

Programação em python

Aprenda o básico na prática

Sobre o curso

O Curso de Aperfeiçoamento Profissional Programação em Python tem por objetivo capacitar profissionais para desenvolver aplicações em linguagem Python, por meio de técnicas de programação, seguindo boas práticas, procedimentos e normas.

```
import time
from selenium import webdriver
from selenium.webdriver.common.by import By
import openpyxl

def get_text_by_xpath(driver, index, complemento) -> str:
    elemento_xpath = '//*[@id="menu-
panel"]/article/div[1]/div/div/section/div[1]/table/tbody/tr[' + str(index) + ']' +
complemento
    return driver.find_element(By.XPATH, elemento_xpath).text

def main():
    # Abrir o navegador e acessar a página da tabela de classificação
    options = webdriver.ChromeOptions()
    options.add_argument("--headless")
    driver = webdriver.Chrome(options=options)

    driver.get("https://www.cbf.com.br/futebol-brasileiro/competicoes/campeonato-
brasileiro-serie-a/2022")

    # Obter a tabela de classificação
    tabela = driver.find_element(By.XPATH, '//*[@id="menu-
panel"]/article/div[1]/div/div/section/div[1]/table')

    # Extrair os dados da tabela
    linhas = tabela.find_elements(By.XPATH, "//*[tr]")
    index = 1
    dados = []
```

O que esperar do curso

Ser capazes de
identificar problemas

Elaborar o algoritmo da
solução

Configurar o ambiente
de desenvolvimento

Programar na
linguagem python

Entender conceitos de
como programar um jogo

Como validar seus
códigos com testes

**Não existe pergunta idiota.
Existe o idiota que não pergunta!**

**Prefiro ser tratado como tolo do
que fingir ser sábio.**

**Chegará a hora que o falso sábio
sucumbirá a fome não saciada da
sua curiosidade**

Introdução

- Como tudo começou
- Algoritmo e Lógica
- Python

```
def __init__(self, path):
    self.file = None
    self.fingerprints = set()
    self.logdups = True
    self.debug = debug
    self.logger = logging.getLogger(__name__)
    if path:
        self.file = open(os.path.join(path, 'requests.log'),
                        'a')
        self.file.seek(0)
        self.fingerprints.update(self.read())

    @classmethod
    def from_settings(cls, settings):
        debug = settings.getbool('DEBUG', False)
        return cls(job_dir(settings), debug)

    def request_seen(self, request):
        fp = self.request_fingerprint(request)
        if fp in self.fingerprints:
            return True
        self.fingerprints.add(fp)
        if self.file:
            self.file.write(fp + os.linesep)

    def request_fingerprint(self, request):
        return request_fingerprint(request)
```

1843 - Augusta Ada Byron (Condessa Ada Lovelace)

Em seu complemento para o Mecanismo Analítico de Charles Babbage escrito em 1843, simplesmente intitulado Notas, ela delineou **quatro conceitos essenciais** que moldaram o nascimento da computação moderna um século mais tarde.

Primeiro, ela imaginou uma máquina de propósito geral, não só pré-programadas.

Segundo, que deveria poder processar mais do que apenas cálculos, mas produções musicais e artísticas.

Terceiro, um esboço de um passo a passo de algoritmo da máquina.

Quarto, máquinas poderiam pensar de forma independente.

DIAGRAM BELONGING TO NOTE D

Number of Operations Nature of Operations	Variables for Data						Working Variables					
	$1V_0$	$1V_1$	$1V_2$	$1V_3$	$1V_4$	$1V_5$	$0V_6$	$0V_7$	$0V_8$	$0V_9$	$0V_{10}$	$0V_{11}$
	+	+	+	+	+	+	+	+	+	+	+	+
	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0
	m	n	d	m'	n'	d'						
1	m				n'							
2	n				m'							
3		d										
4		0			d'							
5		0			0							
6			0		0							
7						0						
8							0					
9								0				
10									0			
11												

$\frac{d'm - d'm'}{mn - m'n} = y$

1936 - Alan Turing - O pai da computação



Foi um matemático e criptógrafo inglês considerado atualmente como o pai da computação, uma vez que, por meio de suas ideias, foi possível desenvolver o que chamamos hoje de computador.

Foi responsável com sua equipe por decifrar a **Enigma**, uma máquina de criptografia alemã utilizada na segunda guerra mundial.

Terceiro, um esboço de um passo a passo de algoritmo da máquina.

O que é algoritmo e fluxograma

Algoritmo é um conjunto sequencial de passos lógicos e finitos que realiza uma tarefa específica. Em termos simples, é uma receita para resolver um problema.

Em programação, um algoritmo é como um roteiro que guia o computador na execução de tarefas. Essas tarefas podem ser simples, como somar dois números, ou complexas, como ordenar uma lista de dados. A habilidade de criar algoritmos eficientes é essencial para resolver problemas de forma estruturada.

Algoritmo para preparar um sanduíche

Objetivo: Preparar um sanduíche simples de pão, queijo e presunto.

Entrada:

1. Pão
2. Queijo
3. Presunto

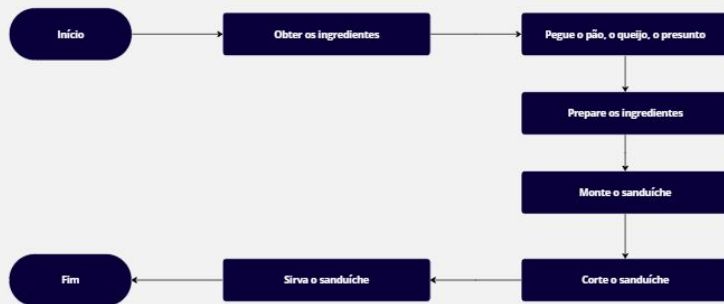
Saída:

Um sanduíche pronto para comer

Passos:

1. Obter os ingredientes.
2. Pegue o pão, o queijo, o presunto.
3. Prepare os ingredientes.
 1. Se o pão estiver congelado, descongele-o.
 2. Se o queijo estiver duro, corte-o em fatias.
 3. Se o presunto estiver inteiro, corte-o em fatias.
4. Montagem do sanduíche.
 1. Coloque uma fatia de pão em um prato.
 2. Coloque as fatias de queijo sobre o pão.
 3. Coloque as fatias de presunto sobre as fatias de queijo.
 4. Coloque a outra fatia de pão sobre os ingredientes.
5. Corte o sanduíche.
 1. Se desejar, corte o sanduíche ao meio ou em fatias.
6. Sirva o sanduíche.





O que é algoritmo e fluxograma



Fluxograma é uma representação visual de um algoritmo. Ele utiliza símbolos gráficos para ilustrar o fluxo de controle do programa, mostrando decisões, loops e operações.

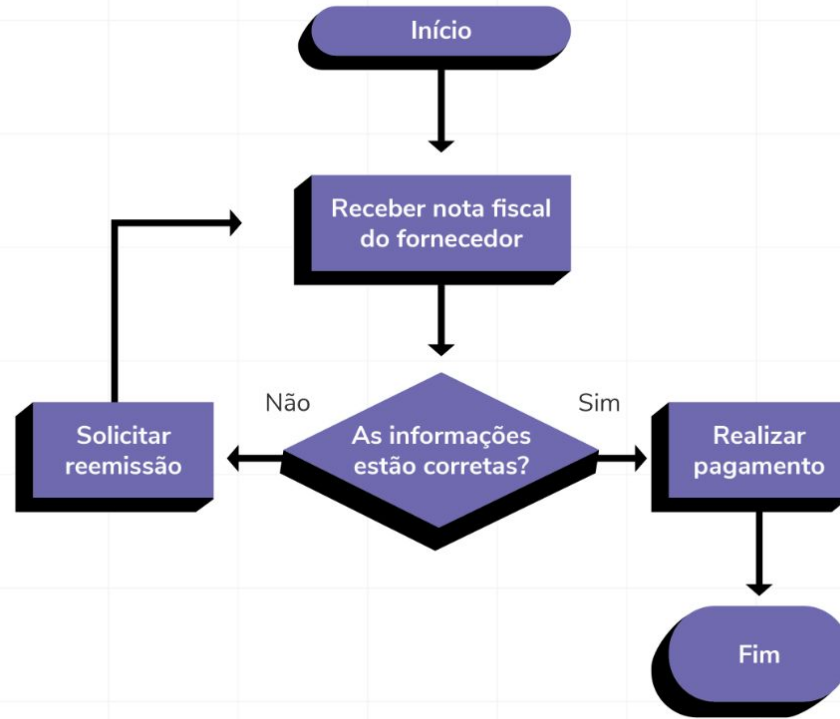
Criar fluxogramas é uma prática comum para planejar e entender a lógica por trás de um algoritmo antes de começar a codificação.

O que é algoritmo e fluxograma

Símbolo	Nome	Quando usar?
	Início ou fim	Sempre que um fluxograma de processo for iniciado ou finalizado.
	Decisão	Na maioria dos processos há momentos em que é preciso tomar decisões. Após o recebimento da nota fiscal de um fornecedor, por exemplo, é preciso conferir se as informações estão corretas. Nessa etapa, é possível incluir no fluxograma um símbolo de decisão com a pergunta: "a NF está correta?".
	Processo	Símbolo mais comum em fluxogramas de processo, serve para indicar etapas simples, que não envolvem tomada de decisão, como "fazer o pagamento do fornecedor".
	Subprocesso	Indica um subprocesso que foi pré-definido.

	Operação manual	Representa tarefas manuais ou não automatizadas que existem dentro do processo.
	Conector	Liga um ponto ao outro. Como alguns fluxogramas são muito extensos, além das linhas pode ser necessário usar o conector para ligar etapas distantes, tornando o fluxograma mais fácil de compreender.
	Documento	Mostra uma etapa do processo em que um documento é gerado, por exemplo quando um vendedor redige uma proposta para um cliente.
	Fluxo de linha	Usado para conectar os símbolos de um processo. Indica a direção em que as etapas ocorrem.

O que é algoritmo e fluxograma



Python - Como surgiu

Python foi criado por Guido van Rossum e teve sua primeira versão lançada em 1991. Van Rossum buscava criar uma linguagem de programação que fosse fácil de ler, escrever e compreender.

O nome "Python" foi inspirado no grupo de comédia britânico "Monty Python", refletindo a ideia de tornar a programação divertida.



Python - Vantagens

Sintaxe Clara e Simples (sem necessidade de chaves e outros vícios das demais linguagens de programação)

Versatilidade (diversas áreas, incluindo desenvolvimento web, automação, análise de dados, inteligência artificial e muito mais)

Comunidade Ativa (fóruns online, grupos de estudo e documentação extensiva estão disponíveis para ajudar novos programadores)

Bibliotecas Abundantes (essas bibliotecas oferecem soluções prontas para diversas tarefas, economizando tempo e esforço)



Hello World

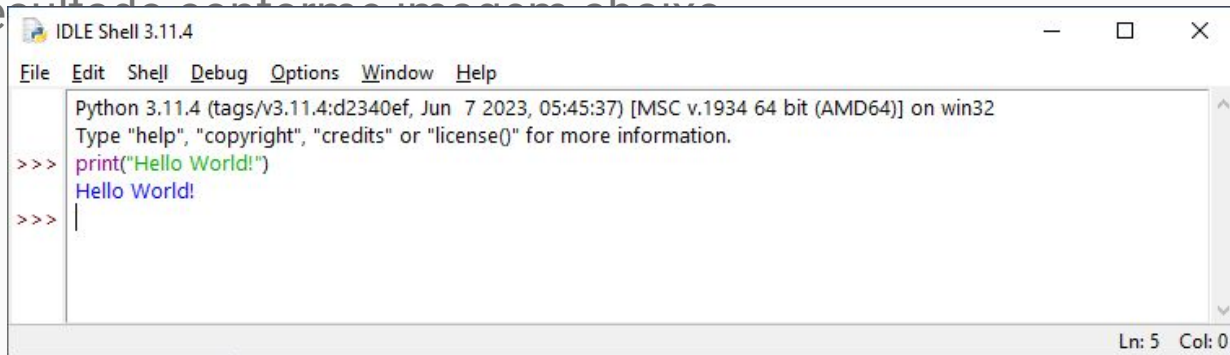
Vamos ao seu primeiro código em python.

Usaremos o IDLE, um ambiente integrado de desenvolvimento e aprendizagem da linguagem.

Procure pelo programa **IDLE Python 3.11.4**, abra-o e digite o comando abaixo:

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

E veja o resultado conforme imagem abaixo:



```
Python 3.11.4 (tags/v3.11.4:d2340ef, Jun 7 2023, 05:45:37) [MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print("Hello World!")
Hello World!
>>> |
```

The screenshot shows the IDLE Python 3.11.4 Shell window. The title bar reads "IDLE Shell 3.11.4". The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main text area displays the Python version and build information, followed by a prompt for help. The user has entered the command `print("Hello World!")` and the output `Hello World!` is displayed. The status bar at the bottom right shows "Ln: 5 Col: 0".

Entendendo o que aconteceu

Para executar alguma ação em python, você precisa instruir o computador a realizar a ação que você deseja.

No nosso exemplo, optamos pelo comando `print()` que tem como objetivo imprimir um texto em tela.

O texto que escolhemos para imprimir foi **“Hello World!”**.

Nesse caso, pudemos ver que para a função `print()` exibir algum texto, ela recebe dentro dos seus parênteses, o texto que deseja ser **“impresso”**.

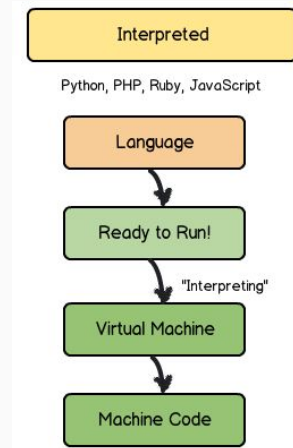
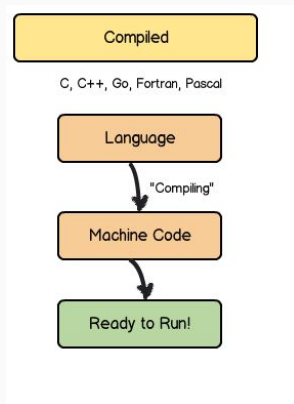
Com isso, o código digitado foi interpretado e nos deu o retorno que esperávamos.

Compilado vs Interpretado - Qual a diferença?

Linguagem compilada traduz todo o código de uma vez antes da execução, gerando um arquivo executável independente (ex: C, C++)

VS

Linguagem interpretada traduz o código linha por linha ou em blocos durante a execução, sem criar um arquivo independente, usando um interpretador (ex: Python, JavaScript).



E no python?

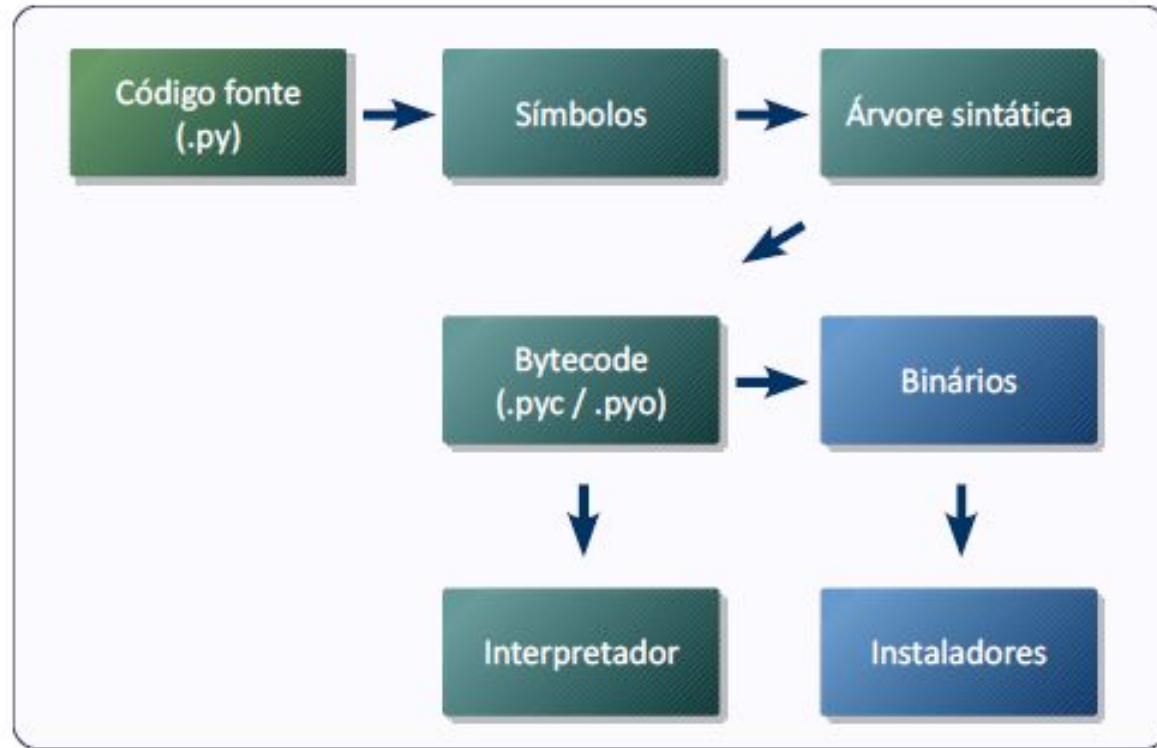


Image retirada do livro **Python para Desenvolvedores**, 2ª edição. Borges, Luiz Eduardo.

Operadores aritméticos

Na base da matemática, o python trabalha nativamente com **7** tipos de operações matemáticas.

Sendo elas listadas na tabela ao lado com o símbolo que é utilizado e a descrição do que ela faz.

+	Soma
-	Diferença / Subtração
*	Multiplicação
/	Divisão entre dois inteiros, resultado em real
//	Divisão entre dois inteiros, resultado truncado.
%	Retorna o resto da divisão
**	Potência (ex: $2^{**}3 = 8$ ou 2^3)

Let's go to the code!!!

Operadores lógicos

Os operadores lógicos servem para validar a informação de 2 ou mais valores e verificar se a expressão é **verdadeira** ou **falsa**.

Esse tipo de teste é conhecido como **Tabela Verdade** em matemática discreta

p	q	$p \vee q$
V	V	V
V	F	V
F	V	V
F	F	F

Operadores lógicos

Como regra, utilizamos a tabela ao lado com suas regras para testar o valor lógico das proposições, assim conseguimos entender quais são as possíveis situações de uma ou mais proposições.

Esses testes são importantes para desenhar corretamente o fluxo do nosso código.

Conectivo	Símbolo	Operação Lógica	Valor Lógico
não	\sim	negação	Terá valor falso quando a proposição for verdadeira e vice-versa.
e	\wedge	conjunção	Será verdadeira somente quando todas as proposições forem verdadeiras.
ou	\vee	disjunção	Será verdadeira quando pelo menos uma das proposições for verdadeira.
se...então	\rightarrow	condicional	Será falsa quando a proposição antecedente for verdadeira e a consequente for falsa.
...se somente se...	\leftrightarrow	bicondicional	Será verdadeira quando ambas as proposições forem verdadeira ou ambas falsas.

Modo de ler:

$\sim p$ – “não p”

$p \wedge q$ – “p e q”

$p \vee q$ – “p ou q”

$p \rightarrow q$ – “se p então q”

$p \leftrightarrow q$ – “p se e somente se q”

Operadores lógicos

João é alto e Maria é baixa. ($p \wedge q$)

p: João é alto

q: Maria é baixa

Se João for alto e Maria for baixa, a frase “João é alto e Maria é baixa” é **VERDADEIRA**.

Se João for alto e Maria não for baixa, a frase “João é alto e Maria é baixa” é **FALSA**.

Se João não for alto e Maria for baixa, a frase “João é alto e Maria é baixa” é **FALSA**.

Se João não for alto e Maria não for baixa, a frase “João é alto e Maria é baixa” é **FALSA**.

p	q	$p \wedge q$
V	V	V
V	F	F
F	V	F
F	F	F

Operadores lógicos

Os operadores lógicos servem para validar a informação de 2 ou mais valores e verificar se a expressão é **verdadeira** ou **falsa**.

Esse tipo de teste é conhecido como **Tabela Verdade** em matemática discreta

<	Menor que
>	Maior que
<=	Menor ou igual à
>=	Maior ou igual à
==	Igual à
!=	Diferente de
not	Negar uma proposição

Entendendo o que aconteceu

Em Python, assim como eu qualquer outra linguagem de programação, existe o conceito de **palavras chaves reservadas**.

São palavras específicas da linguagem e que indicam alguma ação que deve ser entendida pelo interpretador da linguagem.

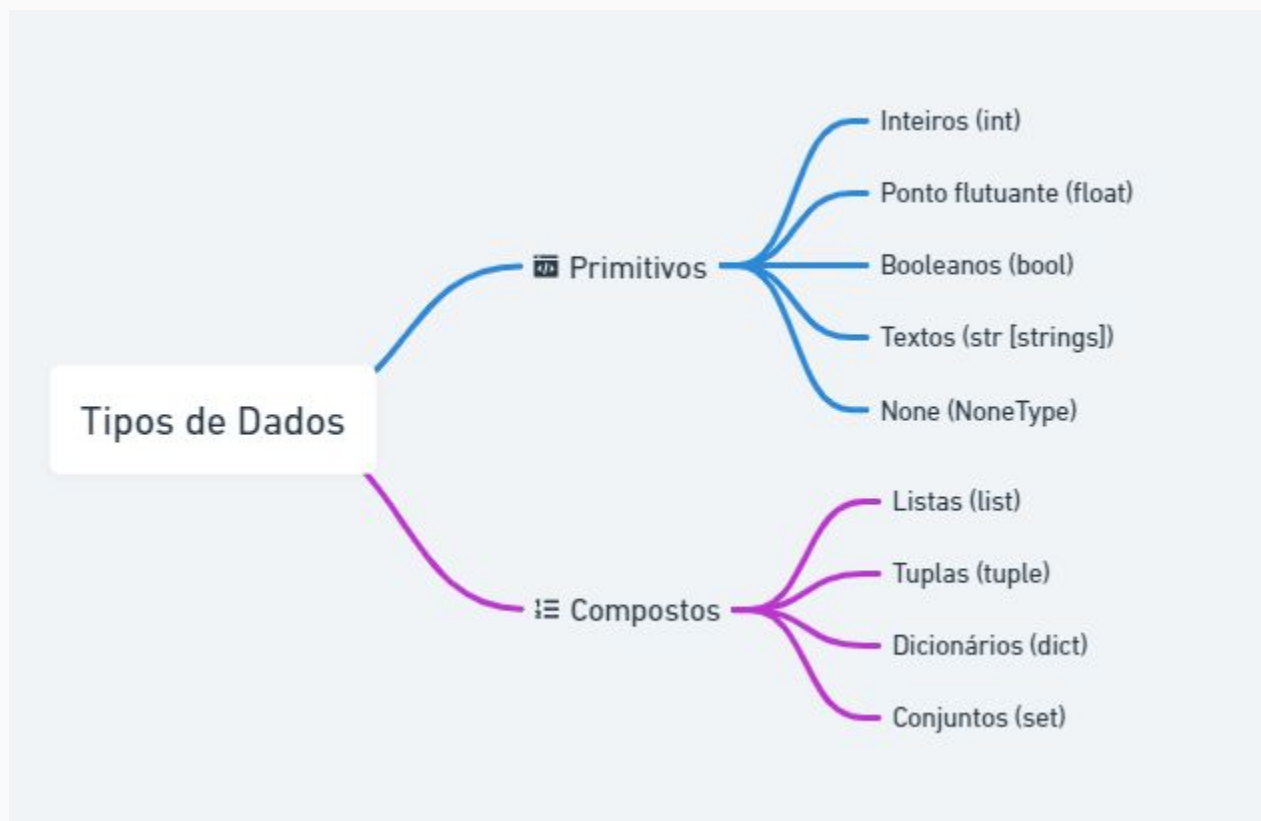
Caso queira ver no IDLE, digite o comando abaixo:

`help("keywords")`

Python Keywords

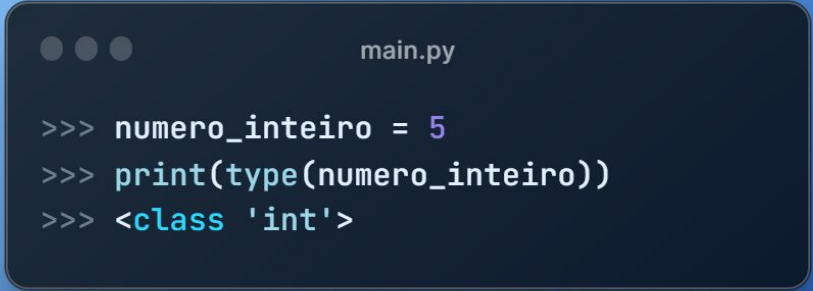
False	def	if	raise
None	del	import	return
True	elif	in	try
and	else	is	while
as	except	lambda	with
assert	finally	nonlocal	yield
break	for	not	
class	from	or	
continue	global	pass	

Tipos de Dados



Inteiro (int)

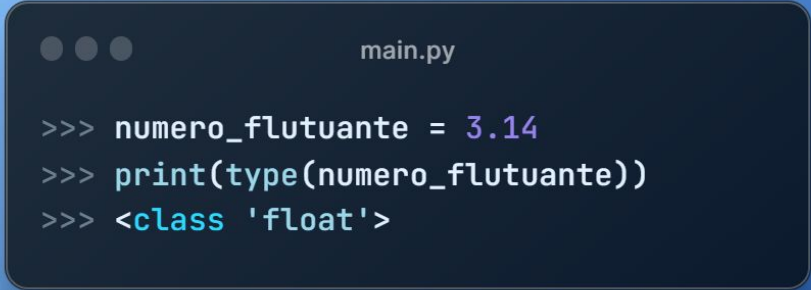
Números inteiros, por exemplo, 5, -10, 100.

A terminal window with a dark blue background and rounded corners. It has three small grey circles in the top-left corner and the filename 'main.py' in the top-right corner. The terminal contains three lines of Python code: the first line assigns the value 5 to the variable 'numero_inteiro'; the second line prints the type of 'numero_inteiro'; and the third line shows the output of the print statement, which is the class 'int'.

```
main.py  
  
>>> numero_inteiro = 5  
>>> print(type(numero_inteiro))  
>>> <class 'int'>
```

Ponto Flutuante (float)

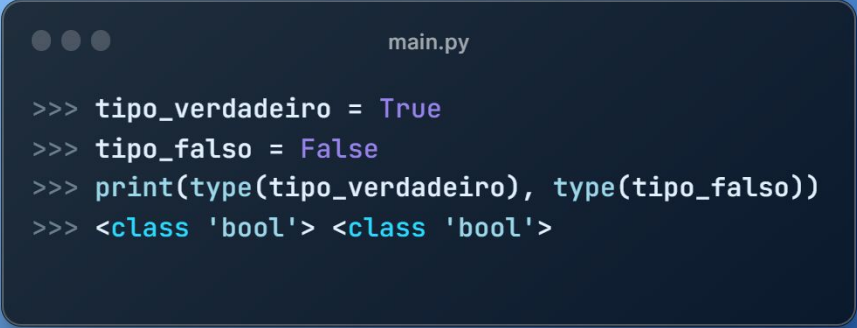
Números com parte decimal, por exemplo, 3.14, -0.5.

A screenshot of a Python REPL window titled 'main.py'. It shows three lines of code: 'numero_flutuante = 3.14', 'print(type(numero_flutuante))', and '<class 'float'>'. The code is color-coded: 'numero_flutuante' is white, '3.14' is purple, 'print' is white, 'type' is white, 'numero_flutuante' is white, 'float' is cyan, and 'class' is cyan. The window has three gray circular window control buttons in the top-left corner.

```
>>> numero_flutuante = 3.14
>>> print(type(numero_flutuante))
>>> <class 'float'>
```

Booleano (bool)


Representa valores lógicos True (verdadeiro) ou False (falso).

A terminal window with a dark background and light blue text. The title bar shows three dots and the filename 'main.py'. The code is as follows:

```
>>> tipo_verdadeiro = True
>>> tipo_falso = False
>>> print(type(tipo_verdadeiro), type(tipo_falso))
>>> <class 'bool'> <class 'bool'>
```

Textos / Strings (str)

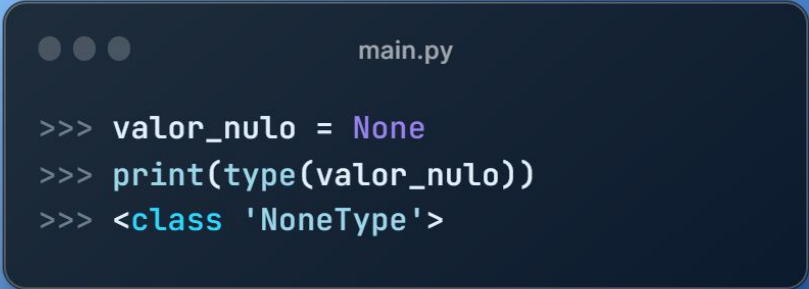
Representa sequências de caracteres, como "Olá, mundo!".

A screenshot of a Python REPL window titled 'main.py'. It shows three lines of code being executed: 'meu_nome = "Tafarel"', 'print(type(meu_nome))', and '<class 'str'>'. The output of the second line is '<class 'str'>'.

```
>>> meu_nome = "Tafarel"  
>>> print(type(meu_nome))  
>>> <class 'str'>
```

Nulo (None)

Representa a ausência de valor ou um valor nulo. Como o zero na matemática.

A terminal window with a dark background and light text. The title bar at the top says 'main.py'. There are three small circles on the left side of the title bar. The terminal shows three lines of Python code: the first line assigns 'None' to 'valor_nulo', the second line prints the type of 'valor_nulo', and the third line shows the output '<class 'NoneType'>'.

```
>>> valor_nulo = None
>>> print(type(valor_nulo))
>>> <class 'NoneType'>
```

Let's go to the code!!!

Tipos de Dados - Compostos (Conjuntos)

Lista (list)

Representa uma coleção ordenada e mutável de elementos.
Os elementos podem ser de qualquer tipo

```
>>> lista = [1, 'Tafarel', 2, '3']
>>> print(lista)
>>> [1, 'Tafarel', 2, '3']
>>> print(type(lista[2]))
>>> <class 'int'>
>>> print(type(lista[3]))
>>> <class 'str'>
```

Tipos de Dados - Compostos (Conjuntos)

Tupla (tuple)

Representa uma coleção ordenada e *imutável* de elementos.
Geralmente usadas para representar coleções fixas de valores.

```
>>> coordenadas = (3.14, 2.71)
>>> tupla_cores = ('vermelho', 'verde', 'azul')
>>> print(tupla_cores)
>>> ('vermelho', 'verde', 'azul')
>>> print(type(tupla_cores))
>>> <class 'tuple'>
```

Tipos de Dados - Compostos (Conjuntos)

Dicionários (dict)

Coleção não ordenada em pares (**chave:valor**).
Cada **chave** deve ser única.

```
>>> dicionario_pessoa = {'nome': 'TafareL', 'idade': 32, 'cidade': 'Araraquara'}
>>> print(dicionario_pessoa)
>>> {'nome': 'TafareL', 'idade': 32, 'cidade': 'Araraquara'}
>>>
>>> print(type(dicionario_pessoa))
>>> <class 'dict'>
>>>
>>> print(dicionario_pessoa['nome'])
>>> TafareL
>>>
>>> print(dicionario_pessoa.get('idade'))
>>> 32
```

Tipos de Dados - Compostos (Conjuntos)

Conjuntos (set)

Coleção não ordenada de elementos únicos.

Útil para operações de conjuntos, como união e interseção.

```
>>> conjuntos_numeros = {1, 2, 3, 4, 5}
>>> print(conjuntos_numeros)
>>> {1, 2, 3, 4, 5}
>>>
>>> print(type(conjuntos_numeros))
>>> <class 'set'>
>>>
>>> print(list(conjuntos_numeros)[0])
>>> 1
```

Let's go to the code!!!

Let's go to the code!!!

if: elif: else:

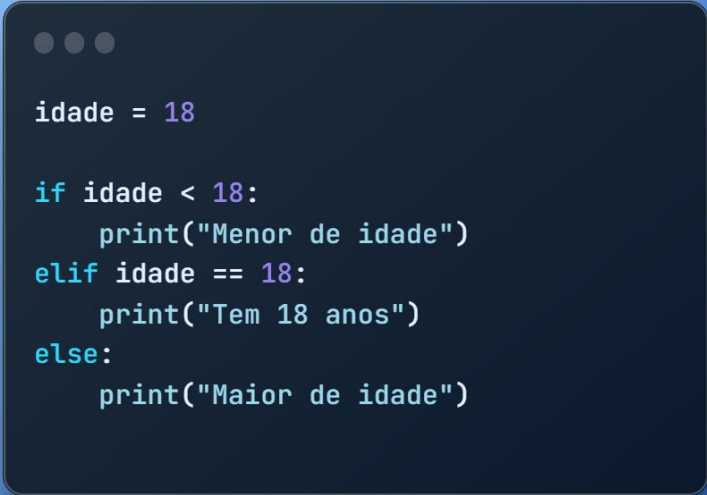
Em Python, **if**, **elif**, e **else** são estruturas de controle de fluxo que permitem tomar decisões no código. Vamos simplificar:

if (se): Usado para verificar se uma condição é verdadeira. Se for, o bloco de código dentro do **if** é executado.

elif (senão se): Se a condição do **if** não for verdadeira, você pode usar o **elif** para verificar outra condição. É como um "caso contrário, se". Se a condição do **elif** for verdadeira, o bloco de código dentro dele é executado.

else (senão): Se nenhuma das condições anteriores for verdadeira, o bloco de código dentro do **else** é executado. É como um "caso contrário". O **else** não possui uma condição própria, pois é a opção final.

if: elif: else:

A dark-themed code editor window with three window control buttons (red, yellow, green) in the top-left corner. The code is written in a light blue/cyan monospace font. It defines a variable 'idade' as 18, then uses an if-elif-else structure to print messages based on the value of 'idade'.

```
idade = 18

if idade < 18:
    print("Menor de idade")
elif idade == 18:
    print("Tem 18 anos")
else:
    print("Maior de idade")
```



Let's go to the code!!!

Laços de repetição - FOR

Em Python, existem dois principais tipos de laços de repetição: **for** e **while**. Vamos dar uma olhada em ambos de forma simplificada:

O laço **for** é utilizado para iterar sobre uma sequência (como uma lista, uma tupla, uma string ou um intervalo numérico).

Neste exemplo, o **for** percorre cada elemento da lista `frutas` e imprime cada fruta.



```
lacos_de_repeticao.py

# Iterando sobre uma lista
frutas = ["maçã", "banana", "uva"]
for fruta in frutas:
    print(fruta)
```

Laços de repetição - WHILE

Em Python, existem dois principais tipos de laços de repetição: **for** e **while**. Vamos dar uma olhada em ambos de forma simplificada:

O laço **while** é utilizado quando você quer repetir um bloco de código enquanto uma condição específica for verdadeira. Aqui está um exemplo:

Neste exemplo, o bloco de código dentro do **while** é executado enquanto a condição **contador <= 5** for verdadeira

.
O contador é incrementado a cada iteração.



```
lacos_de_repeticao.py

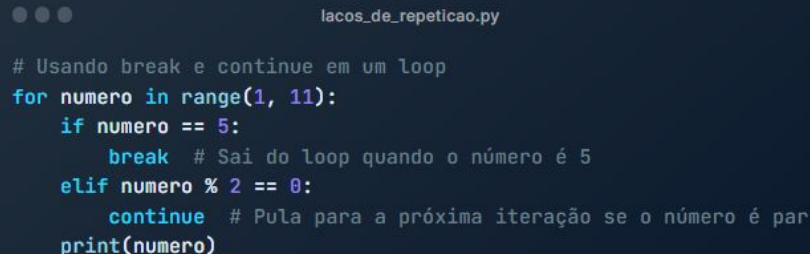
# Usando o while para contar até 5
contador = 1
while contador <= 5:
    print(contador)
    contador += 1
```

Laços de repetição - Controle de Fluxo

Em Python, existem dois principais tipos de laços de repetição: **for** e **while**. Vamos dar uma olhada em ambos de forma simplificada:

Além disso, você pode usar palavras-chave como **break** e **continue** para controlar o fluxo em loops.

break é usado para sair do loop prematuramente, enquanto **continue** é usado para pular para a próxima iteração.



```
lacos_de_repeticao.py

# Usando break e continue em um loop
for numero in range(1, 11):
    if numero == 5:
        break # Sai do loop quando o número é 5
    elif numero % 2 == 0:
        continue # Pula para a próxima iteração se o número é par
    print(numero)
```

Let's go to the code!!!

Funções embutidas

É um bloco de código que realiza uma tarefa específica e pode ser utilizado em diferentes partes de um programa.

Ela é projetada para aceitar dados de entrada (parâmetros ou argumentos), processá-los e, opcionalmente, retornar um resultado.

O conceito de funções ajuda a organizar o código, facilitando a modularidade, reusabilidade e compreensão.

O que são funções

```
def say_hello():  
    print("Hello World!")  
  
def say_hello_to(name: str):  
    print("Hello " + name)  
  
say_hello()  
say_hello_to("Tafarel")
```

Como é a estrutura de uma função

def é a palavra reservada utilizada para criar funções

say_hello e **say_hello_to** são o nome das funções

name é o parâmetro da função (opcional)

str é o tipo de dados do parâmetro (opcional/obrigatório*)

O que são funções

Funções embutidas

Até o presente momento, vimos 4 funções já nesse curso.

`print("Hello World!")` -> exibe o texto que está entre parênteses

`help("keywords")` -> exibe uma lista de ajuda referente a algum parâmetro

`type(meu_nome)` -> informa o tipo do dado armazenado na variável

`input("Digite o seu nome: ")` -> solicita entrada de dados no terminal

Funções embutidas

Existem funções reservadas da linguagem, a qual podemos chamar de nativas/embutidas.

Existem funções criadas pelo próprio programador.

Existem funções que são criadas por terceiros, essas estão agrupadas em pacotes/bibliotecas.

O que são funções

Funções embutidas

`input("Hello World!")` # é uma função interna para SOLICITAR no terminal alguma informação que o usuário forneça, ela fica aguardando você digitar algo.

`print("keywords")` # objetivo de INFORMAR algo no terminal.

`len(lista_pessoas)` # devolve a quantidade de itens de um conjunto

O que são funções

Funções embutidas

range(stop) # representa uma sequência imutável de números e é comumente usado para percorrer um número determinado de vezes em um loop.

range(start, stop, step=1) # segunda “assinatura” da função

round(number, ndigits=None) # arredonda o número para a quantidade de casas decimais informadas (**ndigits**)

open(...[muitos parâmetros]) # usado para fazer a leitura de algum, arquivo geralmente texto ou binários.

O que são funções

Funções embutidas

min(stop) # devolve o menor item de um iterável ou o menor de dois ou mais argumentos.

max(stop) # devolve o maior item de um iterável ou o maior de dois ou mais argumentos.

sum(numeros) # devolve a soma dos números de uma coleção

O que são funções

Funções embutidas

A

[abs\(\)](#)
[aiter\(\)](#)
[all\(\)](#)
[anext\(\)](#)
[any\(\)](#)
[ascii\(\)](#)

B

[bin\(\)](#)
[bool\(\)](#)
[breakpoint\(\)](#)
[bytearray\(\)](#)
[bytes\(\)](#)

C

[callable\(\)](#)
[chr\(\)](#)
[classmethod\(\)](#)
[compile\(\)](#)
[complex\(\)](#)

D

[delattr\(\)](#)
[dict\(\)](#)
[dir\(\)](#)
[divmod\(\)](#)

E

[enumerate\(\)](#)
[eval\(\)](#)
[exec\(\)](#)

F

[filter\(\)](#)
[float\(\)](#)
[format\(\)](#)
[frozenset\(\)](#)

G

[getattr\(\)](#)
[globals\(\)](#)

H

[hasattr\(\)](#)
[hash\(\)](#)
[help\(\)](#)
[hex\(\)](#)

I

[id\(\)](#)
[input\(\)](#)
[int\(\)](#)
[isinstance\(\)](#)
[issubclass\(\)](#)
[iter\(\)](#)

L

[len\(\)](#)
[list\(\)](#)
[locals\(\)](#)

M

[map\(\)](#)
[max\(\)](#)
[memoryview\(\)](#)
[min\(\)](#)

N

[next\(\)](#)

O

[object\(\)](#)
[oct\(\)](#)
[open\(\)](#)
[ord\(\)](#)

P

[pow\(\)](#)
[print\(\)](#)
[property\(\)](#)

R

[range\(\)](#)
[repr\(\)](#)
[reversed\(\)](#)
[round\(\)](#)

S

[set\(\)](#)
[setattr\(\)](#)
[slice\(\)](#)
[sorted\(\)](#)
[staticmethod\(\)](#)
[str\(\)](#)
[sum\(\)](#)
[super\(\)](#)

T

[tuple\(\)](#)
[type\(\)](#)

V

[vars\(\)](#)

Z

[zip\(\)](#)

[__import__\(\)](#)

Escopo de Variáveis

O escopo refere-se à região específica de um programa onde uma variável é acessível. É crucial entender onde as variáveis podem ser usadas e como o Python gerencia esse acesso.

O posicionamento da variável determinará a sua visibilidade e como suas informações podem ser manipuladas.

Variáveis Locais vs. Globais:

- **Variáveis Locais:** São declaradas dentro de uma função e só são acessíveis dentro dessa função. Fora dela, essas variáveis não existem.
- **Variáveis Globais:** São declaradas fora de qualquer função e podem ser acessadas de qualquer lugar no programa. No entanto, elas podem ter limitações em seu uso.

Boas práticas ao criar funções

- **Evitar Uso Desnecessário de Variáveis Globais:** O excesso de variáveis globais pode tornar o código mais difícil de entender e manter.
- **Priorizar Variáveis Locais:** Sempre que possível, use variáveis locais para evitar possíveis efeitos colaterais em outras partes do código.

Let's go to the code!!!

Exceções representam eventos inesperados ou erros durante a execução de um programa.

Permitem lidar com situações de erro de maneira controlada.

Exemplos de erros comuns: divisão por zero, tentativa de acessar um índice inválido em uma lista.

Python fornece exceções específicas para lidar com esses erros.

Tratamento de Erros - Exceções

Estrutura é dividida em 4 blocos:

try: Bloco onde o código que pode gerar exceções é colocado.

except: Bloco que lida com exceções específicas capturadas durante o bloco try.

else: Bloco opcional que é executado se nenhum erro ocorrer no bloco try.

finally: Bloco opcional que é sempre executado, independentemente de ocorrer uma exceção ou não.

Tratamento de Erros - Exceções

```
Untitled-1

# Exemplo de uso da estrutura try, except para lidar com exceções

def dividir_numeros(dividendo, divisor):
    try:
        resultado = dividendo / divisor
        print("Resultado da divisão:", resultado)
    except ZeroDivisionError:
        print("Erro: Divisão por zero não é permitida.")

# Exemplo de chamada da função
dividir_numeros(10, 2) # Saída esperada: Resultado da divisão: 5.0

# Tentativa de divisão por zero
dividir_numeros(8, 0) # Saída esperada: Erro: Divisão por zero não é permitida.
```

Tratamento de Erros - Exceções

```
Untitled-1

# Exemplo usando os blocos try, except, else, finally

def ler_arquivo(nome_arquivo):
    try:
        # Tentativa de abrir o arquivo para leitura
        with open(nome_arquivo, 'r') as arquivo:
            conteudo = arquivo.read()
            print("Conteúdo do arquivo:")
            print(conteudo)

    except FileNotFoundError:
        print(f"Erro: O arquivo '{nome_arquivo}' não foi encontrado.")

    except PermissionError:
        print(f"Erro: Sem permissão para ler o arquivo '{nome_arquivo}'.")

    else:
        # Bloco else é executado se nenhum erro ocorrer no bloco try
        print("Leitura do arquivo concluída com sucesso.")

    finally:
        # Bloco finally é sempre executado, independentemente de ocorrer uma
        # exceção ou não
        print("Operações finais realizadas.")

# Exemplo de chamada da função
nome_do_arquivo = "arquivo.txt"
ler_arquivo(nome_do_arquivo)
```

Módulos e bibliotecas são ferramentas poderosas em Python que permitem organizar e reutilizar código de maneira eficiente.

Módulo: Um arquivo Python que contém definições e instruções. Pode ser importado em outros scripts para usar as funcionalidades definidas nele.

Biblioteca: Um conjunto de módulos que oferecem funcionalidades específicas.

Exemplos populares incluem **NumPy** para computação numérica e **Matplotlib** para visualização de dados.

Importando Módulos

Instrução **import**:

Utilizada para carregar um módulo no script atual.

Exemplo: **import nome_do_modulo**.

Uso de **as** para Alias:

Permite renomear um módulo para facilitar a referência.

Exemplo: **import modulo_longo_nome as ml**.

Importando Módulos

Importando Funções Específicas:

É possível importar apenas as funções necessárias.

Exemplo: **from modulo import funcao**.

É possível importar mais de uma função de uma só vez, basta separá-las por vírgula.

Exemplo: **from modulo import funcao, nova_funcao_exemplo**.

Instalando Bibliotecas Externas

pip: ferramenta de gerenciamento de pacotes em Python. Utilizada para instalar, atualizar e remover pacotes.

Sintaxe Básica:

pip install nome_do_pacote.

Exemplo: **pip install numpy.**

Criando Ambientes Virtuais (Virtual Environments)

Ambientes Virtuais: Isolam projetos Python para evitar conflitos entre bibliotecas. Cada ambiente virtual tem suas próprias dependências.

Criando um Ambiente Virtual: `python -m venv nome_do_ambiente`.

Ativando o Ambiente Virtual:

No Windows: `nome_do_ambiente\Scripts\activate`.

No Unix ou MacOS: `source nome_do_ambiente/bin/activate`.

Criando Ambientes Virtuais (Virtual Environments)

Sempre que estiver trabalhando com o ambiente virtual, tenha por objetivo criar um arquivo para registrar as suas dependências. O comando abaixo deve ser executado sempre que desejar salvar as bibliotecas que você importou para seu ambiente virtual.

Registrando o **requirements.txt**:

pip freeze > requirements.txt (criará o arquivo com as versões de cada biblioteca).

```
requirements.txt
1  et-xmlfile==1.1.0
2  openpyxl==3.1.2
3
```

Criando Ambientes Virtuais (Virtual Environments)

Quando precisar instalar as bibliotecas de um projeto que estiverem no arquivo, utilize o comando abaixo:

Instalando os projetos do **requirements.txt**:

pip install -r requirements.txt (instalará todos as bibliotecas do arquivo).

Boas Práticas:

1. Sempre iniciar o ambiente virtual que for utilizar no projeto.
2. Escolher bibliotecas confiáveis e bem documentadas
3. Documentar as dependências em um arquivo **`requirements.txt`**.