Variáveis e Reuso de Software

Senai Londrina: Pedro Capelari

December 6, 2024

Contents

1 O que são variáveis?

Variáveis são espaços de memória reservados para armazenar valores. Em Python, as variáveis são criadas automaticamente quando você atribui um valor a elas.

2 Regras de Nomeação de Variáveis

- Os nomes de variáveis podem conter letras (a-z, A-Z), números (0-9), e o caractere de sublinhado (_).
- Os nomes de variáveis devem começar com uma letra ou um sublinhado.
- Python diferencia maiúsculas e minúsculas, então `variavel` e `Variavel` são duas variáveis diferentes.

3 Exemplo de Uso de Variáveis

```
# Atribuicao de variaveis
nome = "Joao"
idade = 25
altura = 1.75

# Exibicao dos valores das variaveis
print(f"Nome: {nome}")
print(f"Idade: {idade}")
print(f"Altura: {altura}")
```

4 Tipos de Variáveis

Python é uma linguagem de tipagem dinâmica, o que significa que você não precisa declarar o tipo de variável. O Python irá inferir o tipo baseado no valor atribuído.

- **int**: Números inteiros (ex: 5, -3, 42)
- **float**: Números de ponto flutuante (ex: 3.14, -0.001)
- **str**: Cadeias de caracteres (ex: "Olá, Mundo!")
- **bool**: Valores booleanos (True, False)

5 Conversão de Tipo

```
# Conversão de tipos
numero_str = "123"
numero_int = int(numero_str)

print(type(numero_str))  # <class 'str'>
print(type(numero_int))  # <class 'int'>
```

Reuso de Software em Python

6 O que é Reuso de Software?

Reuso de software é a prática de reutilizar componentes de software em diferentes contextos e aplicações para aumentar a eficiência e reduzir o retrabalho.

7 Funções

Funções são blocos de código reutilizáveis que realizam uma tarefa específica. Em Python, você define uma função usando a palavra-chave `def`.

7.1 Exemplo de Função

```
# Definição de uma função
def saudacao(nome):
    return f"Olá, {nome}!"

# Chamada da função
mensagem = saudacao("Maria")
print(mensagem)
```

8 Módulos

Módulos são arquivos que contêm definições e instruções em Python. Você pode importar um módulo e reutilizar seu código em diferentes partes do seu programa.

8.1 Exemplo de Módulo

Crie um arquivo chamado `meu_modulo.py`:

```
# meu_modulo.py
def adicionar(a, b):
    return a + b

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

Em seu programa principal, você pode importar e usar as funções do módulo:

```
# programa_principal.py
import meu_modulo

resultado_adicao = meu_modulo.adicionar(5, 3)
resultado_subtracao = meu_modulo.subtrair(5, 3)

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

9 Pacotes

Pacotes são coleções de módulos. Eles ajudam a organizar o código em hierarquias de diretórios.

9.1 Exemplo de Pacote

Crie uma estrutura de diretórios como esta:

```
meu_pacote/
    __init__.py
    calculadora.py
```

No arquivo `calculadora.py`:

```
# calculadora.py
def multiplicar(a, b):
    return a * b

def dividir(a, b):
```

```
    if b != 0:
        return a / b
    else:
        return "Divisão por zero não permitida"
```

No seu programa principal:

```
# programa_principal.py
from meu_pacote import calculadora

resultado_multiplicacao = calculadora.multiplicar(4, 5)
resultado_divisao = calculadora.dividir(10, 2)

print(f"Multiplicação: {resultado_multiplicacao}")
print(f"Divisão: {resultado_divisao}")
```

Execução Contínua de Softwares em Python

10 O que é Integração Contínua (CI)?

Integração Contínua é uma prática onde os desenvolvedores fazem commits frequentes de código em um repositório compartilhado. Cada commit aciona automaticamente a construção e os testes do software, garantindo que erros sejam detectados e corrigidos rapidamente.

11 O que é Entrega Contínua (CD)?

Entrega Contínua é uma extensão da CI, onde o software que passou pelos testes é automaticamente implantado em um ambiente de produção ou preparado para implantação. A CD garante que o software esteja sempre em um estado pronto para liberação.

12 Ferramentas Comuns para CI/CD

- **Jenkins**: Uma ferramenta de automação open-source que suporta a construção, teste e implantação de software.
- **Travis CI**: Um serviço de CI/CD que se integra facilmente com projetos hospedados no GitHub.
- **GitHub Actions**: Funcionalidades de CI/CD integradas diretamente no GitHub.

13 Configuração Básica de CI/CD com GitHub Actions

GitHub Actions permite automatizar fluxos de trabalho diretamente no GitHub. Vamos configurar uma simples integração contínua para um projeto Python.

13.1 Estrutura do Projeto

```
meu_projeto/
 .github/
    workflows/
        ci.yml
 src/
    main.py
 tests/
    test_main.py
 requirements.txt
 README.md
```

13.2 Arquivo de Configuração do GitHub Actions: `.github/workflows/ci.yml`

```yaml
name: CI

on: [push, pull_request]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout repository
      uses: actions/checkout@v2

    - name: Set up Python
      uses: actions/setup-python@v2
      with:
        python-version: 3.8

    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt

    - name: Run tests
      run: |
```

```
pytest tests/
```

13.3 Passos Explicados

- **Nome do Workflow**: `name: CI` – Nome do workflow.
- **Eventos de Disparo**: `on: [push, pull_request]` – O workflow é acionado em pushes e pull requests.
  - **Ambiente**: `runs-on: ubuntu-latest` – O trabalho é executado em um ambiente Ubuntu.
  - **Passos**:
    - **Checkout do Repositório**: Usa a ação `actions/checkout` para fazer checkout do código.
    - **Configuração do Python**: Usa a ação `actions/setup-python` para configurar a versão do Python.
    - **Instalação de Dependências**: Instala as dependências listadas em `requirements.txt`.
    - **Execução dos Testes**: Executa os testes com `pytest`.

14 Exemplo de Código Python e Testes

14.1 Código Fonte: `src/main.py`

14.2 Código Fonte: `src/main.py`

```
def soma(a, b):
    return a + b

if __name__ == "__main__":
    print(soma(1, 2))
```

14.3 Teste Unitário: `tests/test_main.py`

```
import pytest
from src.main import soma

def test_soma():
    assert soma(1, 2) == 3
    assert soma(-1, 1) == 0
    assert soma(0, 0) == 0
```

14.4 Arquivo de Requisitos: `requirements.txt`

```
pytest
```

15 Execução Local de CI/CD com Tox

Além de ferramentas baseadas em nuvem, você pode configurar um ambiente local de CI/CD usando Tox, que é uma ferramenta de automação de testes em múltiplos ambientes.

15.1 Instalação do Tox

```
pip install tox
```

15.2 Configuração do Tox: `tox.ini`

```
[tox]
envlist = py38

[testenv]
deps = pytest
commands = pytest
```

15.3 Executando Tox

```
tox
```

16 Benefícios da CI/CD

- **Detecção Precoce de Erros**: Testes automáticos identificam falhas logo após os commits.
- **Entrega Rápida e Segura**: Automação do processo de implantação reduz o tempo de liberação.
- **Melhor Colaboração**: Equipes podem integrar seu trabalho de forma contínua, evitando conflitos de integração.