que Python é "só para IA"** - a linguagem é usada em web, dados, automação, educação, ciência, entre outros.

**Achar que simplicidade = limitação** - a sintaxe simples não impede projetos grandes.




## Aquecimento rápido:

Liste 2 tarefas do cotidiano e escreva os passos de cada uma como se fossem um algoritmo (ex.: “preparar um sanduíche”, “organizar o material da aula”, “chegar até a escola”).

Compare as sequências: o que ficou claro? O que pode ser melhorado?

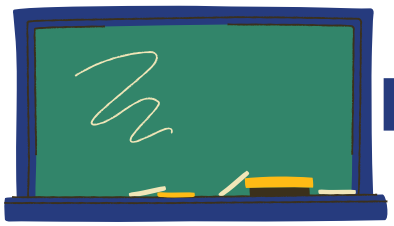
# Atividades do capítulo

1. Quem criou o Python e em que ano?
2. O nome da linguagem tem relação com qual programa de TV?
3. Cite dois motivos para usar Python.
4. Dê um exemplo de algoritmo no cotidiano.
5. Verdadeiro ou falso: Python só serve para projetos pequenos.

## Hora do Desafio:

Escreva, em até 5 passos, um algoritmo para organizar sua mochila.

Extra: transforme os passos em pseudocódigo (frases curtas numeradas) para usar no próximo capítulo.



# Professor em Ação

### Objetivos de aprendizagem:

Reconhecer Python na história recente da computação; explicar, com exemplos, por que a linguagem é uma boa escolha; definir algoritmo com exemplos concretos do cotidiano.

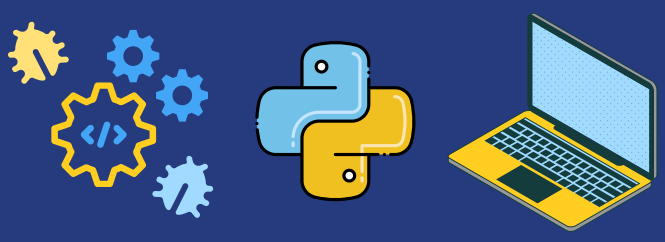
### Roteiro sugerido (20 min):

**1–5 min:** história + curiosidade do nome;
**6–10 min:** por que usar (exemplos reais);
**11–15 min:** o que é algoritmo (exemplos do dia a dia);
**16–20 min:** atividade em dupla (aquecimento) + partilha rápida;

### Avaliação simples:

**Iniciante =** reconhece o criador e cita um exemplo de algoritmo;
**Médio =** explica dois motivos para usar Python e escreve passos claros;
**Avançado =** conecta exemplos à própria realidade e usa pseudocódigo.

# Capítulo 2: Apresentação da IDE



# Objetivo

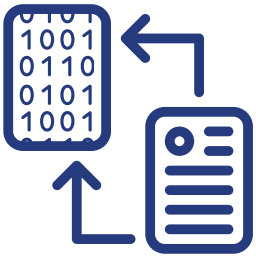
Neste capítulo vamos apresentar o ambiente de desenvolvimento integrado (do inglês, Integrated Development Environment - IDE), local amplamente utilizado pelos desenvolvedores para programar de forma rápida.

# Você vai aprender

- Um pouquinho sobre os responsáveis por traduzir nossa linguagem para o computador;
- Uma definição de IDE;
- Apresentando o ambiente virtual Google Colab.

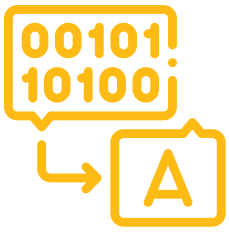
# 2 Compiladores e Interpretadores

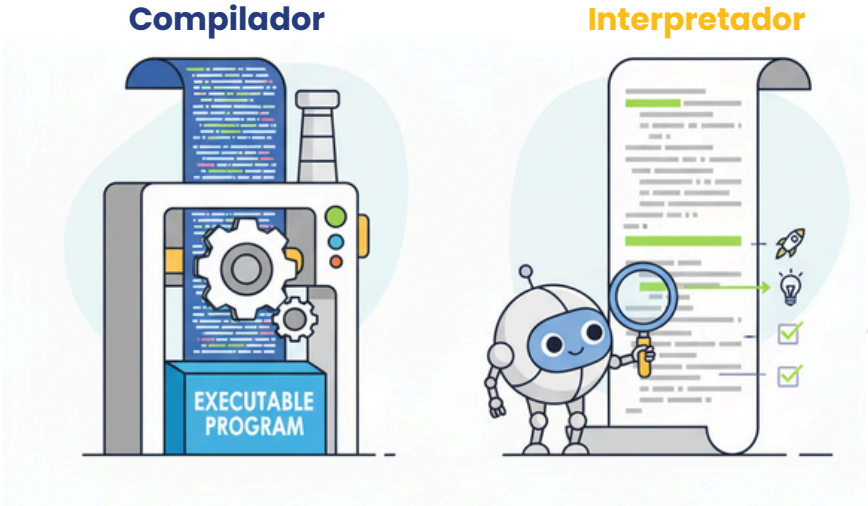
2.1.




Responsáveis por traduzir os códigos para número binário (linguagem entendida pelo computador), os compiladores e interpretadores convertem as instruções dadas pelos programadores para que o computador possa executar os comandos solicitados.

A principal diferença entre os dois é a forma com que eles traduzem o código. Enquanto o compilador traduz o código-fonte todo de uma só vez antes de executá-lo, o interpretador traduz linha a linha, executando imediatamente as linhas após a tradução. O compilador gera um arquivo separado, chamado arquivo executável.

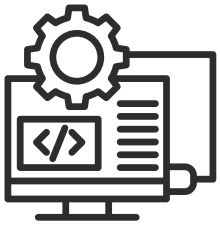



**Descrição:** À esquerda temos a representação de uma grande máquina que representa um compilador onde está sendo inserido pela parte superior um pergaminho na cor azul com linhas de código fonte. Dentro da máquina, engrenagens giram, simbolizando o processo de tradução. Na parte inferior, a máquina produz a seguinte mensagem: "EXECUTABLE PROGRAM" (Programa Executável) representado em um bloco azul. À direita está a representação do interpretador onde um robô segura uma lupa e examina um longo pergaminho branco com linhas de código. Conforme o robô analisa cada linha, setas apontam para ícones que representam ações imediatas como um foguete, uma lâmpada e dois vistos.

A decisão de usar um compilador ou um interpretador acaba variando de acordo com a necessidade ou propósito de cada linguagem e desenvolvedor.


# IDE's



IDE's vem do inglês Integrated Development Environment que em português seria Ambiente de Desenvolvimento Integrado.

Estes ambientes são softwares compostos de várias ferramentas e funcionalidades para melhorar o desempenho e aumentar a velocidade de produção dos sistemas.

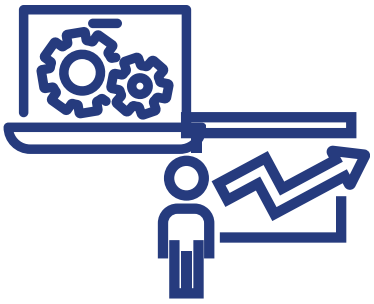
Existem três componentes essenciais em uma IDE, são eles:

**Editor de código:**
Responsáveis por nos permitir escrever e modificar os códigos das linguagens de programação.

**Compilador e/ou interpretador:**
Responsáveis por traduzir a nossa linguagem para a linguagem do computador.

**Debugger:**
Responsável por testar o programa e mostrar, caso haja, locais no código onde existem bugs (erros de sintaxe).

Graças a assimilação de vários equipamentos nos ambientes de desenvolvimento o tempo utilizado para ajuste e integração das ferramentas foram convertidos em tempo de codificação e correção de erros, ajudando também os programadores a manter uma organização no seu fluxo de trabalho.



# 2 O Google Colab

2.3.

O Google Colab é uma IDE e da mesma forma que as outras plataformas do Google é também sincronizada com a nuvem. É um recurso gratuito que nos permite trabalhar/desenvolver simultaneamente com outros colaboradores, caso compartilhado.

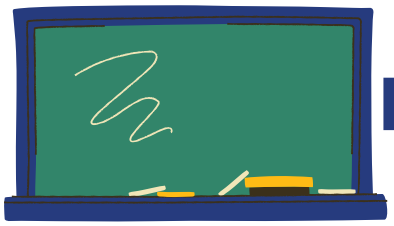
O ambiente virtual também é equipado com notebooks virtuais pré-configuradas além de suportar extensões e complementos, auxiliando na análise e interpretação.

Não se recomenda utilizá-lo para projetos que necessitam de maior segurança e privacidade, mas é uma ótima ferramenta para iniciar no mundo da programação.

**Para mais informações acesse:**
https://colab.research.google.com/#

## Atividades do capítulo

1. Qual a principal diferença entre compiladores e interpretadores?
2. Cite os três principais componentes das IDE's e suas funções.
3. Por que usar IDE's na programação?
4. Cite as vantagens e desvantagens do Google Colab.



# Professor em Ação

## Objetivos de aprendizagem:

Entender o que são Ambientes de Desenvolvimento Integrado; explicar por que são amplamente utilizados pelos programadores.

## Roteiro sugerido (15 min):

**1–5 min:** Compiladores e interpretadores;
**6–10 min:** O que são IDE's e suas características;
**11–15 min:** Por que usar IDE's + apresentando o google colab.

## Avaliação simples:

**Iniciante =** o que são IDE's;
**Médio =** características e motivos de usar as IDE's e também principal diferença entre compilador e interpretador;
**Avançado =** utiliza a IDE do Google Colab para codificar o algoritmo criado no capítulo anterior.

# Capítulo 3: Sintaxe do Python

# Objetivo

Antes de começar a programar de verdade, precisamos conhecer as **regras do jogo**. Essas regras chamamos de **sintaxe**.

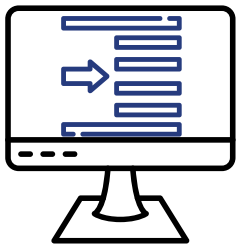
A sintaxe é um conjunto de regras que definem como o programa será escrito e interpretado. Por isso, vamos aprender a base necessária para iniciarmos nossa caminhada codificando em Python.

# Você vai aprender

- A importância da indentação em Python
- As palavras reservadas do Python
- O que são identificadores e como reconhecê-los
- Por que usar comentários (são nossos melhores amigos)

# 3 A indentação em Python

3.1.



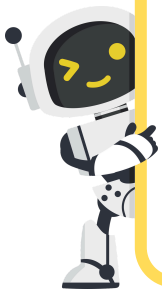
Enquanto outras linguagens enchem a tela de símbolos como **;**, **{ }** ou **( )**, o Python é minimalista. Aqui, a organização do espaço em branco é levada a sério.

Ou seja, se você esquecer uma tabulação ou colocar espaços errados, o código simplesmente não vai rodar.

No Python, diferente de outras linguagens, a indentação se torna algo indispensável para o funcionamento do código. Então cada espaço em branco é essencial para o funcionamento e não deve ser visto como opcional.


**Lembre-se: o Python não aceita bagunça!**

```
"função"
  ("Esta linha está indentada e dentro da função.")

"função"
("Já esta linha não está indentada e não esta dentro da função.")
```



**Cuidado!**

Esta linguagem é **case-sensitive** , ou seja, distingue maiúsculas de minúsculas, tratando Palavra (com apenas uma letra maiúsculas) ≠ palavra (toda em minúscula) ≠ PALAVRA (toda em maiúscula) como identificadores diferentes, isso ocorre tanto nas variáveis quanto nas palavras reservadas.


# Palavras Reservadas

O Python tem palavras que são “mágicas” para a linguagem. São comandos especiais que já vêm com significado próprio.

Assim como qualquer outra linguagem de programação, o Python inclui palavras reservadas, que possuem um significado especial para a linguagem e não devem ser utilizadas fora de contexto, nem podem ser utilizadas para dar nomes a variáveis

As principais palavras reservadas da linguagem estão demonstradas a seguir:

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

```
except = 5

abacaxi = 5
```

```
  File "/tmp/ipython-input-1308548235.py", line 1
    except = 5
    ^
SyntaxError: invalid syntax
```

**Atenção!**

A quantidade de palavras reservadas pode variar de acordo com a versão da linguagem Python que está sendo utilizada!


# Identificadores (dando nomes às coisas)

Os identificadores são os nomes ou rótulos dados por nós para ajudar a distinguir entidades como variáveis, funções, classes, etc. São eles (os identificadores ou nomes) que fazem o código ser mais legível (e menos misterioso).

Existem algumas regras para a escolha dos identificadores, são elas:

- **Podem conter letras (A - Z - a - z), dígitos (0 - 9) e underline ( _ )**
  Exemplo: minhaVarialvel_1
- **Não podem iniciar com dígitos (0 - 9)**
  Exemplo: 1234asdf - não é aceito como identificador
- **Palavras-chaves não podem ser usadas como identificadores!**
  Exemplo: for=5 - o for é um palavra reservada

```
nome_do_aluno = "Maria"
idade2= 10
```

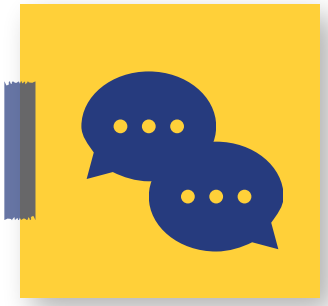
```
2idade = 10
for = "banana"
```

**Não esqueça!**
Python diferencia maiúsculas de minúsculas. Então **Amigo, amigo e AMIGO** são identificadores diferentes.

3

3.3.

# Criando comentários



Ao criarmos códigos muito grandes os comentários se tornam itens essenciais, pois nos ajudam a localizar ou entender a funcionalidade ou o raciocínio utilizado no decorrer do programa.

São como post-its que você deixa dentro do código para se lembrar do que estava pensando. Eles não atrapalham a execução do programa, mas ajudam você (e quem mais for ler seu código) a entender o raciocínio por trás.

No Python há dois tipos de comentários:

**O comentário em linha:** use **#** para transformar tudo que está a partir daquele símbolo até o final da linha em comentário.

```
# este é um comentário em linha
aqui não é mais comentário
```

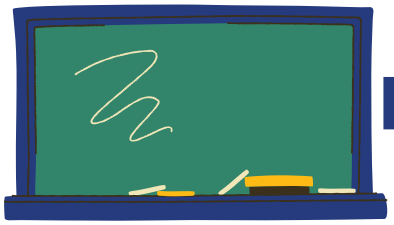
**O comentário em bloco:** use **‘ ‘ ‘** ou **“ “ “** no começo e no fim para escrever várias linhas de comentário.

```
"""este comentário
também é em bloco"""
sdfg
```

```
'''este paragrafo
é em bloco'''
sdfghj
```

# Atividades do capítulo

1. Por que a indentação é considerada indispensável em Python?
2. O que significa dizer que Python é case-sensitive?
3. O que são palavras reservadas?
4. Utilizando o algoritmo criado no capítulo 2, crie identificadores para armazenar os principais passos da sua organização.



# Professor em Ação

### Objetivos de aprendizagem:

Ensinar a base da linguagem Python de forma leve e prática.

### Roteiro sugerido (15 min):

**1–5 min:** Frisar bastante a importância da indentação do código (faça um código errado de propósito para eles verem a diferença).
**6–10 min:** Apresentar as palavras reservadas (pode ser em forma de “cartas mágicas”).
**11–15 min:** Explicar identificadores e brincar com nomes criativos.
**16-20min:** Esclarecer a necessidade e relevância dos comentários ao decorrer do código (Mostrar comentários e pedir que eles inventem explicações engraçadas no código).

### Avaliação simples:

**Iniciante =** reescreva o algoritmo para organizar sua mochila adicionando comentários;
**Médio =** crie mais passos para seu algoritmo, indente-o corretamente e adicione comentários ;
**Avançado =** utilizando tudo que foi aprendido nesta aula crie um algoritmo para tomar banho, seja criativo e não se esqueça de se ensaboar!.

### Viu só?

**Python** não é só sobre código, é sobre dar vida às ideias de forma clara, organizada e até divertida.

# Capítulo 4:

# Apresentando as Variáveis

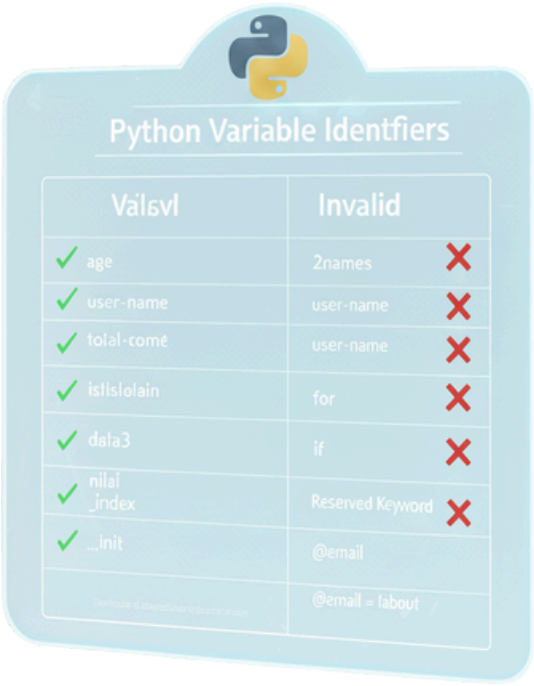
# Objetivo

Orientar sobre como criar e utilizar as variáveis ao codificar em Python. A variável é uma palavra criada pelo programador que corresponde a um valor, a criação de variáveis vem da necessidade de se usar um dado em diferentes contextos, facilitando o reuso e a modificação em grandes programas.

# Você vai aprender

- Como criar nomes das variáveis;
- Conhecendo as variáveis;
- As variáveis numéricas;
- Funcionalidades das *strings*;
- O que é o tipo *boolean*.

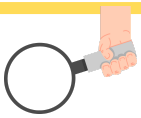
# 4 4.1. Nomes de variáveis



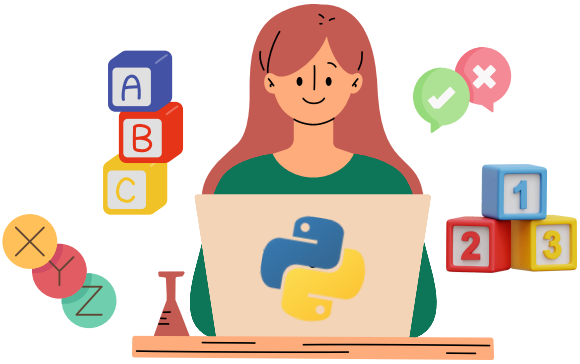

Existem regras e convenções quanto ao nome de uma variável na linguagem. È possível criar uma variável que inicie com letras maiúsculas porém isso é visto como uma quebra de padrão já que na linguagem variáveis são escritas sempre em minúsculas (a comunidade chama de snake_case). Porém essa é uma convenção, se você criar uma variável no código como: Nome ou IDADE, o programa não identifica nenhum erro de sintaxe.

Agora se você iniciar o nome de uma variável com números, como: 1a, isso resultará em um erro de sintaxe na interpretação.

## Curiosidade



A partir da **versão 3** da linguagem é possível adicionar acento nos nomes das variáveis.



# 4 Tipos de variáveis simples

4.2.

As variáveis são espaços alocados na memória que armazenam dados ou expressões que serão utilizados no decorrer do programa, nos permitindo manipular os dados armazenados nelas.

Uma analogia útil é pensar nas variáveis como gavetas de um grande armário ( que seria a memória), cada gaveta guarda apenas um objeto, mas uma gaveta triangular pode apenas guardar objetos triangulares, as gavetas circulares objetos circulares e assim por diante.

Assim como as gavetas possuem diferentes formas e tipos, as variáveis também apresentam classificações distintas.
Os tipos de variáveis disponível em Python são:

**Inteiro (int):** Exemplo: 8 ;
**Ponto Flutuante ou Decimal (float):** Exemplo: 3.14 ;
**String (str):** Exemplo: ola ;
**Boolean (bool):** Exemplo: true / false ;

Existem também tipos que funcionam como coleções, capazes de armazenar mais de um valor - sejam eles do mesmo tipo ou não. Alguns exemplos são:

**List (list):** Exemplo: ['Mônica', 'Magali', 'Cebolinha', 'Cascão'];
**Tuple:** Exemplo: (20, 34, 67, 56, 88);
**Dictionary (dic):** Exemplo: {'Alice': 1.50, 'Brian': 1.70, 'LeBron': 2.06};

A forma como chamamos as variáveis é escrevendo um identificador e utilizando o símbolo de atribuição "=" para armazenar diretamente o valor ou um identificador onde será armazenado o dado.

# 4.3. Variáveis numéricas

## Tipo inteiro (integer)

É a variável que utilizamos quando queremos armazenar um número pertencente ao conjunto dos inteiros;

```
Ano = 2025
```

```
Idade = 20
```

## Tipo ponto flutuante ou decimal (float)

Diferente da variável anterior, utilizamos esta para armazenar números decimais;

```
NotaFinal = 8.5
```

```
Altura = 1.65
```

# 4.4. Variável de texto e suas operações

## Tipo texto (*string*)

A variável *string* é descrita como uma sequência de caracteres, sendo eles letras, dígitos ou símbolos.

| string | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | m | a | n | d | a | | e | | b | o | n | i | t | a |

**Descrição:** A imagem apresenta duas linhas dispostas em sequência. Na linha superior encontra-se a palavra "*string*". A segunda linha é composta por uma sequência de 15 quadrados, e cada um deles representa um caractere da frase "Amanda é bonita", incluindo os espaços entre as palavras.

Entretanto, a linguagem Python não reconhece os acentos gráficos, pois o computador não lê diretamente as letras, elas são convertidas em números binários, a combinação de uma sequência de 0's e 1's é o que determina qual caractere será mostrado na tela do computador.

## Índice de string

Assim como em outras linguagens, as *strings* são consideradas um vetor de caracteres - ou seja, é como se houvesse várias "gavetas" iguais, dispostas lado a lado e interligadas.

Cada caractere na string tem um número que indica a sua posição dentro dela, como uma espécie de índice que mostra onde ele está na sequência.

| string | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | m | a | n | d | a | | e | | b | o | n | i | t | a |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

**Descrição:** Na imagem temos uma linha onde está escrito "string" em preto. Na linha abaixo temos uma sequência de 15 quadrados onde em cada quadrado temos um caracter da frase "Amanda é bonita" ou um espaço. Na última linha temos novamente a sequência de quadrados com a sequência numérica de 0 à 14.

O fato de possuírem uma ordem sequencial torna mais fácil de se manipular, permitindo que seja possível percorrer a string utilizando funções.

Podemos utilizar os colchetes "[ ]" como forma de indicar a posição do vetor que desejamos:

### Exemplo:

```
s = "Bernardo"

print(s[0]) #imprime a primeira letra do nome
print(s[7]) #imprime a última letra do nome
```

```
B
o
```

Utilizamos a função **print( )** para mostrar na tela a informação pedida dentro dos parênteses.


# Funções da string

Dentro do domínio das strings podemos utilizar algumas funções para modificar os dados postos dentro da variável. Uma ótima forma de demonstrar estas aplicações é na prática, por isso vamos abrir nossa IDE e colocar a mão na massa.

Vamos começar definindo uma variável para vermos as mudanças ocorrendo:

```
Nome = "Emanuela da Silva Silveira"
print(Nome)
```

```
Emanuela da Silva Silveira
```

Ao clicarmos no play vemos nosso nome surgir na tela.

## len

A função len é responsável por retornar o tamanho da string, ou seja, a quantidade de gavetas do armário que esta variável ocupa:

**Exemplo:**

```
Nome = "Emanuela da Silva Silveira"
print(len(Nome))
```

```
26
```

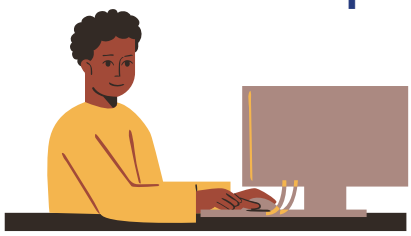
**Não esqueça!**

O espaço conta como um caractere na sequência de strings

### .lower()

Já a lower transforma todas as letras da variável em minúsculas:

**Exemplo:**



```
Nome = "Emanuela da Silva Silveira"
print(Nome.lower())
```

```
emanuela da silva silveira
```

### .upper()

Ao contrário da anterior, este transforma todas as letras em maiúsculas:

**Exemplo:**

```
Nome = "Emanuela da Silva Silveira"
print(Nome.upper())
```

```
EMANUELA DA SILVA SILVEIRA
```

### .title()

Diferente das duas últimas funções, esta transforma apenas a primeira letra das palavras em maiúscula:

**Exemplo:**

```
Nome = "emanuela da silva silveira"
print(Nome.title())
```

```
Emanuela Da Silva Silveira
```

### .count()

Por último, esta é responsável por contar a quantidade de ocorrências de uma palavras especificada dentro dos parênteses da função:

**Exemplos:**

```
Nome = "Emanuela da Silva Silveira"
print(Nome.count("Emanuela"))
```

```
1
```

```
Nome = "Manu Emanuela Manu Silva Manu Silveira Manu"
print(Nome.count("Manu"))
```

```
4
```

**Não esqueça!**

Todas as funções utilizam como base o que está armazenado na variável e não o que foi modificado no decorrer o código!

## Operações com a variável string

Da mesma forma que as funções, as operações nos permitem manipular essa variável.


**Descrição:** A imagem apresenta exemplos visuais de operações com strings em Python. Todas as palavras da imagem estão escritas com uma fonte amarela e estilizadas que imita a textura de uma corda. No canto superior esquerdo, há uma ilustração da palavra "Olá" e "Mundo!", separadas por um grande símbolo preto de adição (+). Acima dessas palavras, um retângulo roxo exibe em branco o comando print("Olá" + "Mundo!!"). Logo abaixo da palavra "Olá", está escrito em amarelo com letras menores "OláMundo". Um pouco abaixo, ainda no lado esquerdo, aparecem três palavras "Olá", alinhadas na horizontal. Acima delas, está um retângulo roxo com o comando na cor branca print("Olá" * 3).Abaixo da sequência de "Olá" aparece o resultado esperado: "OláOláOlá", escrito em azul escuro. No canto direito da imagem, há uma ilustração da palavra "Fatiamento" e acima dela encontra-se uma régua. Um marcador vertical cruza entre os caracteres, destacando o trecho "mento". Acima disso, outro retângulo roxo mostra o comando em branco print('string[7:12]'). Abaixo da palavra fatiada, em letras menores, aparece a palavra "mento" na cor amarela e entre aspas.

## Concatenação

Esta operação junta duas strings em uma só, podendo apenas transformar em uma palavra, ou adicionando um espaço com os parênteses.

**Exemplo:**

```
nome = "Emanuela"
sobrenome = "Silveira"

print(nome + sobrenome)
print(nome + " " + sobrenome)
```

```
EmanuelaSilveira
Emanuela Silveira
```

## Slicing ou fatiamento

Função que permite podemos extrair uma palavra de dentro da outra. Utilizando os índices da string escolhemos as letras que queremos iniciar e finalizar a palavra que será retirada.

**Exemplo:**

```
nome = "Emanuela"
print(nome[1:5])
```

```
manu
```

## Repetição ou duplicação

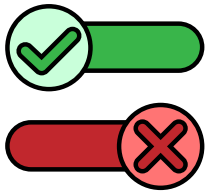
Através deste operador conseguimos fazer com que a palavra ou frase seja repetida várias vezes com apenas um comando.

**Exemplo:**

```
nome = "Emanuela"
nomeEspaco = "Emanuela "
print(nome*5)
print(nomeEspaco*5)
```

```
EmanuelaEmanuelaEmanuelaEmanuelaEmanuela
Emanuela Emanuela Emanuela Emanuela Emanuela
```


# Variável lógica



## Tipo lógico (Boolean)

Esta variável armazena valores do tipo lógico, que podem ser resumidos como verdadeiro ou falso ( True ou False ).

O valor **0** representa False e o valor **1** o True.

São variáveis muito usadas em estruturas condicionais e loops, que serão explicados mais à frente.

**No sexto capítulo, apresentaremos a vocês os operadores, que são utilizamos juntamente com as variáveis para manipular os dados em Python!**

## Hora do Desafio:

Crie um novo algoritmo, desta vez que armazene o estoque de um supermercado utilizando os três tipos de variáveis apresentados no capítulo.

### Exemplos:

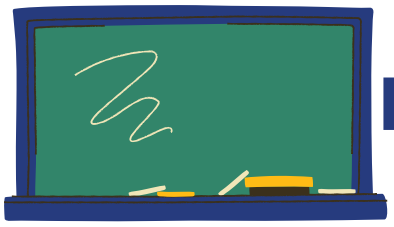
```
#supermercado
produto1 = "tomate"
quantidade1 = 5
preco1 = 2.8
```

```
#supermercado
produto2 = "Laranja"
quantidade2 = 10
preco2 = 7.2
```

```
#supermercado
produto3 = "Batata-doce"
quantidade3 = 7
preco3 =  12.9
```

# Atividades do capítulo

1. Quais foram os tipos de variáveis apresentadas neste capítulo?
2. Escreva 3 variáveis de cada tipo em sua IDE.
3. Escolha 4 das funções do tipo string e altere as variáveis escritas.
4. Utilizando as três strings criadas, treine as operações com string.



# Professor em Ação

### Objetivos de aprendizagem:

Descomplicar o aprendizado das variáveis simples do Python!

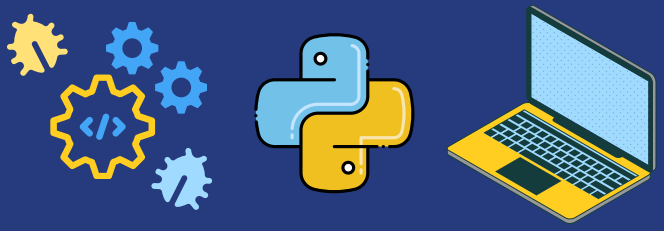
### Roteiro sugerido (15 min):

**1–5 min:** Demonstrar os tipos de variáveis e sua forma de escrita.
**6–10 min:** Frisar a diferença entre as variáveis numéricas.
**11–15 min:** Explicar e mostrar o que cada função das strings faz.
**16-20min:** Orientar sobre o tipo booleano e revisar as características de cada variável.

### Avaliação simples:

**Iniciante =** Com base no algoritmo criado neste capítulo, quais foram os tipos utilizadas em cada variável;
**Médio =** Crie variáveis para o algoritmo de banho criado no capítulo anterior;
**Avançado =** Escolha uma parte de sua rotina diária e crie variáveis.

# Capítulo 5: Variáveis compostas



## Objetivo

Entender as diferenças entre as variáveis primitivas (ou simples), apresentadas no capítulo anterior, e as variáveis compostas, além de conhecer os três tipos de variáveis compostas do Python.

## Você vai aprender

- A principais diferenças entre variáveis simples e compostas;
- O que é uma tupla;
- Como criar listas em Python;
- Finalidade e funções dos dicionários.

# 5 Variáveis Simples VS Compostas

5.1.

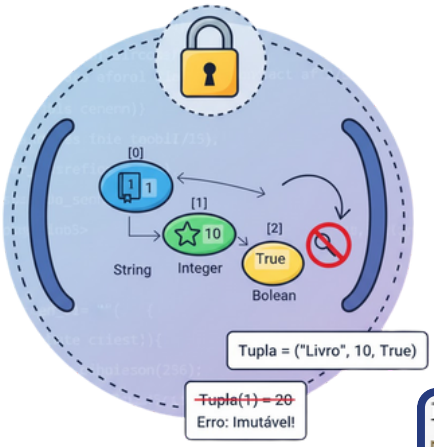
Enquanto as variáveis simples podem acomodar apenas um valor por vez, as variáveis compostas conseguem receber diversas, sendo do mesmo tipo ou não.

Pense nas variáveis simples como uma caixa pequena que só pode guardar uma única coisa de cada vez. Se você colocar algo novo, o valor antigo é substituído. Agora as variáveis compostas são como um organizador com várias prateleiras e gavetas. A variável é o organizador inteiro, e dentro dela você pode guardar vários valores, cada um em sua gaveta. Elas são usadas para agrupar e organizar outros dados.

Entretanto, as variáveis que são armazenadas nos tipos compostos são, de certa forma, variáveis do tipo simples, então, no final das contas é como se ao invés de um nicho do armário as variáveis compostas fossem um armário dentro do armário.


**Descrição:** Na imagem temos três estantes de prateleiras na cor marrom e na parte superior central o título "Memória do computador". À esquerda, na vertical, há a legenda: "exemplo de uma variável composta". Na primeira estante, há um contorno preto que a destaca.Na prateleira de cima, estão várias caixas enfileiradas, cada uma com uma letra, formando a palavra "STRING", nas duas prateleiras inferiores da mesma estante, há uma caixa com "1" e outra com "3.14". Na segunda estante (ao centro), a quarta prateleira tem uma sequência de caixas, cada uma com uma letra formando "STRING".Na terceira estante, apenas a segunda prateleira está ocupada, contendo a caixa "3.14.

# 5 5.1. Tuplas




**Descrição:** A imagem apresenta um diagrama circular onde em seu centro há três elementos internos conectados por setas e indicados por índices entre colchetes. Acima do círculo, há um cadeado dourado e próximo às bordas laterais temos parênteses e no interior deles há uma sequência. Uma forma azul representando o valor "Livro" (string). O próximo índice aponta para uma forma verde com o número 10. O último aponta para uma forma amarela com o valor True, representando um valor booleano. À direita, uma seta vermelha e um símbolo de proibido. Embaixo, há uma anotação de código: Tupla = ("Livro", 10, True). Logo abaixo, uma tentativa de alteração: Tupla[1] = 20 está riscada, seguida da mensagem "Erro: Imutável!"

São variáveis heterogêneas imutáveis, ou seja, diferente dos outros dois tipos, não é possível acrescentar, alterar ou remover as variáveis que foram definidas anteriormente.

Ao declararmos uma tupla utilizamos parêntese "( )" para indicar o início e o fim da variável, e cada item é separado utilizando uma vírgula.

```
linguagens = ("python" , "C" , "JavaScript" , "Lua")
print(linguagens)

linguagens[0] = "C++"
print(linguagens)
```

```
('python', 'C', 'JavaScript', 'Lua')
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipython-input-2205087267.py in <cell line: 0>()
      2 print(linguagens)
      3
----> 4 linguagens[0] = "C++"
      5 print(linguagens)

TypeError: 'tuple' object does not support item assignment
```

Graças a sua imutabilidade, as tuplas são ótimas para guardar valores constantes, como, por exemplo, as coordenadas de um mapa.

```
coordenadas = ('-28.951287334170402' , '-49.46753110935272')
print(coordenadas)
```

```
('-28.951287334170402', '-49.46753110935272')
```

Como citado anteriormente, a tupla é uma variável heterogênea, ou seja, é possível armazenar valores de diferentes tipos de variáveis primitivas.

**Exemplo:**

```
bolsista = ("manu" , 19 , 8 , "Agosto" , 2006 , True)
print(bolsista)
```

```
('manu', 19, 8, 'Agosto', 2006, True)
```

Nas tuplas também é possível utilizar o asterisco para repetir várias vezes todos os dados da tupla:

**Exemplo:**

```
bolsista = ("Manu" , 19 , True)
print(bolsista*2)
```

```
('Manu', 19, True, 'Manu', 19, True)
```

Por mais que não seja possível modificar internamente a tupla podemos deletá-la utilizando o comando del:

**Exemplo:**

```
linguagens = ("Python" , "Lua" , "C" , "C++")
print(linguagens)

del linguagens
print(linguagens)
```

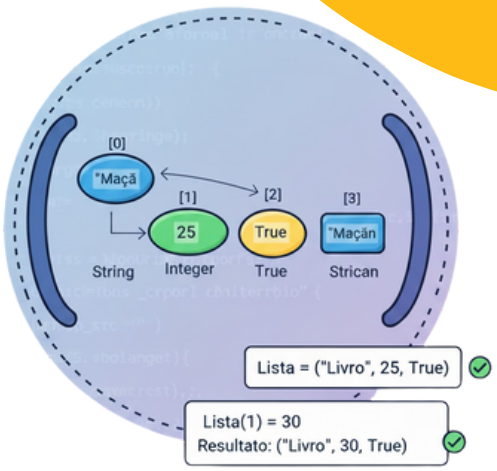
```
('Python', 'Lua', 'C', 'C++')
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipython-input-47251073.py in <cell line: 0>()
      3
      4 del linguagens
----> 5 print(linguagens)

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

# 5 Listas

5.2.

As listas são variáveis compostas mutáveis e heterogêneas. As listas permitem duplicatas, ou seja, o mesmo valor pode ser escrito mais de uma vez no decorrer da variável.

Utilizamos os colchetes [ ] para iniciar e finalizar as listas e da mesma forma que as tuplas, as vírgulas são utilizadas para separar cada um dos dados.




**Descrição:** A imagem mostra um diagrama circular que em seu interior, há quatro elementos representando diferentes tipos de dados com setas ligando-os. Cada elemento é identificados por colchetes com índices, onde o primeiro aponta para uma forma azul com o texto "Maçã", simbolizando um valor do tipo string. O próximo índice aponta para uma forma verde com o valor 25, representando um inteiro. O terceiro aponta para uma forma amarela com o valor True, indicando um booleano. No último aponta para outra forma azul com o texto "Maçã", novamente representando string. No canto inferior direito, há uma anotação de código: "Lista = ('Livro', 25, True)" com um símbolo de confirmação verde ao lado. Embaixo, "Lista = 30" e "Resultado: ('Livro', 30, True)", novamente acompanhados de um check verde.

```
inteiros = [1 , 2 , 3 , 4 , 5]
palavras = ["Abelha" , "Besouro" , "cavalo"]

print(inteiros)
print(palavras)
```

```
[1, 2, 3, 4, 5]
['Abelha', 'Besouro', 'cavalo']
```

Como a lista é mutável, podemos defini-la como nula e ir adicionando valores ao longo do código.

Além da forma tradicional de ler a lista podemos percorrê-la também com um laço de repetição (que será explicado no capítulo 9) :

```
inteiros = [1 , 2 , 3 , 4 , 5]
palavras = ["Abelha" , "Besouro" , "cavalo"]

for str in palavras:
  print(str)
```

```
Abelha
Besouro
cavalo
```

## Alterando a lista

Vamos criar uma lista com matérias escolares:

```
materias = ["Português" , "Matemática" , "História"]
```

Podemos substituir elementos da lista, sobrescrevendo-os:

**Exemplo:**

```
materias = ["Português" , "Matemática" , "História"]
print(materias)

materias[2] = "Geografia"
print(materias)
```

```
['Português', 'Matemática', 'História']
['Português', 'Matemática', 'Geografia']
```

Podemos também alterar um intervalo de valores ao invés de apenas um:

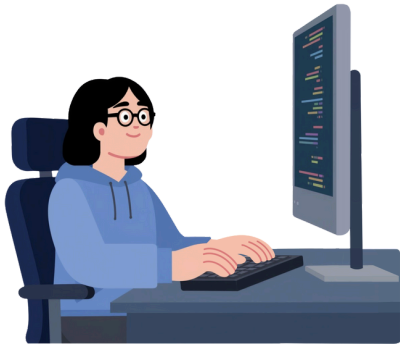
**Exemplo:**

```
materias = ["Português" , "Matemática" , "História"]
print(materias)

materias[2:3] = [ "Artes" , "Educação Física" ]
print(materias)
```

```
['Português', 'Matemática', 'História']
['Português', 'Matemática', 'Artes', 'Educação Física']
```

Para adicionar um item ao final da lista utilizamos o **.append()**:

**Exemplo:**



```
materias = ["Português" , "Matemática" , "História"]
print(materias)

materias.append("Geografia")
print(materias)
```

```
['Português', 'Matemática', 'História']
['Português', 'Matemática', 'História', 'Geografia']
```

Podemos adicionar mais de um item à lista utilizando a função **.extend()**:

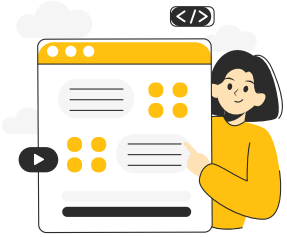
**Exemplo:**

```
materias = ["Português" , "Matemática" , "História"]
print(materias)

materias.extend(["Geografia" , "artes"])
print(materias)
```

```
['Português', 'Matemática', 'História']
['Português', 'Matemática', 'História', 'Geografia', 'artes']
```

Para que possamos escolher a posição que desejamos adicionar o item, utilizamos a função **.insert()**:

**Exemplo:**



```
materias = ["Português" , "Matemática" , "História"]
print(materias)

materias.insert( 2 , "Geografia")
print(materias)
```

```
['Português', 'Matemática', 'História']
['Português', 'Matemática', 'Geografia', 'História']
```

Conseguimos também criar listas dentro de listas:

**Exemplo:**

```
materias = ["Português" , "Matemática" , ["Física" , "Química" , "Biologia"] ,"História"]
print(materias)
```

```
['Português', 'Matemática', ['Física', 'Química', 'Biologia'], 'História']
```

Podemos também deletar itens específicos da lista utilizando o comando **del**:

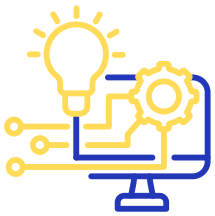
**Exemplo:**

```
materias = ["Português" , "Matemática" ,"História" , "Geografia"]
print(materias)
del(materias[2])
print(materias)
```

```
['Português', 'Matemática', 'História', 'Geografia']
['Português', 'Matemática', 'Geografia']
```

Outra função que se pode utilizar para deletar através do índice é o **.pop()**. Essa função se parece com "desempilhar de uma lista", por isso se não for especificado o índice, ele deleta o último elemento:

**Exemplo:**

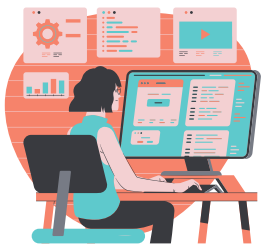


```
materias = ["Português" , "Matemática" ,"História" , "Geografia"]
print(materias)
materias.pop()
print(materias)
```

```
['Português', 'Matemática', 'História', 'Geografia']
['Português', 'Matemática', 'História']
```

Outra forma de deletar um elemento é pelo seu próprio nome com a função **.remove()**:

**Exemplo:**



```
materias = ["Português" , "Matemática" ,"História" , "Geografia"]
print(materias)
materias.remove("História")
print(materias)
```

```
['Português', 'Matemática', 'História', 'Geografia']
['Português', 'Matemática', 'Geografia']
```

A função **.clear()** remove todos os itens de uma lista; entretanto, diferente do comando **del**, os itens não são removidos da memória

**Exemplo:**

```
materias = ["Português" , "Matemática" ,"História" , "Geografia"
print(materias)
materias.clear()
print(materias)
```

```
['Português', 'Matemática', 'História', 'Geografia']
[]
```

## Operações com as listas

De maneira geral, os operadores funcionam de forma semelhante para as variáveis.

### Concatenação

Ao utilizarmos o símbolo de soma ( + ) conseguimos unir duas ou mais listas:

```
impares = [1 , 3 , 5 , 7 , 9]
pares = [2 , 4 , 6 , 8 , 10]

print(pares + impares)
```

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

### Duplicação

Utilizando o símbolo asterisco ( * ) podemos multiplicar por um número natural e repetir várias vezes a mesma lista:

```
materias = ["Português" , "Matemática"]
print(materias*3)
```

```
['Português', 'Matemática', 'Português', 'Matemática', 'Português', 'Matemática']
```

### Ordenando (de modo permanente)

Por meio da função **.sort( )** podemos ordenar os itens dentro da lista permanentemente a partir da linha que é chamada a função:

```
materias = ["Português" , "Matemática" , "Ciências"]

materias.sort()
print(materias)
```

```
['Ciências', 'Matemática', 'Português']
```

```
impares = [1 , 3 , 5 , 7 , 9]
pares = [2 , 4 , 6 , 8 , 10]
concatenacao = pares + impares

concatenacao.sort()
print(concatenacao)
```

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

## Ordenando (de modo momentâneo)

Através da função **sorted()** conseguimos ordenar a lista apenas naquela chamada.

```
materias = ["Português" , "Matemática" , "Ciências"]

print(sorted(materias))
```

```
['Ciências', 'Matemática', 'Português']
```

## Ordenação decrescente (de modo momentâneo)

Ainda utilizando a função **sorted()** podemos organizar de forma decrescente os elementos da função, adicionando o **return** como **true**:

```
numeros = [1,6,4,7,3,8,9,2,5,]

print(sorted(numeros, reverse=True))
```

```
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

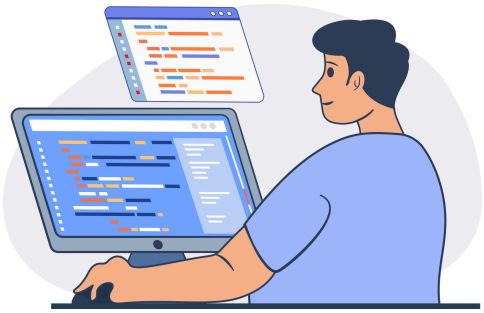

# Dicionários


Os dicionários atuam como variáveis compostas responsáveis por armazenar dados em pares, comumente conhecidos como **chave:valor**. São responsáveis por representar estruturas e relações, levando em consideração que sua organização é imutável - ou seja a ordem definida não mudará.

**Descrição:** A imagem apresenta um diagrama circular onde no topo, está o título "Dicionary" destacado em branco com um fundo azul roxo. Dentro do círculo, há dois símbolos de chaves grandes – uma à esquerda e outra à direita. No centro, há três caixas retangulares de diferentes cores, representando pares chave-valor, a caixa verde "idade" com valor 30, a amarela "ativo" com valor True, a caixa azul "nome" com valor "Alice", há linhas que conectam as chaves aos seus respectivos valores. À esquerda, há um ícone do Python e um símbolo de "mutabilidade". À direita, aparece um símbolo de proibição sobre uma chave com a legenda "Chaves DEVEM ser únicas!". Na parte inferior, há caixas brancas com exemplos de código Python com símbolos de verificação verde, indicando operações válidas na caixa superior: dicionario = {'nome': Alice, 'idade': 30, 'ativo': True} e na caixa inferior: dicionario['idade'] = 35 => {'nome': Alice, 'idade': 35, 'ativo': True}.

As chaves são valores imutáveis e não podem repetir seu identificador em outras chaves, responsáveis por permitir o acesso aos valores. Frequentemente utilizamos strings como as chaves, mas números ou tuplas também podem ser utilizadas como chaves.

Já o valor pode conter diversos tipos de elementos, sejam variáveis simples ou compostas. Eles representam os dados ou informações associados a determinada chave. Sendo mutáveis, podem ser definidos e redefinidos ao decorrer do código.



Aplicamos as chaves { } no início e fim dos dicionários e da mesma forma que as anteriores, as vírgulas são utilizadas para separar os pares chave:valor.

```
livro = {"Título": "Percy Jackson e os Olimpianos" , "Autor": "Rick Riordan"}
```

## Observe que:

É possível quebrar os dicionários em várias linhas, pois o Python desconsidera os espaços em branco dentro das variáveis.

Isso torna mais fácil e organizado escrever grandes dicionários utilizando várias linhas.

```
Alunos = {
    "Nome": "Alice",
    "idade": 9,
    "Pais": [
        {"nome": "Mateus", "idade": 38},
        {"nome": "Alessandra", "idade": 36}
    ]
}
```

## Alterando valores

Como foi citado na introdução do dicionário, as chaves não são alteráveis, visto que são elas nos ajudam a chegar nos valores do dicionário. Entretanto, os valores associados às chaves são alteráveis.

```
Alunos = {
    "Nome" : "Alice"  ,
    "Idade" : 9
}
print(Alunos)
Alunos["Idade"] = 10
print(Alunos)
```

```
{'Nome': 'Alice', 'Idade': 9}
{'Nome': 'Alice', 'Idade': 10}
```

## Adicionar nova chave

Podemos também criar uma nova classe dentro de um dicionário já existente, por isso também não é incomum que os dicionários sejam criados vazios e que os valores sejam adicionados no decorrer do código.

```
Alunos = {
    "Nome" : "Alice"  ,
    "Idade" : 9
}
print(Alunos)
Alunos["Série"] = "5º ano"
print(Alunos)
```

```
{'Nome': 'Alice', 'Idade': 9}
{'Nome': 'Alice', 'Idade': 9, 'Série': '5º ano'}
```

## Limpando os valores

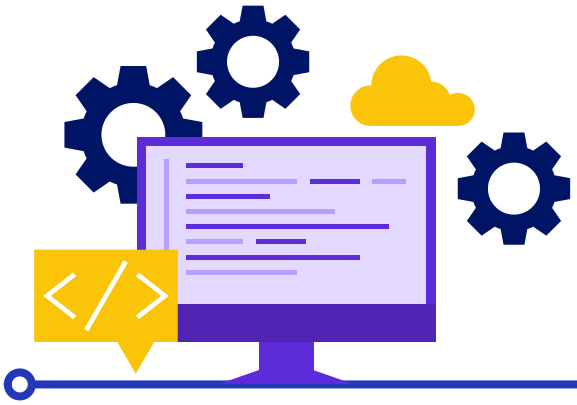
Podemos utilizar uma função **.clear( )** para limpar todos os dados do dicionário, deixando - o completamente limpo.

```
Alunos = {
    "Nome" : "Alice"  ,
    "Idade" : 9
}
print(Alunos)
Alunos.clear()
print(Alunos)
```

```
{'Nome': 'Alice', 'Idade': 9}
{}
```

## Removendo chaves utilizando o .pop( )

Da mesma maneira que na lista podemos utilizar o **.pop( )** para deletar a chave desejada.



```
Alunos = {
    "Nome" : "Alice"  ,
    "Idade" : 9 ,
    "Série" : "5º ano"
}
print(Alunos)
Alunos.pop("Série")
print(Alunos)
```

```
{'Nome': 'Alice', 'Idade': 9, 'Série': '5º ano'}
{'Nome': 'Alice', 'Idade': 9}
```

## Removendo chaves utilizando o del

Podemos também utilizar a função del, tanto para deletar apenas uma parte do dicionário quanto o dicionário inteiro.

```
Alunos = {
    "Nome" : "Alice"  ,
    "Idade" : 9 ,
    "Série" : "5º ano"
}
print(Alunos)
del Alunos["Série"]
print(Alunos)
```

```
{'Nome': 'Alice', 'Idade': 9, 'Série': '5º ano'}
{'Nome': 'Alice', 'Idade': 9}
```

```
Alunos = {
    "Nome" : "Alice"  ,
    "Idade" : 9 ,
    "Série" : "5º ano"
}
print(Alunos)
del Alunos
print(Alunos)
```

```
{'Nome': 'Alice', 'Idade': 9, 'Série': '5º ano'}
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipython-input-3466163643.py in <cell line: 0>()
      6 print(Alunos)
      7 del Alunos
----> 8 print(Alunos)

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

## Chaves e valores

Existe uma função que retorna os identificadores das chaves (**.keys( )**) e outra que retorna os identificadores dos valores (**.values( )**).


```
Alunos = {
    "Nome" : "Alice"  ,
    "Idade" : 9 ,
    "Série" : "5º Série"
}
print(Alunos)
Valores = Alunos.values()
print(Valores)
```

```
{'Nome': 'Alice', 'Idade': 9, 'Série': '5º Série'}
dict_values(['Alice', 9, '5º Série'])
```

```
Alunos = {
    "Nome" : "Alice"  ,
    "Idade" : 9 ,
    "Série" : "5º Série"
}
print(Alunos)
Chaves = Alunos.keys()
print(Chaves)
```

```
{'Nome': 'Alice', 'Idade': 9, 'Série': '5º Série'}
dict_keys(['Nome', 'Idade', 'Série'])
```

## Copiando o dicionário

Utilizando a função **.copy( )** podemos copiar o dicionário para um outro identificador.

```
Alunos = {
    "Nome" : "Alice"  ,
    "Idade" : 9 ,
    "Série" : "5º Série"
}
print(Alunos)
copia = Alunos.copy()
print(copia)
```

```
{'Nome': 'Alice', 'Idade': 9, 'Série': '5º Série'}
{'Nome': 'Alice', 'Idade': 9, 'Série': '5º Série'}
```

```
Alunos = {
    "Nome" : "Alice"  ,
    "Idade" : 9 ,
    "Série" : "5º Série"
}
copia = Alunos.copy()
Alunos["Idade"] = 10

print(Alunos)
print(copia)
```

```
{'Nome': 'Alice', 'Idade': 10, 'Série': '5º Série'}
{'Nome': 'Alice', 'Idade': 9, 'Série': '5º Série'}
```

Essa função nos permite editar o dicionário e manter a alteração caso queira chamá-la novamente.

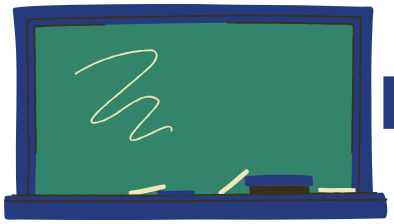
# Atividades do capítulo

1. Quais são as principais diferenças entre variáveis simples e variáveis compostas ?
2. Qual a principal diferença na forma de escrita das três variáveis compostas apresentadas?
3. O que são tuplas e quais suas características?
4. O que são listas e quais suas características?
5. O que são dicionários e quais suas características?

## Vamos criar!

Com base no que foi visto neste capítulo crie três códigos:

- **Tuplas**: com as coordenadas da sua casa;
- **Listas**: receita para preparar um prato;
- **Dicionários**: certidão do seu animal de estimação.



# Professor em Ação

## Objetivos de aprendizagem:

Ensinar os alunos a criar, utilizar e diferenciar as variáveis compostas.

## Roteiro sugerido (30 min):

**1–5 min:** Explicar as principais diferenças entre as variáveis compostas e as simples;
**6–10 min:** Como criar e utilizar as tuplas;
**11–20 min:** Funções e operações das listas;
**20-25 min:** Diferenças das listas e dicionários e funções do dicionário;
**26-30 min:** Revisar as características e diferenças dos tipos de variáveis compostas.

## Avaliação simples:

**Iniciante =** Explique as principais funções de cada variável apresentada neste capítulo;
**Médio =** Crie um dicionário que contenha listas e dicionários;
**Avançado =** Utilizando as funções apresentadas modifique a lista e o dicionário criados anteriormente.


# Capítulo 6: Operadores

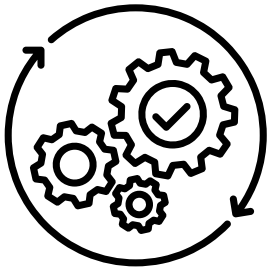
## Objetivo

Você vai aprender a compreender a lógica e as utilidades dos operadores, além de aplicá-los corretamente.

## Você vai aprender

- O que são os operadores.
- Operadores aritméticos.
- Operadores de atribuição.
- Operadores de comparação.
- Operadores lógicos.

# 6.1. O que são os operadores?



Imagine os operadores em Python como os "verbos de ação" do seu código. Se as suas variáveis são os personagens da história, os operadores são os comandos que fazem esses personagens interagirem, se juntarem ou se compararem. Sem a ação desses operadores, seus dados seriam como estátuas paradas, esperando por uma ação que nunca aconteceria.

No Python temos vários tipos de operadores, mas os que serão ensinados são os mais conhecidos e utilizados pelos programadores

Operadores Aritméticos,
Operadores Relacionais,
Operadores de Atribuição,
Operadores Lógicos.

# 6.2. Operadores Aritméticos

São os responsáveis por realizar as expressões algébricas, com eles podemos fazer uma calculadora com Python.

| | | |
|---|---|---|
| + | Adiciona dois operandos | 3 + 4 |
| - | Subtrai o operando | 5 - 2 |
| * | Multiplica dois operandos | 6 * 2 |
| / | Divide o operando | 49 / 7 |
| % | Módulo, resto da divisão | 10 % 3 |
| // | Divisão arredondada | 25 // 2 |
| ** | Expoente | 6 ** 2 |

```
x = 10
print(x + 4)
```

```
14
```

```
x = 10
print(x - 4)
```

```
6
```

```
x = 10
print(x * 4)
```

```
40
```

```
x = 10
print(x / 4)
```

```
2.5
```

```
x = 10
print(x % 4)
```

```
2
```

```
x = 10
print(x // 4)
```

```
2
```

```
x = 10
print(x ** 4)
```

```
10000
```

# 6.3. Operadores de Atribuição

São operadores que simplificam a sintaxe dos operadores aritméticos, atribuindo e realizando as operações em Python.

| | | |
|---|---|---|
| = | Utilizado para atribuir valores | x = 3 |
| += | Soma um valor ao numero em x | x += 1 |
| -= | Subtrai um valor ao número em x | x -= 1 |
| *= | Multiplica o número em x por um valor | x *= 2 |
| /= | Divide o número em x por um valor | x /= 5 |
| %= | Retorna o resto da divisão | x %= 1 |
| //= | Arredonda o quociente da divisão | x //= 3 |
| **= | Eleva o valor de x ao valor atribuido | x **=3 |

```
x = 10
print(x)
```

```
10
```

```
x = 10
x+=2
print(x)
```

```
12
```

```
x = 10
x-=2
print(x)
```

```
8
```

```
x = 10
x*=2
print(x)
```

```
20
```

```
x = 10
x/=2
print(x)
```

```
5.0
```

```
x = 10
x%=2
print(x)
```

```
0
```

```
x = 10
x//=2
print(x)
```

```
5
```

```
x = 10
x**=2
print(x)
```

```
100
```

# 6.4. Operadores Relacionais

Responsáveis pelas comparações, são muitas vezes utilizados em laços de repetição e principalmente em estruturas condicionais

| | | |
|---|---|---|
| > | Maior que | 7 > 4 |
| < | Menor que | 3 < 9 |
| == | Igual a | 2 == 2 |
| != | Diferente de | 1 != 3 |
| >= | Maior ou igual a | x >= 6 |
| <= | Menor ou igual a | x <= 8 |


# Operadores Lógicos

São a base da tomada de decisões no código, pois usados para unir ou modificar condições de Verdadeiro (True) ou Falso (False).

| | |
|---|---|
| **and (E)** | Retorna **True** se ambas as condições forem verdadeiras. |
| **or (ou)** | Retorna **True** se pelo menos uma condição for verdadeira. |
| **not (não)** | Inverte o valor, ou seja se é **True**, vira **False**. |

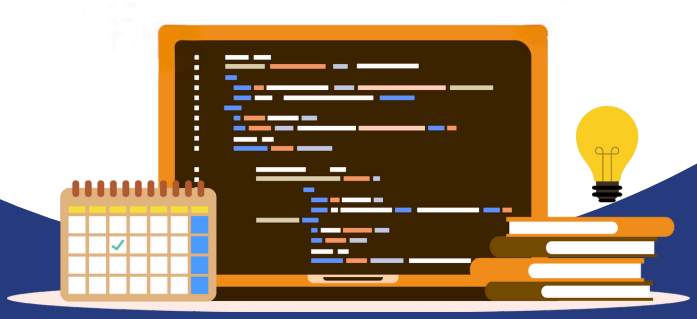

## Hora do Desafio:

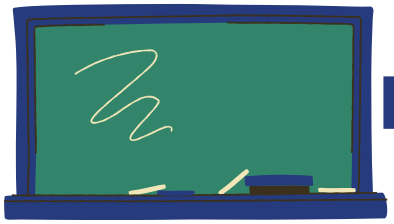
Em um parque de diversões há várias montanhas-russas que, dependendo da idade ou altura, algumas pessoas não podem utilizar. Escreva um programa que verifique se a pessoa pode ou não andar o brinquedo.

## REGRAS DO BRINQUEDO ##
-Idade mínima: 10 anos
-Altura Mínima: 130 cm

## Resposta do Desafio:

```
# Regras do Brinquedo
idadeMinima = 10
alturaMinima = 130

# Dados da pessoa
idadePessoa = #definir
alturaPessoa = #definir

atendeIdade = idadePessoa >= idadeMinima
atendeAltura = alturaPessoa >= alturaMinima
podeEntrar = atendeIdade and atendeAltura

print("---Regras do brinquedo---")
print(f"Idade mínima para entrada: {idadeMinima} anos")
print(f"Altura mínima para entrada: {alturaMinima} cm")
print("\n")
print(f"Idade da criança: {idadePessoa} anos")
print(f"Altura da criança: {alturaPessoa} cm")
print(f"Atende à idade mínima? {atendeIdade}")
print(f"Atende à altura mínima? {atendeAltura}")
print(f"Acesso ao brinquedo liberado? {podeEntrar}")
```





# Professor em Ação

## Objetivos de aprendizagem:

Apresentar os operadores aos alunos e suas funções, explicando o comportamento dos 4 tipos de operadores.

## Roteiro sugerido (20 min):

**1–5 min:** Explicar o que são os operadores e suas funções.
**6–10 min:** Ensinar sobre os operadores aritméticos.
**11–15 min:** Apresentar os operadores de atribuição e as diferenças entre eles e os operadores aritméticos.
16**–20 min:** Finalizar explicando sobre os operadores de comparação e os lógicos, frisando bem a interação dos operadores lógicos com os outros operadores.

## Avaliação simples:

**Iniciante =** Explique as principais funções de cada operador apresentado neste capítulo.
**Médio =** Crie um boletim escolar para calcular a média e dizer se o aluno foi aprovado, está em recuperação ou se foi reprovado.
**Avançado =** Adicione mais brinquedos ao nosso parque de diversões, seja criativo!.

## Não esqueça!

As variáveis não podem ter o mesmo identificador!

# Capítulo 7: Interagindo com o usuário

## Objetivo

É comum pedirmos aos usuários do programa informações, neste capítulo vamos aprender como podemos solicitar e armazenar esses dados em variáveis.

## Você vai aprender

- Como podemos solicitar informações?

7

7.1.

# Como interagimos com o usuário?

Ao criarmos um código é comum que necessitemos pedir informações do usuário para integrarmos no código, assim somos capazes de realizar tarefas levando em consideração as informações específicas de cada usuário.

Para isso utilizamos a função de entrada **input( )**, na qual o input é atribuído a uma variável e no interior dos parênteses colocamos o que desejamos solicitar ao usuário.

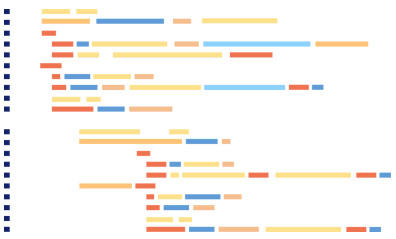
```
nome = input("digite seu nome: ")
print(nome)
```

```
digite seu nome: Manu
Manu
```

De forma geral, o input entende qualquer variável como string, então ao adicionarmos o operador de soma, ele vai entender como concatenação e não realizará a soma dos valores:

```
valor1 = input("digite um número: ")
valor2 = input("digite um número: ")
print(valor1 + valor2)
```

```
digite um número: 2
digite um número: 4
24
```



Para resolvermos esse problema devemos adicionar a função que desejamos transformar o input:

```
valor1 = int(input("digite um número: "))
valor2 = int(input("digite um número: "))
print(valor1 + valor2)
```

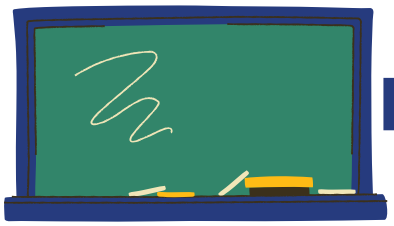
```
digite um número: 2
digite um número: 4
6
```

## Agora sim!

Viu como foi fácil? Quem diria que de forma simples e rápida seríamos capazes de conversar com o nosso código, não é?

# Atividades do capítulo

1. Qual é o objetivo da função input() em um programa Python?
2. Qual o tipo de variável padrão do input?
3. Como fazemos para ler um tipo int ou float?
4. Crie um exemplo para cada tipo de variável simples.



# Professor em Ação

### Objetivos de aprendizagem:

Ensinar como interagir com o usuário.

### Roteiro sugerido (10 min):

**1–5 min:** Explicar o que é a função input.
**6–10 min:** Demonstrar o erro que ocorre conforme o modo default da função input e como consertá-lo.

### Avaliação simples:

**Iniciante =** Peça ao usuário para digitar seu nome e sua idade e mostre-os na mesma frase na tela.
**Médio =** Crie um cadastro de pet, pedindo o nome, raça, altura e peso!.
**Avançado =** No capítulo anterior criamos um parque de diversão, agora que já sabemos como armazenar as informações do usuário, modifique o código para que seja ele o responsável por adicionar as informações da pessoa que entrará no brinquedo.


# Capítulo 8:

# Estruturas Condicionais

## Objetivo

No dia a dia vivemos nos perguntando se fazemos isso ou aquilo. No Python temos como realizar esse dois caminhos também! Hoje vamos aprender sobre o if e seus companheiros, que nos permitem seguir um caminho diferente dependendo da variável.

## Você vai aprender

- O que seria uma estrutura condicional?;
- If;
- Elif;
- Else;


# O que são as estruturas condicionais?


Às vezes temos que tomar decisões baseadas nas variáveis do dia a dia, se chover, abrir o guarda-chuva, se não chover fechar o guarda-chuva e por aí vai. No código também precisamos colocar condicionais, assim, por exemplo, poderíamos fazer um sistema, se chover o varal mecanizado é puxado para dentro de casa, senão, ele fica na rua!


**Atenção!**

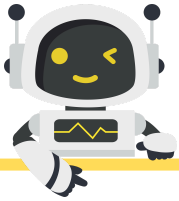
Para lidar com as estruturas condicionais é fundamental saber utilizar os operadores!


# Se chover, abra o guarda-chuva! (variável if)

Ao nos depararmos com situações em que, dependendo do que fizermos, temos consequências diferentes, pensamos por um momento: "Faço isso ou aquilo?". No Python não é diferente, utilizamos a palavra reservada if (que traduzindo para o português significa "se") para representar essa condição da nossa ação, conforme o exemplo a direita:

```
tempo = input("Digite o tempo:")

if tempo == "chuva":
  print("Não esqueça o guarda-chuva")
```

```
Digite o tempo:chuva
Não esqueça o guarda-chuva
```



Como demonstrado no exemplo acima, pela variável e a condição serem compatíveis a expressão que está após os dois pontos (:) será realizada.

## Atenção!

Como não utilizamos parênteses ou ponto e vírgula para indicar o início e fim da estrutura, lembre-se sempre de indentar o código

```
if tempo == "chuva":
print("Não esqueça o guarda-chuva")
```

```
  File "/tmp/ipython-input-3103557100.py", line 4
    print("Não esqueça o guarda-chuva")
    ^
IndentationError: expected an indented block after 'if' statement on line 3
```


# A última condição (variável else)

Mesmo tendo várias opções, sempre chega um momento em que se nada funcionar temos uma última condição, essa última é o else (que é traduzido como “senão”). Então se nenhuma das hipóteses anteriores funcionar utilizamos o else, ele não precisa de uma condição, visto que só é acionado caso nenhuma outra condição seja preenchida.

## Exemplo:

```
tempo = input("Digite o tempo: ")

if tempo == "chuva":
  print("Não esqueça do guarda-chuva")
elif tempo == "nublado":
  print("Talvez chova, é bom levar o guarda-chuva")
elif tempo == "nevando":
  print("Se encasaque bem para não pegar um resfriado!")
elif tempo == "nublado":
  print("Cuidado com a baixa visibilidade")
else:
  print("Aproveite seu dia!")
```

```
Digite o tempo: sol
Aproveite seu dia!
```


# Continuando a estrutura (variável elif)

Infelizmente, nossa vida não depende apenas de uma variável, e é nesse momento em que a linguagem Python se diferencia das outras linguagens. Enquanto nas outras temos apenas duas variáveis - o if e o else -, no Python temos o elif que utilizamos de forma intermediária.

Então, ao invés de a estrutura da condicional ser assim:

```
tempo = input("Digite o tempo: ")

if tempo == "chuva":
  print("Não esqueça do guarda-chuva")
else tempo == "nublado":
  print("Talvez chova, é bom levar um guarda-chuva")
```

```
  File "/tmp/ipython-input-3519151527.py", line 5
    else tempo == "nublado":
         ^
SyntaxError: expected ':'
```

Se você fizer da forma mostrada anteriormente irá ocorrer um erro de sintaxe, já que o Python não permite condições junto do else, por isso utilizamos o elif:

```
tempo = input("Digite o tempo: ")

if tempo == "chuva":
  print("Não esqueça do guarda-chuva")
elif tempo == "nublado":
  print("Talvez chova, é bom levar um guarda-chuva")
```

```
Digite o tempo: nublado
Talvez chova, é bom levar um guarda-chuva
```

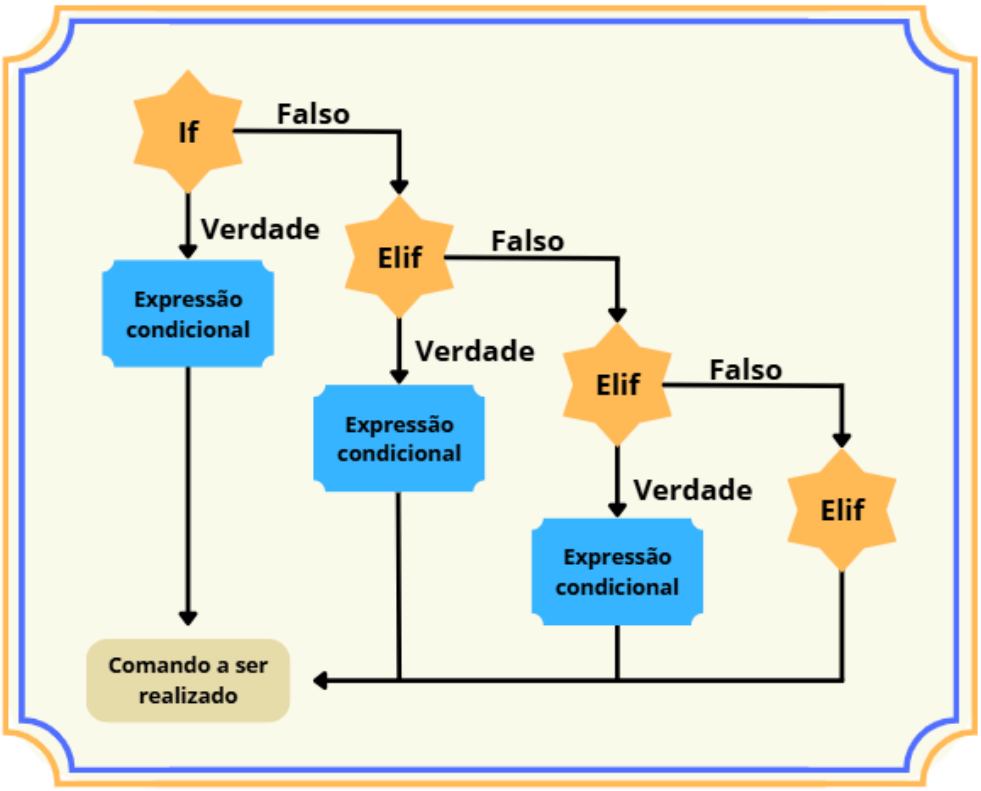
## Não esqueça!

É possível utilizar o else sem utilizar o elif

```
tempo = input("Digite o tempo: ")

if tempo == "chuva":
  print("Não esqueça do guarda-chuva")
else:
  print("Aproveite seu dia!")
```

```
Digite o tempo: sol
Aproveite seu dia!
```

As estruturas condicionais funcionam sempre da mesma maneira, se a linha do if for falsa, ela passa para a próxima condição (seja ela elif ou else) e quando encontrar a condição verdadeira ou chegar no else, o bloco de código será executado.




## Hora do Desafio:

Compare números de 0 a 9, vendo se eles são:

Ímpares
Pares
Maior que 5
Menor que 5
Igual a 5

## Respostas:

```
num = int(input("Digite um número: "))

if num==5:
  print("o número é cinco")
elif num%2 == 0 and num<5 and num<=10:
  print("o número é par e menor que cinco")
elif num%2 == 1 and num<5 and num<=10:
  print("o número é ímpar e menor que cinco")
elif num%2 == 0 and num>5 and num<=10:
  print("o número é par e maior que cinco")
elif num%2 == 1 and num>5 and num<=10:
  print("o número é ìmpar e maior que cinco")
else :
  print("o número é maior que nove")
```

# Atividades do capítulo

1. Por que utilizamos as estruturas condicionais?
2. Como iniciamos e finalizamos as estruturas condicionais?
3. Qual a utilidade do elif e por que não utilizamos diretamente o else?

# Professor em Ação

## Objetivos de aprendizagem:

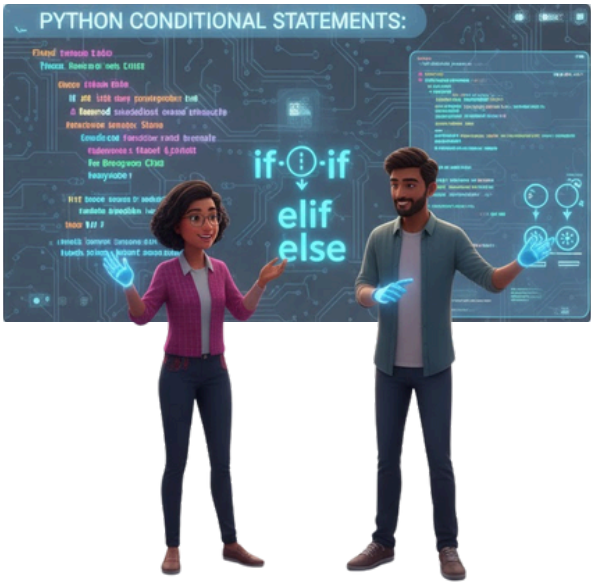
Explicar como utilizar as estruturas condicionais no Python.

## Roteiro sugerido (20 min):

**1–5 min:** Explicar o que é uma estrutura condicional e sua utilidade no nosso dia a dia.
**6–10 min:** Demonstrar como utilizar o if, frisar a importância da indentação.
**11–15 min:** Apresentar o elif e o else e por que utilizamos o elif ao invés do else if.
16**–20 min:** Esclarecer como funciona a estrutura condicional e a forma como o compilador lê a estrutura.

## Avaliação simples:

**Iniciante =** Crie um programa pra saber se uma pessoa já pode votar.
**Médio =** Compare se os números são negativos, positivos ou zero.
**Avançado =** Crie um cadastro onde comparamos o nome, e-mail e senha de um usuário, caso todas as condições estejam corretas informe que está logado.



# Capítulo 9:

# Estruturas de Repetição

# Objetivo

A ideia principal das estruturas de repetição é automatizar tarefas repetitivas. Em vez de escrever o mesmo comando várias vezes, podemos escrever apenas uma vez e pedir para o computador repetir quando for necessário. Isso economiza tempo e evita erros. Por isso, aprender a utilizar laços de repetição pode mudar totalmente sua forma de programar.

# Você vai aprender

- O que é uma estrutura de repetição?.
- O que significa while.
- O que significa for.
- Como utilizar a função **range( ).**
- Qual a diferença entre eles.

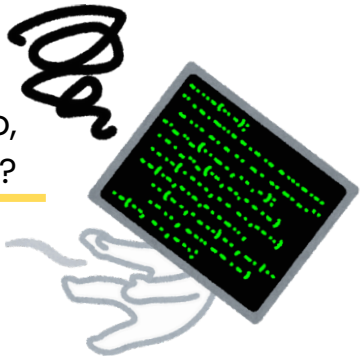

# O que é uma estrutura de repetição?

Imagine que você queira pedir para o computador escrever os números de 1 até 10.

Sem repetição, você teria que escrever:

```
print(1)
print(2)
print(3)
...
print(10)
```

Isso dá muito trabalho, não é?

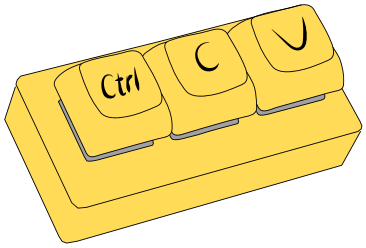


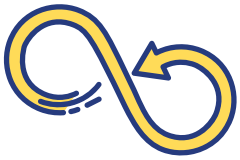
Com um laço de repetição, você pode dizer apenas:

"Repita até chegar ao número 10!"

Assim, o computador entende que deve fazer várias vezes a mesma coisa.

Não é incomum nos código termos que repetir muitas vezes a mesma parte do código, entretanto copiar e colar a mesma função várias e várias vezes além de não ser nenhum pouco funcional pode gerar muitos bugs.



Uma forma de resolver isso foi a criação dos laços de repetição, que são comandos utilizamos para repetir quantas vezes precisarmos de uma certa parte do código.

Existem laços que repetem uma tarefa um número exato de vezes e outros que continuam trabalhando enquanto uma condição for verdadeira.

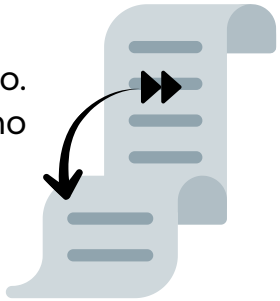
## Atenção!

Os dois laços de repetição ensinados são comuns nas outras linguagens também, entretanto cada linguagem tem sua própria sintaxe!

Há duas palavras reservadas que utilizamos para o controle de fluxo das estruturas condicionais, são elas **break** e **continue**.

O **break** serve para parar o laço de repetição.

Por exemplo, imagine que você está contando de 1 até 100, mas quer parar no número 20. O break faz isso.



Já o **continue** serve para pular uma parte e continuar o laço. É bem útil em casos que tem muitas condições em um mesmo loop, melhorando o tempo de resposta do programa.

Por exemplo, você está contando de 1 até 10, mas não quer mostrar o número 5. O continue ignora o 5 e segue para o próximo número.

## O laço while

O comando **while** é uma palavra inglesa que significa "enquanto", e é uma ótima forma de decorar o que esta função faz, visto que enquanto tal condição não for verdade, o loop continuará a ocorrer.

```
tempo = input("digite o tempo: ")
while tempo == "chuva":
  print("Use o guarda-chuva")
  tempo = input("digite o tempo: ")
```

Esse código vai escrever na tela **"Use o guarda-chuva"**, até que outra palavra seja atribuída à variável tempo que não seja "chuva" .

# 9 O laço while true

9.3.

Outra maneira de utilizarmos o while, é com while true, em que ao invés de colocarmos uma variável que define quando o loop finaliza ou inicia, ele sempre será verdade. Por isso ao utilizarmos o while true é indispensável a aplicação das estruturas condicionais!

**Mas como finalizamos o laço então?**

A resposta é mais simples do que aparenta, utilizaremos a função break, então sendo verdadeira a condição na qual o break está sendo solicitado a execução do loop é finalizada.

Veja o exemplo abaixo:

```
print("1-sim --- 2-não")
while True:
  menu = int(input("Deseja continuar com a previsão do tempo?"))
  if menu == 1:
    tempo = input("Digite o tempo:")
    if tempo == "chuva":
      print("leve um guarda-chuva")
    elif tempo == "sol":
      print("aproveite seu dia!")
  elif menu == 2:
    print("até a proxima!")
    break
  else:
    print("digite um opção válida")
```

```
1-sim --- 2-não
Deseja continuar com a previsão do tempo?1
Digite o tempo:chuva
leve um guarda-chuva
Deseja continuar com a previsão do tempo?1
Digite o tempo:sol
aproveite seu dia!
Deseja continuar com a previsão do tempo?3
digite um opção válida
Deseja continuar com a previsão do tempo?2
até a proxima!
```

O código acima é um menu de previsão do tempo.
Ele fica repetindo a pergunta:

- Quer continuar? (1 = sim, 2 = não).
- Se você disser sim, ele pergunta se está sol ou chuva.
- Ele dá um conselho para cada caso.
- Se você disser não, o código termina.
- Se você digitar algo errado, ele pede para corrigir.

Fácil, né?

# 9 O laço for

9.4

O laço for é como dizer:

"Para cada coisa dentro de uma lista, faça tal ação."

Pense em uma fila de alunos na escola: o for vai chamar um por um e dizer o nome deles. O for é um dos laços de repetição mais utilizados, porém sua construção em Python é bem diferente em relação às outras linguagens.

Observe o código ao lado:

```
numeros = [1 , 2 , 3 , 4 , 5]

for sequencia in numeros:
  print(sequencia)
```

**Entenda:**

- Nele definimos a variável números para armazenar a lista de números;

```
numeros = [1 , 2 , 3 , 4 , 5]
```

- Depois usamos o comando **for**, que diz ao computador:
  "Pegue cada número dessa lista, um por um, e faça algo com ele."
- A palavra sequência é como uma etiqueta temporária. A cada volta do loop, ela recebe o próximo número da lista.
- A palavra in significa "dentro de". Ou seja: pegue cada item dentro da lista de números.
- Por fim, usamos o print(sequência) para mostrar cada número na tela.

**Resultado:**

```
1
2
3
4
5
```

### Dica importante:

É muito comum usar for junto com condições (if e else). Assim podemos criar verificações, por exemplo, para saber se um número é par ou ímpar.

Neste exemplo, o computador olha cada número da lista e decide:

- Se é par, mostra uma mensagem.
- Se é ímpar, mostra outra.

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8]

for numero in numeros:
    if numero % 2 == 0:
        print(f"O número {numero} é PAR.")
    else:
        print(f"O número {numero} é ÍMPAR.")
```

**Resultado:**

O número 1 é ÍMPAR.
O número 2 é PAR.
O número 3 é ÍMPAR.
O número 4 é PAR.
O número 5 é ÍMPAR.
O número 6 é PAR.

## Hora do Desafio:

Desenvolva um código com uma lista de nomes: “Ana”, “Lucas”, “Maria”, e “Pedro”. utilizando a estrutura de repetição ***for*** mostre na tela cada nome identificando quem é líder da turma e quem apenas faz parte do grupo

### Lembre-se:

O **for** não serve só para repetir coisas, mas também para **verificar condições** e dar respostas diferentes dependendo da situação.

**Resultado:**

```
nomes = ["Ana", "Lucas", "Maria", "Pedro"]

for nome in nomes:
    if nome == "Maria":
        print(f"{nome} é a líder do grupo.")
    else:
        print(f"{nome} faz parte do grupo.")
```

```
Ana faz parte do grupo.
Lucas faz parte do grupo.
Maria é a líder do grupo.
Pedro faz parte do grupo.
```


## Função range( )

A função **range()** é usada quando queremos **definir exatamente quantas vezes** o programa deve repetir uma ação.

Está função é utilizada para colocar uma quantidade finita de repetições, então, ao invés de utilizarmos listas para ter uma determinada quantidade de repetições do loop, nós podemos definir a quantidade de vezes.

**Dica importante:**

Você pode trocar o número dentro do range() para repetir mais ou menos vezes.

- **range(3)** → repete 3 vezes.
- **range(10)** → repete 10 vezes.

**Exemplo:**

```
quantidade = input("Digite cinco alimentos: ")
for i in range(5):
  alimento = input("Alimento: " )
  print(alimento)
```

**Entenda o código acima:**

- O computador começa pedindo: “Digite cinco alimentos.”
  (Isso é só para o usuário saber que precisa escrever cinco vezes).
- Depois usamos: for i in range(5):
  Isso quer dizer: repita 5 vezes.
  Na 1ª vez, o computador pede um alimento.
  Na 2ª vez, pede outro alimento.
  E assim por diante, até completar as 5 repetições.
- Dentro do laço, temos:
  alimento = input("Alimento: ")
  print(alimento)
- Aqui o usuário digita um alimento, e o computador mostra o que foi digitado.

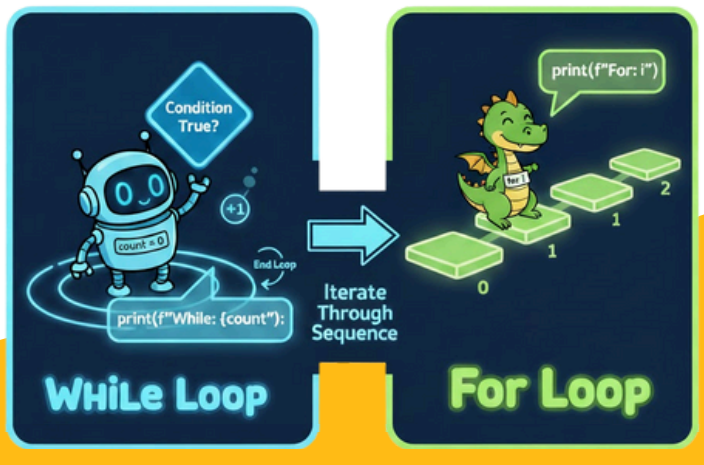
**Resultado:**

```
Digite cinco alimentos: manga
Alimento: arroz
arroz
Alimento: feijão
feijão
Alimento:
```

9.6

## While VS For – quando usar ?

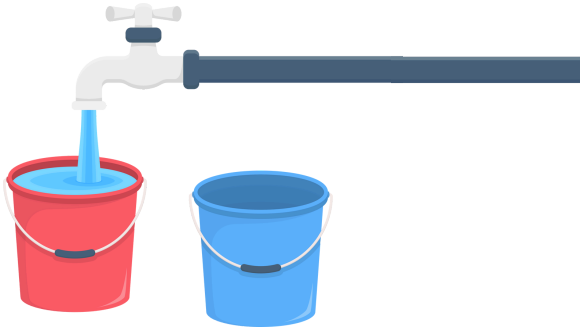
Mesmo sendo similar, cada laço de repetição se encaixa melhor em situações diferentes.




**Descrição:** A imagem possui um fundo azul escuro com dois retângulos e compara o “While Loop” e o “For Loop”. À esquerda há um robô azul com uma expressão amigável segurando uma placa com a condição “Condition True?”, abaixo dele, há um contador rotulado como “count = ...”, o robô está acompanhado pela instrução de código print(f"While: {count}") destacada no chão. Uma pequena seta mostra o incremento do contador (+1) e outra indica o ponto de saída do laço (“End Loop”). Ao centro, uma seta azul aponta para a direita e contém o texto: “Iterate Through Sequence” (Itera através de uma sequência), sinalizando a transição entre os dois tipos de laço. À direita, um simpático dragão verde caminha sobre uma sequência de blocos numerados (0, 1, 2), cada um simbolizando um passo da iteração. Na fala do dragão mostra print(f"For: i").

O ***while*** é geralmente utilizado quando não se sabe exatamente quantas vezes será necessário repetir o loop, mas conhecemos a condição de parada.

Por exemplo, quando temos que encher um balde de água, continuamos enchendo o balde enquanto ele não estiver cheio. A repetição para no momento em que a condição "o balde não está cheio" se torna falsa.

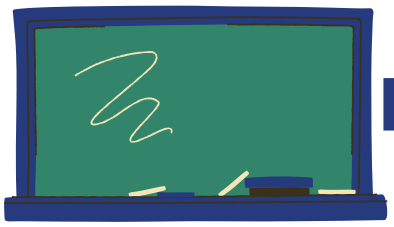


Já o for é quando sabemos quantas vezes será preciso reproduzir a ação ou quando precisamos percorrer uma sequência pré-definida de itens.

Por exemplo, quando queremos checar a lista de compras, há uma quantidade definida de itens e você fará o mesmo caminho: para cada item na sua lista de compras, procure-o na prateleira e coloque no carrinho. Ao finalizar todos os itens se encerra a lista.

# Atividades do capítulo

1. Por que utilizamos as estruturas de repetição?
2. Dê dois exemplos utilizando o while?
3. Qual a diferença do for com e sem o range?
4. Qual o melhor momento para utilizar o while? E o For?
5. *Crie 4 exemplos utilizados as 4 maneiras de operar os laços de repetição.*



# Professor em Ação

## Objetivos de aprendizagem:

O que são estruturas de repetição, quais os principais loops utilizados em Python e como criá-los.

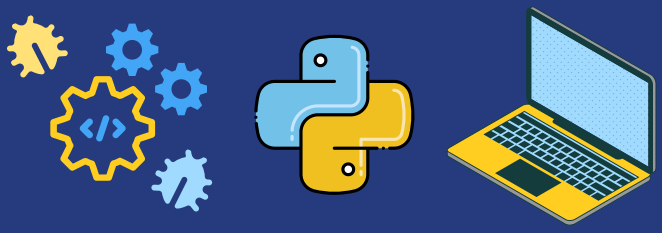
## Roteiro sugerido (20 min):

**1–5 min:** Apresentar as estruturas de repetição e seu funcionamento.
**6–10 min:** Como criar um while e seu irmão while True.
**11–15 min:** Como criar um for e utilizar a função range( ).
16**-20 min:** Explicar as diferenças entre o while e o for.

## Avaliação simples:

**Iniciante =** Crie uma lista com 4 números e dentro de um laço de repetição leia os números e some-os.
**Médio =** Crie um programa capaz de resolver um número fatorial.
**Avançado =** Faça um programa que recebe dois números e mostra todos os números que estão entre eles, além de somar todos os números do intervalo.

# Capítulo 10: Tratamento de Erros



# Objetivo

Quanto mais extensos nossos códigos ficam, mais comum se tornam os erros, entretanto existem alguns que não são apenas erros de sintaxe, neste capítulo vamos aprender como lidar com eles.

# Você vai aprender

- O que são as exceptions.
- Como lidar com os erros.
- Funções complementares.
- Criando exceções.

# 10 O que são as exceptions?

10.1.

Não é incomum, ao executar o nosso código, aparecem avisos de erros na execução do programa ou exceções que levam a um comportamento inesperado ou ao interrompimento do código, isso são as **exceptions**, e quer dizer que um problema impede nosso programa de concluir a tarefa para qual foi programado.

É comum o compilador ou interpretador mostrar para nós o tipo de erro ou exceção. Alguns erros comuns são:

- **VallueError**: erro de valor, ocorre quando o valor que desejamos e o valor que recebemos são diferentes;
- **ZeroDivisionError**: erro de divisão por zero, ocorre quando tentam dividir um número por zero;
- **IndexError**: erro de índice, ocorre quando tentamos acessar um index inexistente em alguma variável;
- **TypeError**: erro de tipo, ocorre quando tentamos fazer operações com tipos não compatíveis de variáveis;
- **KeyError**: erro de chave, ocorre quando tentamos acessar uma chave inexistente de um dicionário.

Outro erro muito comum mas que abrange vários outros é o **SyntaxErrors** (erros de sintaxe), ou seja quando erramos coisas como:

- esquecemos de fechar aspas, parênteses, chaves;
- não indentamos corretamente;
- não utilizamos dois pontos para indicar função, repetição ou condição;
- entre outros.

# 10 Como lidar com os erros

10.2

Para que, cada vez que um erro ou exceção apareça em nosso código, não fosse necessário ter que interromper a execução e nos mostrar o aviso de erro, foi criado um bloco de código capaz de seguir opções alternativas.

A estrutura é semelhante às condicionais, entretanto, ao invés de utilizarmos o if e o else, usamos o try e o except. O try é a tentativa do código, se não houver erros o código continua sendo traduzido, mas caso haja erro o except entra em ação.

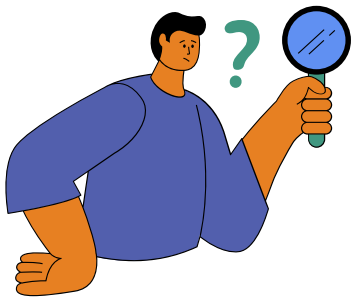
E então o código fica assim:

```
#tente:
try:
  num = int(input("Digite um número inteiro: "))
  print(f"O número digitado foi {num}")
# caso ocorra erro, faça:
except ValueError:
  print("O que foi digitado não é um número inteiro")
```

Então, caso o número digitado pelo usuário seja um inteiro, o programa irá mostrá-lo ao usuário, caso seja de outro tipo, será executado o segundo bloco do código, ou seja o except.

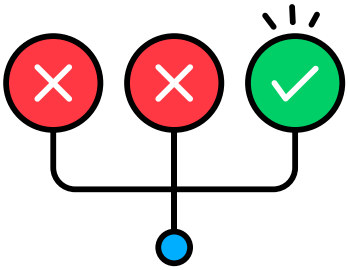
Pode ocorrer também a intervenção de mais de um tipo de erro no mesmo bloco, para isso existem as capturas de múltiplas exceções, podendo ocorrer de duas formas:

- Quando queremos tratar mais de um erro da mesma forma:



```
try:
    dividendo = int(input("Digite o dividendo: "))
    divisor = int(input("Digite o divisor: "))
    resultado = dividendo/divisor

except(ZeroDivisionError, ValueError):
    print("Número inválido ou divisor igual à zero")
```

- quando queremos tratar diferentes erros e diferentes formas:



```
try:
    dividendo = int(input("Digite o dividendo: "))
    divisor = int(input("Digite o divisor: "))
    resultado = dividendo/divisor

except ValueError:
    print("Numero digitado inválido")

except ZeroDivisionError:
    print("Divisão por zero não permitida")
```


## Funções complementares

Podemos também utilizar outras palavras chaves com o try e o except, um exemplo é o pass, que ao ser utilizado com o except podemos apenas ignorar o erro e pular para o próximo bloco de código.


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

try:
    indice = lista.index(5)
    encontrado = True

except ValueError:
    # O 'ValueError' acontece porque 5 não está na lista.
    pass
```

Neste caso, ao executarmos o código, caso alguma das exceções queira interromper o funcionamento, o programa apenas pulará o código e não mostrará nenhum resultado. Porém devemos tomar cuidado ao utilizar o pass, pois ele esconde bugs e torna o código muito difícil de depurar, por isso deve ser usado com cautela.

Outro bloco que podemos usar para complementar o tratamento de erros é o ***else***, após o *try* ser realizado e concluído **sem levantar nenhuma exceção,** ele realizará o que se encontra dentro do bloco do else.


```
try:
    idade = int(input("Digite sua idade: "))

except ValueError:
    print("Isso não é um número válido!")

else:
    print(f"Sucesso! Você tem {idade} anos.")
```

Há também outra palavra-chave que podemos utilizar para mostrar uma frase padrão no final do código, ela aparece independente do que ocorra no try e except, seu nome é finally. Ela é principalmente utilizada para ações de limpeza que ocorrerão no código.


```
try:
    print("Entrei no quarto e acendi a luz.")

    print("Achei minhas chaves.")

except Exception:
    print("caso houvesse execessão seria tratada aqui!")

finally:
    # Roda SEMPRE, não importa o que aconteceu
    print("Saindo do quarto e apagando a luz.")
```

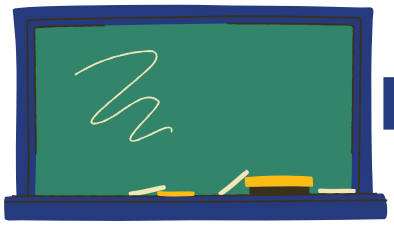

# Criando exceções

Às vezes será necessário criar alertas para o nosso código. Para isso, podemos utilizar as estruturas condicionais e criar mais de um caminho, mas conseguimos utilizar também a estrutura do try e except para nos ajudar com isso. É possível ainda utilizarmos a palavra reservada raise.

```
try:
    idade = int(input("Digite sua idade: "))
    print("\n")
    print("Verificando idade...")

    if idade < 18:
        raise ValueError("Idade inválida! Não pode menor de idade.")

    # Esta linha só roda se a exceção (raise) não for verdade
    print(f"Bem-vindo! Você tem {idade} anos.")

except ValueError as e:
    print(f"ENTRADA BARRADA! Motivo: {e}")
```

# Atividades do capítulo

1. De acordo com o texto o que é uma exceção?
2. Qual a principal diferença entre um SyntaxError (Erro de Sintaxe) e uma exceção como por exemplo o ValueError?
3. O que acontece no bloco try e o que acontece no bloco except?
4. Para que serve a palavra-chave raise?



# Professor em Ação

### Objetivos de aprendizagem:

Aprender os tipos de erros e exceções e como contorná-las, além da utilização das funções complementares e da criação de exceções.

### Roteiro sugerido (10 min):

**1–5 min:** Ensinar sobre os erros e exceções que podem aparecer ao decorrer do código.
**6–10 min:** Explicar como criar uma forma de contornar esses problemas para não interromper a execução do código.
**11-15 min:** Mostras as palavras-chaves que podemos utilizar para complementar o try e except e como cada uma funciona.
**16-20 min:** Explicar sobre a criação de exceções e como criá-las.

### Avaliação simples:

**Iniciante =** Se um bloco try for executado com sucesso, quais são os dois outros blocos opcionais que podem ser executados e em que ordem eles aparecem?
**Médio =** Crie um programa que use try, except, else e finally para pedir dois números, tentar dividir um pelo outro e mostrar uma mensagem final, mesmo que ocorra erro.
**Avançado =** Criar um programa que consulta o preço de um item em um cardápio (dicionário), tratando o erro que ocorre se o cliente pedir um item que não existe.

# Capítulo 11: Funções

# Objetivo

Durante a codificação é normal utilizarmos estruturas iguais em partes totalmente diferentes do código, por isso há uma maneira de ter uma parte do código que seja reutilizável!

Hoje vamos aprender o que são esses blocos reutilizáveis, como criar e como utilizá-los ao decorrer do programa.

# Você vai aprender

- O que são funções.
- Como criamos uma função?.
- O que são os parâmetros.
- Variáveis locais e globais.
- Recursividade.


# O que são as funções e por que utilizá-las

Podemos imaginar as funções como blocos de lego onde podemos encaixá-los onde precisarmos e mais de uma vez caso necessário.

Ao longo do livro a palavra “função” já foi citado muitas vezes, mas o que ela realmente faz?

Elas são criadas para modificar ou realizar uma ação no código, como, por exemplo no capítulo 5 onde apresentamos as funções das variáveis, elas são capazes de modificar as strings a partir do momento que chamadas. Muitas funções já são atribuídas à linguagem, mas cada programador pode criar suas próprias funções no decorrer do software.

**Temos três tipos básicos de função:**

**Funções nativas:**
São funções que já estão atreladas a linguagem Python e estão sempre disponíveis, não sendo necessário importá-las.

**Funções de biblioteca:**
São funções que estão guardadas dentro de outros módulos do Python podendo ser chamadas através das bibliotecas para utilizarmos em nosso código.

**Funções personalizadas:**
São funções que nós enquanto programadores podemos criar e definir para realizar tarefas específicas do nosso programa.

```
import math #importação da biblioteca
            #para utilizar a função

nome.upper() #função nativa
math.pi() #função importada
def saudacao(): #função personalizada
```

# 11 Como criamos uma função?

11.2.

Antes de iniciarmos a criação das funções temos que entender algumas características das funções:


- Todos possuem um identificador;
- Possuem parâmetros;
- Armazenam seus parâmetros nos ( );
- Podem ou não retornar um valor;

Ao criarmos uma função devemos seguir um padrão, iniciamos com a palavra reservada **def** seguido pelo nome da função e os parênteses.

Os **parâmetros de entrada**, ou seja, os dados que fornecemos, devem ser colocados entre parênteses que ficam no final da função().

Para marcar a finalização do cabeçalho da função e o início do corpo da mesma, utilizamos **dois pontos,** conforme a imagem a seguir:

```
nome =input("digite seu nome: ")

def saudacao(): #função
  print(f"Seja bem vindo {nome}!") #corpo da função

saudacao() #chamando a função
```

```
digite seu nome: Manu
Seja bem vindo Manu!
```

# 11 Parâmetros

11.3.

Os parâmetros são as informações que a função recebe para poder realizar uma ação. É como se as funções fossem uma máquina de fazer suco e os parâmetros das frutas.

Eles tornam as funções flexíveis e reutilizáveis, sem os parâmetros, cada vez que fossemos trocar a fruta do suco precisaríamos criar uma máquina nova. E também podemos utilizar mais de um parâmetro por função.

Utilizando funções é comum escutarmos sobre os argumentos, então qual seria a diferença dos parâmetros e argumentos?

Os parâmetros são as informações que a função deseja receber, já os argumentos são as informações recebidas pela função.

Toda função pode receber parâmetros ou argumentos, mas não significa que eles sempre serão obrigatórios.


## Variáveis globais e locais

A principal diferença entre elas é sua visibilidade.

As variáveis globais são criadas fora do corpo de qualquer função ou estrutura e podem ser chamadas ao decorrer de qualquer parte do código.

Já as variáveis locais são criadas dentro de funções ou estruturas e só podem ser chamadas dentro de onde foi criada. Normalmente por que só serão utilizadas naquela função liberando espaço e melhorando a sintaxe do código, conforme a imagem:

```
nome = "manu" # variável global

def cabecalho():
  saudacao = "olá" # variável local
  print(f"{saudacao}, seja bem-vinda {nome}")

cabecalho()

print(nome)
print(saudacao) #erro
```

```
olá, seja bem-vinda manu
manu
<function saudacao at 0x7c55ff784180>
```

11

11.5.

# O que é recursividade?

Recursividade é a ação de uma função chamar a si mesma durante sua execução. Podem simplificar grandes problemas, dividindo-os em problemas menores.

**Pense em bonecas russas**: para encontrar a boneca pequena, você precisa abrir uma versão um pouco maior dela. Você repete esse processo até encontrar a última boneca, a menor de todas, que não abre mais.

```python
def fatorial(n):
    # Isso impede que a função se chame infinitamente.
    if n == 1 or n == 0:
        print(f"Caso base atingido: fatorial({n}) = 1")
        return 1
    else:
        print(f"Chamando recursivamente: {n} * fatorial({n-1})")
        return n * fatorial(n - 1)

resultado = fatorial(4)
print("\n")
print(f"O resultado final de fatorial(4) é: {resultado}")
```

**Resultado:**

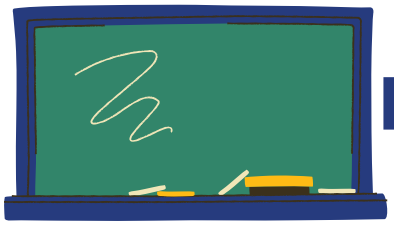
```
Chamando recursivamente: 4 * fatorial(3)
Chamando recursivamente: 3 * fatorial(2)
Chamando recursivamente: 2 * fatorial(1)
Caso base atingido: fatorial(1) = 1


O resultado final de fatorial(4) é: 24
```

## Atenção!

A recursividade é eficiente em alguns cenários, mas caso não seja utilizada com moderação pode resultar em falta de memória para o processamento do programa!

# Atividades do capítulo

1. Qual a finalidade das funções?
2. Quais as diferenças entre variáveis locais e globais?
3. Explique com suas palavras o que é recursividade.
4. Quais as características de uma função?



# Professor em Ação

### Objetivos de aprendizagem:

Entender o que são as funções, suas características, como criá-las e por que utilizá-las.

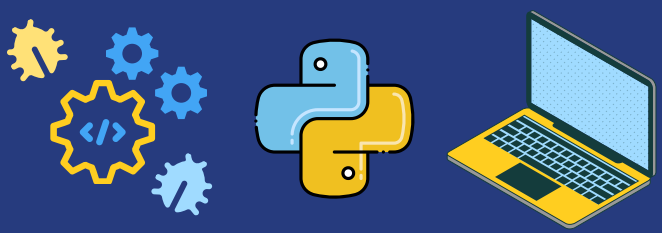
### Roteiro sugerido (10 min):

**1–5 min:** Ensinar sobre o que são as funções.
**6–10 min:** Explicar como criar uma função.
**11-15 min:** Esclarecer o que são parâmetros e as diferenças entre variáveis locais e globais.
**16–20 min:** Apresentar a recursividade.

### Avaliação simples:

**Iniciante =** Crie uma código com uma função capaz de retornar a média de um aluno.
**Médio =** Peça ao usuário para criar uma senha de até 8 caracteres e verifique com uma função se a senha é válida.
**Avançado =** Crie uma função recursiva capaz de mostrar a Sequência de Fibonacci. A sequência de Fibonacci é uma série de números onde cada número é a soma dos dois anteriores.

# Capítulo 12: As Bibliotecas em Python



# Objetivo

Como citado no capítulo anterior, é comum que tenhamos a necessidade de importar funções de outras coleções, elas são conhecidas na comunidade de programação como bibliotecas.

Neste capítulo iremos apresentar as bibliotecas a vocês e como utilizá-las

# Você vai aprender

- O que são as bibliotecas.
- Como utilizá-la.
- Principais bibliotecas.

12

12.1.

# O que são as bibliotecas?

As bibliotecas são agrupamentos de funções e objetos pré-escritos que podem ser reutilizados em vários projetos, facilitando a realização de tarefas complexas.

Elas foram desenvolvidas para ajudar a resolver problemas específicos e oferecer recursos adicionais que não fazem parte do núcleo da linguagem.

## Para que servem as bibliotecas?

Pode até parecer boba a utilização das bibliotecas, mas elas otimizam muito a nossa produção. As principais vantagens da utilização das bibliotecas são:

**Melhoria do desempenho:**

Permitindo que foquemos na criação do código e em resolver os bugs ao invés de desaproveitar o tempo em criação de funções.

**Maior organização do código:**

Como elas ficam armazenadas em outro local, ao implementarmos as bibliotecas não teremos mais linhas e linhas de código com funções adjunto ao código principal, tornando a leitura mais fluida e nos permitindo ter maior visibilidade.

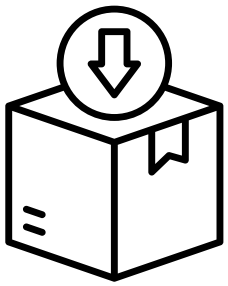
**Facilidade de inserção:**

As bibliotecas foram criadas para serem de fácil integração com os mais diversos códigos, tornando simples sua utilização.

**Ferramentas avançadas :**

As bibliotecas possuem uma ampla coleção de funcionalidades que podemos utilizar para facilitar a resolução de problemas.

## A biblioteca Padrão



A biblioteca Padrão é uma coleção de módulos e pacotes que são instalados automaticamente junto com o interpretador Python. Isso significa que, sem precisar executar nenhum comando de instalação, você já tem acesso a funcionalidades robustas e eficientes para realizar uma imensa variedade de tarefas.

Alguns exemplos são as funções como .len, .cout, .range e muitas outras funções que foram apresentadas no decorrer do livro sem termos que fazer nenhuma implementação pois elas fazem parte da biblioteca padrão do Python.


# Como instalar e utilizar as bibliotecas

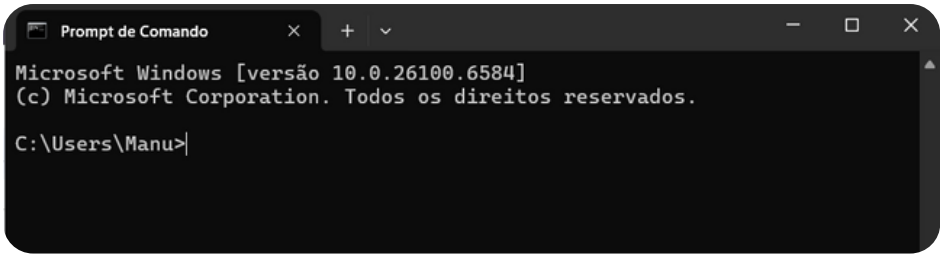
Agora que já sabemos o que são e utilidades das bibliotecas, vamos dar um upgrade no nosso ambiente de trabalho para que possamos usar as bibliotecas.

## Como instalamos ?

No **Google Colab** já temos um ambiente integrado com as bibliotecas.

Entretanto, nas outras IDE's precisamos instalar o pacote que nos permite acessá-las antes de conseguirmos utilizá-las efetivamente.

Geralmente a instalação é feita através do **terminal** ou **prompt de comando (cmd)** e utilizando um gerenciador de pacotes chamado **pip**.



Esta é a interface do cmd, dependendo do sistema operacional do seu computador alguns comandos básicos serão diferentes, mas os únicos comandos que precisaremos para instalar as bibliotecas serão o **cd (change directory)** e o **install**.

Escreveremos o comando: **Python -m pip install**

```
Usage:
  C:\Users\Manu\AppData\Local\Microsoft\WindowsApps\PythonSoftwareFoundat
ion.Python.3.11_qbz5n2kfra8p0\python.exe -m pip <command> [options]
```

Talvez apareça um aviso informando que não é possível fazer a alteração sem estrar na pasta base do arquivo. Então, precisamos navegar até o local que desejamos alterar.

Ali já nos foi dado o caminho que precisamos chegar, atualmente estamos na pasta \Users\Manu e precisamos percorrer um longo caminho. E para adentrar em algum diretório (pasta) usamos o comando **cd**.

```
Usage:
  C:\Users\Manu\AppData\Local\Microsoft\WindowsApps\PythonSoftwareFoundat
ion.Python.3.11_qbz5n2kfra8p0\python.exe -m pip <command> [options]

no such option: --upgrade

C:\Users\Manu>cd AppData

C:\Users\Manu\AppData>
```

Agora já estamos dentro da pasta AppData, e assim continuaremos até o local necessário.

Ao chegarmos no último diretório escreveremos o código **\Python.exe -m pip install**.

```
C:\Users\Manu\AppData\Local\Microsoft\WindowsApps>cd PythonSoftwareFounda
tion.Python.3.11_qbz5n2kfra8p0

C:\Users\Manu\AppData\Local\Microsoft\WindowsApps\PythonSoftwareFoundatio
n.Python.3.11_qbz5n2kfra8p0>
C:\Users\Manu\AppData\Local\Microsoft\WindowsApps\PythonSoftwareFoundatio
n.Python.3.11_qbz5n2kfra8p0>\python.exe -m pip install
```

E assim teremos instalado no nosso computador o necessário para utilizarmos as bibliotecas.

## Importando as bibliotecas

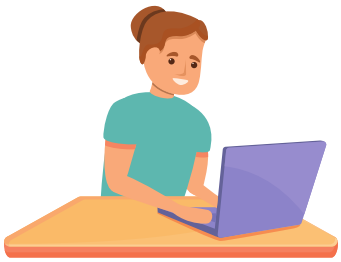
Agora que já temos elas instaladas em nossa IDE podemos utilizá-las. Fazemos isso de uma forma muito simples, apenas escrevemos import e o nome da biblioteca que queremos utilizar.

```
import math
```

Há uma maneira de renomearmos a biblioteca que vamos utilizar, simplificando mais ainda quando formos chamá-las, na sequência do nome da biblioteca escrevemos as (do inglês como) e o nome que desejamos:

```
import math as m
```

A partir deste comando, a biblioteca já disponível para ser utilizada no seu código:



```
import math as m

print(m.pi)
```

```
3.141592653589793
```

### Não esqueça!

Sempre que for utilizar alguma função da biblioteca é necessário chamar a biblioteca como no exemplo anterior, em que antes de utilizar a função pi foi colocado m. Se tentarmos utilizar a função diretamente, sem a chamar a biblioteca, ocorrerá um erro, conforme podemos observar na imagem a seguir

```
import math as m

print(pi)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipython-input-3952984989.py in <cell line: 0>()
      1 import math as m
      2
----> 3 print(pi)

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

# 12 Quais as principais bibliotecas

12.3.

Existem várias bibliotecas que podem ser importadas, às vezes será necessário fazer novamente a instalação com o **pip install**, mas as principais e mais utilizadas já estão disponíveis para você!

As bibliotecas podem ser utilizadas nas mais diversas aplicações como:

- Criação de sites e APIs
- Ciências de dados
- Visualização de dados
- Inteligência artificial
- Automações
- Criação de interfaces gráficas
- E muitos outros

Sem contar que cada biblioteca possui várias funções em seu escopo, nos permitindo ter uma quantidade incrível de ferramentas.

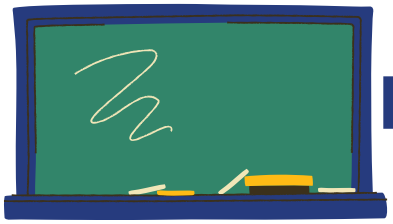
Algumas das bibliotecas preferidas dos desenvolvedores são:

- **Numpy** (Numerical Python) - é uma biblioteca de manipulação de dados utilizada para cálculos numéricos e científicos.
- **Pandas** - é uma biblioteca de estrutura de dados que oferece uma grande variedade de ferramentas para análise de dados.
- **Requests** - é a biblioteca capaz de realizar requisições em HTTP, permitindo uma fácil comunicação entre APIs e sites.
- **Matplotlib** - é uma das principais bibliotecas de visualização de dados em Python.
- **Django** e **Flask** - são duas bibliotecas que permitem a construção de aplicações web, a escolha entre eles depende muito das preferências do programador e de onde será aplicado.

A linguagem Python é bem ampla e colaborativa, com o Python Package Index (PyPI) como seu principal repositório. Sempre atualizado com a ajuda dos diversos desenvolvedores do Python, o PyPI já oferece mais de 500 mil bibliotecas, e esse número continua aumentando.

# Atividades do capítulo

1. O que são as bibliotecas?
2. Por que utilizar bibliotecas no seu código?
3. Explique com suas palavras como instalar bibliotecas em outras IDE.
4. Escolha duas bibliotecas das que foram apresentadas e escreva por que elas chamaram sua atenção.



# Professor em Ação

### Objetivos de aprendizagem:

Aprender sobre as bibliotecas como instala-las e utilizá-las.

### Roteiro sugerido (20 min):

**1–5 min:** O que são as bibliotecas e a biblioteca padrão.
**6–15 min:** Como podemos instalar as bibliotecas em outras IDEs e como utilizá-las.
**16–20 min:** Apresentar algumas bibliotecas.

### Avaliação simples:

**Iniciante =** O que é uma biblioteca em Python e por que ela é importante?.
**Médio =** Explique por que usamos a biblioteca random e cite um exemplo de situação em que ela seria útil.
**Avançado =** Use a biblioteca datetime para mostrar a data e hora de hoje, junto com uma mensagem divertida.(Exemplo: “Hoje é 31/10/2025 — Dia de praticar Python!”).

# Capítulo 13:

# Programação Orientada a Objeto

# Objetivo

Até aqui nosso códigos são listas de instruções que o computador segue. Entretanto, para resolver grandes problemas, essa forma de sintaxe pode tornar mais difícil a manutenção do código. Por esse motivo, utilizamos a programação orientada a objetos (POO), que é uma nova forma de estruturar o código.

# Você vai aprender

- O que é POO?.
- O que são e como utilizar as classes.
- O que são e como utilizar os objetos.
- O que são os atributos.
- O que são os métodos.
- Quais as principais aplicações do POO.

## Mas afinal...

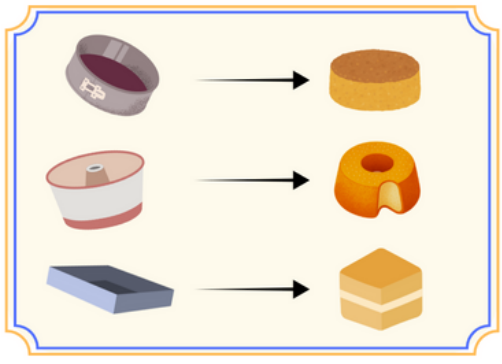
# 13.1. O que é a orientação a objetos?


Como citado na introdução deste capítulo, a programação orientada à objeto (POO) é uma nova forma de estruturar o nosso código. Em vez de organizar o código em torno das funções, como uma grande lista de tarefas - abordagem conhecida como programação procedural -, organizamos o código em torno dos objetos o que torna o programa mais intuitivo e organizado.

Ele pode ser dito também como uma forma de aproximar a elaboração do código ao mundo real, onde cada objeto possui duas partes principais: as características (atributos) e as ações (métodos), da mesma maneira que muitas coisas no nosso dia a dia.

**Descrição:** A imagem apresenta dois lados, no esquerdo mostra um conjunto de elementos mecânicos como engrenagens cinzas, caixas e blocos marrons e azuis que estão ligados por fios as várias caixas cinzas que estão alinhadas e rotuladas como "Tarefa 1", "Tarefa 2", "Tarefa 3" e "Tarefa 4". O caminho que conecta o topo das engrenagens até cada tarefa é longo e tortuoso. A legenda em tom azul escuro abaixo diz "PROGRAMAÇÃO PROCEDURAL – Lista de Tarefas". No lado direito é exibido um cenário lúdico semelhante a um "mundo em miniatura". Acima desse mundo, há uma nuvem azul com a palavra "CLASSE" em branco e embaixo a logo do Python. Setas partem da nuvem e apontam para três instâncias diferentes: carro (com atributos relacionados: Cor, Marca, Modelo, etc.), pássaro (com atributos como Velocidade, Altura, etc.) e casa (com atributos de Cor, Andares, e outras características). A legenda abaixo diz: "PROGRAMAÇÃO ORIENTADA A OBJETOS – Mundo de Objetos", em tom de azul escuro.

# 13.2. O que são as classes?



As classes são como moldes que usamos para criar objetos. Quando criamos uma classe, estamos definindo um tipo de dado especial e personalizado que podemos usar várias vezes no nosso código, tornando tudo mais organizado e fácil de entender.

Por exemplo, uma forma de bolo, todas podem conter o mesmo bolo, mas terão formatos diferentes, da mesma maneira que cada forma pode fazer bolos de diversos sabores. A forma em si não é o bolo, ela apenas define a estrutura do bolo, se será redondo, quadrado, retangular e assim por diante.

Nas classes também definimos os atributos (características) e os métodos (ações) que um objeto daquele tipo terá.

Para criar uma classe usamos a palavra reservada **class** seguida do nome que definimos para ela.

**Algumas convenções quanto ao nome da classe:**

- Começar com letra maiúscula
- Não possui espaço entre as palavras
- Camel Case (a primeira letra de cada palavra será maiúscula)

```
class MassaDoBolo:
  pass
```

**Atenção**

No exemplo acima utilizamos a palavra **pass** para não dar erro ao criar a classe sem nenhum atributo ou método.


## O que são os objetos?

Enquanto as classes são a ideia ou formato, os objetos são a figura real ou final das classes. Como citado anteriormente de uma mesma classe podemos criar vários objetos, ou seja, cada objeto é independente entre si. Eles vieram de um mesmo molde, mas o que acontece com um objeto não afeta os outros.

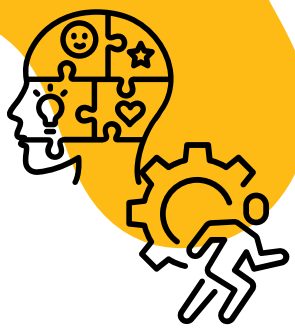
Para criar um objeto utilizamos um identificador e atribuímos a ele a classe que criamos anteriormente.

**Atenção!**

Os objetos são sempre criados fora da classe, dentro da classe ficam apenas os atributos e métodos dos objetos

```
class MassaDoBolo:
  pass

tipo1 = MassaDoBolo()
```

Como criamos o objeto



Cada objeto tem um estado e um comportamento:

- **Estado**
  São as características atuais do seu objeto, essas características são armazenadas nos atributos.
- **Comportamento**
  O que cada objeto pode fazer, são as ações que você permite ao objeto fazer e elas são armazenadas nos métodos.

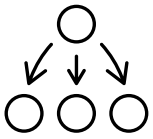
Ou seja, o objeto é uma entidade fundamental da POO. É a "coisa" real que guarda suas próprias informações (estado) e sabe como realizar suas próprias ações (comportamento).


# O que são os atributos?


Os atributos são variáveis pertencentes a classe que definirão as características ou o estado dos objetos criados a partir daquela classe.

```
class MassaDoBolo:
  ingredientes = ["ovos" , "farinha" , "açúcar" , "leite" , "azeite"]

tipo1 = MassaDoBolo()
```

Neste exemplo nós adicionamos o atributo ingredientes, ou seja todos os tipos de bolo criados a partir dessa classe terão esses atributos que foram pré-definidos.



Como esses atributos são variáveis, nós podemos alterá-los utilizando funções, como:

```
class MassaDoBolo:
  ingredientes = ["ovos" , "farinha" , "açúcar" , "leite" , "azeite"]

tipo1 = MassaDoBolo()
print(tipo1.ingredientes)
tipo1.ingredientes = ["ovos" , "farinha" , "açúcar" , "leite" , "manteiga"]
print(tipo1.ingredientes)

tipo2 = MassaDoBolo()
tipo2.ingredientes.append("chocolate")
print(tipo2.ingredientes)
```

No exemplo acima utilizamos fizemos uma substituição dos atributos por atribuição e também utilizando a função .append( ).

Existem dois tipos de atributos:

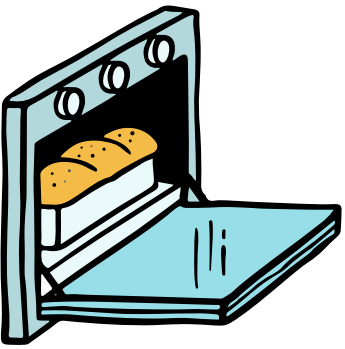
- **Os atributos de instância** - que são específicos de cada objeto.
- **Os atributos de classe** - que são compartilhados por todos os objetos (filhas) da classe.


## O que são os métodos ?


Os métodos são as ações que os objetos podem realizar, e da mesma forma que os atributos, eles são definidos na classe.

Uma de suas características é que os métodos podem ser sobrescritos em classes filhas, permitindo que cada uma tenha um comportamento personalizado para a mesma ação.

Para criar um método devemos iniciá-lo usando a palavra-chave def dentro da classe, incluindo self como o primeiro parâmetro, como no exemplo abaixo:



```
class MassaDoBolo:
  tipo = "massa"
  ingredientes = ["ovos" , "farinha" , "açúcar" , "leite" , "azeite"]
  def assar(self , temperatura):
    print(f"Bolo {self.tipo} será assado em {temperatura}")

tipo1 = MassaDoBolo()
tipo1.tipo = "padrão"

tipo1.assar("180º")
```

```
Bolo padrão será assado em 180º
```

A principal diferença entre uma função normal e um método é que como os métodos estão inclusos dentro da classe eles possuem acesso aos atributos, graças ao parâmetro self, que serve como ponte permitindo que o método acesse e manipule os atributos.

13.6

# Princípios da POO

Em orientação a objetos temos várias características que nos ajudam a ter um código mais polido, mas entre elas temos os quatro pilares da programação orientada a objeto que são:

## Encapsulamento


Ele pode agrupar os atributos e métodos dos objetos dentro de uma classe, mas seu principal objetivo é proteger o acesso e as modificações do objeto, permitindo ou não que os dados sejam alterados.

É como se criássemos uma hotel, o encapsulamento são os quartos, cada objeto tem o seu e nenhum dos objetos tem acesso ou pode fazer alterações de outro objeto, a não ser que chame a lista de métodos públicos, como se chamassem a classe para um encontro no saguão do hotel.

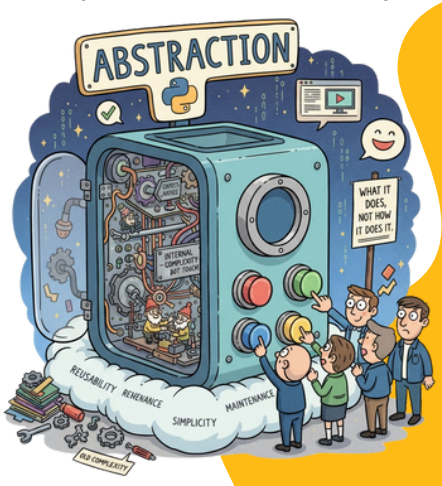
Ao criarmos uma classe, o padrão é ser o encapsulamento público que permite alterar os dados dentro e fora da classe.

Entretanto, como a principal função do encapsulamento é justamente a segurança, podemos criar um encapsulamento privado, no qual os dados que estão dentro dele não podem ser acessados fora da classe.

## Abstração

A abstração é responsável por simplificar as ações ao usuário, ela esconde as coisas complexas e mostra apenas aquilo que é realmente necessário. Isso torna o desenvolvimento mais fácil, além de ajudar na reutilização, manutenção e melhoria do programa. Em outras palavras, a abstração concentra-se no que um objeto faz, e não em como ele realiza essa ação, focando em esconder a complexidade para simplificar o uso do objeto.

Isso também diminui o 'impacto da mudança'. Ou seja, podemos otimizar ou alterar completamente o funcionamento interno de uma classe e, contanto que sua interface (seus 'métodos públicos') permaneça a mesma, o restante do código continua funcionando sem precisar de nenhuma modificação.

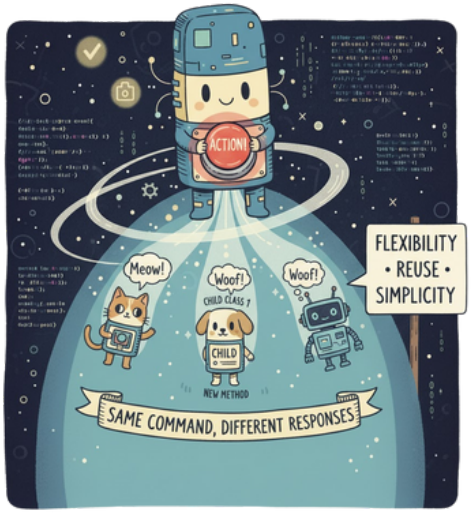


## Herança


De forma simples, a herança nos permite criar novas classes (chamada de filha ou subclasse) dependentes de classes já existentes (chamada de pai ou superclasse), definindo uma ordem entre as classes. A classe filha, criada a partir da classe pai, "herda" todos os seus atributos e métodos, e podemos adicionar coisas novas ou modificar o que foi herdado.

A herança nos permite reaproveitar o código, evitando que escrevemos o mesmo código várias vezes. Além de nos permitir ter uma maior organização e facilitando a manutenção, pois ao alterar a classe pai todos os seus filhos serão corrigidos automaticamente.

## Polimorfismo




O último pilar é a habilidade de um objeto assumir mais de uma forma. Isso é evidente quando os objetos de classes filhas derivadas de um mesmo pai, respondem a mesma ação (método), mas adaptado ao comportamento de cada subclasse. Ou seja, o polimorfismo permite que você use o mesmo comando para objetos diferentes, e cada um responderá de sua maneira única.

Essa característica torna o código mais flexível e genérico, ele nos permite interagir com diferentes objetos da mesma maneira através de uma interface comum. Em vez de escrever código repetitivo e complexo para tratar cada caso, podemos simplesmente contar com uma interface comum. Isso facilita a reutilização e diminui drasticamente o impacto de futuras mudanças no sistema.

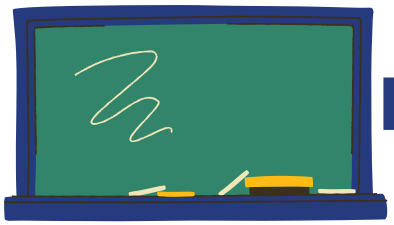
# Atividades do capítulo

1. O que você entendeu por Programação Orientada a Objeto?
2. O que são as classes e por que utilizamo

# Atividades do capítulo

3 Qual a utilidade dos objetos e sua ligação com o conceito das classes?

4 O que são os atributos e os métodos e qual a importância deles para este tipo de programação?



# Professor em Ação

### Objetivos de aprendizagem:

Ensinar a como programar no formato orientado a objeto.

### Roteiro sugerido (30 min):

**1–5 min:** O que é a programação orientada a objeto?.
**6–15 min:** O que são as classes e os objetos e qual a ligação desses dois conceitos.
**16–25 min:** Apresentar o que são e como utilizar os atributos e métodos.
**26-30 min:** Apresentar os quatro pilares da programação orientada a objeto.

### Avaliação simples:

**Iniciante =** O que é uma classe em Python e o que ela representa?.
**Médio =** Explique para que servem as classes e objetos, e cite um exemplo de algo do mundo real que poderia virar uma classe em Python.
**Avançado =** Crie duas classes: Cachorro e Gato. Cada uma deve ter um método falar() que mostre:
"Au au" e "miau" .

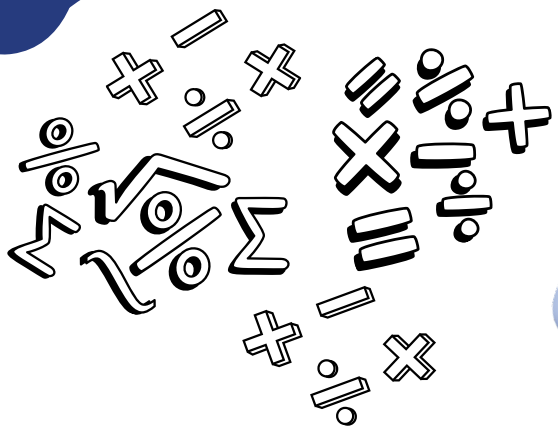
# Capítulo 14: Fazendo uma calculadora

## Objetivo

Utilizar tudo o que foi aprendido até o momento para criar uma calculadora em Python

## Você vai aprender

- Criando uma calculadora.



# 14

## 14.11 Enunciado

Agora que já temos um vasto conhecimento de Python vamos fazer um desafio.
Utilizando tudo que já foi ensinado crie uma calculadora que seja capaz de:

- somar.
- subtrair.
- multiplicar.
- dividir (mostrar o resto da divisão e o resultado ).
- potenciação.
- radiciação.
- comparar dois números ( se é maior, igual ou menor ).
- fatorial.

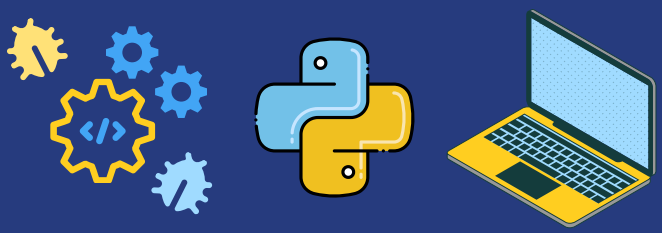

# Resposta

```
import math

def exibir_menu():
    print("\n--- CALCULADORA PYTHON ---")
    print("1. Somar")
    print("2. Subtrair")
    print("3. Multiplicar")
    print("4. Dividir (Resultado e Resto)")
    print("5. Potenciação")
    print("6. Radiciação (Raiz Quadrada)")
    print("7. Comparar dois números")
    print("8. Fatorial")
    print("0. Sair")

while True:
    exibir_menu()
    opcao = input("Escolha uma opção: ")

    if opcao == '0':
        print("Saindo da calculadora. Até mais!")
        break

# Lógica para operações que precisam de apenas 1 número
if opcao == '6': # Radiciação
    num = float(input("Digite o número para calcular a raiz quadrada: "))
    if num >= 0:
        print(f"A raiz quadrada de {num} é {math.sqrt(num):.2f}")
    else:
        print("Erro: Não existe raiz real de número negativo.")

elif opcao == '8': # Fatorial
    num = int(input("Digite um número inteiro para o fatorial: "))
    if num >= 0:
        print(f"O fatorial de {num} é {math.factorial(num)}")
    else:
        print("Erro: Fatorial apenas para números positivos.")
```


# Resposta

```
# Lógica para operações que precisam de 2 números
elif opcao in ['1', '2', '3', '4', '5', '7']:
    num1 = float(input("Digite o primeiro número: "))
    num2 = float(input("Digite o segundo número: "))

    if opcao == '1': # Somar
        print(f"Resultado: {num1} + {num2} = {num1 + num2}")

    elif opcao == '2': # Subtrair
        print(f"Resultado: {num1} - {num2} = {num1 - num2}")

    elif opcao == '3': # Multiplicar
        print(f"Resultado: {num1} * {num2} = {num1 * num2}")

    elif opcao == '4': # Dividir (Resto e Resultado)
        if num2 != 0:
            resultado = num1 // num2 # Divisão inteira
            resto = num1 % num2      # Resto da divisão
            print(f"Resultado inteiro: {resultado}")
            print(f"Resto da divisão: {resto}")
        else:
            print("Erro: Divisão por zero não é permitida.")

    elif opcao == '5': # Potenciação
        print(f"Resultado: {num1} elevado a {num2} = {num1 ** num2}")

    elif opcao == '7': # Comparar
        if num1 > num2:
            print(f"{num1} é MAIOR que {num2}")
        elif num1 < num2:
            print(f"{num1} é MENOR que {num2}")
        else:
            print(f"Os números são IGUAIS")

else:
    print("Opção inválida! Tente novamente.")
```

# Capítulo 15: Desafios Finais



# Objetivo

Utilizar tudo o que foi aprendido até o momento para resolver estes cinco desafios finais!

# Você vai aprender

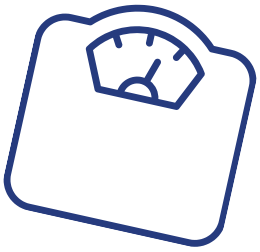
- Calculadora de IMC  (Índice de Massa Corporal.
- Conversor de unidades métricas.
- Agenda de aniversários.
- Registro de despesas.
- Conversor de tempo de estudo.

15.1.

## Calculadora de IMC (Índice de Massa Corporal)

Crie uma programa que seja capaz de imprimir o valor do IMC de uma pessoa, e mostre se ela está:

acima do peso (entre 25.0 a 29.9);
obeso ( a cima de 30.0);
abaixo do peso (menor que 18.5);
peso normal (18.5 a 24.9);



A fórmula é: IMC = Peso /(Altura x Altura).
**lembre-se** de pedir a altura e peso com o input.

15.2.

## Conversor de Unidades Métricas

Frequentemente necessitamos converter as unidades de medida, por isso vamos criar um programa que seja capaz de fazer isso para nós.

Crie um conversor:

- de km para m e vice versa
- de m para cm e vice versa
- de km para cm e vice versa

**Dica**: crie um menu e utilize funções;

15.3

## Agenda de Aniversários

Vamos criar um programa que guarda as datas de aniversário de amigos e familiares. Para isso vamos utilizar os dicionários e listas. Você pode criá-los vazio ou com alguns aniversários.

O programa deve ser capaz de:

- imprimir as datas contidas no dicionário;
- adicionar mais datas de aniversário.

15.4

## Registro de Despesas

No dia a dia é comum ocorrer muitos gastos, por isso vamos criar um programa em que possamos anotar e somar as despesas:

O programa deve:

- realizar uma soma até que você digite o valor zero ("Digite o valor da despesa (ou 0 para parar):
- imprimir a soma das despesas do dia.

15.5

## Conversor de Tempo de Estudo

É bem comum que os cronômetros de estudos sejam apenas em minutos, então para não ficar quebrando a cabeça convertendo o tempo de minutos para hora vamos criar um programa para isso:

- pergunte o total de minutos.
- converta os valores.
- imprima o resultado.


# Resposta

## Calculadora de IMC (Índice de Massa Corporal)

```python
# Pedindo os dados ao usuário
peso = float(input("Digite o seu peso (em kg): "))
altura = float(input("Digite a sua altura (em metros): "))

# Calcular o IMC usando a fórmula IMC = Peso / (Altura * Altura)
imc = peso / (altura * altura)

# Exibe o valor calculado com 2 casas decimais
print(f"Seu IMC é: {imc:.2f}")

# Verificar a classificação
if imc < 18.5:
    print("Situação: Abaixo do peso")

elif 18.5 <= imc <= 24.9:
    print("Situação: Peso normal")

elif 25.0 <= imc <= 29.9:
    print("Situação: Acima do peso")

else:
    print("Situação: Obeso")
```

# 15

15.6

## Resposta

## Conversor de Unidades Métricas

```python
def exibir_menu():
    print("\n--- CONVERSOR DE MEDIDAS ---")
    print("1. Quilômetros (km) para Metros (m)")
    print("2. Metros (m) para Quilômetros (km)")
    print("3. Metros (m) para Centímetros (cm)")
    print("4. Centímetros (cm) para Metros (m)")
    print("5. Quilômetros (km) para Centímetros (cm)")
    print("6. Centímetros (cm) para Quilômetros (km)")
    print("0. Sair")

# Funções de Conversão
def km_para_m(valor):
    return valor * 1000

def m_para_km(valor):
    return valor / 1000

def m_para_cm(valor):
    return valor * 100

def cm_para_m(valor):
    return valor / 100

def km_para_cm(valor):
    return valor * 100000

def cm_para_km(valor):
    return valor / 100000
```

```python
# Programa Principal
while True:
    exibir_menu()
    opcao = input("Escolha uma opção: ")

    if opcao == '0':
        print("Encerrando o conversor.")
        break

    if opcao in ['1', '2', '3', '4', '5', '6']:
        valor = float(input("Digite o valor a ser convertido: "))

        if opcao == '1':
            print(f"{valor} km equivalem a {km_para_m(valor)} m")
        elif opcao == '2':
            print(f"{valor} m equivalem a {m_para_km(valor)} km")
        elif opcao == '3':
            print(f"{valor} m equivalem a {m_para_cm(valor)} cm")
        elif opcao == '4':
            print(f"{valor} cm equivalem a {cm_para_m(valor)} m")
        elif opcao == '5':
            print(f"{valor} km equivalem a {km_para_cm(valor)} cm")
        elif opcao == '6':
            print(f"{valor} cm equivalem a {cm_para_km(valor)} km")
    else:
        print("Opção inválida, tente novamente.")
```


# Resposta

## Agenda de Aniversários

```python
def exibir_menu():
    print("\n--- AGENDA DE ANIVERSÁRIOS ---")
    print("1. Ver lista de aniversários")
    print("2. Adicionar novo aniversário")
    print("0. Sair")

# Criação do dicionário (pode começar vazio ou preenchido)
agenda = {
    "Mariana": "15/03",
    "João": "22/07",
    "Tia Sofia": "05/12"
}

while True:
    exibir_menu()
    opcao = input("Escolha uma opção: ")

    if opcao == '0':
        print("A sair do programa...")
        break
  #mostra os aniversários que temos na agenda
  elif opcao == '1':
      print("\n--- Aniversariantes ---")
      if len(agenda) == 0:
          print("A agenda está vazia.")
      else:
          for nome, data in agenda.items():
              print(f"Nome: {nome} | Data: {data}")

  elif opcao == '2':
      # Adicionar mais datas de aniversário
      nome_novo = input("\nDigite o nome da pessoa: ")
      data_nova = input("Digite a data (ex: DD/MM): ")

      agenda[nome_novo] = data_nova
      print(f"Aniversário de {nome_novo} adicionado com sucesso!")

  else:
      print("Opção inválida. Tente novamente.")
```


# Respostas

## Registro de Despesas

```
print("--- CONTROLE DE GASTOS DIÁRIOS ---")
total_despesas = 0

while True:
    #Pedimos o valorda dispesa
    valor = float(input("Digite o valor da despesa (ou 0 para parar): "))

    if valor == 0:
        break
    #continua somando até o valor zero ser digitado
    total_despesas += valor

# Exibimos o resultado final
print(f"\nA soma das despesas do dia: R$ {total_despesas:.2f}")
```

## Conversor de Tempo de Estudo

```
print("--- CONVERSOR DE TEMPO DE ESTUDO ---")

# Minutos totais
total_minutos = int(input("Digite o total de minutos passados: "))

# Converter os valores
horas = total_minutos // 60

# O módulo (%) pega o que sobrou da divisão (os minutos restantes)
minutos_restantes = total_minutos % 60

# Resultado
print(f"\nTempo total: {horas} hora(s) e {minutos_restantes} minuto(s).")
```

# Referências


ALI AWAN, Abid. **Tratamento de exceções e erros em Python** (2024). DATACAMP. Disponível em: https://www.datacamp.com/pt/tutorial/exception-handling-python. Acesso em: 10 de nov. 2025.

BORGES, Luiz Eduardo. **Python para desenvolvedores** . 1. ed. São Paulo: Novatec Editora, 2015. 318 p.

BESSA, André. **Compilador vs interpretador nas linguagens de programação** (2023). ALURA. Disponível em: https://cursos.alura.com.br/forum/topico-compilador-vs-interpretador-nas-linguagens-de-programacao-314787. Acesso em 27 ago. 2025.

CARVALHO, Caroline. **O que é Python? – um guia completo para iniciar nessa linguagem de programação** (2023). ALURA. Disponível em: https://www.alura.com.br/artigos/python?. Acesso em 26 ago. 2025.

CARVALHO, Guilherme. **Entendendo a Programação Orientada a Objetos com Python** (2023). MEDIUM. Disponível em: https://medium.com/@guilhermerdcarvalho/paradigma-orientado-ao-objeto-poo-em-python-a107d35bee3f. Acesso em 30 de set. de 2025.

CLEMENTE, Paulo. **Python, Um Guia sobre Programação Orientada a Objetos.** ROCKETSEAT. Disponível em: https://www.rocketseat.com.br/blog/artigos/post/python-poo. Acesso em 30 de set. de 2025.

DIDÁTICA TECH. **Tudo sobre variáveis em Python! Aprenda com exemplos práticos** (2024). DIDÁTICA TECH. Disponível em: https://didatica.tech/tudo-sobre-variaveis-em-python-aprenda-com-exemplos-praticos/. Acesso em 01 de set. de 2025.

Fabio. **Estruturas de condições em Python** (2016). DEVMEDIA. Disponível em: https://www.devmedia.com.br/estruturas-de-condicao-em-python/37158. Acesso em 22 de set. de 2025.

Fabio. **Estruturas de repetição em Python** (2020). **DEVMEDIA**. Disponível em: https://www.devmedia.com.br/estruturas-de-repeticao-em-python/41551. Acesso em 24 de set. de 2025.

FACCIONI, Juliano. **Dicionário em python: o guia definitivo para iniciantes** (2024). ASIMOV. Disponível em: https://hub.asimov.academy/blog/dicionario-python/. Acesso em 11 de set. de 2025.

FACCIONI, Juliano. **Range em Python: descubra o que é a função range e como usá-la** (2024) . ASIMOV. Disponível em: https://hub.asimov.academy/blog/range-em-python/. Acesso em 25 de set. de 2025.

GOMES, Ana Maria. **Como Criar Funções em Python** (2024). ASIMOV. Disponível em: https://hub.asimov.academy/tutorial/como-criar-funcoes-em-python/. Acesso em 29 de set. de 2025.

LOPES, Erikson. **Strings no python** (2023). PYTHON ACADEMY. Disponível em: https://pythonacademy.com.br/blog/strings-no-python. Acesso em 01 de set. de 2025.

MENEZES, Nilo Ney Coutinho. **Introdução à programação com Python** . 2. ed. São Paulo: Novatec Editora, 2016. 328 p.

ORESTES, Yan. **Tupla no Python: o que é, como criar e manipular e suas diferenças com as Listas** (2018) . ALURA. Disponível em: https://www.alura.com.br/artigos/conhecendo-as-tuplas-no-python?srsltid=AfmBOopo3zwBurjlrxPXW8i0twW9vDoFQjaoEFfTidNGhgcSEjTYCLyR. Acesso em 05 de set. de 2025.

PAIVA, Fábio Augusto Procópio de et al. Introdução a Python com aplicações de sistemas operacionais. Natal: Editora IFRN, 2020.

**Python Iluminado** (2025). PYTHON ILUMINADO. Disponível em: https://pythoniluminado.netlify.app/. Acesso em 28 ago. 2025.

PYTHON INSTITUTE. **Python® – the language of today and tomorrow** (2025). PYTHON INSTITUTE. Disponível em: https://pythoninstitute.org/about-python. Acesso em 26 ago. 2025.

PYTHON SOFTWARE FOUNDATION. **Estrutura de dados** (2025). PYTHON. Disponível em: https://docs.python.org/pt-br/3.13/tutorial/datastructures.html#dictionaries. Acesso em: 18 de set. de 2025.

RAMOS, Vinícius. **Estruturas de repetição usando loops while no python** (2023). PYTHON ACADEMY. Disponível em: https://pythonacademy.com.br/blog/loops-com-while-no-python. Acesso em 24 de set. de 2025.

RAMOS, Vinícius. **Programação Orientada a Objeto no Python: Introdução** (2023). PYTHON ACADEMY. Disponível em: https://pythonacademy.com.br/blog/introducao-a-programacao-orientada-a-objetos-no-python. Acesso em: 08 de out. de 2025.

RAMOS, Vinícius. **Tratamento de erros e exceções no python com try/except** (2023). PYTHON ACADEMY. Disponível em: https://pythonacademy.com.br/blog/tratamento-erros-excecoes-no-python. Acesso em: 10 de nov. de 2025.

RED HAT. **Ambiente de desenvolvimento integrado (IDE)** (2023). REDHAT. Disponível em: https://www.redhat.com/pt-br/topics/platform-engineering/what-is-ide. Acesso em 27 ago. 2025.

SHIMABUKURO, Igor; LIMA, Lucas. **O que é algoritmo? Entenda como funciona o conjunto de instruções de um programa** (2025). TECNOBLOG. Disponível em: https://tecnoblog.net/responde/o-que-e-algoritmo. Acesso em 26 ago. 2025.

SOARES, Adriano. **For Python: aprenda a utilizar a loop for com exemplos** (2024). ASIMOV. Disponível em: https://hub.asimov.academy/blog/for-python/. Acesso em 23 de set. de 2025.

SOARES, Adriano. **If, elif, e else: entendendo as estruturas condicionais em Python** (2024). ASIMOV. Disponível em: https://hub.asimov.academy/blog/if-elif-e-else-entendendo-as-estruturas-condicionais-em-python/. Acesso em 22 de set. de 2025.

SOARES, Adriano. **Try e except em Python - Entenda como lidar com erros** (2024). ASIMOV. Disponível em: https://hub.asimov.academy/blog/try-except-python/. Acesso em: 10 de nov. 2025.

VALINOR, Rodrigo. **Google Colab: descubra o que é e como usar essa ferramenta** (2025). REMESSA ONLINE. Disponível em: https://www.remessaonline.com.br/blog/google-colab/. Acesso em 27 ago. 2025.

W3SCHOLLS. **Python Lists** (2025). W3SCHOLLS. Disponível em: https://www.w3schools.com/python/python_lists.asp. Acesso em 08 de set. de 2025.

W3SCHOLLS. **Python Dictionaries** (2025). W3SCHOLLS. Disponível em: https://www.w3schools.com/python/python_dictionaries.asp. Acesso em 11 de set. de 2025.

# Sobre as autoras:

### Emanuela da Silva Silveira

Graduanda em Engenharia de Computação pela Universidade Federal de Santa Catarina (UFSC). Possui técnico integrado em Informática para à internet pelo Instituto Federal Catarinense - Campus Sombrio (IFC - CS). Membro do Laboratório de Tecnologias Computacionais (LabTeC - UFSC)

### Marta Adriana Machado da Silva

Doutora em Engenharia e Gestão do Conhecimento (UFSC) e Mestre e Bacharel em Ciências da Computação (UFSC). Com mais de 20 anos de experiência no Ensino Superior, atua como Docente, e autora de 3 livros didáticos (publicados pela UNESC em 2022): “Raciocínio Lógico e Computacional”, "Programação de Computadores" e "Estrutura de Dados". Possui vasta experiência em produção científica, com 7 capítulos de livros e 13 artigos completos publicados em periódicos especializados (como ETD e RENTE). Sua pesquisa inclui temas de Inteligência Artificial, inovação educacional, Realidade Aumentada (RA) e Laboratórios Remotos.

### Eliane Pozzebon

Professora Titular do Departamento de Computação da Universidade Federal de Santa Catarina (UFSC) e atua na área de Inteligência Artificial. Coordena o Laboratório de Tecnologias Computacionais (LabTeC-UFSC) e lidera o Grupo de Pesquisa em Tecnologias Computacionais certificado pelo CNPq. Possui pós-doutorado em Inteligência Artificial pela Universidade Federal do Rio Grande do Sul (UFRGS), doutorado em Engenharia Elétrica e mestrado em Ciência da Computação pela UFSC. Suas pesquisas envolvem inteligência artificial aplicada à educação e à saúde, tecnologias imersivas, reconhecimento facial e iniciativas voltadas à inclusão de mulheres na computação.

# Gostou desta leitura? Confira também:

**Visite http://labtec.ufsc.br/ebooks para conhecer a a coleção completa e conferir outros lançamentos.**