

formação em

# dados<sup>+</sup>

Python zero

dnc



# **Everton Menezes**

Head de Growth

@escoladnc

**Engenheiro Mecânico**

**Black Belt em Lean Six  
Sigma**

**Especialista em Projetos, BI,  
Estatística e Data Analysis**

**Experiência com:**

**Fortran, C++, Java Script,  
Python**

**Desenvolvedor de  
aplicativos de Robot  
Process Automation**

**Aplicações de modelos de  
Algoritmos Genéticos e  
Redes Neurais**

# O que veremos neste curso:

1

Introdução ao  
Python 0

2

Tipos de  
Dados

3

Operadores  
Aritméticos

4

Trabalhando com  
Strings

5

Condicionais

6

Loops

# O que veremos neste curso:

**7**

Listas e Tuplas

**8**

Dicionários

**9**

Funções Python  
e Lambda

**10**

Classes e  
Pacotes



# Alinhamento de Expectativas

**Objetivo:** Dar os primeiros passos no mundo da programação e ser capaz de utilizar lógica para resolver problemas por meio da programação utilizando o Python.

Só é possível Aprender <-> Fazendo

Regra nº 1 do curso: Faça os Exercícios

Regra nº 2 do curso: Siga a regra nº 1

# Módulo 1: Introdução ao Python Zero

**1**

Introdução à  
programação

**2**

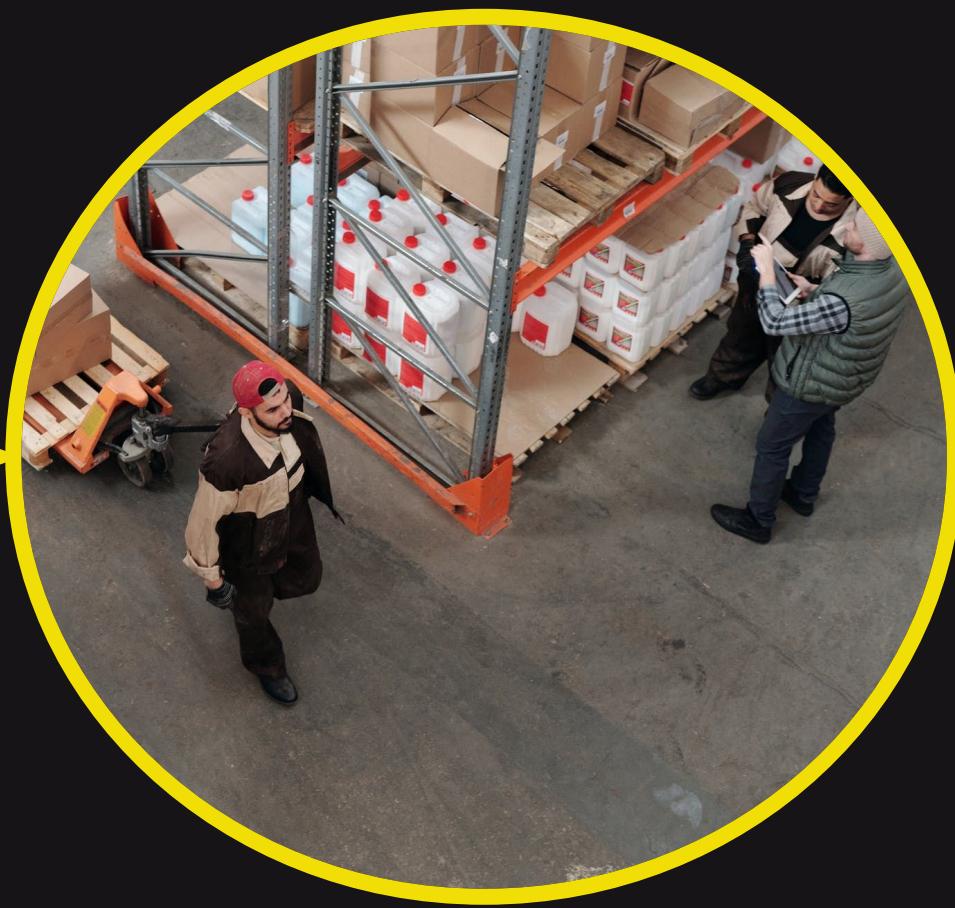
Por que Python?

**3**

Interpretadores  
e IDEs

# Introdução à Programação

Como fazer mais com menos?



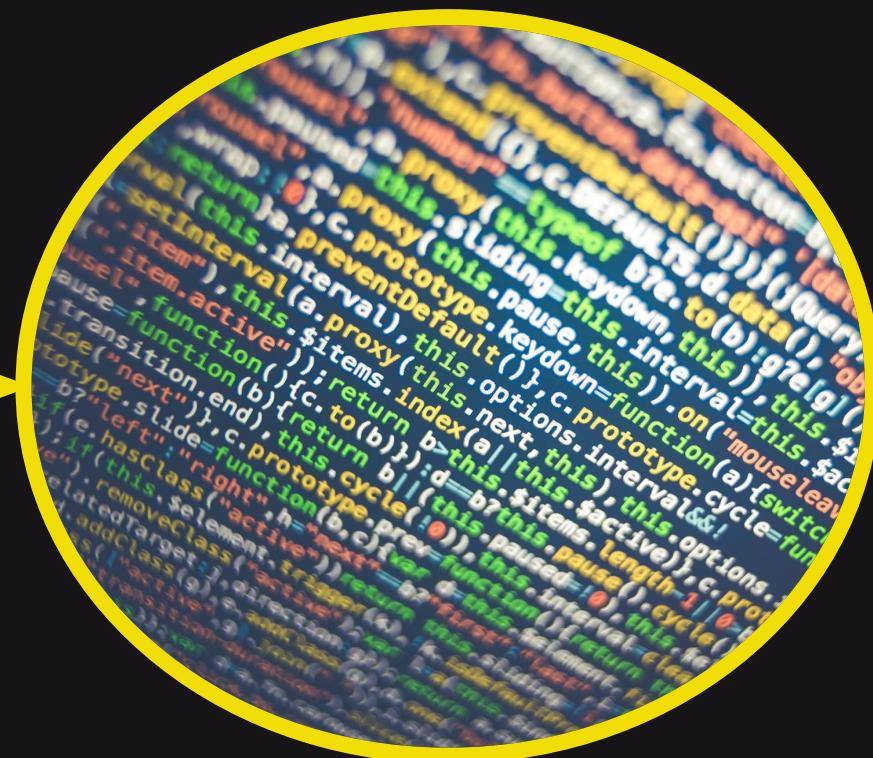
# Introdução à Programação

Como fazer ainda mais com muito menos?



# Introdução à Programação

Como tomar melhores decisões e mais rápido?



# **Tríplice da Aplicação da Programação**



**Games**



**Automações**



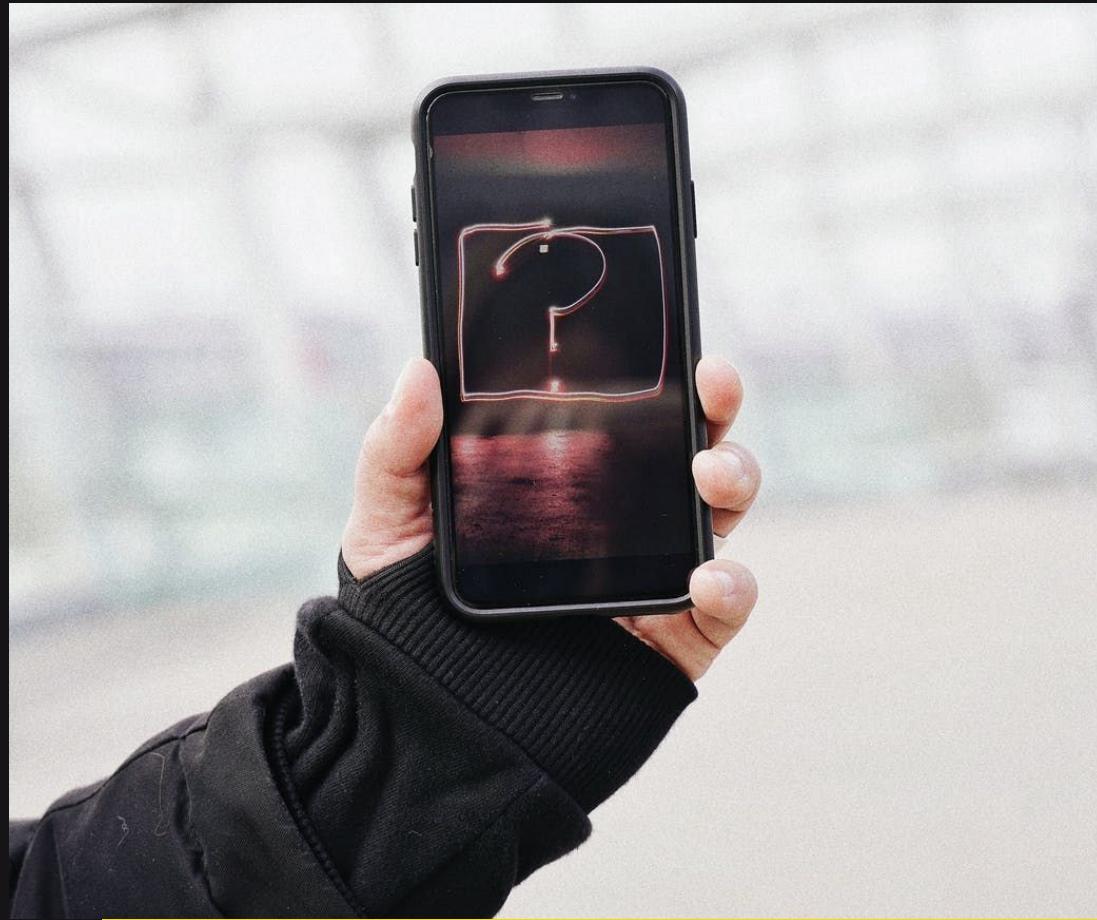
**Tomadas de Decisões**

# **Tomada de Decisão**

**Você já parou pra pensar quantas decisões você toma num dia?**

- O que comer?
- Aonde comer?
- O que vestir?
- O que comprar?
- Qual caminho pegar?

...



# Tomada de Decisão

## Simples: O que vou comer?



- Tipo de comida?
- Quanto dinheiro tenho?
- Tempo pra comer?
- Qualidade da comida?

Opções:  
Lanchão vs Méqui

# **Tomada de Decisão**

## **Complexa: Aonde invisto R\$ 100 mi?**

- Como está a situação financeira da empresa?
- Como estão os indicadores financeiros por fábrica? E por produto? E por região?
- Como está o mercado?
- Como está o juros?
- Como estão os concorrentes?
- Quais são os outros fatores externos? Previsão do tempo? Pandemias?

...

**Obs.: Seres humanos erram de 30 a 50%  
das tomadas de decisões.**

# **Bem-vindo ao mundo da Programação**

**Como fazer mais com menos?**

**Como tomar decisões melhores e mais rápido?**

**Como melhorar e facilitar a vida dos seres  
humanos?**

# Por que Python?

Basicamente porque é uma linguagem de simples e de propósito geral.



Mas como assim???

# Propósito de Lingua de programação?

PHP  
HTML  
Java Script

Desenvolvimento Web  
(sites)

C++  
Java

Desenvolvimento de  
Sistemas e Softwares

Fortran

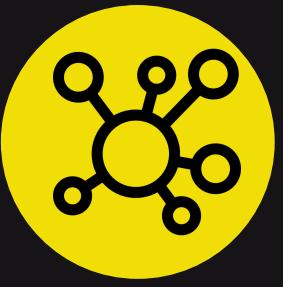
Voltado para Análises  
Numéricas



# Características do Python



Fácil e  
Intuitivo



Multiplataforma



Código Aberto



Linguagem  
Organizada



Orientada a  
Objetos



Bibliotecas  
“ilimitadas”

# Aplicações

Áreas: Desde Inteligência Artificial até biotecnologia e computação 3D.

Algumas aplicações:

- Games: Battlefield 2
- IA e Machine Learning: Skyscanner
- Casas inteligentes: raspberry e arduino
- Sistemas de recomendações: Youtube, Netflix
- Reconhecimento de imagem: Face ID, Google Car

# Python

Uma das linguagens mais **Simples** de se aprender.

É **Flexível**, pois pode ser usada nas mais diversas aplicações, desde Web até Data Science

Possui “infinitos” **Pacotes** que facilitam nossa vida na programação.

# Interpretadores e IDEs

O Python é uma linguagem interpretada, é um passo a passo!



Mas como assim???

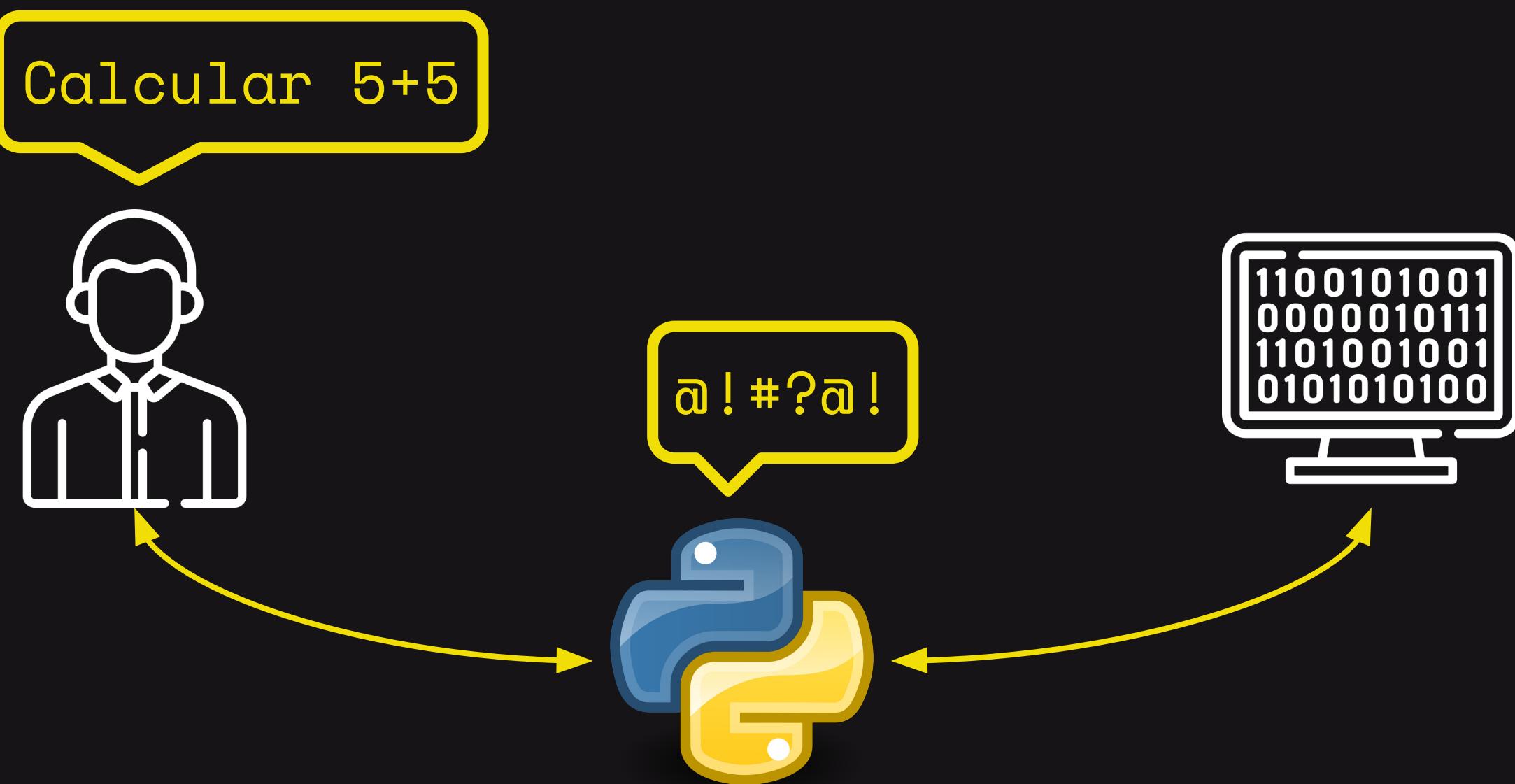
# O que é interpretar?

Interpretar = Traduzir

Eu quero um Lanche



# O que é interpretar em programação??



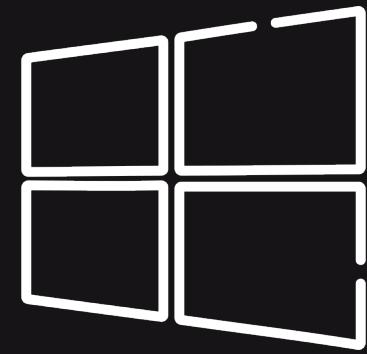
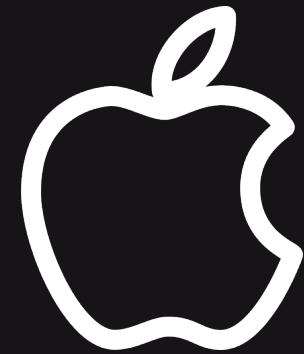
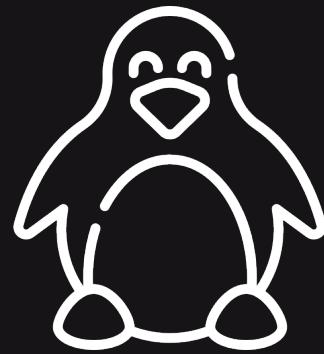
# Características de uma Linguagem Interpretada

- É executado linha a linha
  - Mais fácil de achar erros*
  - Mais devagar a execução*
- A interpretação é executada no computador do usuário
  - Facilidade na edição*
  - Menor segurança*



# Terminais e OS

O Python e os Sistemas Operacionais:



*download*

Os terminais são os locais no qual executamos os nossos códigos:

cmd ou PowerShell(win), idle (python), shell (linux), terminal (mac)

# PowerShell

```
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos os direitos reservados.

Instale o PowerShell mais recente para obter novos recursos e aprimoramentos!
https://aka.ms/PSWindows

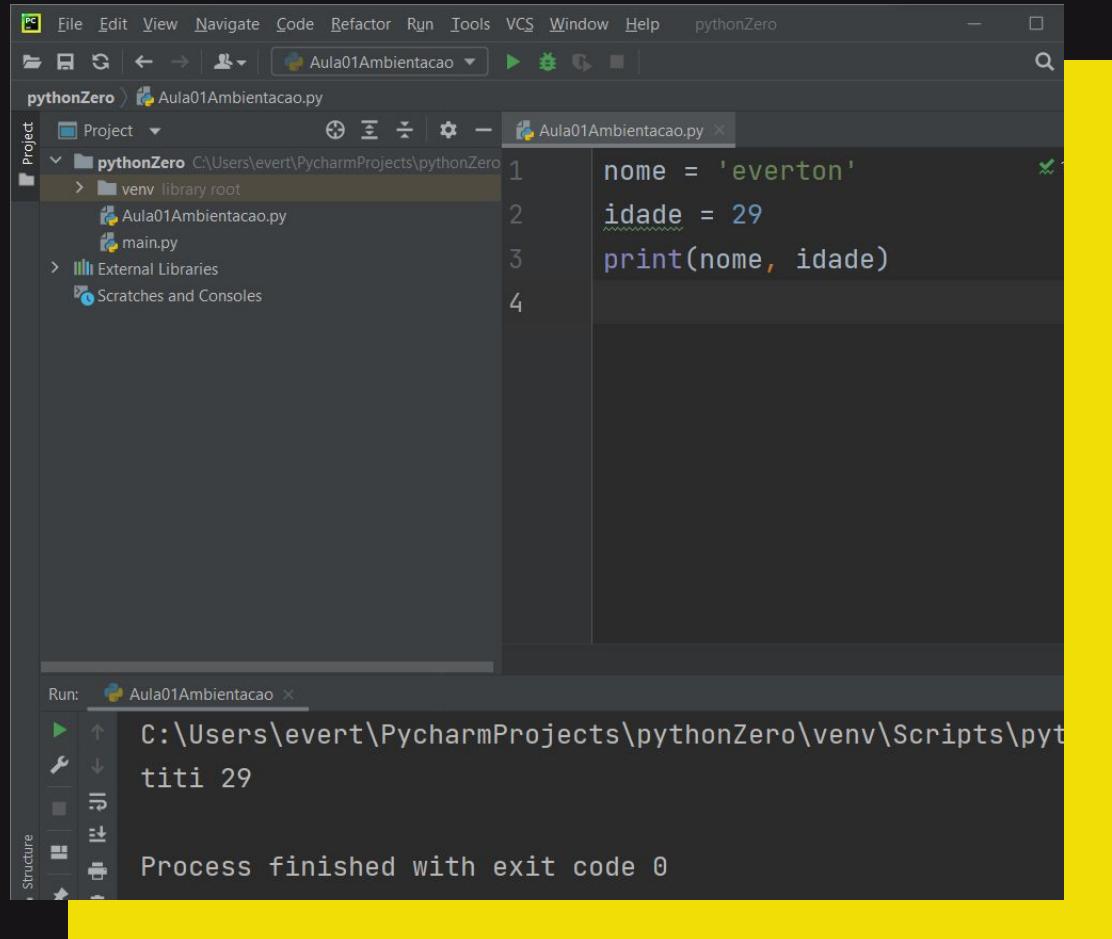
PS C:\Users\ever> python
Python 3.9.7 (tags/v3.9.7:1016ef3, Aug 30 2021, 20:19:38) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information. •
>>>
```

```
>>> 2+2
4
```

```
>>> print('Olá, mundo')
Olá, mundo
```

# **IDEs - Integrated Development Environment**

## **Ambiente de Desenvolvimento Integrado**



A screenshot of the PyCharm IDE interface. The top navigation bar includes File, Edit, View, Navigate, Code, Refactor, Run, Tools, VCS, Window, Help, and pythonZero. The main window shows a project named 'pythonZero' with files 'Aula01Ambientacao.py', 'venv library root', 'Aula01Ambientacao.py', and 'main.py'. The code editor displays the following Python code:

```
nome = 'everton'  
idade = 29  
print(nome, idade)
```

The bottom terminal window shows the output of running the script: 'titi 29'. A status bar at the bottom indicates 'Process finished with exit code 0'.

**Editor de código-fonte**  
+  
**Interpretador**  
+  
**Debugger**

**PyCharm, Sublime, ...**

**Google Colab**

# Google Colab

Ambiente de programação em Python da Google para democratizar ainda mais o estudo e a prática de programação.

- Interpretador fica na CPU da Google
- Fácil compartilhamento
- Não precisa baixar nem instalar nada

The screenshot shows the Google Colab interface. At the top, there's a navigation bar with 'Arquivo', 'Editar', 'Ver', 'Inserir', 'Ambiente de execução', 'Ferramentas', and a user profile icon. Below the bar, there's a toolbar with buttons for '+ Código', '+ Texto', and 'Copiar para o Drive'. To the right of the toolbar are 'Compartilhar', 'Conectar', 'Editar', and a dropdown menu. The main area has a dark background with white text. It starts with a welcome message: 'Olá, este é o Colaboratory' and 'comece a usá-lo abaixo!'. A section titled 'Primeiros passos' explains that the document is an interactive notebook (Colab) where you can write and execute Python code. It shows a code cell with the following Python script:

```
[ ] seconds_in_a_day = 24 * 60 * 60  
seconds_in_a_day
```

The output of the code is '86400'. Below the code cell, there's a note about executing the code by clicking it or pressing 'Command/Ctrl+Enter'. It also mentions that variables defined in one cell can be used in others.

```
[ ] seconds_in_a_week = 7 * seconds_in_a_day  
seconds_in_a_week
```

# **Interpretadores**

**Eles interpretam o desejo do programador em código-máquina.**

**São executados passo a passo, facilitando a detecção e conserto de erros.**

# Mão na Massa

Agora chegou a hora que vocês estavam esperando...

Let's CODE



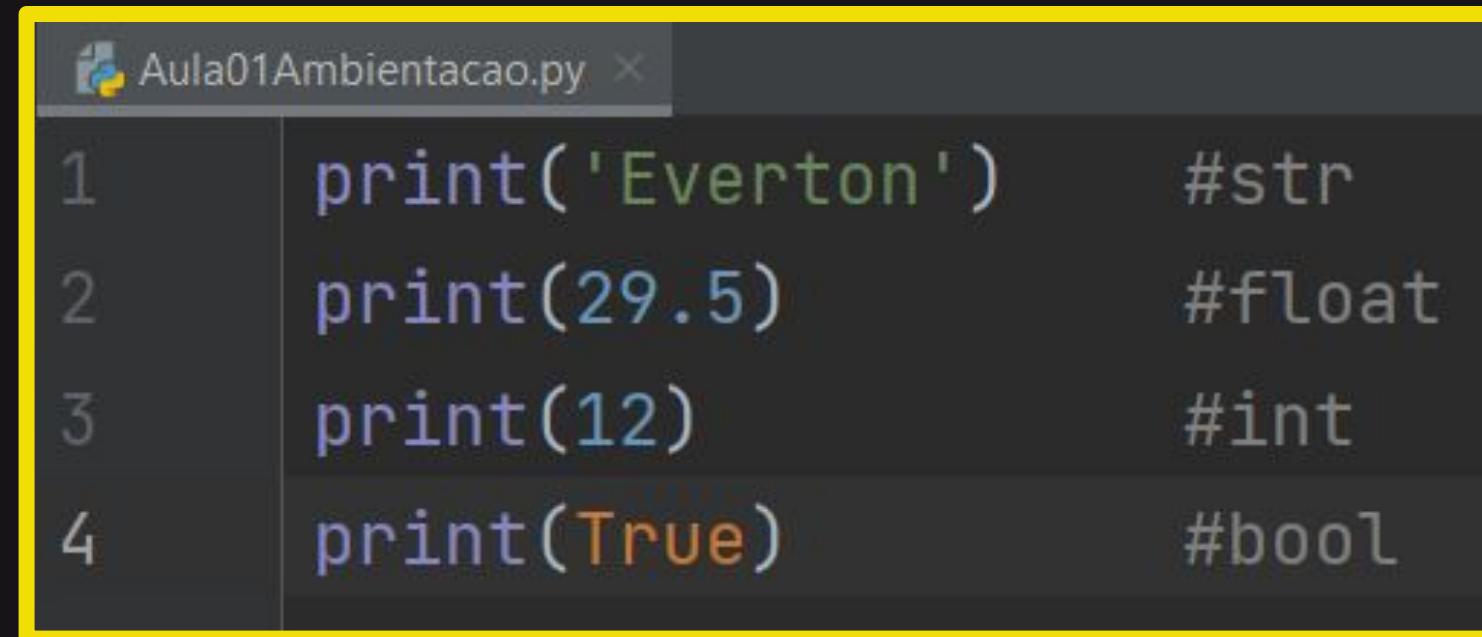
# Módulo 2:

## Tipos de Dados



# Tipos de Dados

Os tipos básicos são: int e float para números, string para textos e booleana para V ou F.

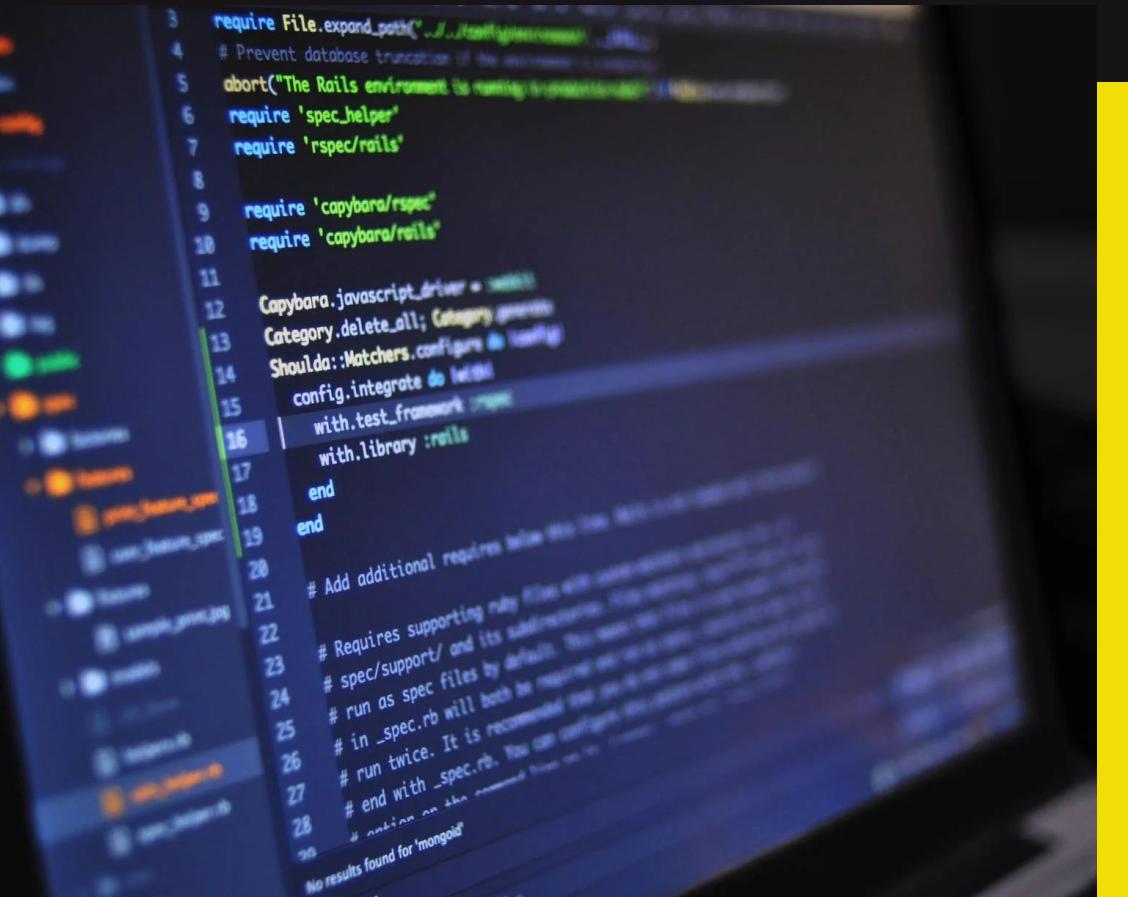


```
Aula01Ambientacao.py
1 print('Everton')      #str
2 print(29.5)           #float
3 print(12)              #int
4 print(True)            #bool
```

# Antes de tudo

## Como iremos estudar:

1. Assista a aula teórica  
**[teoria]**
2. Abra o arquivo base -  
*próxima atividade*
3. Acompanhe a aula  
prática **[pratica]**
4. **Code** junto comigo  
durante a aula
5. Faça os exercícios



```
3 require File.expand_path('../config/environment', __FILE__)
4 # Prevent database truncation if the database needs
5 abort("The Rails environment is running in production mode!
6 require 'spec_helper'
7 require 'rspec/rails'

8 require 'capybara/rspec'
9 require 'capybara/rails'

10 Capybara.javascript_driver = :webkit
11 Category.delete_all; Category.create!(name: "Frontend")
12 Shoulda::Matchers.configure do |config|
13   config.integrate do |with|
14     with.test_framework :rspec
15     with.library :rails
16   end
17 end
18
19 # Add additional requires below this line if necessary
20
21 # Requires supporting files with custom matchers and
22 # helper methods under 'spec/support/' and its subdirectories. These must be
23 # run as spec files by default. You can specify this as :support_files
24 # in _spec.rb will both be run as
25 # run twice. It is recommended that you do not name files under 'spec/
26 # end with _spec.rb. You can configure this pattern with the :file_name
27 # option in the configuration of RSpec::Rails::Configuration.
28
29 # option in the configuration of RSpec::Rails::Configuration.
30
31 # No results found for 'mongoid'
32
33 Mongoid.configure do |config|
34   config.connect_to('mongoid')
35   config.buffer = true
36 end
```

# Trabalhando com texto

Textos em Python são colocados entre aspas  
**'str'**

Vamos mostrar a mensagem para o usuário:  
Olá, mundo!

Olá, mundo !

#nossa frase

' Olá, mundo ! '

#texto entre ' '

```
print(' Olá, mundo ! ') #função imprimir
```

```
Olá, mundo !
```

# Entendendo a estrutura

```
print('Olá, mundo!')      #função imprimir
```

String [Tipo de Dado] = de cadeia  
de caracteres 'str'

print() [Função] = imprime pro  
usuário o que estiver dentro do: ()

# [Comentário] = após o # tudo é  
desprezado na linha

*Observe as cores!!!*

## Outros exemplos - str

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

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

```
print('10') #número
```

```
print(" ") #espaço em branco
```

```
print() #linha em branco
```

```
print('!@#!#?@!') #caracter especial
```

# Trabalhando com número

Números em Python são colocados direto e podem ou não ter casas decimais.

7            #int  
7.2        #float

Podemos imprimir números?

```
print(7)
```

```
7
```

```
print(7.2)
```

```
7.2
```

# Podemos imprimir uma soma?

```
print(7 + 2)
```

```
9
```

```
print(7.2 + 2.8)
```

```
10.0
```

```
print('7' + '2')
```

```
72
```

# Podemos juntar strings?

```
print('Meu nome é' + ' Everton')
```

```
Meu nome é Everton
```

```
print('Meu nome é', ' Everton')
```

```
Meu nome é Everton
```

```
print('Eu tenho', 3, 'cachorros')
```

```
Eu tenho 3 cachorros
```

# Podemos juntar strings?

```
print('Eu tenho' + 3 + ' cachorros')
```

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

Na função **print()** podemos misturar tipos de variáveis desde que elas sejam separadas por **,**

Já o operador **+** só pode ser usado com tipos de dados compatíveis.

Int + int

soma

Int + float

soma

Str + Str

concatenar

# Declarando variáveis

lanche	=	'x-burguer'
preco	=	10.90
qtd	=	2

Variáveis

Atribuição

Valores  
armazenados

# **Podemos fazer contas?**

```
lanche = 10.9  
batata = 5.9  
refri = 3  
conta = lanche + batata + refri  
print(conta)  
  
19.8
```

# **Podemos fazer impressão composta?**

```
lanche = 10.9
batata = 5.9
pgto = 'cartão'
conta = lanche + batata
print('Você pediu um lanche de: R$', lanche, 'e'
      'uma batata de: R$', batata, 'totalizando: R$', conta,
      '. O seu pagamento será feito no: ', pgto)
```

Você pediu um lanche de: R\$ 10.9 e uma batata  
de: R\$ 5.9 totalizando: R\$ 16.8 . O seu  
pagamento será feito no: cartao

# input()

Para melhorarmos a interface com o usuário, podemos pedir para que ele insira os valores de cada variável usando o **input()**

```
lanche = input('Qual lanche você quer?')  
print('Você quer um: ', lanche)
```

```
... Qual lanche você quer? x-burguer
```

```
Você quer um: x-burguer
```

```
input( ..... )
```



Frase descrevendo a informação  
que o usuário deve inserir.  
O dado sempre vem como str

# **type ()**

No input e em outros casos, podemos nos deparar com Tipos de Dados indesejados. Para isso podemos validar utilizando a função: **type()**

```
type('Everton')
```

```
str
```

```
type(5)
```

```
int
```

```
type(57.2)
```

```
float
```

## Tipo de Dado = **bool**

O último tipo de dado que veremos é o **bool**.  
Ela pode ser **True** ou **False** (verdadeira ou falso)

```
pgtoCartao = True  
pgtoDinheiro = False  
type(pgtoCartao)  
bool
```

**True** e **False** sempre com letra maiúscula e sem aspas. Muito utilizado para fazer comparações

# **Tipos de Dados**

**str [cadeia de caracteres] = 'Py' , '7' , “áçac”**  
**int [inteiro] = 5 , 7 , 2**  
**float [real] = 5.2 , 7.0 , 3.14**  
**bool [booleano] = True , False**

# **Funções**

**print( ) = imprime uma valor ou frase**  
**type( ) = verifica o Tipo de Dado de uma variável**  
**input( ) = pede pro usuário fornecer um valor (str)**

# Mão na Massa

Agora chegou a hora que vocês estavam esperando...

Let's CODE



# Módulo 3: Operadores Aritméticos

**1**

Aritmética

**2**

Transformando  
variáveis: `int()`,  
`float()`, `str()`

**3**

Função `print()`  
modificada

# Operadores Aritméticos

Os operadores mais usados são: mais, menos, vezes, dividir e potência

```
print(7+2) = 9  
print(7-2) = 5  
print(7*2) = 14  
print(7/2) = 3.5  
print(7**2) = 49
```

# Operadores

7	+	2	==	9
7	-	2	==	5
7	*	2	==	14
7	/	2	==	3.5
7	**	2	==	49
7	//	2	==	3
7	%	2	==	1

soma  
subtração  
multiplicação  
divisão  
potência  
divisão inteira  
resto da divisão

Operadores



Igualdade



# Operadores só funcionam com números?

Dos operadores acima, existem 2 que conseguimos usar com strings:

+

→ concatena

\*

→ repete n vezes

```
print('Olá' + ' mundo!')
```

```
Olá mundo!
```

```
print('Olá' *5)
```

```
Olá Olá Olá Olá Olá
```

# Mas...

## Qual a aplicação?

Como mostrar para o usuário?

Bem Vindo

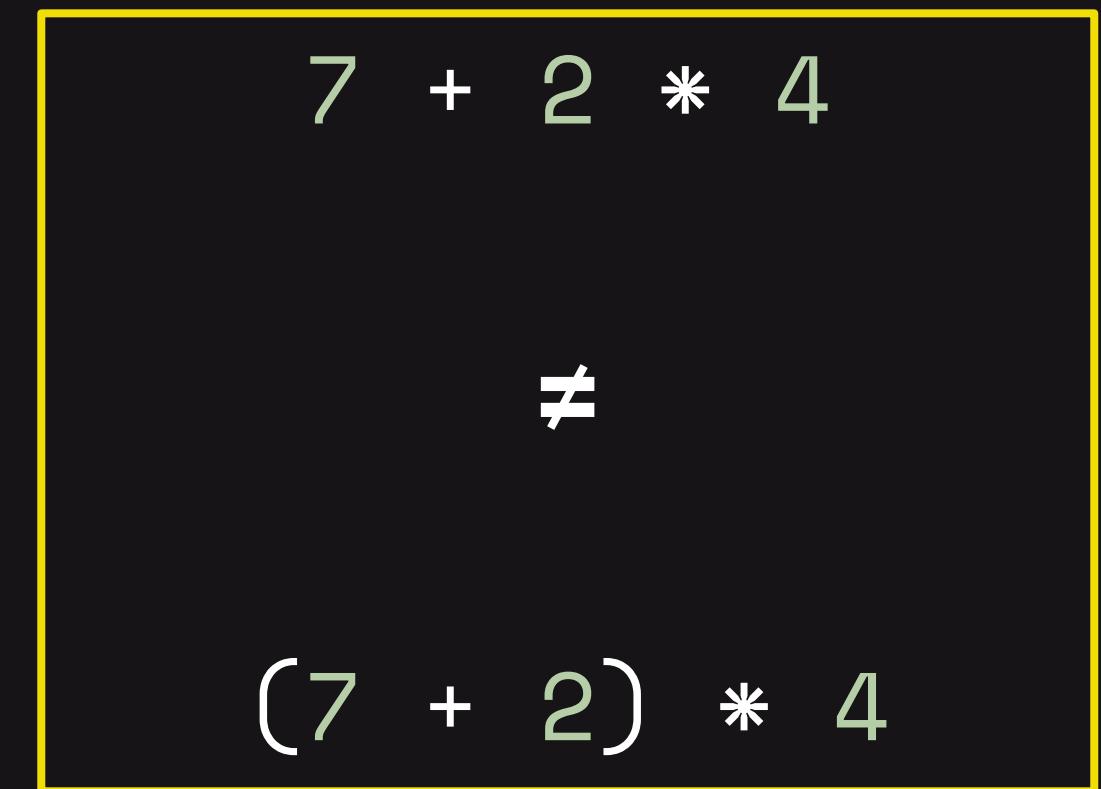
```
print('-----')
print('      Bem Vindo      ')
print('-----')
```

```
print('-*21)
print('-*6 + 'Bem Vindo' + ' -*4)
print('-*21)
```

# Voltando para aritmética

## Ordem de precedência:

1.	( )
2.	**
3.	* / % //
4.	+ -



*Tome cuidado com as contas e valide os resultados do seu programa!!!*

# Transformando variáveis

Vamos agora aprender as funções:

`int()` #Converte variável para inteiro

`float()` #Converte variável para real

`str()` #Converte variável para string

Quando iremos usar?

Quando as variáveis vierem num formato que não conseguimos manipular como queremos.

Exemplo: após o `input()`

# int()

```
int(1.857)
```

```
1
```

```
int(True)
```

```
1
```

```
int('4')
```

```
4
```

```
int(False)
```

```
0
```

```
int('tex34')
```

```
ValueError: invalid literal for int() with  
base 10: 'tex34'
```

# float( )

float(1)

1.0

float(True)

1.0

float('4')

4.0

float(False)

0.0

float('tex34')

ValueError: could not convert string to  
float: 'tex34'

# **str( )**

`str(1)`

`'1'`

`str(True)`

`'True'`

`str(2.758)`

`'2.758'`

`str(False)`

`'False'`

# **print() com .format()**

```
# Imprimindo na forma tradicional
print('Você pediu um lanche de: R$', 
lanchePreco, 'e uma batata de: R$', 
batataPreco, 'totalizando: R$', pedidoValorTotal)
```

**Repare no tanto de aspas e vírgulas**

```
# Imprimindo com .format
print('Você pediu um lanche de: R${} e uma
batata de: R${} totalizando:
R${}'.format(lanchePreco,batataPreco,pedidoValor
Total))
```

# Entendendo a estrutura do print( ) com .format

```
a = 2  
b = 3  
print ('Texto {} mais texto {}'.format(a,b))
```

{ } [Máscaras] = ficam na  
posição das variáveis

.format(a,b) [Método] =  
separa as variáveis que  
estão dentro do print()

```
Texto 2, mais texto 3
```

# Delimitando tamanho de números impressos

```
a = 1  
b = 7  
print('a dividido por b é: {}'.format(a/b))
```

```
a dividido por b é: 0.14285714285714285
```

```
a = 1  
b = 7  
print('a dividido por b é: {:.5f}'.format(a/b))
```

```
a dividido por b é: 1.42857
```

# Entendendo a estrutura da Máscara

```
a = 245  
b = 2.835895  
print ('|{:10d}| e |{:10.4f}|'.format(a,b))
```

```
| 245 | e | 2.8359 |
```

{ 0:10d }  
0 → 1ª variável no  
format  
: → separador  
10 → separar 10 espaços  
para o número  
d → número inteiro

{ 1:10.4f }  
1 → 2ª variável no  
format  
: → separador  
10 → separar 10 espaços  
para o número  
. → separador decimal  
4f → limitar em 4 floats  
(algarismos após vírgula)

# Operações aritméticas

( ),      \*\* ,      \*      /      %      // ,      +      -

# Conversão de variáveis

`int()` = converte para inteiro

`float()` = converte para real

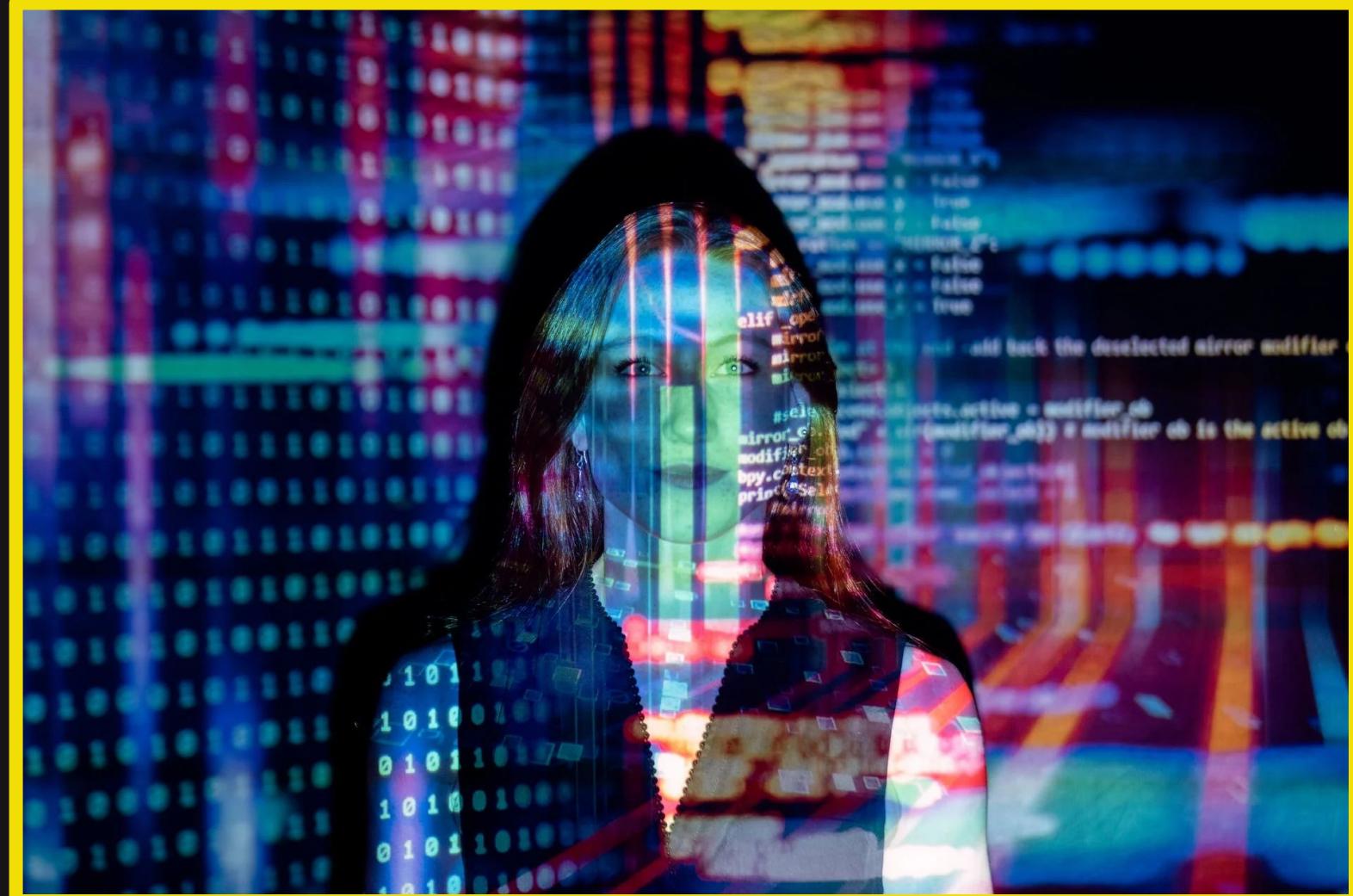
`str()` = converte para string

# `print()` com `.format()`

`print('textos { }'.format( ))`

# Mão na Massa

Let's CODE



# Módulo 4:

## Condicionais

1

Comparadores

2

Estrutura  
simples: if-else

3

Estrutura  
composta:  
if-elif-else

4

Múltiplas  
Condições  
and e or

5

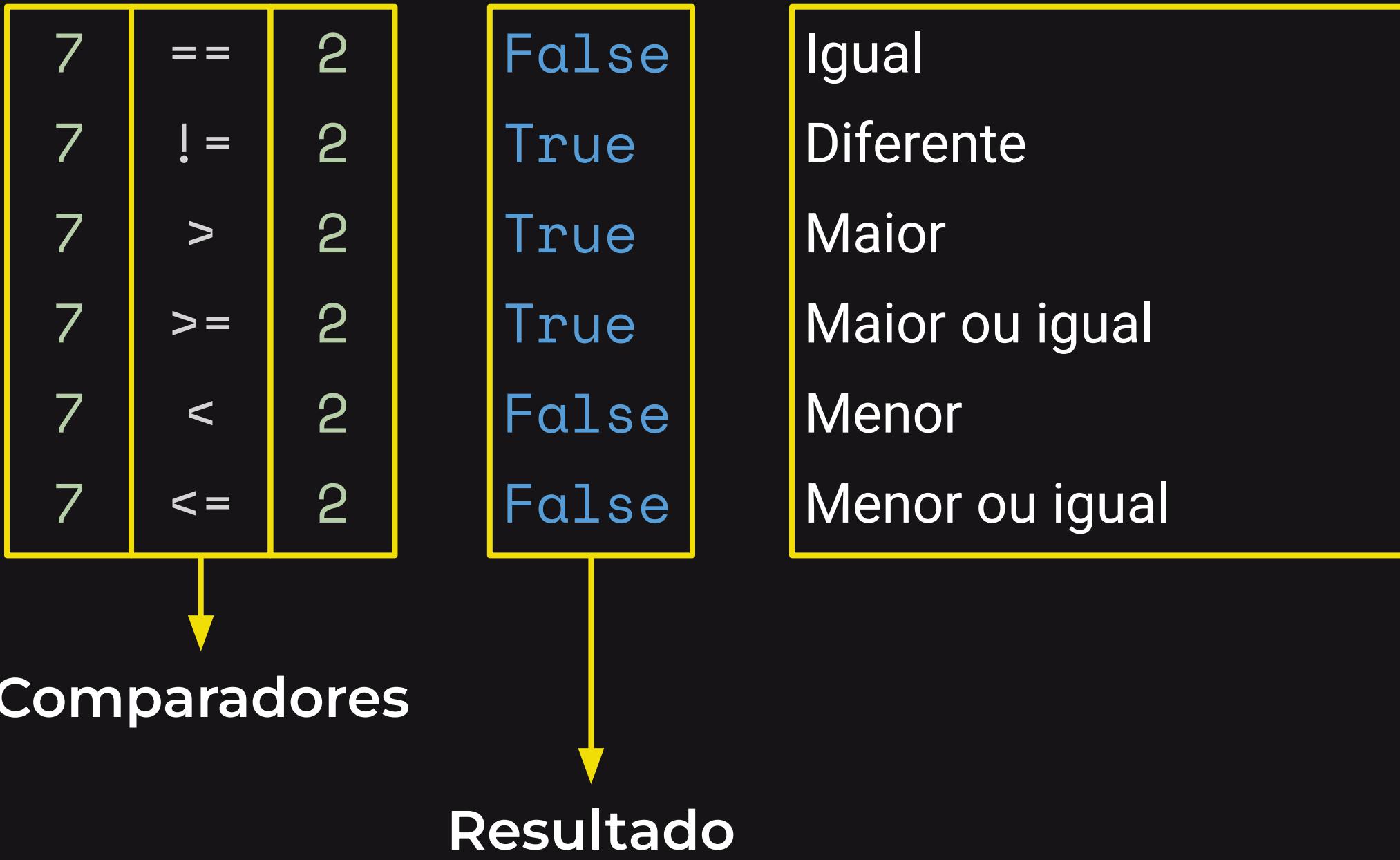
Verificar variáveis:  
.isnumeric( ),  
.isalpha( ), ...

# Condicionais

Se algo é verdade, faça uma coisa,  
senão faça outra coisa

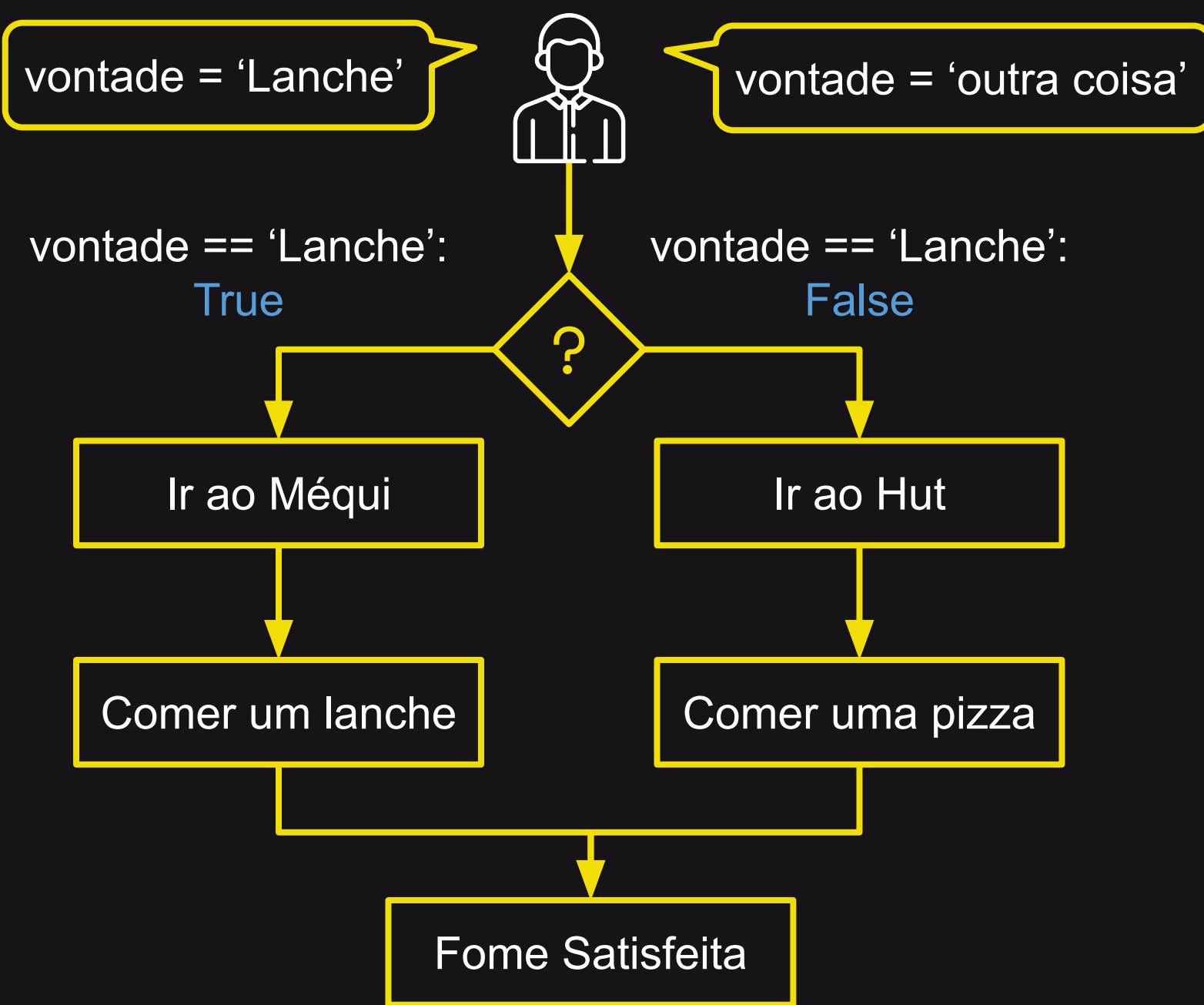
```
if True:  
    print('É True')  
else:  
    print('É False')
```

# Comparadores



# O que são condições?

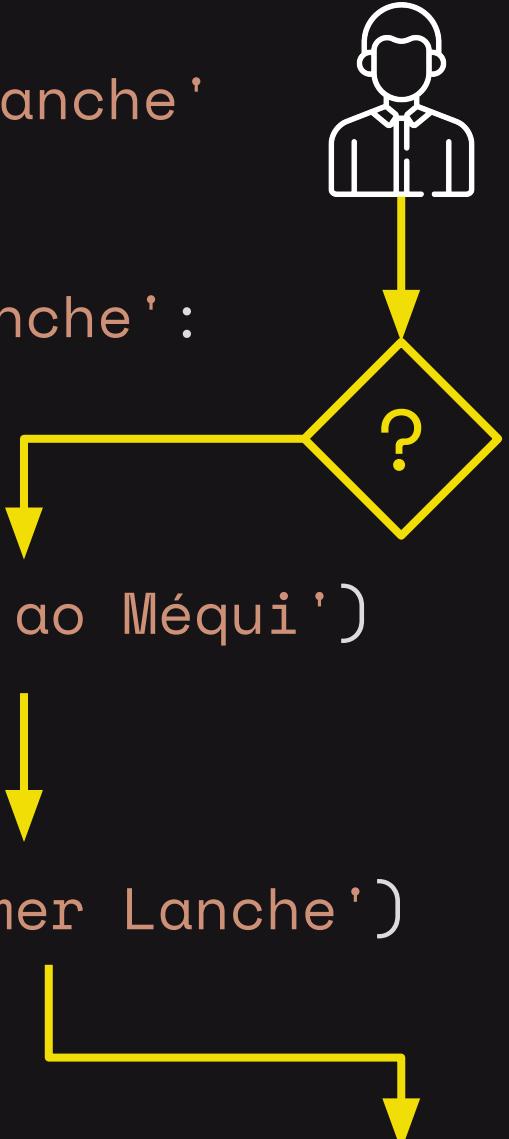
Exemplo: Estou com fome e quero decidir aonde eu vou comer!!!



# De condições para código

## Parte 1: True

```
vontade = 'Lanche'  
  
if vontade == 'Lanche':  
    print('Ir ao Méqui')  
  
    print('Comer Lanche')  
  
    print('Fome Satisfeita')
```

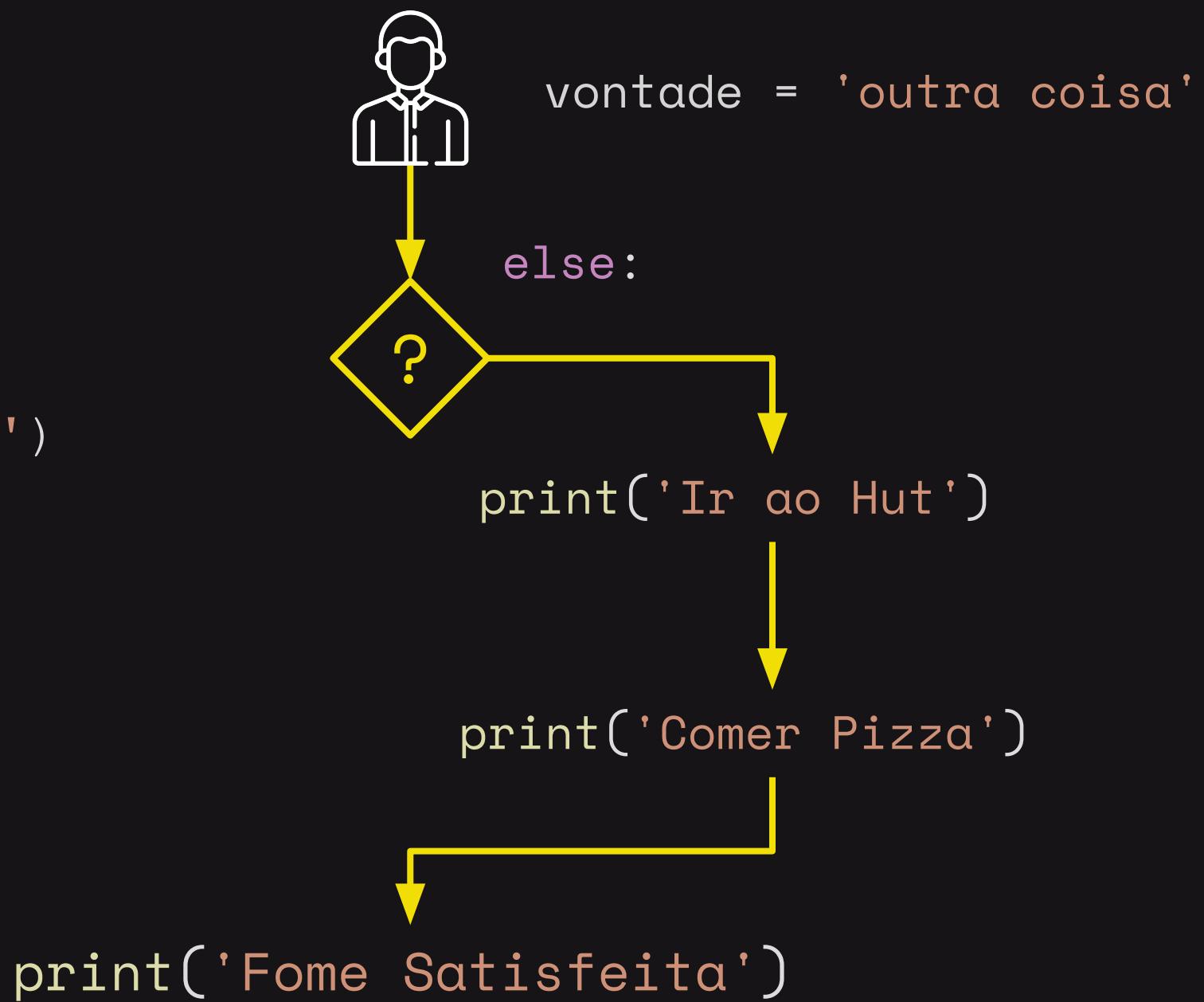


```
vontade = 'Lanche'  
if vontade == 'Lanche':  
    print('Ir ao Méqui')  
    print('Comer Lanche')  
    print('Fome Satisfeita')
```

# De condições para código

## Parte 2: False

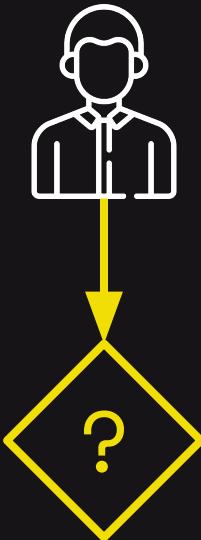
```
else:  
    print('Ir ao Hut')  
    print('Comer Pizza')
```



# Juntando os Códigos:

```
vontade=input('vontade= ')
```

```
if vontade == 'Lanche':  
    print('Ir ao Méqui')  
    print('Comer Lanche')
```



```
else:  
    print('Ir ao Hut')  
    print('Comer Pizza')
```

```
print('Fome Satisfeita')
```

---

```
vontade = input('vontade = ')  
if vontade == 'Lanche':  
    print('Ir ao Méqui')  
    print('Comer Lanche')  
else:  
    print('Ir ao Hut')  
    print('Comer Pizza')  
print('Fome Satisfeita')
```

# Estrutura do if

```
vontade = input('vontade = ')
if vontade == 'Lanche':
    print('Ir ao Méqui')
    print('Comer Lanche')
else:
    print('Ir ao Hut')
    print('Comer Pizza')
print('Fome Satisfeita')
```

- if → inicio da condição
- True → teste de verdade
- :
- início do bloco da verdade
- Bloco da condição verdadeira
- else → inicio da condição falsa
- :
- início do bloco da condição falsa
- Bloco da condição falsa
- Fim do if
- Ações fora do if

Atenção à  
Indentação

TAB

**Mas... e se tivermos**

**mais de 2 condições?**

As condicionais trabalham com variáveis

Booleanas: True or False

Então não é possível fazer dentro do if !!!

Mas e agora???

**Vamos fazer um if dentro**

**da condição falsa**

# Condicionais aninhados

Se vontade == ‘Lanche’ -> ir ao Méqui

Se vontade == ‘Pizza’ -> ir ao Hut

Se vontade == ‘Mexicano’ -> ir ao Bell

```
vontade = input('vontade = ')
if vontade == 'Lanche':
    print('Ir ao Méqui')
    print('Comer Lanche')
else:
    if vontade == 'Pizza':
        print('Ir ao Hut')
        print('Comer Pizza')
    else:
        print('Ir ao Bell')
        print('Comer Tacos')
print('Fome Satisfeita')
```

input:	vontade = Lanche
output:	Ir ao Méqui Comer Lanche

input:	vontade = Pizza
output:	Ir ao Hut Comer Pizza

input:	vontade = Mexicano
output:	Ir ao Bell Comer Tacos

# Hora de simplificar

**else + if = elif**

```
vontade = input('vontade = ')
if vontade == 'Lanche':
    print('Ir ao Méqui')
    print('Comer Lanche')

else:
    if vontade == 'Pizza':
        print('Ir ao Hut')
        print('Comer Pizza')
    else:
```

elif vontade == 'Pizza':

# Resumo

## Estrutura completa if

```
if condição1 == validação1:  
    Bloco Verdade condição 1  
elif condição2 == validação2:  
    Bloco Verdade condição 2  
elif condição3 == validação3:  
    .  
    .  
    .  
else:  
    Bloco condições Falsas
```

# Múltiplas Condições

## Operadores Lógicos

E se precisarmos validar 2 ou mais condições de uma só vez?

**and** → ambas as condições precisam ser satisfeitas simultaneamente

```
if True and True:
```

True: Bloco Verdade

```
if True and False:
```

False: Bloco False

**or** → pelo menos 1 condição precisa ser satisfeita

```
if True or True:
```

True: Bloco Verdade

```
if True or False:
```

True: Bloco Verdade

```
if False or False:
```

False: Bloco False

# Exemplo and

Se eu tiver mais que 20 reais e preço do lanche for menor que 10 reais, então eu compro o lanche:

```
if saldo >=20 and precoLanche <10:  
    print('Comprar lanche')  
else:  
    print('Não comprar lanche')
```

saldo = 20 precoLanche = 9.9	Comprar Lanche
saldo = 20 precoLanche = 12	Não comprar Lanche
saldo = 18 precoLanche = 9.9	Não comprar Lanche

# Exemplo or

Se você tiver média nas notas > 7 ou nota na prova final > 7 você está aprovado.

```
if media > 7 or notaFinal > 7:  
    print('Aprovado')  
else:  
    print('Reprovado')
```

media = 8 notaFinal = 6	Aprovado
media = 6 notaFinal = 8	Aprovado
media = 6 notaFinal = 6	Reprovado

# Verificando variáveis

Será que o usuário digitou uma variável válida?

Será que pegamos do Banco de Dados uma variável que podemos fazer as contas?

```
.isnumeric( )    #o input é numérico  
.isalpha( )      #o input é texto  
.isalnum( )      #o input é texto + num
```

Como usar?

```
var = input('Digite algo: ')  
var.isalpha()
```

True or  
False

# Exemplos

```
var = input('Digite algo: ')
type(var)
```

```
Digite algo: texto
str
```

```
var.isalpha()
```

```
True
```

```
var.isalnum()
```

```
False
```

```
var.isnumeric()
```

```
False
```

# Exemplos

```
var = input('Digite algo: ')
type(var)
```

```
Digite algo: 125
str
```

```
var.isalpha()
```

```
False
```

```
var.isalnum()
```

```
False
```

```
var.isnumeric()
```

```
True
```

# Estrutura if - elif - else

```
if condição1 == validação1:  
    Bloco Verdade condição 1  
elif condição2 == validação2:  
    Bloco Verdade condição 2  
elif condição3 == validação3:  
    Bloco Verdade condição 3  
...  
else:  
    Bloco condições Falsas
```

==  
!=  
>  
>=  
<  
<=

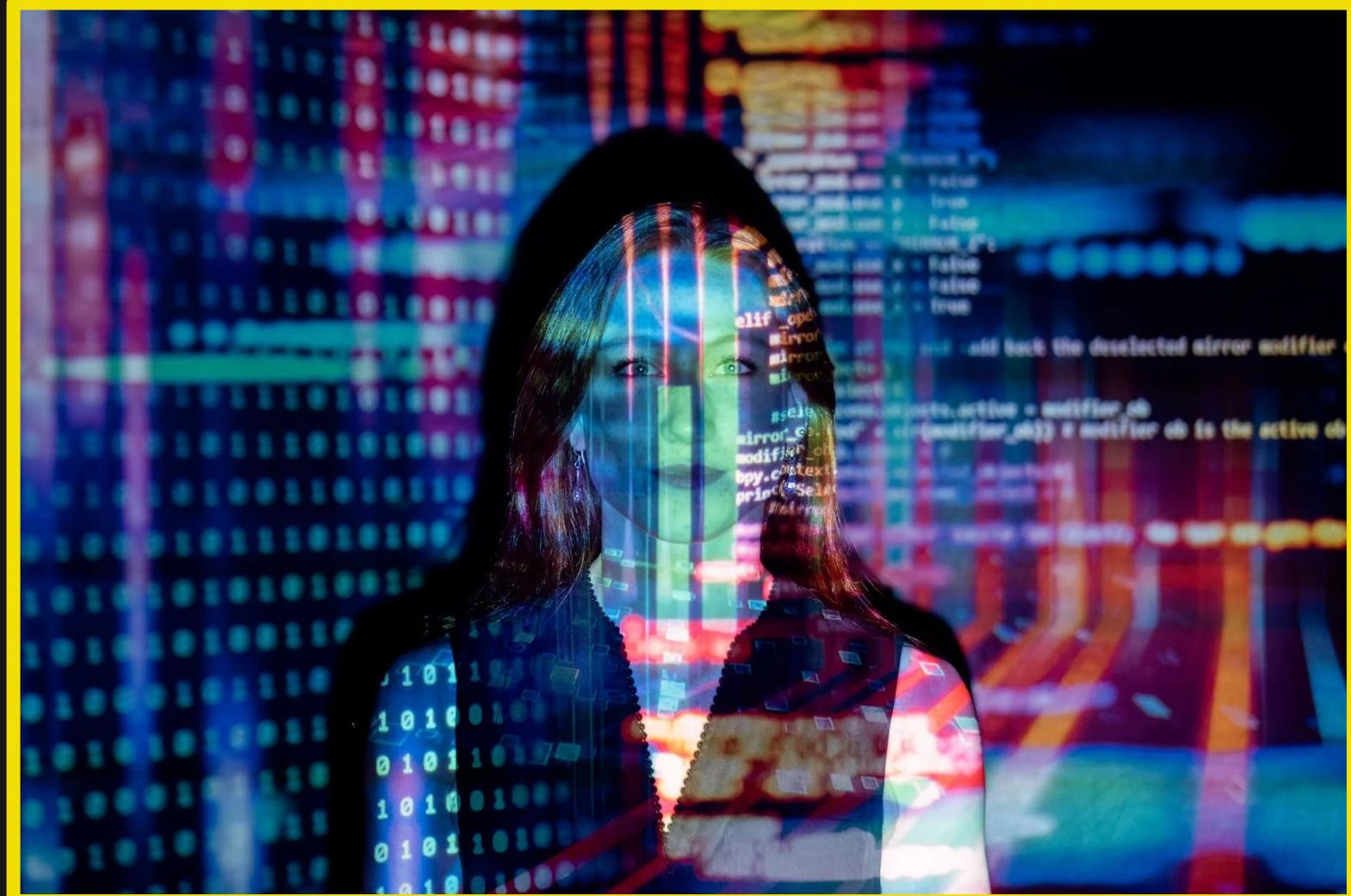
and  
or

## Teste de variável

```
.isnumeric( ) #o input é numérico  
.isalpha( )   #o input é texto  
.isalnum( )   #o input é texto + num
```

# Mão na Massa

Let's CODE



# Módulo 5: Repetições

**1**

Estruturas de  
Repetições

**2**

Estrutura  
for

**3**

Estrutura  
while

**4**

Interrompendo  
um loop:  
`break`

# Repetições

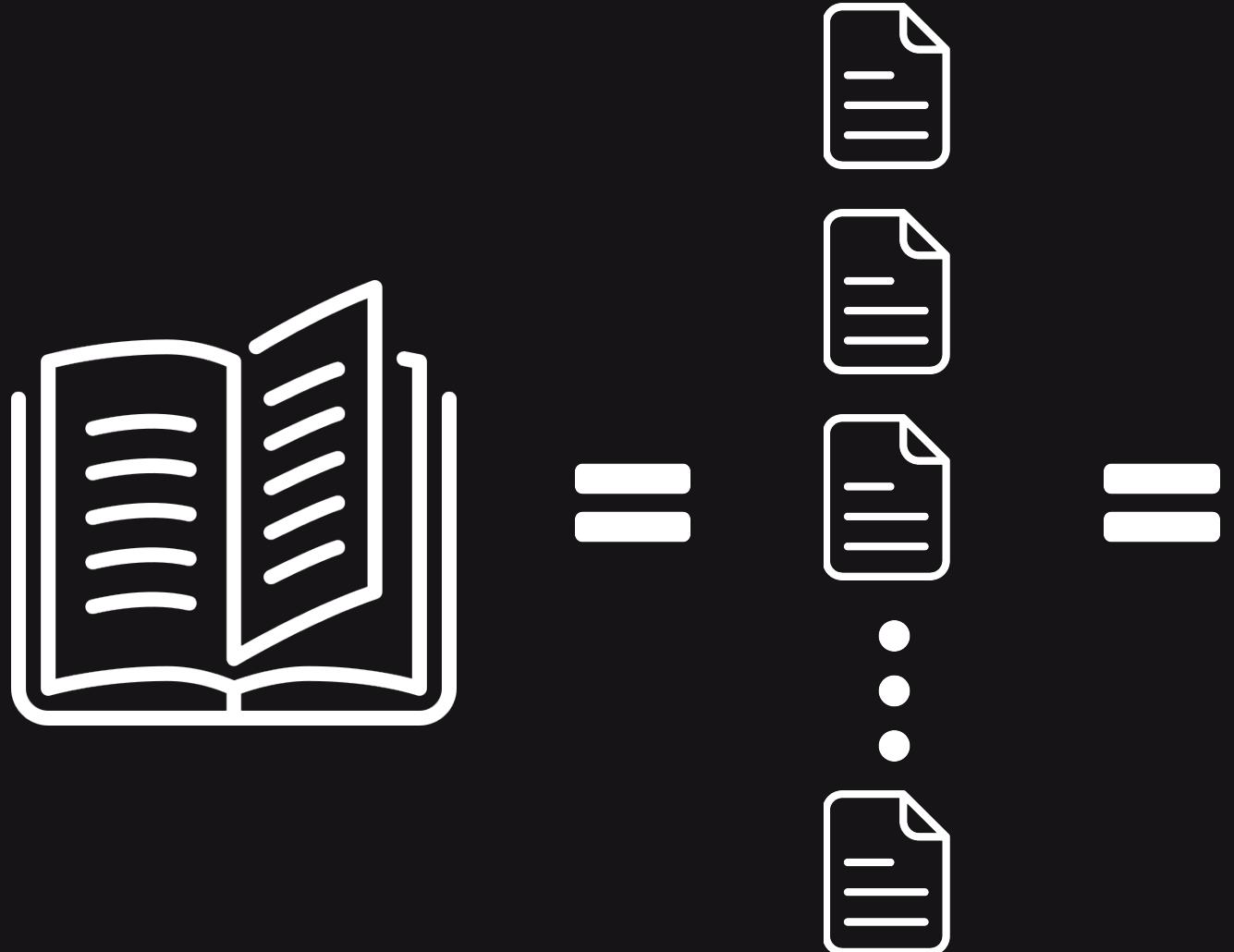
Faça algum processo repetitivo até ele terminar

```
for i in passo:  
    print('For: Execute')  
  
j = True  
while j == True:  
    print('While: Execute')  
    j = False
```

# O que é uma repetição?

É algo que precisamos executar várias vezes.

Exemplo: Ler um livro de 20 páginas



```
print('Ler página')
print('Ler página')
print('Ler página')
print('Ler página')
print('Ler página')
```

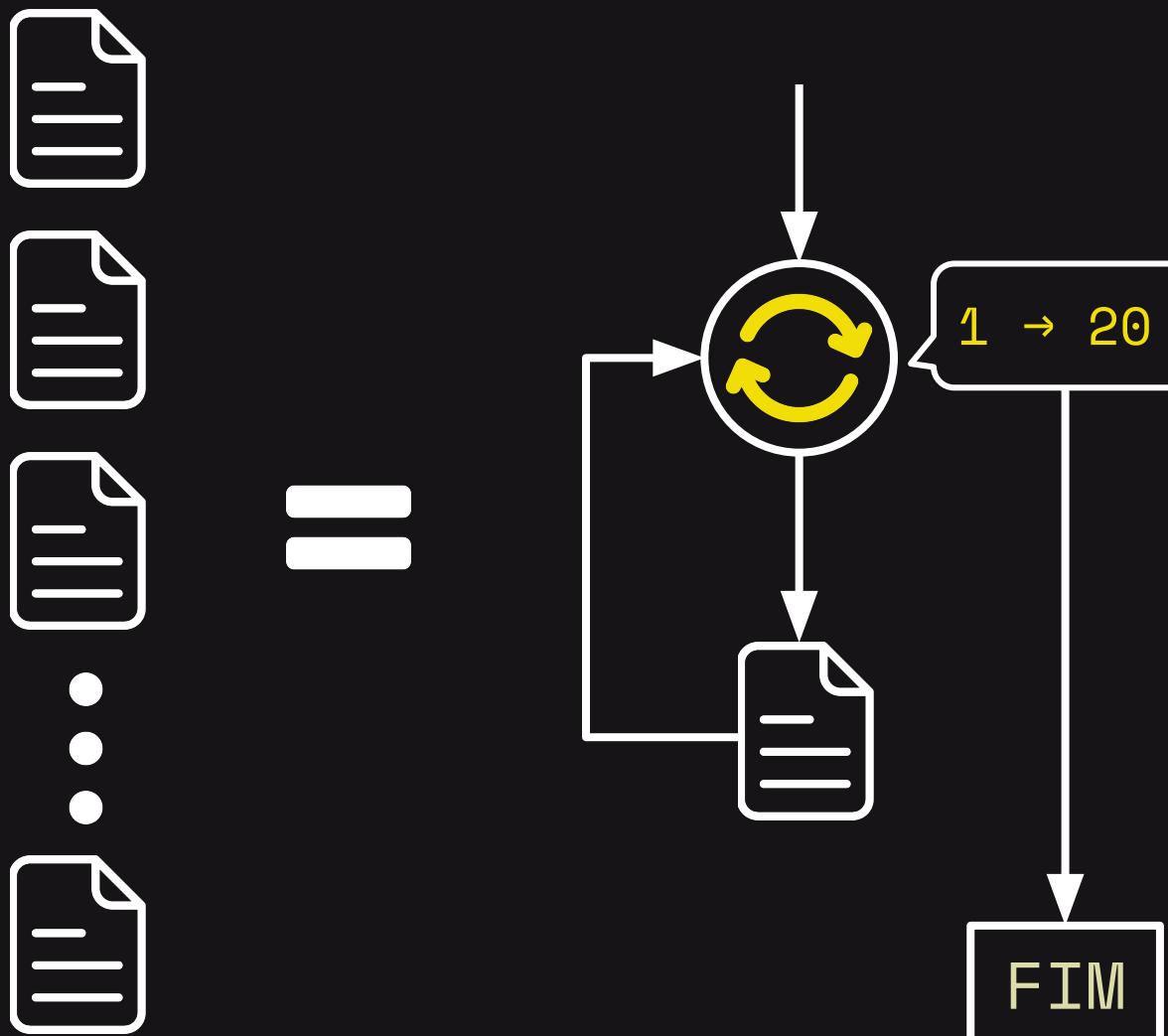
...

```
print('Ler página')
print('Ler página')
```

# O que é uma repetição?

É algo que precisamos executar várias vezes.

Exemplo: Ler um livro de 20 páginas



```
print('Ler página')
```

...

```
print('Ler página')
```

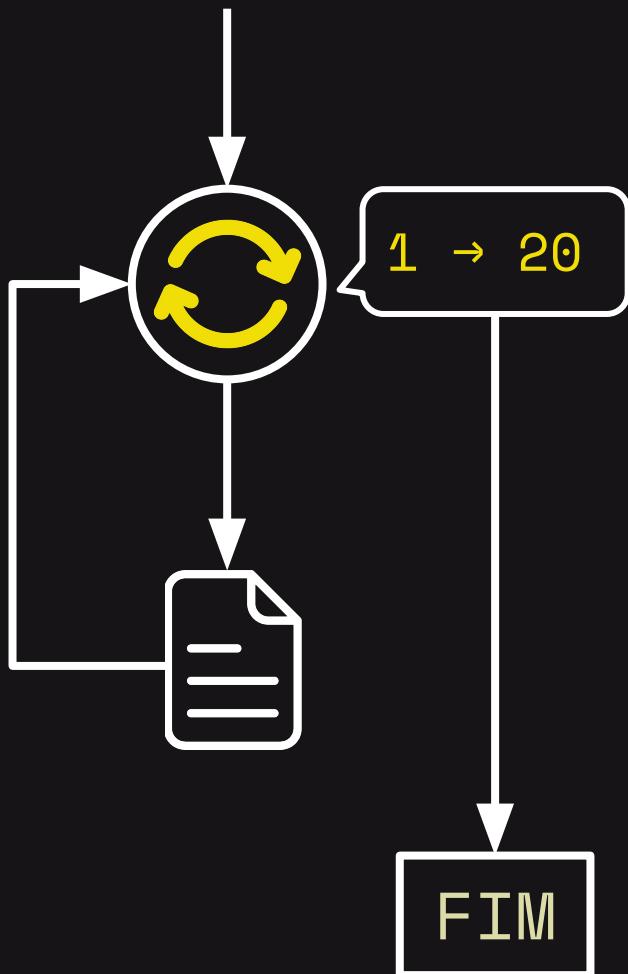
de 1 até 20

```
    print('Ler página')
```

```
print('Fim')
```

# Transformado em código

Exemplo: Ler um livro de 20 páginas



```
de 1 até 20  
print('Ler página')  
print('Fim')
```

```
for i in range(1,20):  
    print('Ler página')  
print('Fim')
```

# Entendendo a Estrutura do `for`

Exemplo: Ler um livro de 20 páginas

```
for i in range(1,20):  
    print('Ler página')  
    .....  
print('Fim')
```

`for` → inicio do loop  
`i` → variável de controle  
`range(1,20)` → variação do i  
1, 2, ..., 19, **20**

Bloco de ações repetitivas

Fim do loop

# Vamos rodar um código?

```
for i in range(1,6):
    print('i = {} Ler a página'.format(i))
print('Fim')
```

```
i = 1 Ler a página
i = 2 Ler a página
i = 3 Ler a página
i = 4 Ler a página
i = 5 Ler a página
Fim
```

# Modificando o

## range()

Estrutura:

`range(inicio, fim, passo)`

    inicio → número inteiro inicial

    fim → número inteiro final

    passo → quantia somada em cada loop

`range(1, 7, 1)` → 

1, 2, 3, 4, 5, 6, 7
---------------------

`range(10, 16, 2)` → 

10, 12, 14, 16
----------------

`range(5, 1, -1)` → 

5, 4, 3, 2, 1
---------------

# Exemplo

Exemplo: Some os números ímpares consecutivos iniciando em 1 e terminando em 9.

```
soma = 0
for i in range(1,9+1,2):
    soma = soma + i
    print('i = {} e soma = {}'.format(i,soma))
print('soma = {}'.format(soma))
```

```
i = 1 e soma = 1
i = 3 e soma = 4
i = 5 e soma = 9
i = 7 e soma = 16
i = 9 e soma = 25
soma = 25
```

## Exemplo 2

Exemplo: Somar 3 números digitados pelo usuário

```
soma = 0
for i in range(1,3+1):
    n = int(input('Digite um número: '))
    soma = soma + n
print('n = {} e soma = {}'.format(n,soma))
print('soma = {}'.format(soma))
```

Digite um número: 1

n = 1 e soma = 1

Digite um número: 5

n = 5 e soma = 6

Digite um número: 4

n = 4 e soma = 10

soma = 12

# Cálculos incrementais:

No mundo da programação é normal pegarmos um valor anterior e atualizarmos ele com alguma soma, subtração ou multiplicação. Como vimos nos 2 últimos exemplos.

Existe uma maneira mais resumida de fazer essa operação.

```
soma = 10
```

```
soma = soma + 1
```

→

```
soma += 1
```

soma = 11

```
soma = 10
```

```
soma = soma - 5
```

→

```
soma -= 5
```

soma = 5

# Cálculos incrementais:

No mundo da programação é normal pegarmos um valor anterior e atualizarmos ele com alguma soma, subtração ou multiplicação. Como vimos nos 2 últimos exemplos.

Existe uma maneira mais resumida de fazer essa operação.

```
soma = 10
```

```
soma = soma * 2
```

→

```
soma *= 2
```

soma = 20

anterior	=	incremento
anterior	-	incremento
anterior	*	incremento
anterior	/	incremento

anterior	=	incremento
anterior	-	incremento
anterior	*	incremento
anterior	/	incremento

**e se não soubermos**

**a quantidade de passos?**

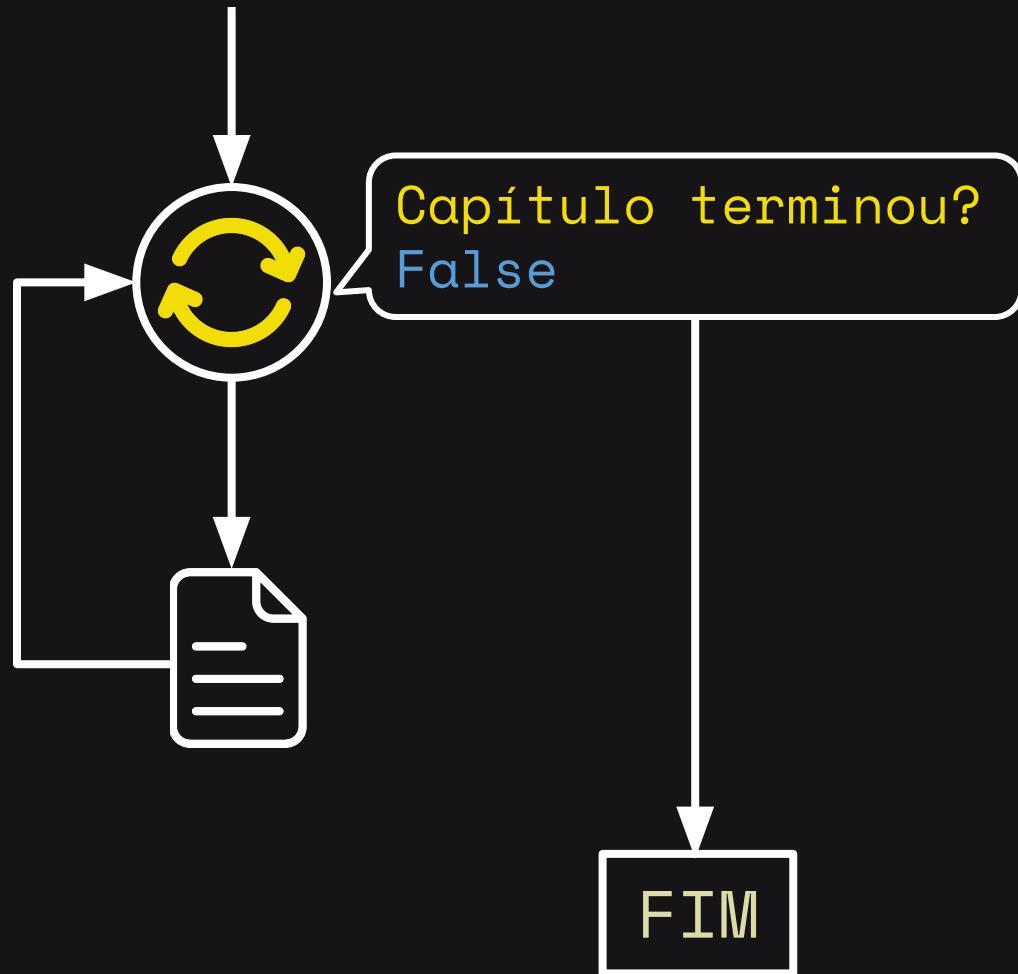
Como já vimos, o for é uma estrutura de repetição que possui um começo e um fim bem determinados.

Quando não sabemos onde termina um loop, precisamos usar outros métodos.

**usaremos o: while**

# Exemplo

Quero ler um capítulo de um livro, mas não sei quantas páginas possui este capítulo



```
print('Ler página')
```

...

```
print('Ler página')
```

```
enquanto False:
```

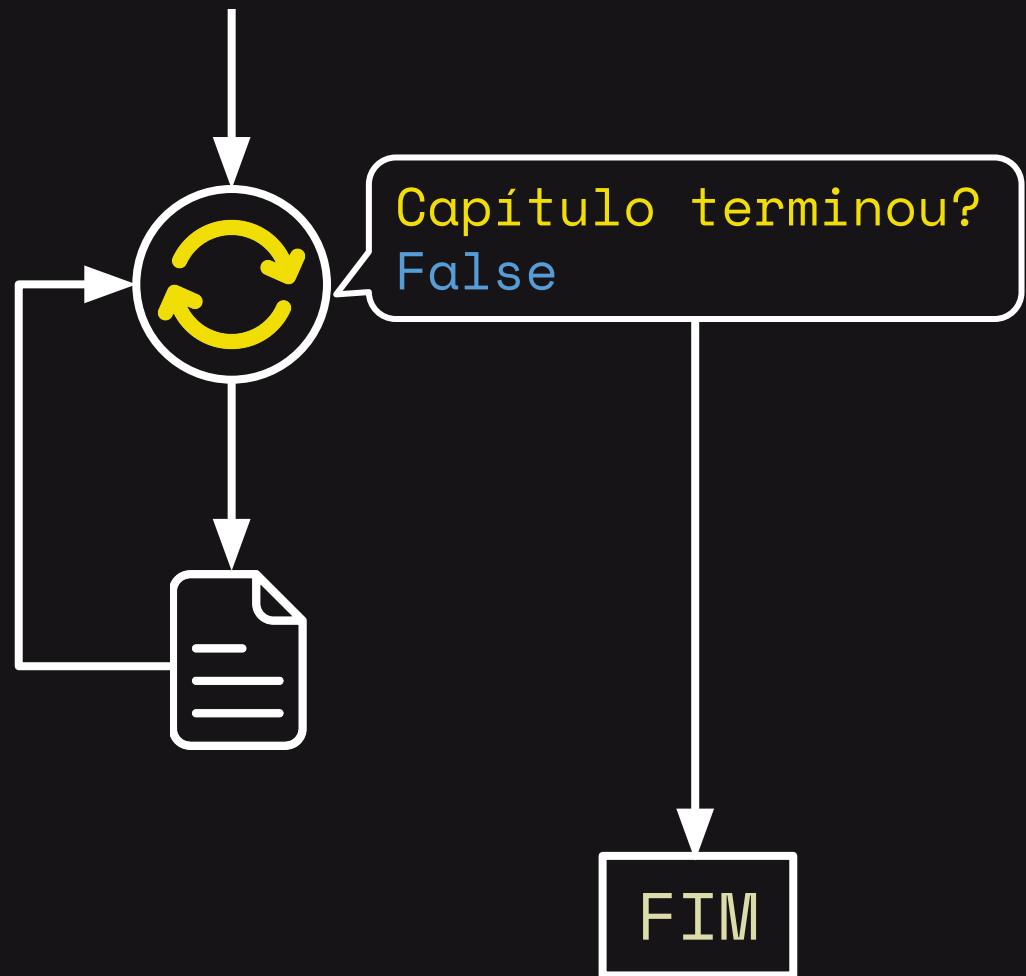
```
    terminou
```

```
    print('Ler página')
```

```
    print('Fim')
```

# Transformado em código

Quero ler um capítulo de um livro, mas não sei quantas páginas possui este capítulo



```
enquanto False:  
    terminou  
    print('Ler página')  
    print('Fim')
```

```
fim = False  
while fim == False:  
    print('Ler página')  
    fim = input('Terminou?')  
    print('Fim')
```

# Entendendo a Estrutura do while

Quero ler um capítulo de um livro, mas não sei quantas páginas possui este capítulo

```
fim = 'n'  
while fim == 'n':  
    print('Ler página')  
    fim = input('Terminou?')  
print('Fim')
```

Fim → declaração de variável de controle

while → inicio do loop  
fim → variável de controle  
== False → condição de teste

Bloco de ações repetitivas

Fim do loop

# Vamos rodar um código?

```
fim = 'n'  
while fim == 'n':  
    print('Ler página')  
    fim = input('Terminou?')  
print('Fim')
```

Ler a página  
Terminou? n  
Ler a página  
Terminou? n  
...  
Ler a página  
Terminou? s  
Fim

# **while**

## **com controle numérico**

**Exemplo:** Digite números quaisquer até a soma deles ultrapassar 20.

```
soma = 0
while soma < 20:
    n = float(input('Digite um número: '))
    soma += n
print('a soma final foi de {:.2f}'.format(soma))
```

# Vamos rodar um código?

```
soma = 0
while soma < 20:
    n = float(input('Digite um número: '))
    soma += n
print('a soma final foi de {:.2f}'.format(soma))
```

```
Digite um número: 5
Digite um número: 4
Digite um número: 1
Digite um número: 5
Digite um número: 6
a soma final foi de: 21.00
```

# Risco dos loops

## loop infinito

Exemplo: Vamos calcular a soma de números digitados pelo usuário

```
soma = 0
while True:
    n = float(input('Digite um número: '))
    soma += n
print('a soma final foi de {:.2f}'.format(soma))
```

```
Digite um número: 5
Digite um número: 4
Digite um número: 1
...
```

# comando break

O comando “break” serve para interromper e sair do loop assim que ele é usado.

```
soma = 0
while True:
    n = float(input('Digite um número: '))
    soma += n

    continua = input('Quer continuar (s/n): ')
    if continua == 'n':
        break

print('a soma final foi de {:.2f}'.format(soma))
```

# **Exemplos de uso**

- ❑ Caso o usuário tenha digitado algum valor que o programa não consiga compilar
- ❑ Caso o programa esteja tendendo para um resultado impossível (ML)
- ❑ Caso a condição principal do programa seja satisfeita antes do final do loop (ML)
- ❑ Cadastro de múltiplas entradas em bancos de dados
- ❑ Adição de itens no carrinho de compras numa loja online

# Estrutura de loop

```
passo = range(1,10)
for i in passo:
    print('For: Execute')
```

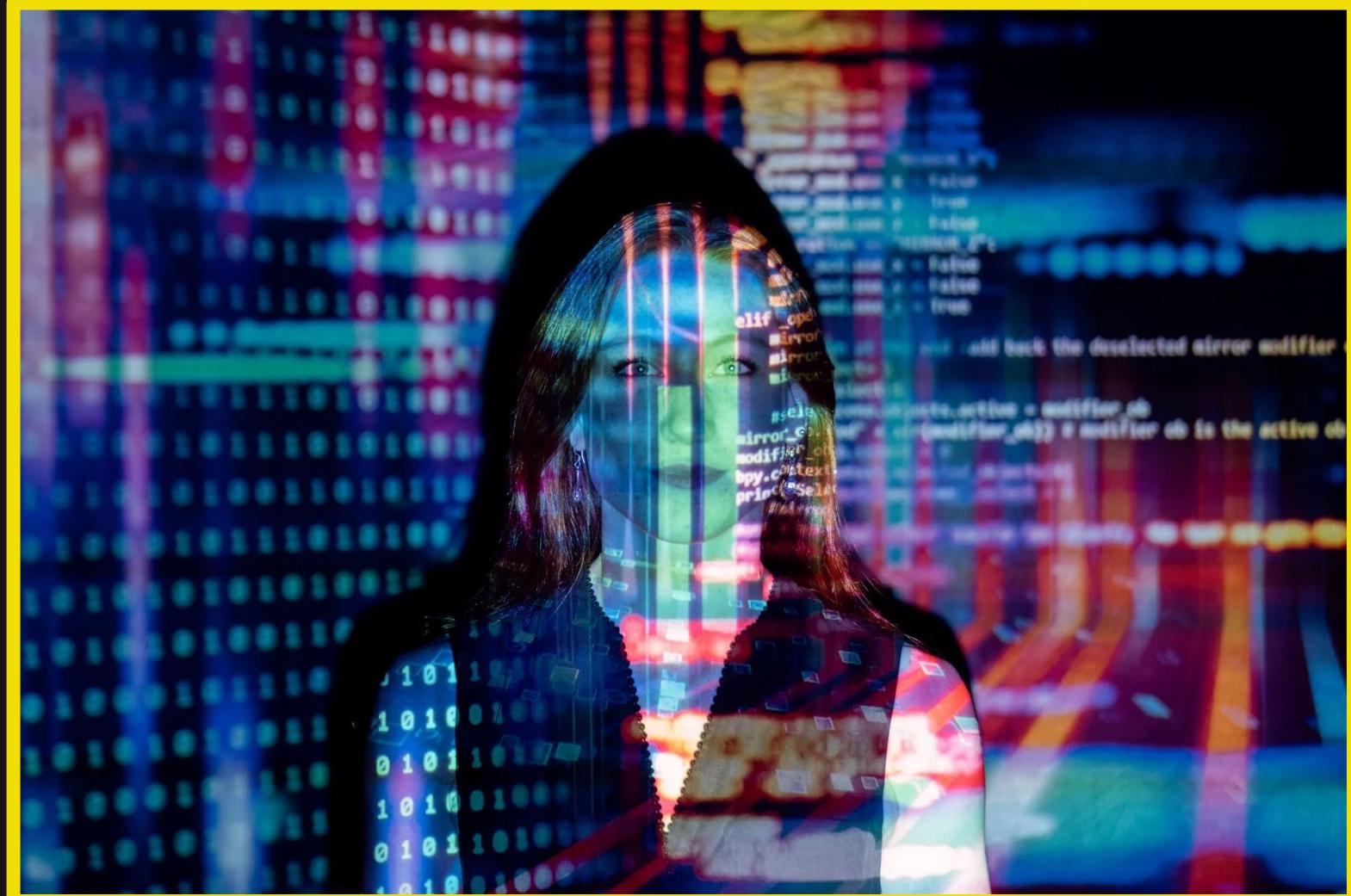
```
j = True
while j == True:
    print('While: Execute')
    j = False
```

## break

```
while j == True:
    ...
    if True:
        break
```

# Mão na Massa

Let's CODE



# Módulo 6: Variáveis Compostas - Tuplas

1

O que são variáveis compostas

2

O que são Tuplas

3

Algumas funções com tuplas

4

Tuplas dentro de loops

# Tuplas

As tuplas são variáveis compostas imutáveis

```
Tupla = ('var1', 'var2', 'var3')
```

# Variáveis simples

Variáveis simples guardam somente 1 informação por vez e ocupam somente 1 espaço na memória

```
var1 = 'Lanche'
```

```
[ 'Lanche' ]
```

```
var1 = 'Batata'
```

```
[ 'Batata' ]
```

```
var1 = 'Refri'
```

```
[ 'Refri' ]
```

# Variáveis compostas

Variáveis compostas guardam várias informações de uma só vez

Pos	[0]	[1]	[2]
var = [	‘Lanche’	‘Batata’	‘Refri’ ]

Acessando os valores:

```
var[0] == ‘Lanche’
```

```
var[1] == ‘Batata’
```

```
var[2] == ‘Refri’
```

# Tuplas

## Como declarar

```
var = [ 'Lanche', 'Batata', 'Refri' ]
```

Para declarar seus valores, basta colocar cada um separado por vírgula dentro de parênteses:

```
tupla = ('Lanche', 'Batata', 'Refri')
```

Após declarado, podemos acessar seus valores

# Acessando seus valores

```
tupla = ('Lanche', 'Batata', 'Refri')
```

Acessando a tupla completa:

```
print(tupla) ('Lanche', 'Batata', 'Refri')
```

Acessando uma posição específica:

```
print(tupla[0]) Lanche
```

```
print(tupla[1]) Batata
```

# Fatiamento

```
[0] [1] [2] [3] [4]  
tupla = (2, 6, 4, 5, 7)
```

Acessando termos consecutivos

print(tupla[0:2])	(2, 6)
print(tupla[1:4])	(6, 4, 5)

Acessando do termo N até o último e vice-versa

print(tupla[1:])	(6, 4, 5, 7)
print(tupla[:3])	(2, 6, 4)

Acessando os últimos termos

print(tupla[-1])	(7)
print(tupla[-3])	(4)

# Definição de Tupla

Tuplas são variáveis compostas imutáveis

Declarando:

```
tupla = ('Lanche', 'Batata', 'Refri')
```

Modificando:

```
tupla[1] = 'Milk Shake'  
tupla = ('Lanche', 'Milk Shake', 'Refri')
```

Após declarada, é impossível mudar uma tupla

# Tipos de dados

## dentro da Tupla

As variáveis compostas aceitam qualquer tipo de dados:

```
tupla1 = ('Lanche', 'Batata', 'Refrí')
tupla2 = (2, 6, 4, 5, 7)
tupla3 = (True, False, False)
```

Aceitam também tipos misturados de dados

```
tupla4 = ('Lanche', 10.9, False)
```

```
print(type(tupla4))
print(type(tupla4[0]))
print(type(tupla4[1]))
print(type(tupla4[2]))
```

```
<class 'tuple'>
<class 'str'>
<class 'float'>
<class 'bool'>
```

# Funções com Tuplas

```
t1 = (0, 6, 4, 2, 8)
```

```
t2 = (1, 3, 3, 5, 3, 5, 7)
```

Tamanho da Tupla

len(t1)	5
sorted(t1)	[0, 2, 4, 6, 8]

Ordenação

*Repare que está dentro de [ ] então o output não é uma tupla!*

Contagem termos

t2.count(3)	3
t2.index(7)	6
t2.index(3)	1
t2.index(3,3)	4

Posição de termos

Concatenar tuplas

t2 + t1	(1, 3, 3, 5, 3, 5, 7, 0, 6, 4, 2, 8)
---------	--------------------------------------

# Tuplas dentro de for

Até agora, rodamos nossos loopings dentro de um range. Porém podemos também rodar dentro de tuplas e listas:

```
t1 = (2, 6, 4, 5, 7)

for i in t1:
    print('i = {} - Valor da tupla'.format(i))
print('Fim')
```

```
i = 2 - Valor da tupla
i = 6 - Valor da tupla
i = 4 - Valor da tupla
i = 5 - Valor da tupla
i = 7 - Valor da tupla
Fim
```

# Tuplas dentro de for

E se a tupla só tiver texto?

```
t2 = ('Lanche', 'Batata', 'Refri')

for i in t2:
    print('Você quer pedir um(a) {}?'.format(i))
print('Fim')
```

Você quer pedir um(a) Lanche?

Você quer pedir um(a) Batata?

Você quer pedir um(a) Refri?

Fim

# Exemplo

Exemplo: Dada a tupla abaixo. Calcule a soma de seus valores.

```
t1 = (2, 6, 4, 5, 7)
```

```
soma = 0
for i in t1:
    soma += i
print('soma = {}'.format(soma))
```

```
soma = 24
```

# **E se quisermos mostrar o índice da tupla no for?**

Aí vamos usar a função enumerate()

```
t2 = ('Lanche', 'Batata', 'Refri')

for ind, valor in enumerate(t2):
    print('Item {} é um(a): {}'.format(ind, valor))
print('Fim')
```

```
Item 0: é um(a): Lanche
Item 1: é um(a): Batata
Item 2: é um(a): Refri
Fim
```

# Estrutura da Tupla

Tuplas são variáveis compostas imutáveis

```
tupla = ('var1', 2, True)
```

```
tupla[0]    tupla[0:2]    tupla[:2]    tupla[1:]
```

## Funções

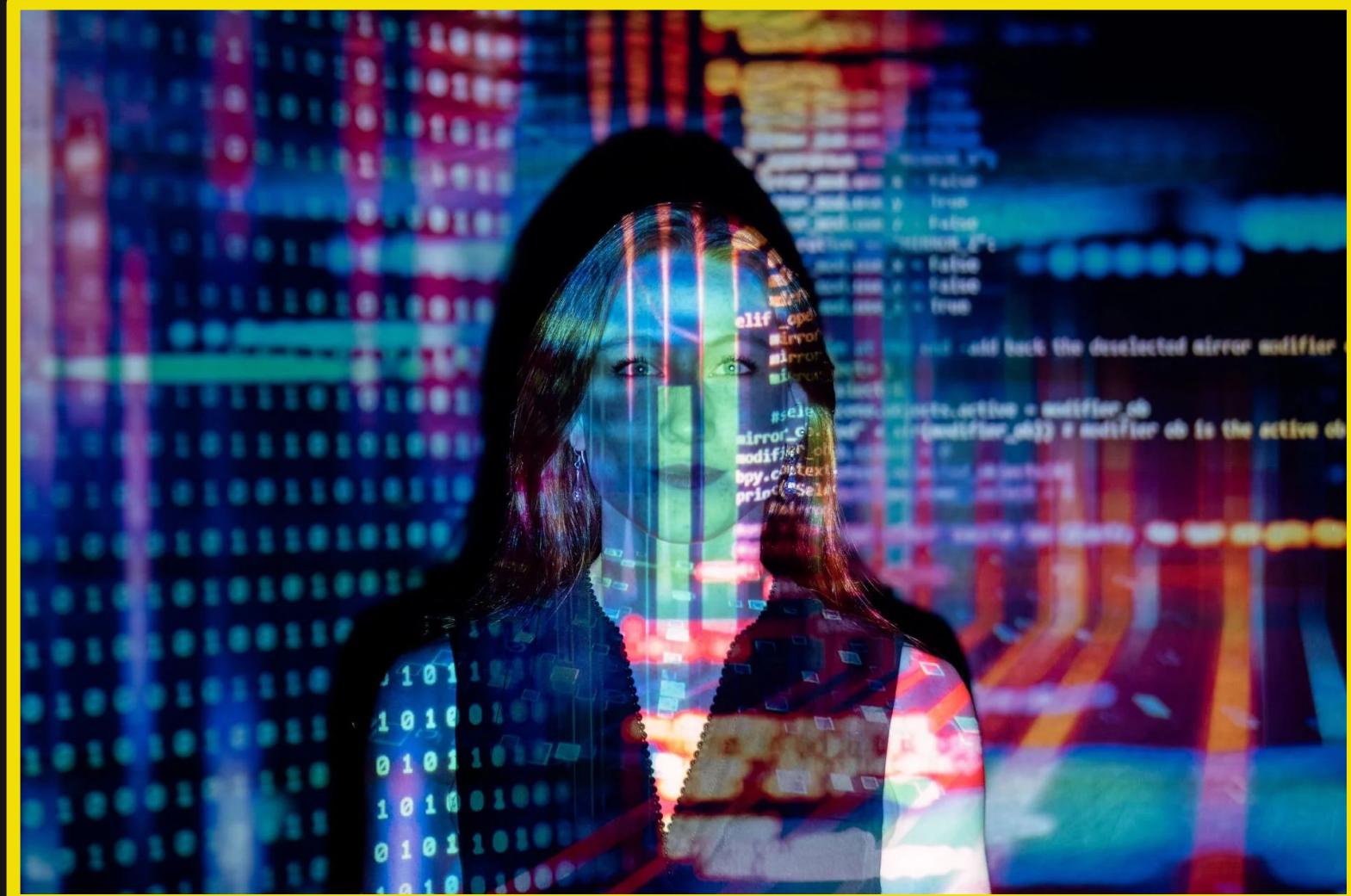
len(t1)	→ qtd de termos
sorted(t1)	→ ordenar
t1.count(valor)	→ contar a qtd de “valor”
t1.index(valor)	→ achar a 1ª posição de “valor”
t1.index(valor, pos)	→ achar “valor” a partir da “pos”
t1 + t2	→ concatenar tuplas

## Tupla dentro de for

```
for i in t1:  
    print('ações')
```

# Mão na Massa

Let's CODE



# Módulo 7: Variáveis Compostas - Listas



# Listas

As listas são variáveis compostas mutáveis

```
Lista = [ 'var1' , 'var2' , 'var3' ]
```

# Estrutura das Listas

Para declarar seus valores, basta colocar cada um separado por vírgula dentro de colchetes:

Pos	[0]	[1]	[2]
	-----	-----	-----
var = [	'Lanche'	'Batata'	'Refri'
	-----	-----	-----

## Tuplas:

```
tupla = ('Lanche', 'Batata', 'Refri')
```

## Listas:

```
lista = ['Lanche', 'Batata', 'Refri']
```

# Listas vs Tuplas

Todas as técnicas que vimos até agora funcionam tanto com **Listas** quanto com **Tuplas**:

- ❑ Acessar itens: [0] [0:2] [:3] [3:] [-1]
- ❑ Métodos: .index( ) - .count( )
- ❑ Funções: len( ) - sorted( )
- ❑ Concatenar: lista1 + lista2
- ❑ Lista dentro de for: for i in lista:  
                          for i,v in enumerate(lista):

Além de tudo isso, podemos modificar elementos, deletar, adicionar e muito mais!!!

# Funções com Listas

Como já falamos a maior vantagem das listas é poderem ser modificadas. Vamos ver as principais modificações:

```
l1 = [2, 6, 4, 5, 7]
```

## ❑ Modificando um termo

```
l1[2] = 3
```

```
l1 = [2, 6, 3, 5, 7]
```

```
l1[0:2] = [0,1]
```

```
l1 = [0, 1, 3, 5, 7]
```

# Funções com Listas

```
l1 = [0, 1, 3, 5, 7]
```

## □ Adicionando termos

- .append( )

```
l1.append(7)
```

```
l1 = [0, 1, 3, 5, 7, 7]
```

- .insert( pos, valor )

```
l1.append( 3, 4 )
```

```
l1 = [0, 1, 3, 4, 5, 7, 7]
```

# Funções com Listas

```
l1 = [0, 1, 3, 4, 5, 7, 7]
```

## □ Deletando termos

□ del l1[ pos ]

```
del l1[3]
```

```
l1 = [0, 1, 3, 5, 7, 7]
```

□ var = lista.pop( pos )

```
var = l1.pop(2)
```

```
l1 = [0, 1, 5, 7, 7]           var = 3
```

□ .remove( valor )

```
l1.remove(7)
```

```
l1 = [0, 1, 5, 7]
```

# **if com lista**

```
l1 = [0, 1, 3, 4, 5, 7, 7]
```

Quando deletamos itens de listas, é comum aparecer mensagem de erro, caso aquele item não exista. Para isso existe um método simplificado de aplicarmos o if:

```
if valor in lista:  
    lista.remove(valor)
```

**Exemplo:**

```
if 3 in l1:  
    lista.remove(3)  
  
l1 = [0, 1, 4, 5, 7, 7]
```

# Funções com Listas

```
l1 = [0, 2, 6, 4, 8, 1, 3]
```

## ❑ Ordenando a lista

❑ sorted( lista )

```
sorted(l1)
```

```
l1 = [0, 1, 2, 3, 4, 6, 8]
```

❑ .sort( )

```
.sort()
```

```
l1 = [0, 1, 2, 3, 4, 6, 8]
```

```
.sort(reverse = True)
```

```
l1 = [8, 6, 4, 3, 2, 1, 0]
```

# Declarando

## Listas em branco

É muito comum declararmos uma lista em branco para que depois o usuário vá adicionando novos termos.

```
lista = []
lista = list()
```

```
lista.append(novo valor)
```

**Criando uma lista com range:**

```
lista = list( range(inicio, fim, passo) )
```

# Copiando e Clonando Listas

```
l1 = [0, 2, 6, 4, 8, 1, 3]
```

Quando declaramos uma nova variável e igualamos a uma lista, ela na verdade vira um “clone” da lista original. Qualquer modificação em uma das duas, irá impactar as duas.

```
l2 = l1  
l2.pop()
```

```
print(f'l1 = {l1}')  
print(f'l2 = {l2}')
```

```
l1 = [0, 2, 6, 4, 8, 1]  
l2 = [0, 2, 6, 4, 8, 1]
```

# Copiando e Clonando Listas

```
l1 = [0, 2, 6, 4, 8, 1, 3]
```

E se quisermos fazer somente uma cópia da lista e deixarmos as 2 independentes? Então temos que mudar a maneira de declarar a segunda lista.

```
l2 = l1[:]  
l2.pop()
```

```
print(f'l1 = {l1}')  
print(f'l2 = {l2}')
```

```
l1 = [0, 2, 6, 4, 8, 1, 3]  
l2 = [0, 2, 6, 4, 8, 1]
```

# Exemplos

## Listas dentro de for

Exemplo 1: Dada as listas abaixo, some os seus itens

```
l1 = [0, 2, 6, 4, 8, 1, 3]
```

```
l2 = [3, 2, 1, 7, 8, 2, 5]
```

```
l1 = [0, 2, 6, 4, 8, 1, 3]
l2 = [3, 2, 1, 7, 8, 2, 5]
l3 = []
```

```
for i, valor in enumerate(l1):
    l3.append(l1[i] + l2[i])
print(l3)
```

```
[3, 4, 7, 11, 16, 3, 8]
```

# Exemplos

Exemplo 2: Faça o mesmo para essas listas:

```
l1 = [0, 2, 6, 4, 8, 1, 3]
```

```
l2 = [3, 2, 1, 7, 8]
```

```
l1 = [0, 2, 6, 4, 8, 1, 3]
```

```
l2 = [3, 2, 1, 7, 8]
```

```
l3 = []
```

```
for i, valor in enumerate(l1):  
    l3.append(l1[i] + l2[i])  
print(l3)
```

# Função zip

O erro anterior acontece porque o programa tenta acessar uma posição dentro de uma lista que não existe. Para contornarmos este problema temos o zip.

```
for val1, val2 in zip(l1, l2):
```

```
l1 = [0, 2, 6, 4, 8, 1, 3]
l2 = [3, 2, 1, 7, 8]
l3 = []

for v1, v2 in zip(l1, l2):
    print(f'v1 = {v1} e v2 = {v2}')
    l3.append(v1 + v2)
print(l3)
```

```
v1 = 0, v2 = 3
v1 = 2, v2 = 2
v1 = 6, v2 = 1
v1 = 4, v2 = 7
v1 = 8, v2 = 8
[3, 4, 7, 11, 16]
```

# Estrutura das Listas

Listas são variáveis compostas **mutáveis**

l1 = ['var1', 2, True]

l1[pos] = valor

l1 = [] l1 = list()

l1 = list(range(i,f,p))

l1 = l2 → clone

l1 = l2[:] → cópia

## Funções

del l1[pos]

→ deleta item da “posição”

var = l1.pop(pos)

→ deleta da “pos” e salva em “var”

l1.remove(valor)

→ deleta o 1º “valor” da lista

l1.append(valor)

→ adiciona “valor” no final

l1.insert(pos,valor)

→ adiciona “valor” na “posição”

l1.sort(reverse)

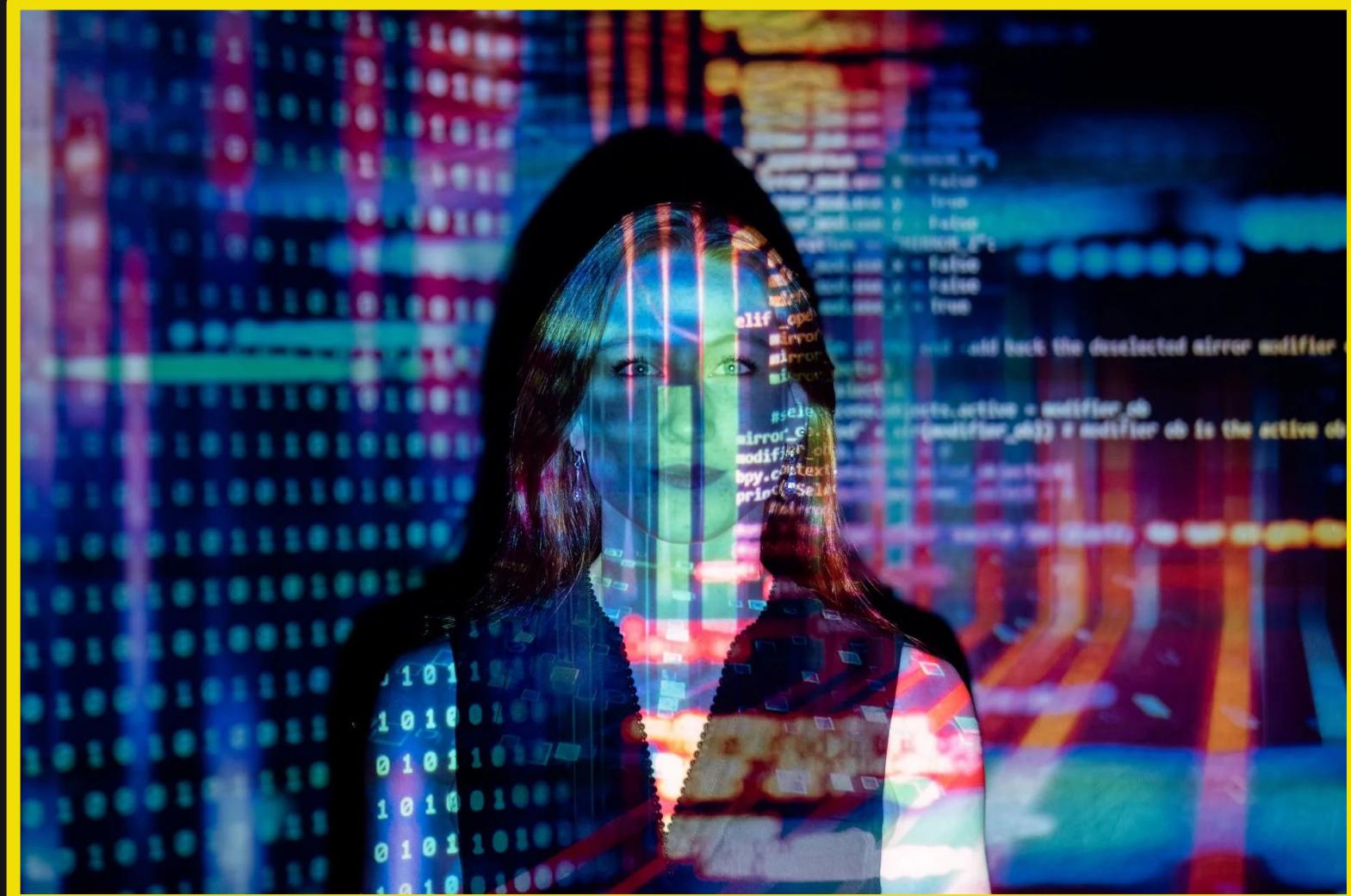
→ ordena lista True/False

## Lista dentro de for

```
for v1, v2 in zip(l1, l2):  
    print('ações')
```

# Mão na Massa

Let's CODE



# Listas dentro de Listas

As listas aceitam qualquer tipo de dados,  
até outras listas.



```
Lista = [ [ 'v1' ,1] , [ 'v2' ,2] , [ 'v3' ,3] ]
```

# Estrutura das

## Listas dentro de Listas



```
lanche = [ 'Lanche' , 10.9 ]
```

```
batata = [ 'Batata' , 5.5 ]
```

```
refri = [ 'Refri' , 3.9 ]
```

```
cardapio = [ lanche , batata , refri ]
```

# Estrutura composta



```
lanche = ['Lanche', 10.9]
batata = ['Batata', 5.5]
refri = ['Refri', 3.9]

cardapio = [lanche, batata, refri]

cardapio = [[ 'Lanche' , 10.9] ,
            [ 'Batata' , 5.5] ,
            [ 'Refri' , 3.9]]
```

# Acessando os valores



```
cardapio[0]      == ['Lanche', 10.9]
```

```
cardapio[0][1] == 10.0
```

```
cardapio[2]      == ['Refri', 3.9]
```

```
cardapio[2][0] == 'Refri'
```

*Obs.: este tipo de estrutura é  
uma introdução a dicionários*

# Declarando

## Listas compostas

### ❑ Declaração por partes:

```
lanche = [ 'Lanche' , 10.9]
batata = [ 'Batata' , 5.5]
refri = [ 'Refri' , 3.9]
cardapio = [lanche, batata, refri]
```

### ❑ Declaração direta:

```
cardapio = [ [ 'Lanche' , 10.9] , [ 'Batata' , 5.5] ,
[ 'Refri' , 3.9] ]
```

# **Declarando**

## **Listas compostas**

### **❑ Declaração com append:**

```
cardapio = []
comida = []
comida.append('Lanche')
comida.append(10.9)
cardapio.append(comida[:])
```

# Declarando

## Listas compostas

### ❑ Declaração com append e input do usuário:

```
cardapio = []
comida = []
for c in range(0,3):
    comida.append(str(input('Nome da comida: ')))
    comida.append(float(input('Preço da comida: ')))
cardapio.append(comida[:])
comida.clear()      # limpando a memória da variável
print(cardapio)
```

# Acessando os valores

## ❑ Acesso as listas:

```
cardapio[0] == ['Lanche', 10.9]
```

```
cardapio[1] == ['Batata', 5.5]
```

```
cardapio[2] == ['Refri', 3.9]
```

## ❑ Acesso aos valores da lista:

```
cardapio[0][1] == 10.9
```

```
cardapio[1][0] == 'Batata'
```

```
cardapio[2][0] == 'Refri'
```

# Acessando os valores

## □ Acesso os valores dentro de um for

```
for c in cardapio:  
    print(f'c = {c}, c[0] = {c[0]}, c[1] = {c[1]}')
```

```
c = ['lanche', 10.9], c[0] = lanche, c[1] = 10.9  
c = ['batata', 5.5], c[0] = batata, c[1] = 5.5  
c = ['refri', 3.5], c[0] = refri, c[1] = 3.5
```

# Estrutura Listas Compostas

```
composta = [[ 'v0' , 0 ] , [ 'v1' , 1 ] , [ 'v2' , 2 ] ]  
composta[0] == [ 'v0' , 0 ] → composta[0][0] == 'v0'  
composta[1] == [ 'v1' , 1 ] → composta[1][0] == 'v1'
```

```
for c in composta:  
    c[0] == [ 'v0' , 0 ] → c[1] == [ 'v1' , 1 ] ...
```

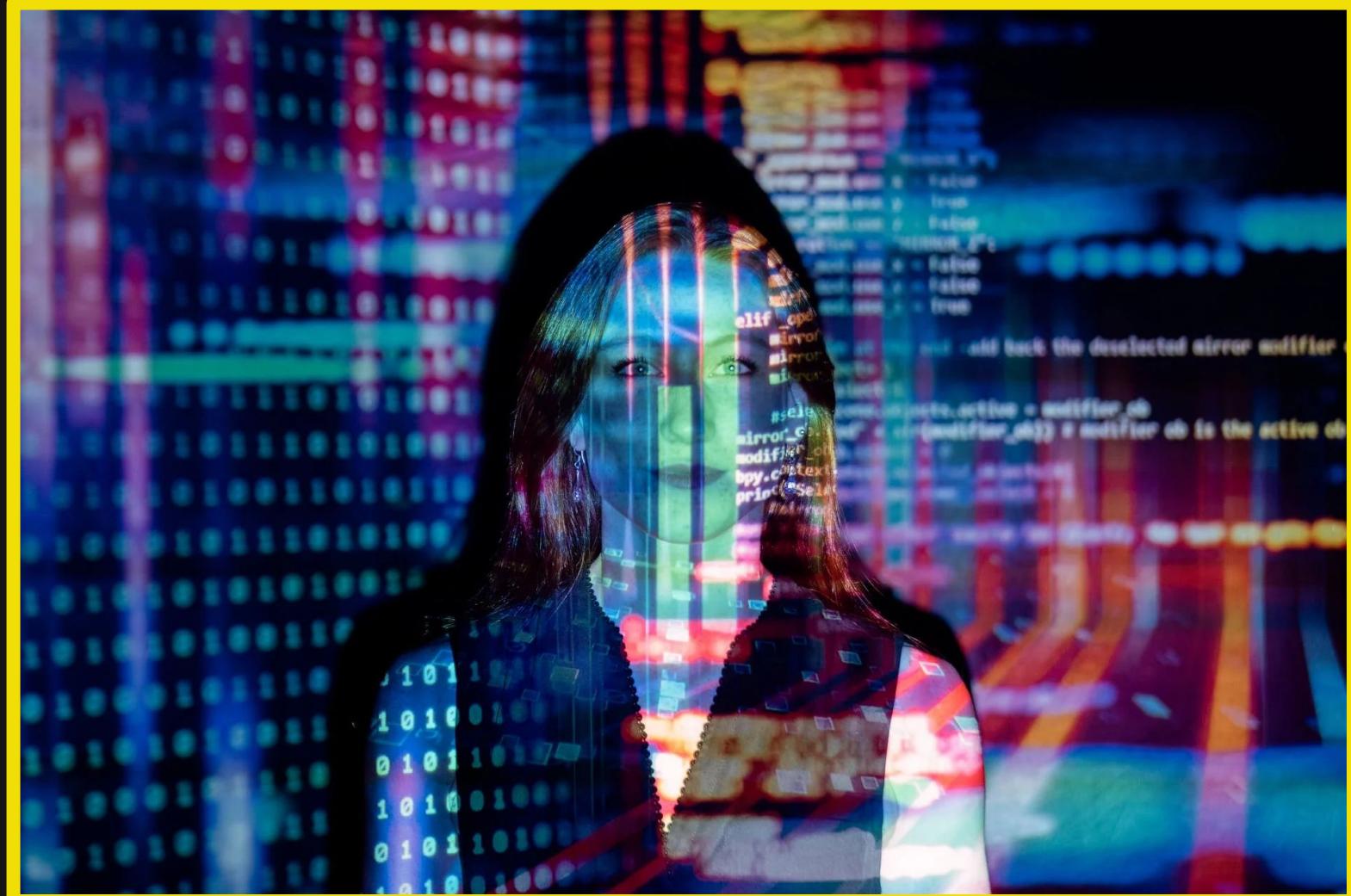
```
for c in range(i,f):  
    l.append((input('l[0]: ')))  
    l.append((input('l[1]: ')))  
    composta.append(l[:])  
    l.clear()
```

# Funções

```
var.clear() → limpa a memória da variável
```

# Mão na Massa

Let's CODE



# Módulo 7: Dicionário

**1**

O que são dicionários

**2**

Funções com dicionários

**3**

Dicionários dentro de Loops

**4**

Dicionários dentro de listas

# Dicionários

Dicionários são listas com índices textuais

```
dic = { 'chave1': 'valor' ,  
        'chave2': 'valor2' ,  
        'chave3': 'valor3' }
```

# Dicionários vs Listas

Dicionários são estruturas de dados, semelhantes a listas, que as posições ao invés de serem numéricas são chaves.

	nome:	idade:	cidade:
Chaves =			
dicio = [	‘Everton’	29	‘São José’
Pos	[0]	[1]	[2]

**Lista:**

```
lista = [ ‘Everton’ , 29 , ‘São José’ ]
```

**Dicio:**

```
dicio = { ‘nome’ : ‘Everton’ ,  
          ‘idade’ : 29 ,  
          ‘cidade’ : ‘São José’ }
```

# Estrutura dos Dicionários

Existem 3 pontos importantes na estrutura de um dicionário:



# Manipulando os dados

```
dicio = {}           dicio = dict()  
dicio = { 'nome' : 'Everton',  
          'idade' : 29,  
          'cidade' : 'São José' }
```

□ Acessando os valores:      `dicionario[ 'chave' ] == value`

```
dicio[ 'nome' ] == 'Everton'  
dicio[ 'idade' ] == 29  
dicio[ 'cidade' ] == 'São José'
```

□ Modificando os valores:      `dicionario[ 'chave' ] = value`

```
dicio[ 'nome' ] = 'Thiago'
```

□ Criando nova chave:      `dicionario[ 'chave' ] = value`

```
dicio[ 'peso' ] = 65
```

# Manipulando os dados

```
dicio = { 'nome' : 'Everton',  
          'idade' : 29,  
          'cidade' : 'São José' }
```

## ❑ Acessando o dicionário:

### ❑ Completo: dionario

```
print(dicio)
```

```
{'nome': 'Thiago', 'idade': 29, 'cidade': 'São José'}
```

### ❑ Valores: dionario.values()

```
print(dicio.values())
```

```
dict_values(['Everton', 29, 'São José'])
```

### ❑ Chaves: dionario.keys()

```
print(dicio.keys())
```

```
dict_keys(['nome', 'idade', 'cidade'])
```

# Manipulando os dados

```
dicio = { 'nome' : 'Everton',  
          'idade' : 29,  
          'cidade' : 'São José' }
```

## □ Acessando o dicionário:

### □ Items: dionario.items()

```
print(dicio.items())  
dict_items([('nome', 'Everton'), ('idade', 29),  
           ('cidade', 'São José')])
```

## □ Deletando os valores: del dionario['chave']

```
del dicio['cidade']  
print(dicio)  
{'nome': 'Everton', 'idade': 29}
```

# Dicionários

## dentro de for

Como dicionários são muito similares a listas, também podemos passar por eles dentro de for:

```
for v in dicio.values():    Values: Everton  
    print(f'Values: {v}')      Values: 29
```

```
for k in dicio.keys():    Keys: nome  
    print(f'Keys: {k}')       Keys: idade
```

```
for k, v in dicio.items():  k:nome e v:Everton  
    print(f'k:{k} e v:{v}')   k:idade e v:29
```

# Dicionários

## dentro de listas

Como dicionários são muito similares a listas, também podemos passar por eles dentro de for:

```
pessoas = [ { nome: 'Everton', idade: 29, cidade: 'São José' }, { nome: 'Thiago', idade: 34, cidade: 'São Paulo' }, { nome: 'André', idade: 22, cidade: 'Lorena' } ]
```

```
[ { 'nome': 'Everton', 'idade': 29, 'cidade': 'São José' },  
  { 'nome': 'Thiago', 'idade': 34, 'cidade': 'São Paulo' },  
  { 'nome': 'André', 'idade': 22, 'cidade': 'Lorena' } ]
```

# Acessando os dados

```
pessoas =  
[ { 'nome': 'Everton', 'idade': 29, 'cidade': 'São José' } ,  
{ 'nome': 'Thiago', 'idade': 34, 'cidade': 'São Paulo' } ,  
{ 'nome': 'André', 'idade': 22, 'cidade': 'Lorena' } ]
```

- ❑ **For dentro de For:** Primeiro acesse o mais externo para depois o mais interno.

```
for p in pessoas:  
    for k, v in p.items():  
        print(f'Key: {k} , Value: {v}')
```

# Inserindo dados

❑ **Método .copy():** Quando queremos inserir um dicionário dentro de uma lista, não podemos usar o método do fatiamento para fazer a cópia da lista temporária, então faremos:

```
dicio = {}  
lista = []  
for d in range(ini, fim):  
    dicio[ 'Key1' ] = input('Valor1 = ')  
    dicio[ 'Key2' ] = input('Valor2 = ')  
    lista.append(dicio.copy())
```

# Estrutura dos Dicionários

Dicionários são listas com índices textuais

dicio = {}	dicio = dict()
dicio = { 'key1': 'value', 'key1': 'value'}	
dicio['key1'] == 'value1'	dicio.values() → valores
dicio.keys() → chaves	dicio.items() → items
del l1[pos]	→ deleta item da “posição”

## Listas dentro de for

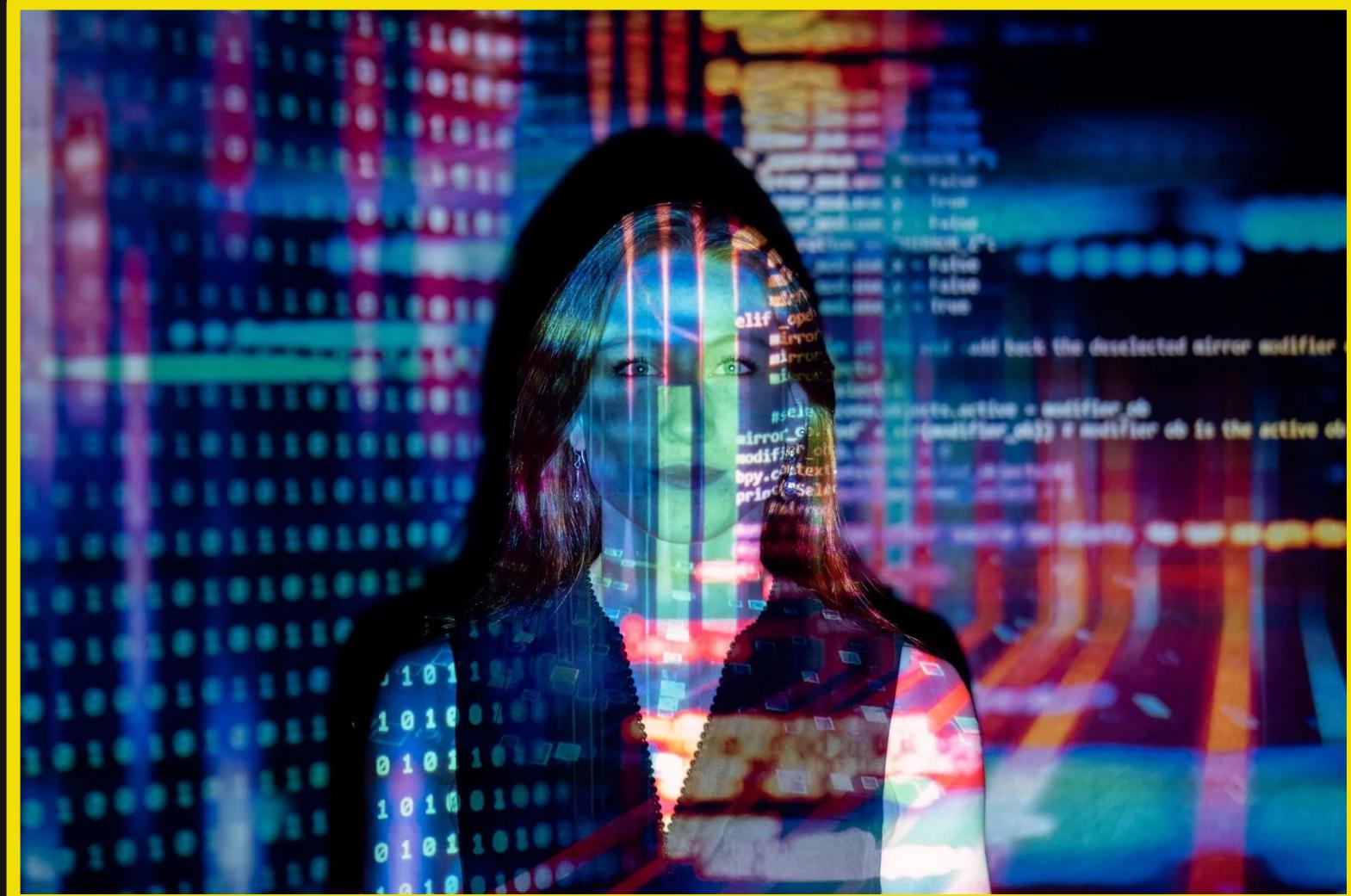
```
for k, v in dicio.items():
    print(f'k: {k} e v: {v}')
```

```
for dicio in listas:
    for k, v in dicio.items():
        print(f'Key: {k}, Value: {v}')
```

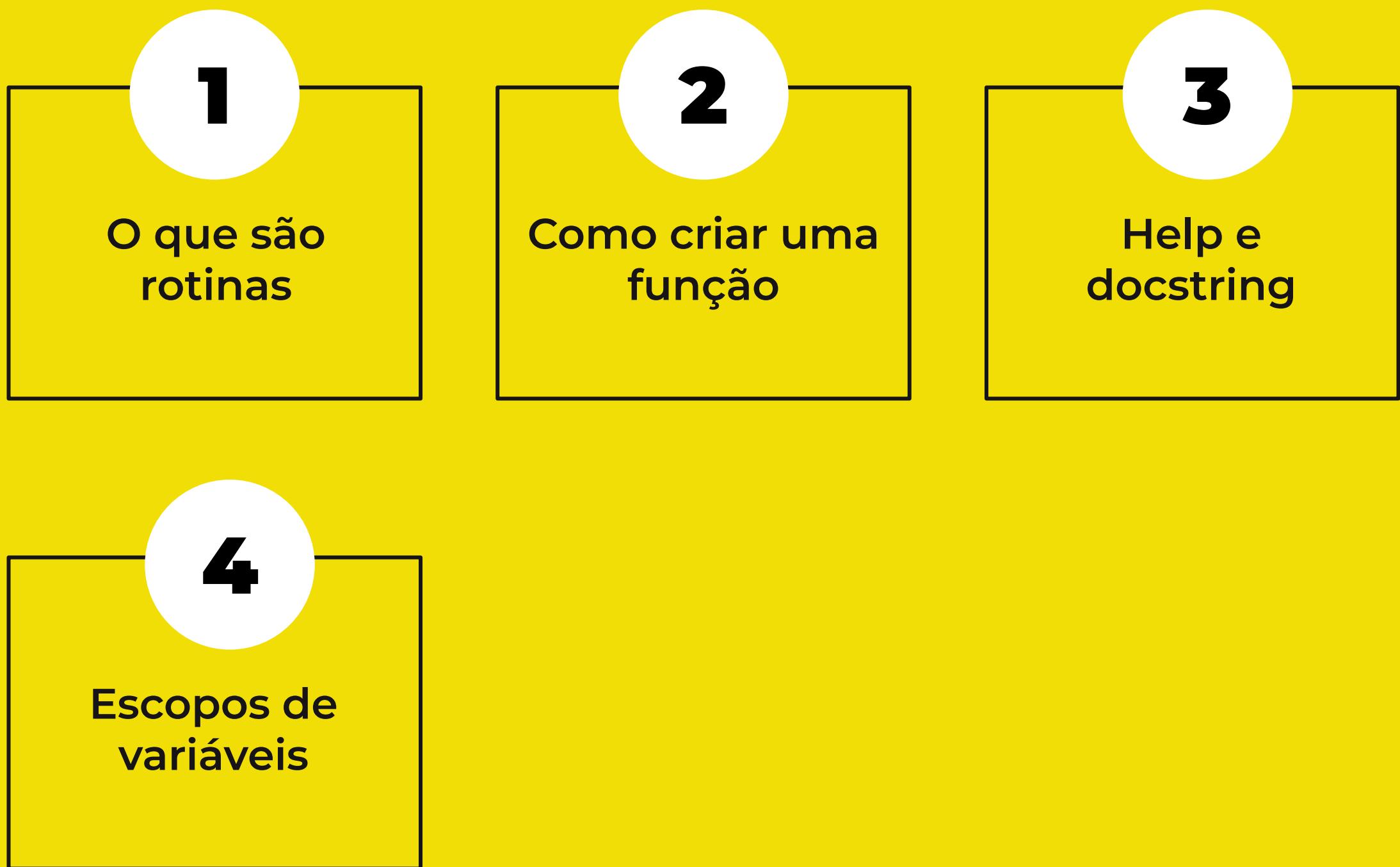
```
lista.append(dicio.copy())
```

# Mão na Massa

Let's CODE



# Módulo 8: Funções



# Funções

Funções são rotinas, ações que são executadas várias vezes nos programas.

```
def funcao():  
    print('Função')
```

```
funcao()
```

# Rotinas

Rotinas são tarefas repetitivas, que executamos várias vezes, sempre da mesma maneira.



# **Rotina na programação**

**Rotina em programação é tudo que é feito várias vezes:**

- No mesmo programa**
- Programas diferentes**

## **Funções Nativas**

**As funções nativas do Python nada mais é do que rotinas criadas por outros desenvolvedores que são utilizadas por milhões de outros desenvolvedores:**

- print( ), len( ), range( ), int( ), float( )**
- bibliotecas**

# Como criar uma função

```
def funcao():
    print('Função')

funcao()
```

Toda função vem no começo do programa e é declarada por `def`. Durante o programa principal, toda vez que você precisa usar essa função você “chama” ela e executa essa subrotina.

- ❑ Imagina ter que codar o `print` toda vez?
- ❑ Imagina ter que criar um `for` toda vez para contar a quantidade de posições em uma lista? (`len`)

# Estrutura das funções

```
def funcao(par):  
    print('Função')  
  
# Programa Principal  
funcao()
```

def → inicio da função  
funcao(par) → nome da função  
par → parâmetros de entrada

Ações executadas dentro da função / subrotina

Espaço em branco = fim das subrotinas

Chamada da função no programa principal

As funções podem ou não ter parâmetros de entrada e podem ou não ter parâmetros de saída.

# Algumas funções

## Funções sem parâmetro de entrada:

Faça uma função que imprima “Olá Mundo”

```
# Função 1:  
def ola():  
    print('Olá Mundo')
```

```
# Programa Principal
```

```
ola()
```

```
...  
ola()
```

```
...  
ola()
```

```
ola()
```

Olá Mundo

Olá Mundo

Olá Mundo

# Algumas funções

## Funções com parâmetro de entrada:

Faça uma função `soma a + b`:

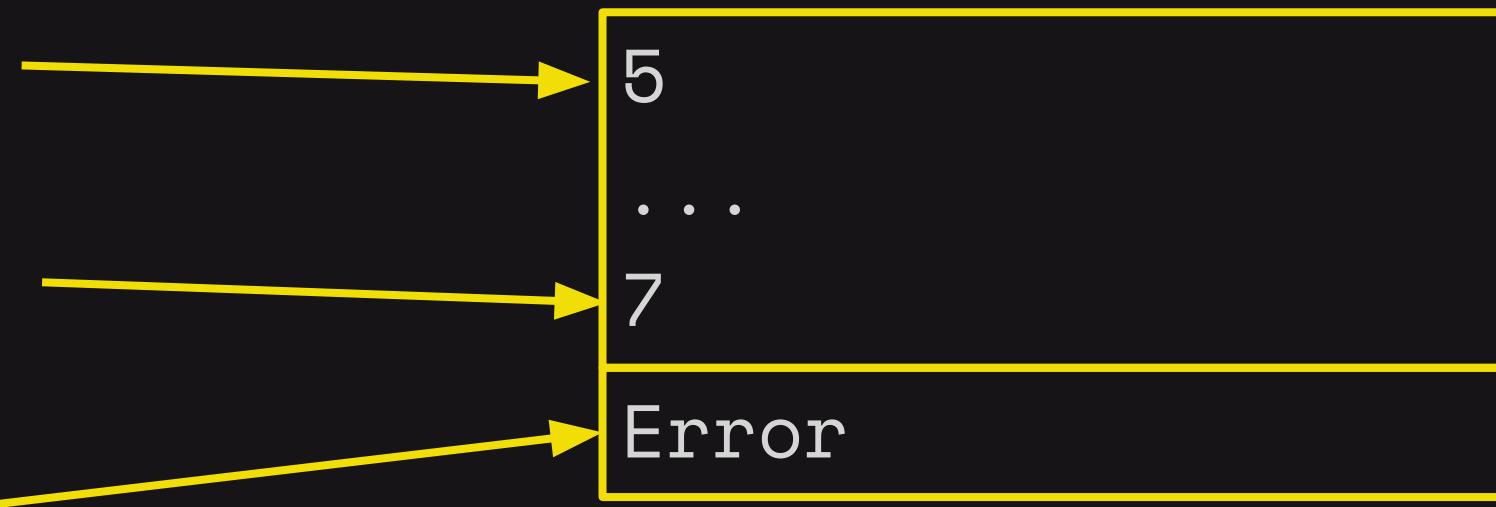
```
# Função 2:  
def soma(a, b):  
    print(a + b)
```

```
# Programa Principal
```

```
soma(2, 3)
```

```
...  
soma(3, 4)
```

```
...  
soma(-2)
```



# Algumas funções

- Funções com parâmetro opcionais:  
Faça uma função some a + b:

```
# Função 2:  
def soma(a=0, b=0):  
    print(a + b)
```

```
# Programa Principal
```

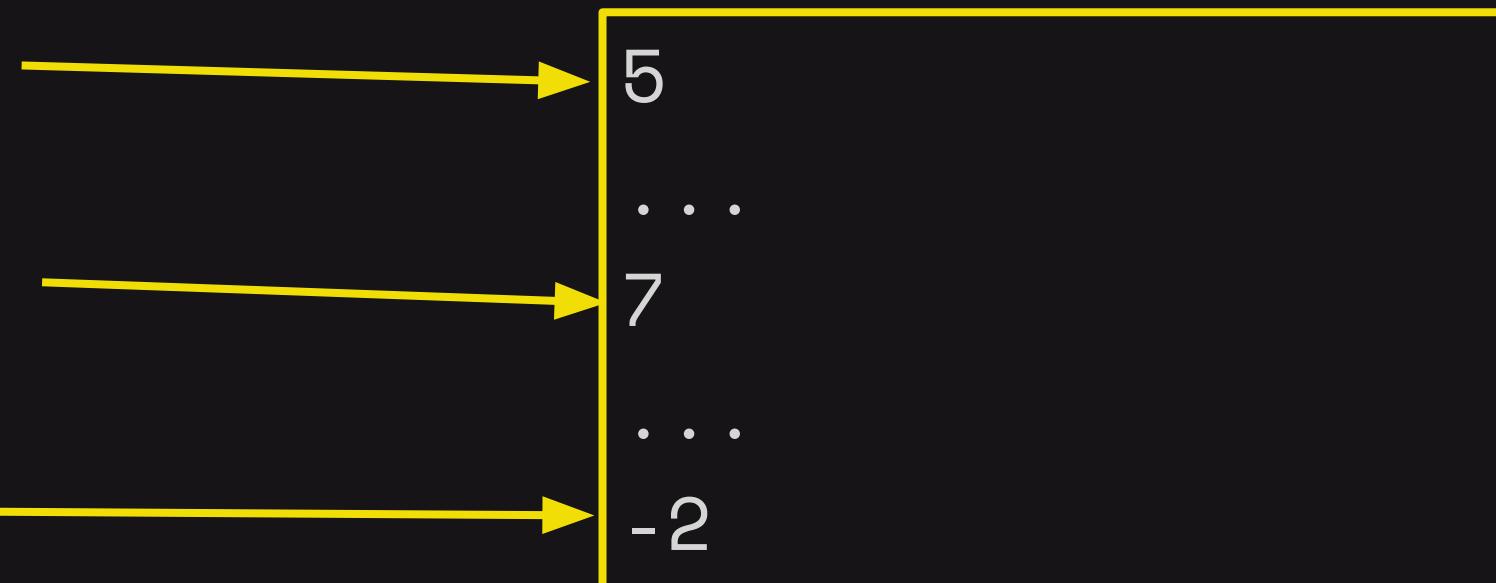
```
soma(2, 3)
```

```
...
```

```
soma(3, 4)
```

```
...
```

```
soma(-2)
```



# Algumas funções

## ❑ Funções com saída (retorno):

Faça uma função some a + b:

```
# Função 2:  
def soma(a=0, b=0):  
    s = a + b  
    return s
```

```
# Programa Principal
```

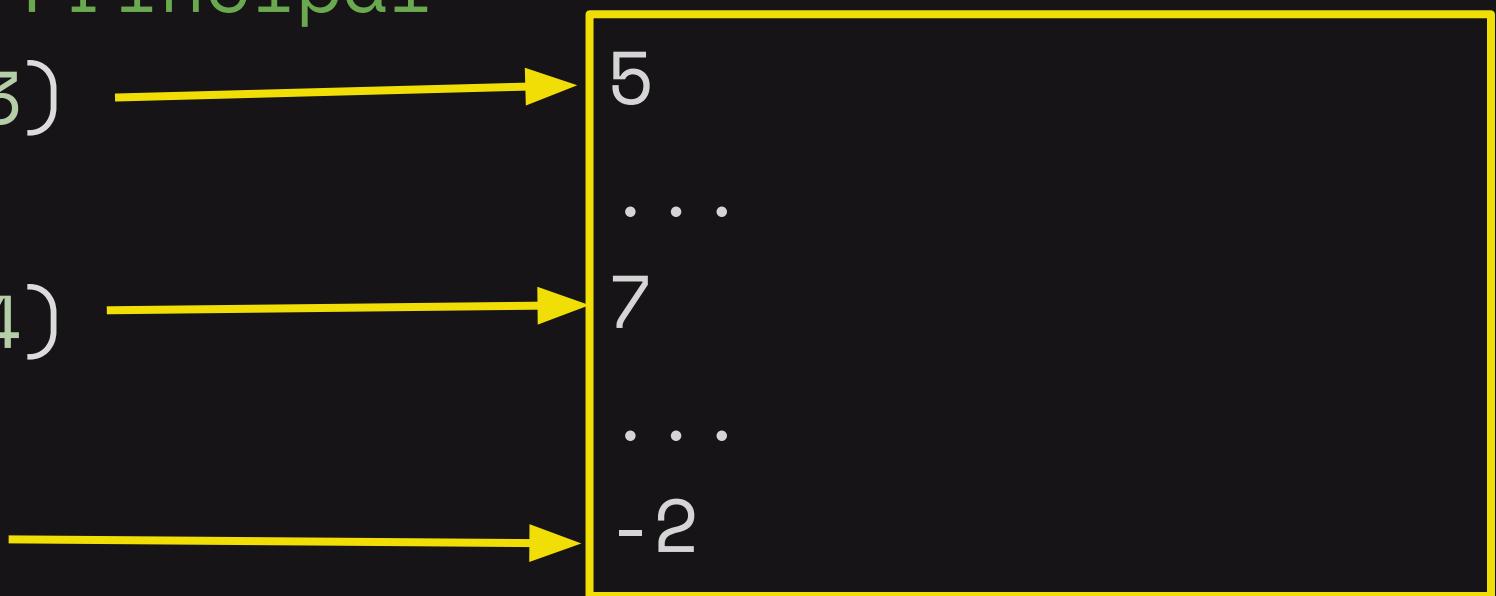
```
s=soma(2, 3)
```

```
...
```

```
a=soma(3, 4)
```

```
...
```

```
x=soma(-2)
```

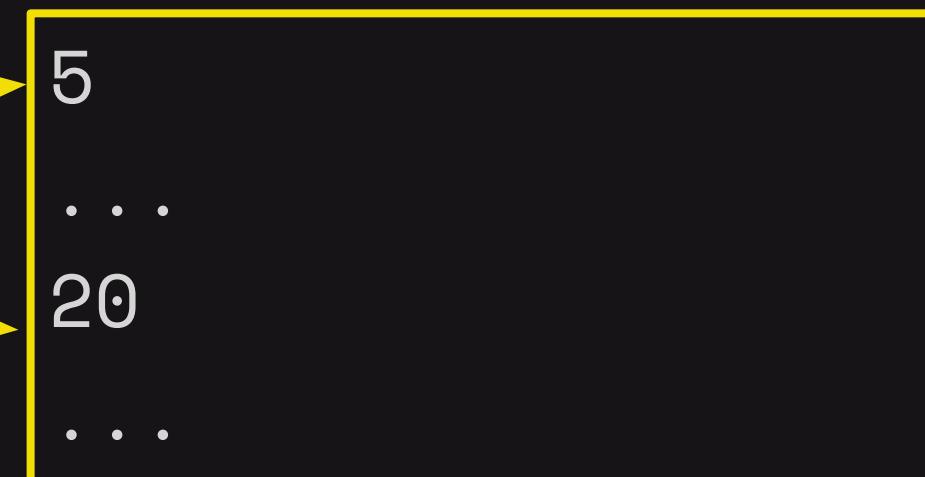


# Algumas funções

- Funções com quantidade indefinida de parâmetros:  
Faça uma função some a + b:

```
# Função 2:  
def soma(* num):  
    s=0  
    for n in num:  
        s += n  
    return s
```

```
# Programa Principal  
soma(2, 3)  
...  
soma(3, 4, 8, 5)  
...
```



# Help e docstring

O help do Python nos ajuda a mostrar quais são os parâmetros principais, opcionais, retornos e funcionalidades de qualquer função:

```
help(print)
```

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

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

Prints the values to a stream, or to sys.stdout by default.

Optional keyword arguments:

file: a file-like object (stream); defaults to the current sys.stdout.

sep: string inserted between values, default a space.

end: string appended after the last value, default a newline.

flush: whether to forcibly flush the stream.

# Help e docstring

E como fazemos com nossas próprias funções? Nós mesmos podemos escrever este “help”.

```
# Função 2:  
def soma(a=0, b=0):  
    """  
  
    Função soma: soma 2 números  
    a: parâmetro opcional de entrada  
    b: parâmetro opcional de entrada  
    retorno: s  
    Autor: Everton  
    """  
  
    s = a + b  
    return s
```

# Escopo de variável

```
-----|  
| # Função 1:  
| def soma(a, b):  
|     c = 5  
|     s = a + b  
|     return s  
|-----|  
  
# Programa Principal  
c = 10  
x = soma(2, 3)  
-----|
```

Escopo Local

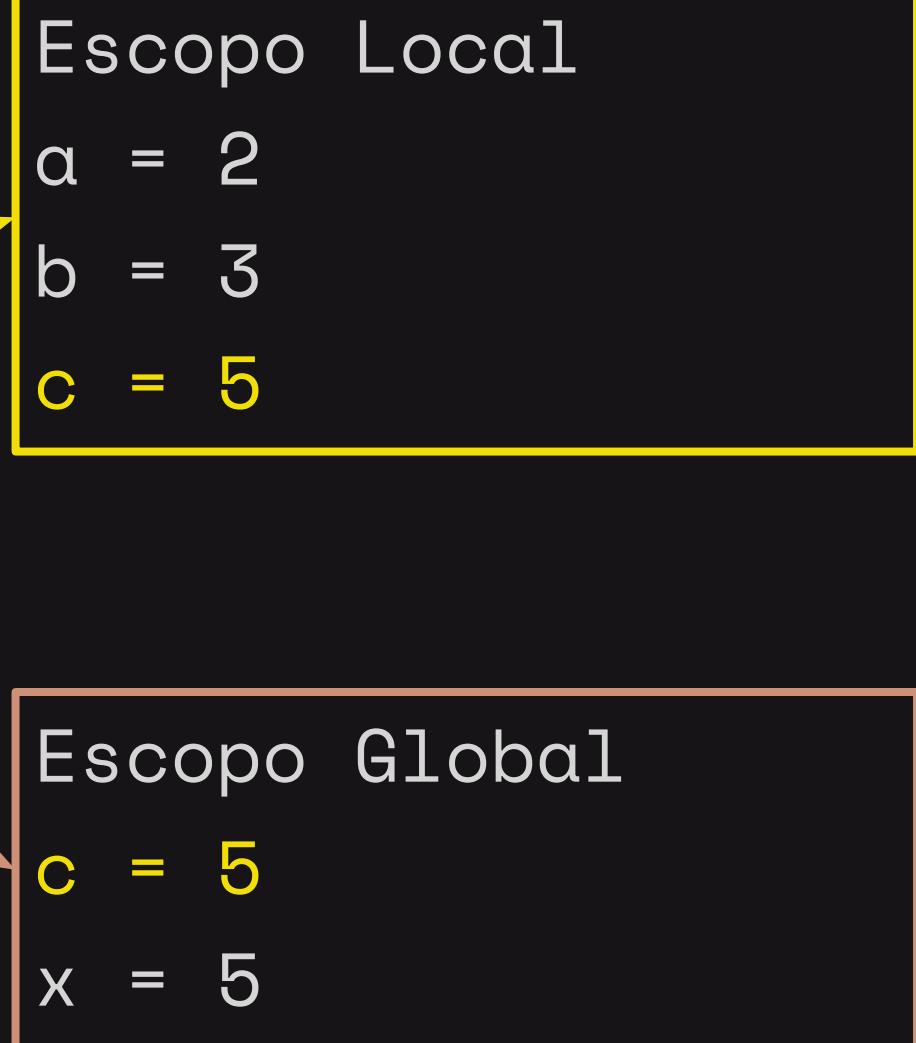
a = 2  
b = 3  
c = 5

Escopo Global

c = 10  
x = 5

# Escopo de variável

```
|-----|  
| # Função 1:  
| def soma(a, b):  
|     global c  
|     c = 5  
|     s = a + b  
|     return s  
|-----|  
  
# Programa Principal  
c = 10  
x = soma(2, 3)  
|-----|
```



# Funções

Funções são rotinas, ações que são executadas várias vezes nos programas.

```
def funcao(par):  
    """  
    docstring  
    """  
    print('Função')  
  
funcao()
```

```
funcao()  
funcao(a,b)  
funcao(a=0,b=0)  
funcao(*num)  
funcao(list)  
return var  
global var
```

# Help e docstring

```
help(funcao)
```

```
docstring
```

# Mão na Massa

Let's CODE

