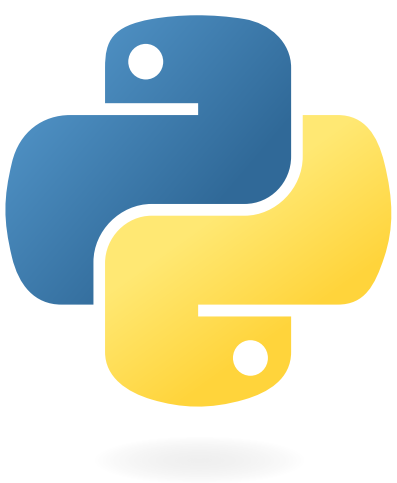
Introdução à Programação em



python™

https://evandro-crr.github.io/intro-python

Evandro Chagas Ribeiro da Rosa

Função & Modularização

Como organizar seu código para ser:

- Mais fácil de implementar
- Mais fácil de entender
- Mais fácil de testar
- Mais fácil de dar manutenção

Função

Função é um bloco de código com um nome

```
def mensagem():
  print("Olá da função, mensagem")

mensagem()
```

Sintaxe:

```
def <nome da função>(<lista de argumentos>):
  <corpo da função>
```

- Funções são definidas com a instrução `def`.
- Para executar o código definido na função, é necessário *chamar* a função.
- Uma função pode ser chamada inúmeras vezes.

Passando Informação para a Função

```python
def saudacao(nome: str):
  """Imprime uma saudação para 'nome'."""

  print(f"Olá, {nome.title()}")

saudacao('joão')
saudacao('maria')
```

- Argumentos permitem passar informações para dentro de uma função.
- A anotação de tipo do argumento é chamada de *type hint*.
- O *type hint* é ignorado pelo interpretador, mas é útil para documentar o código.
- A string no início da função é chamada de *docstring* e serve para documentar o que a função faz.

Retornando Valores de uma Função

```python
def somar(a: int, b: int) -> int:
  """Soma dois valores."""
  resultado = a + b
  return resultado

print(somar(11, 12))
```

- É possível retornar valores de uma função usando a instrução `return`
- A instrução do tipo de retorno é ignorada pelo interpretador
- O valor retornado por uma função pode ser usado em uma expressão

```python
print(somar(2, 5) * 3)
```

Passagem por Valor e por Referência

- Objetos do tipo `int`, `float`, `str`, `bool` e `tuple` são passados por valor
- Objetos do tipo `list` e `dict` são passados por referência

```
>>> def somar_mais_um(a):
...     a += 1
...
>>> a = 10
>>> somar_mais_um(a)
>>> print(a)
10
```

```
>>> def adicionar_um(lista):
...     lista.append(1)
...
>>> lista = [1, 2, 3]
>>> adicionar_um(lista)
>>> print(lista)
[1, 2, 3, 1]
```

Exemplo: Jogo da Velha (1/4)

```
def main():
    """Função principal."""
    jogadores = {0: "X", 1: "O"}
    vencedor = None

    tabuleiro = [["*" for _ in range(3)] for _ in range(3)]
    mostrar(tabuleiro)

    for i in range(9):
        jogador = jogadores[i % 2]
        jogar(tabuleiro, jogador)
        mostrar(tabuleiro)
        if venceu(tabuleiro, jogador):
            vencedor = jogador
            break

    if vencedor is not None:
        print(f"O {vencedor} venceu!!!")
    else:
        print("Deu velha!")

main()
```

Exemplo: Jogo da Velha (2/4)

```
def mostrar(tabuleiro: list[list[str]]):
    """Mostra o tabuleiro na tela.

    Args:
        tabuleiro (list[list[str]]): Tabuleiro
    """
    for linha in tabuleiro:
        print("|" + "|".join(linha) + "|")
```

Exemplo: Jogo da Velha (3/4)

```python
def posicao(jogador: str) -> tuple[int, int]:
    """Pega do usuário uma posição do tabuleiro.

    Args:
        jogador (str): Jogador atual

    Returns:
        tuple[int, int]: Posição escolhida
    """
    while True:
        try:
            x, y = map(int, input(f"({jogador}) Posição: ").split())
            if 0 <= x < 3 and 0 <= y < 3:
                break
        except Exception:
            continue

    return x, y
```

```python
def jogar(tabuleiro: list[list[str]], jogador: str):
    """Joga na posição indicada pelo jogador.

    Args:
        tabuleiro (list[list[str]]): Tabuleiro
        jogador (str): Jogador atual
    """
    while True:
        x, y = posicao(jogador)
        if tabuleiro[x][y] == "*":
            break

    tabuleiro[x][y] = jogador
```

Exemplo: Jogo da Velha (4/4)

```python
def venceu(tabuleiro: list[list[str]], jogador: str) -> bool:
    """Verifica se o jogador venceu.

    Args:
        tabuleiro (list[list[str]]): Tabuleiro
        jogador (str): Jogador atual

    Returns:
        bool: True se o jogador venceu, False caso contrário
    """
    return (
        any(all(tabuleiro[j][i] == jogador for i in range(3)) for j in range(3))
        or any(all(tabuleiro[i][j] == jogador for i in range(3)) for j in range(3))
        or all(tabuleiro[i][i] == jogador for i in range(3))
        or all(tabuleiro[i][2 - i] == jogador for i in range(3))
    )
```

Valor Padrão de Argumentos

```python
def saudacao(
    nome: str,
    nome_meio: str = "",
    ultimo_nome: str = "",
):
    nome_completo = nome + " " + nome_meio + " " + ultimo_nome
    print(f"Olá, {nome_completo.strip()}")


saudacao("João", "Carvalho", "Silva")
saudacao("Maria")
```

- É possível definir o valor padrão dos últimos argumentos
- Quando um argumento não for fornecido, o valor padrão será usado
- A ordem dos argumentos importa na chamada

Argumentos Nomeados

Podemos passar os argumentos de uma função de duas formas:

1. Por posição
2. Por nome (Keyword Arguments)

```
def saudacao(
    nome: str,
    nome_meio: str = "",
    ultimo_nome: str = "",
):
    nome_completo = ( nome + " "
      + nome_meio + " " + ultimo_nome)
    print(f"Olá, {nome_completo.strip()}")


saudacao("João", ultimo_nome="Silva")
saudacao(ultimo_nome="Carvalho", nome="Maria")
```

- É possível definir os argumentos usando seus nomes
- A ordem em que os argumentos nomeados são fornecidos não importa

Número Arbitrário de Argumentos

```python
def somar(inicial, *numeros) -> int:
    total = inicial
    for numero in numeros:
        total += numero
    return total

print(somar(1, 2, 3))
```

- É possível passar um número arbitrário de argumentos
- Os argumentos são armazenados em uma tupla

- A ordem dos argumentos é preservada na tupla

Número Arbitrário de Argumentos Nomeados

```python
def aprovado(corte, **nomes) -> dict[str, str]:
    situacao = {}

    for nome, nota in nomes.items():
        if nota >= corte:
            situacao[nome] = "Aprovado"
        else:
            situacao[nome] = "Reprovado"

    return situacao


print(aprovado(7, Alice=8, Bob=5, Charlie=7))
```

- É possível passar um número arbitrário de argumentos nomeados
- Os argumentos são armazenados em um dicionário e a chave é uma string

Funções Anônimas (Lambdas)

```
operacoes = {
    "+": lambda a, b: a + b,
    "-": lambda a, b: a - b,
    "*": lambda a, b: a * b,
    "/": lambda a, b: a / b,
}


def calcular(operacao, a, b):
    operador = operacoes[operacao]
    return operador(a, b)


print(calcular("+", 2, 3))
```

- Uma função lambda é uma função anônima, ou seja, sem nome.
- Esse tipo de função pode ser definida em uma única linha.
- Aceita apenas uma única expressão.
- O retorno da função é o resultado da expressão.
- **Sintaxe**:

```
lambda <lista de parâmetros>: <expressão>
```

Módulos em Python

Podemos utilizar funções que não são definidas diretamente no nosso código, importando módulos externos.

```python
import random

numero = random.randint(1, 10)

print(numero)
```

```python
from random import randint

numero = randint(1, 10)

print(numero)
```

- Um **módulo** é um arquivo Python que pode conter funções, variáveis e outras construções.
- A **biblioteca padrão** do Python traz diversos módulos prontos para uso.
- Podemos **criar** nossos próprios módulos usando arquivos `.py`.

Exemplo de Módulo

`matematica.py`

```python
def somar(a, b):
    return a + b

def subtrair(a, b):
    return a - b
```

`principal.py`

```python
import matematica

resultado_soma = matematica.somar(5, 3)
resultado_subtracao = matematica.subtrair(10, 7)

print(f"Soma: {resultado_soma}")
print(f"Subtração: {resultado_subtracao}")
```

O arquivo `matematica.py` é utilizado como um módulo dentro de `principal.py`.

⚙ Exercícios 1

Sistema de Gerenciamento de Estoque Iterativo

- Implemente um programa interativo de gerenciamento de estoque para uma loja. O programa deve exibir um menu de opções e permitir que o usuário realize as seguintes operações repetidamente, até que ele opte por sair:

Cada opção deve ser uma função

1. Cadastrar produto: O usuário poderá cadastrar um novo produto informando seu **nome**, **quantidade** e **preço unitário**.
2. Remover produto
3. Consultar produto
4. Relatório completo
5. Valor total em estoque
6. Sair

⚙ Exercício 2

Sistema de Biblioteca

- Implemente um sistema de gerenciamento de uma biblioteca, com as seguintes funcionalidades :
  1. Cadastro de livros: O usuário deverá cadastrar os livros disponíveis na biblioteca, fornecendo informações como **título**, **autor** e **número de exemplares**.
  2. Empréstimo de livros
  3. Devolução de livros
  4. Relatório final

Cada opção deve ser uma função

⚙ Exercício 3

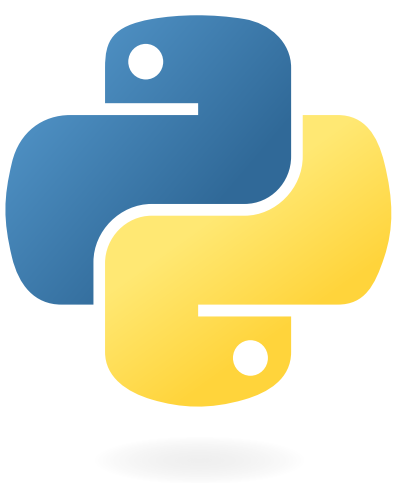
Calculadora com Notação Prefixada

- Implemente uma calculadora que receba expressões em *notação prefixada* e retorne o resultado da operação. Na notação prefixada, o operador vem antes dos operandos. A calculadora deve suportar as quatro operações básicas: adição, subtração, multiplicação e divisão.

```
print(calcular_prefixo("+ 2 3"))
# Saída esperada: 5
print(calcular_prefixo("* 4 5"))
# Saída esperada: 20
print(calcular_prefixo("- 10 4"))
# Saída esperada: 6
print(calcular_prefixo("/ 8 2"))
# Saída esperada: 4.0
print(calcular_prefixo("+ * 2 3 4"))
# Interpretação: (2 * 3) + 4 = 6 + 4 = 10
print(calcular_prefixo("- + 5 6 * 2 3"))
# Interpretação: (5 + 6) - (2 * 3) = 11 - 6 = 5
print(calcular_prefixo("/ + 10 5 3"))
# Interpretação: (10 + 5) / 3 = 15 / 3 = 5.0
print(calcular_prefixo("* + 1 2 + 3 4"))
# Interpretação: (1 + 2) * (3 + 4) = 3 * 7 = 21
print(calcular_prefixo("- / 20 4 + 3 2"))
# Interpretação: (20 / 4) - (3 + 2) = 5 - 5 = 0
print(calcular_prefixo("+ * 2 3 / 10 2"))
# Interpretação: (2 * 3) + (10 / 2) = 6 + 5 = 11
```

https://pt.wikipedia.org/wiki/No

Introdução à Programação em



python™

https://evandro-crr.github.io/intro-python

Evandro Chagas Ribeiro da Rosa