














Python Cheat Sheet


Despertando o Pythonista que existe em você!




Temas	 Guia prático com comandos essenciais do Python. Ideal para iniciantes em programação, ciência de dados e automação.
# Introdução à linguagem	 O que é Python, para que serve, vantagens e primeiras instruções
# Introdução às IDEs	 Configurando e utilizando editores como VSCode, Jupyter e Google Colab
# Atribuições e expressões	 Como declarar variáveis e realizar operações básicas
# Números e operações	 Tipos numéricos, operadores aritméticos e funções matemáticas
# Condicionais	 Uso de if, elif e else para decisões no código
# Estruturas de repetição	 Laços for e while, e controle de fluxo (break, continue)
# Funções	 Como criar funções (def), argumentos, escopo e return
# Strings	 Criação, f-strings, métodos úteis e manipulação de texto
# Listas	 Listas em Python, indexação, métodos, mutabilidade
# Sequências	 Visão geral de listas, tuplas e strings como coleções ordenadas
# Dicionários	 Estrutura chave: valor, acesso, atualização e métodos
# Classes	 Programação orientada a objetos: definição de classes e métodos





Despertando o Pythonista que existe em você!



Introdução à linguagem	Como escrever, executar e organizar comandos. Aprenda a sintaxe básica e como a indentação define blocos de código.	 
# Comentários e impressão	<code>print("Olá, mundo!")</code> # Exibe no console	
# Identação	<code>if True:</code> <code>print("Indentado corretamente")</code>	
# Bibliotecas	<p># Uma biblioteca em Python é uma coleção de módulos que contêm funções, classes e outros objetos que podem ser reutilizados. Exemplos:</p> <p> Pandas: Manipulação de dados tabulares (DataFrames), como excel do python</p> <p> NumPy: Operações matemáticas e manipulação de arrays</p> <p> Matplotlib: Criação de gráficos básicos</p>	

Introdução às IDEs	Ambientes de desenvolvimento (IDEs) com seus componentes para programar, testar e depurar seu código com eficiência.	 
# Exemplos de IDE	 Google Colab,  Jupyter Notebook,  VSCode,  PyCharm	
# Componentes	 Editor de código,  Terminal/console,  Área de execução e saída,  Integração com Git,  Autocompletar,  Depurador	

Atribuições e Expressões	Atribuição de valores a variáveis, identificar tipos de dados e realizar expressões aritméticas e lógicas no Python.	 
# Atribuição simples	<code>pi = 3.1459</code> # Atribui um número decimal (float) à variável pi	
# Atribuições múltiplas	<code>a, b = 3, 4</code> # Atribui 3 a 'a' e 4 a 'b' na mesma linha	
# Tipos de variáveis	<code>a = 5</code> # inteiro (int) <code>b = 3.75</code> # ponto flutuante (float) <code>c = True</code> # booleano (bool) <code>type(a) -> int</code> # retorna <class 'int'> <code>type(b) -> float</code> # retorna <class 'float'> <code>type(c) -> bool</code> # retorna <class 'bool'>	
# Conversão de tipo	<code>int(3.8) -> 3</code> #  retorna 3 <code>str(10) -> 10</code> #  retorna "10"	
# Reatribuições	<code>z += 10</code> #  equivalente a <code>z = z + 10</code>	





Despertando o Pythonista que existe em você!

Números e Operações matemáticas	Domine as principais operações e funções para cálculos e decisões numéricas no Python!	 GitHub 
# Expressões com operadores aritméticos	<pre>2 + 3 # ➕ soma → 5 8 / 4 # ÷ divisão → 2.0 (sempre retorna float) 8 % 5 # ↻ módulo → 3 (resto da divisão) 2 * 2 # ✖ multiplicação → 4 2 ** 3 # ⬆ exponenciação → 8 (2 elevado à 3)</pre>	
# Funções aritméticas	<pre>abs(-7) # 1 2 valor absoluto → 7 pow(2, 3) # ⬆ potência → 8 (2**3) sum([1, 2, 3]) # ➕ soma elementos → 6 min([3, 2]) # ▼ menor valor → 2 max([3, 2]) # ▲ maior valor → 3</pre>	
# Conversão de tipos numéricos	<pre>int(3.9) # ↻ converte float → int → 3 float(4.2) # ↻ permanece float → 4.2</pre>	
# Usando Math para funções mais avançadas	<pre>import math math.sqrt(16) # ✓ raiz quadrada → 4.0 math.floor(2.7) # ▼ arredonda para baixo → 2 math.ceil(2.1) # ▲ arredonda para cima → 3 math.log10(100) # 📊 log base 10 → 2.0 math.sin(math.pi) # ↻ seno de pi → 0.0</pre>	





Despertando o Pythonista que existe em você!

Condicionais	Use estruturas condicionais para tomar decisões no código com base em comparações lógicas e valores booleanos.	 GitHub
# Valores booleanos	<pre># Os valores booleanos podem assumir apenas dois valores type(True) # → <class 'bool'> type(1 == 1) # → <class 'bool'> (resultado da comparação é booleano)</pre>	
# Operadores Relacionais	<pre>x == 5 #Igual a x != 5 #Diferente de x > 5 #Maior que x < 5 #Menor que x >= 5 #Maior ou igual a x <= 5 #Menor ou igual a</pre>	
# Testes condicionais simples	<pre>idade = 18 if idade >= 18: print("✅ Você pode votar.")</pre>	
# if...else	<pre>idade = 16 if idade >= 18: print("✅ Você pode votar.") else: print("❌ Você ainda não pode votar.")</pre>	
# if...elif...else	<pre>nota = 7 if nota >= 9: print("🏆 Excelente") elif nota >= 6: print("✅ Aprovado") else: print("❌ Reprovado")</pre>	
# Operadores lógicos	<pre>idade = 20 tem_titulo = True # Operador AND if idade >= 18 and tem_titulo: print("Pode votar.") # Operador OR if idade < 18 or not tem_titulo: print("Não pode votar.") # O operador NOT inverte o valor booleano: print(not True) # False</pre>	
# Tratamento de erros	<pre>valor = "vinte" try: idade = int(valor) print(f"Sua idade é {idade}") except ValueError: print("⚠️ Valor inválido: não é possível converter para número.")</pre>	





Despertando o Pythonista que existe em você!

Estrutura de Repetição	Repita blocos de código com as estruturas for e while, úteis para percorrer sequências.	 GitHub 
# Loop for	<pre>for i in 'Hello': print(i) # Imprime cada letra da palavra "Hello"</pre>	
# Loop for sobre lista	<pre>nomes = ["Ana", "Bruno", "Carlos"] for nome in nomes: print("Olá,", nome) # Percorre cada item da lista nomes. Saúda cada nome na lista</pre>	
# Loop for com range	<pre>for i in range(1, 6, 1): # Início, fim (exclusivo), passo print(i) # Imprime de 1 a 5</pre>	
# Loop while	<pre>contador = 0 while contador < 3: print("Contador:", contador) contador += 1 # Executa enquanto a condição for verdadeira.</pre>	
# Operadores lógicos	<pre>for i in range(1, 6): print(i)</pre>	
# Break	<pre>for i in range(5): if i == 3: break # Interrompe o loop quando i for 3 print(i)</pre>	
# Continue	<pre>for i in range(5): if i == 2: continue # Pula a iteração quando i for 2 print(i)</pre>	
# Else com Loop	<pre>for i in range(3): print(i) else: print("Fim do loop!")</pre>	





Despertando o Pythonista que existe em você!

Funções	Declare e utilize funções para organizar e reutilizar blocos de código.	 GitHub 
# Boas práticas com funções	<ul style="list-style-type: none"># Boas práticas com funções# Use nomes descritivos (ex: calcular_media)# Escreva funções pequenas e com uma única responsabilidade# Sempre que possível, use return ao invés de print, para maior reutilização	
# Definindo uma função simples	<pre>def saudacao(): "Esta função vai retornar a saudação" print("Olá!") # 🇺🇸 A palavra-chave def cria uma função chamada saudacao, que imprime uma mensagem.</pre>	
# Chamando uma função	<pre>saudacao() # ▶ Executa o código dentro da função definida.</pre>	
# Função com parâmetro	<pre>def saudar(nome): print("Olá,", nome) saudar("Ana") # 📄 A função recebe um argumento (nome) e usa esse valor no corpo da função.</pre>	
# Função com múltiplos parâmetros	<pre>def soma(a, b): return a + b print(soma(3, 5)) # Saída: 8</pre>	
# Função com valor padrão	<pre>def apresentar(nome="Visitante"): print("Bem-vindo,", nome) # Se nenhum argumento for passado, o valor padrão "Visitante" será usado.</pre>	
# Função com return	<pre>def quadrado(numero): return numero ** 2 # 🏹 Return envia um valor de volta para quem chamou a função. Aqui, retorna o quadrado do número.</pre>	
# Função help()	<pre>help(print) # 📖 Exibe a documentação da função print.</pre>	
# Função Lambda	<pre>quadrado = lambda x: x ** 2 print(quadrado(4)) # Saída: 16 # ⚡ Lambda cria uma função anônima. É útil para funções curtas, geralmente passadas como argumento.</pre>	
# Input	<pre>def saudacao(): nome = input("Digite seu nome: ") print(f"Olá, {nome}!") # Use input() para capturar dados do usuário no console.</pre>	





Despertando o Pythonista que existe em você!

Strings	Aprenda a trabalhar com textos no Python: como criar, combinar, acessar e transformar strings.	 GitHub 
# Criando strings	<pre>minha_string = "Um copo de água" outra_string = 'Dois copos de água' # ✨ Strings podem ser criadas com aspas simples ou duplas</pre>	
# Cuidados com as aspas	<pre>msg = 'vou usar o celular só para "usar a calculadora"' msg2 = "copo d'água" msg3 = 'copo d\'água' # Usando barra invertida # 💡 Quando há conflito entre aspas, use o tipo oposto ou caractere de escape (\).</pre>	
# Operações com strings	<pre>"piu" * 3 # 'piupiu' "piu" + "piu" # 'piupiu' # 💬 Você pode repetir ou concatenar strings com * e +</pre>	
# Quebra de linha	<pre>msg = "Oi,\ntudo bem?" print(msg) # \n insere uma nova linha dentro da string.</pre>	
# Funções úteis	<pre>frase = " ciência de dados " print(frase.strip()) # ✂ Remove espaços print(frase.upper()) # ✂ CAIXA ALTA print(frase.lower()) # ✂ caixa baixa print(frase.replace("ciência", "arte"))</pre>	
# f-string (formatação moderna)	<pre>nome = "Ana" print(f"Olá, {nome}!") # 🙌 Olá, Ana! # 🧩 Use f-strings para inserir variáveis ou expressões diretamente no texto</pre>	





Despertando o Pythonista que existe em você!

Listas	Use listas para armazenar coleções de dados, organizadas em ordem.	 GitHub
# Criando listas	<pre>numeros = [1, 2, 3, 4] misturada = [1, "oi", True, [9, 8]] vazia = []</pre> <p># 🇺🇲 Listas podem conter diferentes tipos de dados, inclusive outras listas.</p>	
# Acessando itens da lista	<pre>lista = ["a", "b", "c", "d"] print(lista[0]) # Primeiro item: 'a' print(lista[-1]) # Último item: 'd'</pre> <p># 🔍 Os índices começam em 0. Índices negativos contam do fim para o início.</p>	
# Modificando itens	<pre>lista[1] = "novo"</pre> <p># ✏️ Listas são mutáveis – você pode substituir valores.</p>	
# Tamanho da lista	<pre>len(lista) # Retorna a quantidade de itens</pre>	
# Adicionando itens	<pre>lista.append("e") # ➕ Adiciona no final lista.insert(1, "x") # ➕ Insere na posição 1</pre>	
# Removendo itens	<pre>lista.remove("x") # ✖ Remove o primeiro "x" lista.pop() # ✖ Remove o último lista.pop(0) # ✖ Remove o primeiro</pre>	
# Fatiamento da lista	<pre>sublista = lista[1:3] # 🎯 Itens da posição 1 até 2 (exclui o 3)</pre>	
# Iteração sobre listas	<pre>for item in lista: print(item)</pre>	
# Verificação de listas	<pre>"b" in lista # ✅ True se 'b' estiver presente</pre>	
# Ordenando listas	<pre>numeros = [3, 1, 4, 2] numeros.sort() # 🔄 Altera a lista original sorted(numeros) # 🔄 Retorna nova lista ordenada</pre>	





Despertando o Pythonista que existe em você!

Sequências	Sequências são coleções ordenadas de dados. As principais são tuplas, listas e strings.	 GitHub
# Tipos de sequências	<code>minha_lista = ['a', 'b', 'c']</code> # 📦 lista <code>minha_tupla = ('a', 'b', 'c')</code> # 🔒 tupla <code>minha_string = "abc"</code> # 📝 string	
# Criando tuplas	<code>uma_tupla = (1, 2, 3, 4)</code> # Elementos são separados por vírgulas e envolvidos por parênteses. <code>tupla_mista = (10, "texto", True)</code> # Tuplas podem conter tipos variados <code>tupla_unitaria = (5,)</code> # ⚠️ Tupla com um único elemento, necessário usar vírgula para ser tupla	
# Verificando o tipo	<code>type(uma_tupla)</code> # <class 'tuple'>	
# Tuplas são imutáveis	<code>tupla[0] = 100</code> # ❌ Erro! # Não é possível alterar os elementos de uma tupla após sua criação.	
# Indexação	<code>print(minha_lista[1])</code> # b <code>print(minha_tupla[1])</code> # b <code>print(minha_string[1])</code> # b # 🗨️ Acesso pelo índice. Lembre-se: começa em 0!	
# Segmentação	<code>print(minha_tupla[1:])</code> # ('b', 'c') <code>print(minha_string[:2])</code> # 'ab' # [início:fim] pega elementos do início até (fim - 1).	
# Métodos para tuplas	<code>uma_tupla.count(2)</code> # Conta quantas vezes o valor 2 aparece <code>uma_tupla.index(2)</code> # Retorna a posição do valor 2	
# Acessando elementos	<code>print(uma_tupla[0])</code> # Primeiro elemento <code>print(uma_tupla[-1])</code> # Último elemento	
# Comprimento	<code>len(minha_lista)</code> # 3 <code>len(minha_string)</code> # 3 <code>len(minha_tupla)</code> # 3	
# Concatenação	<code>(1, 2) + (3, 4)</code> # (1, 2, 3, 4)	
# Verificação	<code>'d' not in minha_tupla</code> # 🔍 True	
# Conversão de uma tupla	<code>uma_lista = list(uma_tupla)</code> # Cria uma nova lista com os mesmos elementos da tupla original. A tupla em si não é modificada.	





Despertando o Pythonista que existe em você!

Dicionários	São úteis quando você quer associar um identificador (chave) a uma informação (valor), como em cadastros, registros ou perfis.	 
# Criando um dicionário	<pre>aluno = { "nome": "Ana", "idade": 22, "curso": "Python" }</pre> <p># 📌 As chaves devem ser únicas e geralmente são strings. Os valores podem ser de qualquer tipo e repetidas.</p>	
# Acessando valores	<pre>print(aluno["nome"]) # Ana print(aluno.get("idade")) # 22 # 💡 Use .get() quando quiser evitar erro caso a chave não exista.</pre>	
# Adicionando ou modificando valores	<pre>aluno["email"] = "ana@email.com" # novo par aluno["idade"] = 23 # altera valor existente</pre>	
# Removendo elementos	<pre>del aluno["curso"] # remove o par com chave "curso" aluno.pop("idade") # remove e retorna o valor</pre>	
# Verificação de chaves	<pre>"nome" in aluno # True "telefone" not in aluno # True</pre>	
# Iterando sobre o dicionário	<pre>for chave in aluno: print(chave, "→", aluno[chave])</pre>	
# Métodos para dicionários	<pre>aluno.keys() # retorna as chaves aluno.values() # retorna os valores aluno.items() # retorna tuplas (chave, valor)</pre>	
# Dicionários aninhados	<pre>turma = { "aluno1": {"nome": "Ana", "idade": 22}, "aluno2": {"nome": "Bruno", "idade": 24} } #Permite organizar dados estruturados em camadas, como registros ou tabelas.</pre>	



Despertando o Pythonista que existe em você!

Classes	Classes são modelos que descrevem como criar objetos (instâncias) , com atributos (dados) e métodos (comportamentos) .	 GitHub 
# O que é uma classe?	<pre>a = [1, 0] print(type(a)) # <class 'list'> #list é uma classe embutida. Quando criamos [1, 0], estamos instanciando um objeto da classe list. # 🧱 Classe = molde # 🧑 Objeto = instância baseada nesse molde</pre>	
# Criando sua própria classe	<pre>class Pessoa: # class: define uma nova classe. def __init__(self, nome): # init: método construtor self.nome = nome # self.nome cria um atributo associado ao objeto def apresentar(self): print(f"Olá, eu sou {self.nome}")</pre>	
# Utilizando um objeto	<pre>ana = Pessoa("Ana") ana.apresentar() # Olá, eu sou Ana</pre>	
# Adicionado um comportamento (método)	<pre>class Pessoa: def __init__(self, nome, idade): self.nome = nome self.idade = idade def apresentar(self): print(f"Olá, eu sou {self.nome} e tenho {self.idade} anos.") def envelhecer(self, anos): self.idade += anos ana = Pessoa("Ana", 30) ana.apresentar() # Olá, eu sou Ana e tenho 30 anos. ana.envelhecer(5) ana.apresentar() # Olá, eu sou Ana e tenho 35 anos.</pre>	

PATIENTS2.PYTHON

Python Cheat Sheet

Despertando o Pythonista que existe em você!

A Patients2Python é a Primeira e Única Comunidade dedicada exclusivamente ao aprendizado e aplicação de Ciência de Dados de Saúde.

 Missão:

Desenvolver líderes capazes de reunir expertise clínica, gestão estratégica e domínio tecnológico, para que, de fato, estejam aptos para abordar os desafios complexos dos sistemas de saúde modernos, a ponto de conduzi-los com assertividade rumo ao Objetivo Quintuplo de aprimorar resultados clínicos e administrativos, sem que comprometer a equidade e experiência de pacientes e profissionais ao longo da jornada de cuidado.

