LINGUAGEM DE PROGRAMAÇÃO APLICADA

AULA 6

Prof. Vinicius Pozzobon Borin

CONVERSA INICIAL

Ao longo desta etapa, você aprenderá sobre:

- ferramentas de suporte ao desenvolvedor;
- testes unitários (unit test);
- geração de executável.

TEMA 1 – FERRAMENTAS DE SUPORTE AO DESENVOLVEDOR: CODE COMPLETION E REFACTORINGS

Uma IDE (*Integrated Development Environment)*, como o PyCharm, desempenha um papel crucial no desenvolvimento de software, especialmente para linguagens de programação como Python. Aqui estão algumas maneiras pelas quais uma IDE pode ser importante:

- Facilidade de Desenvolvimento: IDEs oferecem uma variedade de recursos que tornam o processo de codificação mais eficiente e menos propenso a erros. Isso inclui realce de sintaxe, conclusão de código, verificação de erros em tempo real e formatação automática.
- Gerenciamento de Projetos: IDEs como o PyCharm permitem que os desenvolvedores organizem seus projetos de forma eficaz, com ferramentas para gerenciar dependências, configurar ambientes virtuais e integrar sistemas de controle de versão como o Git.
- Depuração avançada: Uma IDE oferece recursos avançados de depuração, como pontos de interrupção, inspeção de variáveis, rastreamento de pilha e execução passo a passo do código. Isso é essencial para identificar e corrigir bugs no código.
- Integração com ferramentas de terceiros: IDEs muitas vezes integram-se a uma variedade de ferramentas e serviços de terceiros que são úteis no desenvolvimento de software, como bancos de dados, servidores web, frameworks e bibliotecas.
- Refatoração de código: IDEs oferecem recursos de refatoração que permitem aos desenvolvedores reestruturarem o código de forma segura e eficiente, ajudando a melhorar sua qualidade e manutenção.
- Suporte a testes automatizados: com integração a frameworks de teste, as IDEs facilitam a escrita, execução e depuração de testes

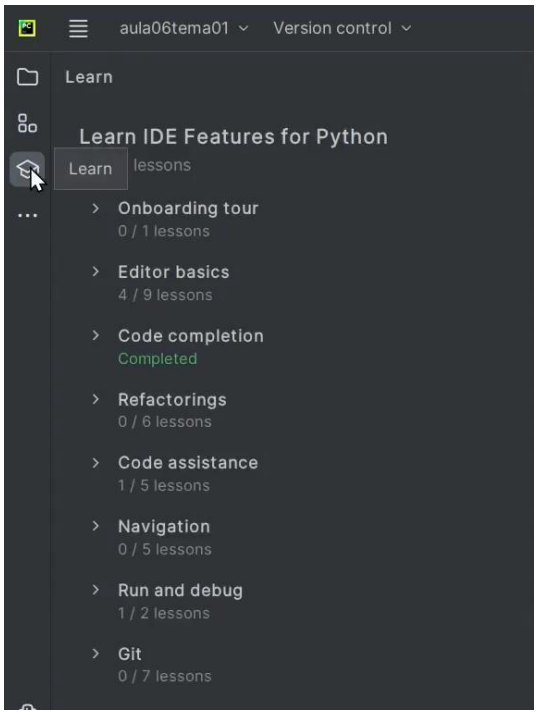
automatizados, o que é fundamental para garantir a qualidade do software.

O menu “Learn” no PyCharm é uma seção dedicada que oferece recursos educacionais para desenvolvedores, especialmente aqueles que estão aprendendo Python ou aprimorando suas habilidades de programação. Essa seção geralmente inclui recursos como tutoriais, cursos, documentação e exemplos de código que visam ajudar os usuários a entenderem melhor a linguagem Python e como usar o PyCharm de maneira mais eficaz.

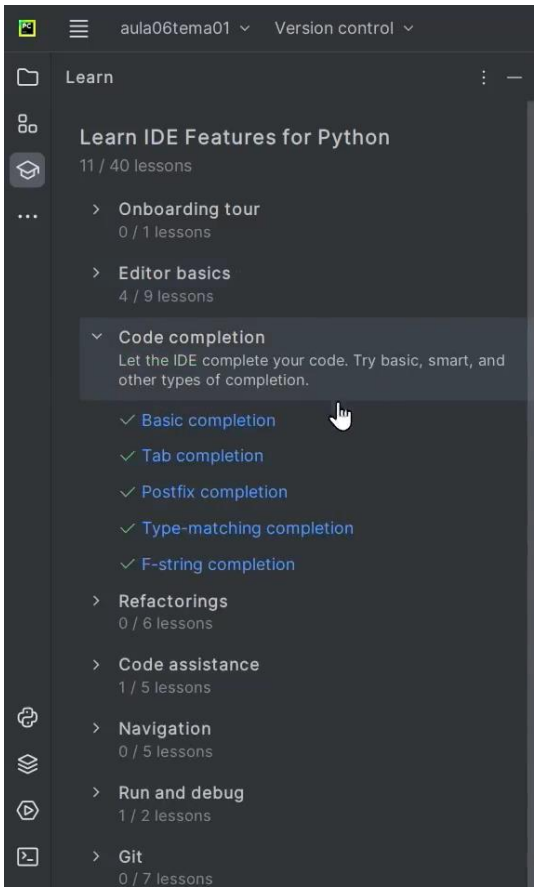
Aqui estão alguns dos recursos comuns que você pode encontrar no menu “Learn” do PyCharm:

- Tutoriais interativos: estes são tutoriais passo a passo que ajudam os usuários a aprender conceitos fundamentais de Python e também a usar recursos específicos do PyCharm. Eles geralmente incluem exemplos de código e exercícios práticos.
- Documentação integrada: a documentação oficial do Python e do PyCharm pode ser acessada diretamente do IDE. Isso permite que os desenvolvedores consultem rapidamente informações sobre sintaxe, bibliotecas, APIs e recursos específicos do PyCharm enquanto estão codificando.
- Exemplos de código: o PyCharm pode oferecer uma variedade de exemplos de código prontos para uso, que os usuários podem explorar para entender como implementar diferentes funcionalidades em Python ou como usar recursos específicos do IDE.

Para acessar a ferramenta de Learn do PyCharm, acesse no menu lateral esquerdo:



Para acessar o tutorial de code completion, acesse:



TEMA 2 – FERRAMENTAS DE SUPORTE: CODE ASSISTANCE E DEBUGGING

2.1 Code Assistance

O “Code Assistance” (ou assistência de código) em uma IDE de programação refere-se a um conjunto de recursos projetados para auxiliar os desenvolvedores durante o processo de escrita de código. Esses recursos visam aumentar a produtividade, reduzir erros e melhorar a experiência geral de desenvolvimento. Aqui estão alguns dos principais aspectos do “Code Assistance”:

- Realce de sintaxe: a IDE destaca diferentes partes do código com cores diferentes, facilitando a visualização e a compreensão da estrutura do código. Isso ajuda os desenvolvedores a identificar rapidamente erros de sintaxe e problemas de formatação.
- Conclusão de código (*Code Completion*): ao digitar, a IDE sugere automaticamente opções para completar o código com base no contexto atual. Isso inclui nomes de variáveis, métodos, funções, palavras-chave e até mesmo trechos de código predefinidos. A conclusão de código ajuda a reduzir o tempo gasto em digitação e minimiza erros de digitação.
- Inspeção de código (*Code Inspection*): a IDE realiza análises estáticas do código em tempo real para identificar possíveis erros, inconsistências ou práticas de codificação não recomendadas. Isso pode incluir avisos sobre variáveis não utilizadas, problemas de estilo de código, possíveis exceções e muito mais.
- Refatoração de código (*Code Refactoring*): a IDE oferece ferramentas para reestruturar e otimizar o código de forma segura e eficiente. Isso inclui a renomeação de variáveis, extração de métodos, modificação de assinaturas de funções, remoção de código redundante e outras transformações que melhoram a qualidade e a legibilidade do código.
- Navegação no código: a IDE permite que os desenvolvedores naveguem facilmente pelo código-fonte, pulando para definições de classes, funções, variáveis e outros elementos, bem como navegando por referências cruzadas. Isso facilita a compreensão da estrutura do projeto e a localização de informações relevantes.

No geral, a assistência de código em uma IDE é projetada para tornar o processo de desenvolvimento mais eficiente, ajudando os desenvolvedores a escrever, entender e modificar o código com mais facilidade e precisão. Isso é especialmente útil em linguagens de programação complexas ou projetos de grande escala, onde a compreensão e a manutenção do código podem ser desafios significativos.

2.2 Debug

O debug em programação é o processo de identificar, isolar e corrigir erros em um programa de computador. Os erros podem variar de simples erros de sintaxe a problemas mais complexos de lógica ou comportamento do programa. O processo de debug é essencial para garantir que um programa funcione corretamente e produza os resultados desejados. Aqui estão alguns dos principais aspectos do debug em programação:

- Identificação de erros: o primeiro passo no processo de debug é identificar que um erro ocorreu. Isso pode ser indicado por mensagens de erro, comportamento inesperado do programa, ou resultados incorretos.
- Reprodução do erro: o próximo passo é reproduzir o erro de forma consistente. Isso pode envolver executar o programa várias vezes com entradas diferentes ou em diferentes condições de execução para entender as circunstâncias em que o erro ocorre.
- Isolamento do erro: uma vez que o erro foi reproduzido, o próximo passo é isolar a causa do erro. Isso pode envolver a revisão do código-fonte relevante, a identificação de variáveis ou operações problemáticas, e a análise do fluxo de execução do programa para entender onde o erro está ocorrendo.
- Depuração ativa: durante o processo de isolamento do erro, os desenvolvedores geralmente usam técnicas de depuração ativa para examinar o estado do programa em tempo de execução. Isso pode incluir a inserção de pontos de interrupção no código, inspeção de variáveis, rastreamento de pilha e execução passo a passo do código para entender como o programa está se comportando.
- Correção do erro: uma vez que a causa do erro foi identificada, os desenvolvedores podem fazer as correções necessárias no código para

resolver o problema. Isso pode envolver a correção de erros de sintaxe, ajuste de lógica de programação, manipulação de exceções, entre outras ações.

- Testes de regressão: após a correção do erro, é importante realizar testes de regressão para garantir que a correção não introduziu novos problemas no programa. Isso pode envolver a execução de testes automatizados ou manuais para verificar se o programa está funcionando conforme o esperado.

O processo de debug pode ser facilitado pelo uso de ferramentas de desenvolvimento integradas (IDEs) que oferecem recursos avançados de depuração, como pontos de interrupção, inspeção de variáveis, rastreamento de pilha e execução passo a passo do código. Essas ferramentas ajudam os desenvolvedores a entender melhor o comportamento do programa e a identificar e corrigir erros com mais eficiência.

TEMA 3 – COMEÇANDO COM UNIT TEST

Testes unitários, ou unit tests, são uma prática de desenvolvimento de software em que partes individuais (ou unidades) do código são testadas isoladamente para garantir que funcionem como esperado. Uma unidade pode ser uma função, um método de uma classe ou até mesmo uma pequena parte de código que executa uma determinada funcionalidade.

Em termos simples, os testes unitários verificam se cada parte do código faz o que é suposto fazer. Eles são escritos pelo desenvolvedor e têm como objetivo validar se as unidades de código estão produzindo os resultados corretos para diferentes entradas ou cenários.

Por exemplo, se você tem uma função que soma dois números, um teste unitário para essa função pode verificar se a soma de 2 e 3 é realmente 5. Ou se você tem um método que calcula o preço total de uma compra com desconto, o teste unitário pode verificar se o cálculo está sendo feito corretamente para diferentes combinações de itens e descontos.

Os testes unitários são importantes porque ajudam a garantir a qualidade do código. Eles fornecem uma maneira de verificar se as unidades de código estão produzindo os resultados esperados, o que ajuda a evitar bugs e problemas de funcionamento. Além disso, facilitam a detecção de regressões.

Se alguma mudança no código quebra uma funcionalidade existente, os testes unitários podem detectar isso imediatamente, permitindo uma correção rápida antes que o problema se torne mais grave.

Vejamos um exemplo envolvendo classes e métodos. Imagine uma classe Calculadora, que faz soma e subtração.

```
# Classe que queremos testar
class Calculadora:
    def soma(self, a, b):
        return a + b

    def subtracao(self, a, b):
        return a - b
```

Um exemplo de teste para cada um dos métodos pode ser visto a seguir.

```
import unittest

# Classe de teste para a classe Calculadora
class TestCalculadora(unittest.TestCase):
    # Teste para o método soma
    def test_soma(self):
        calc = Calculadora()
        resultado = calc.soma(2, 3)
        self.assertEqual(resultado, 5)

    # Teste para o método subtração
    def test_subtracao(self):
        calc = Calculadora()
        resultado = calc.subtracao(5, 3)
        self.assertEqual(resultado, 2)
```

Note que:

- A classe TestCalculadora herda de unittest.TestCase e contém métodos de teste para os métodos da classe Calculadora.
- Os métodos de teste têm nomes começando com test_ para que o módulo unittest os reconheça como testes.

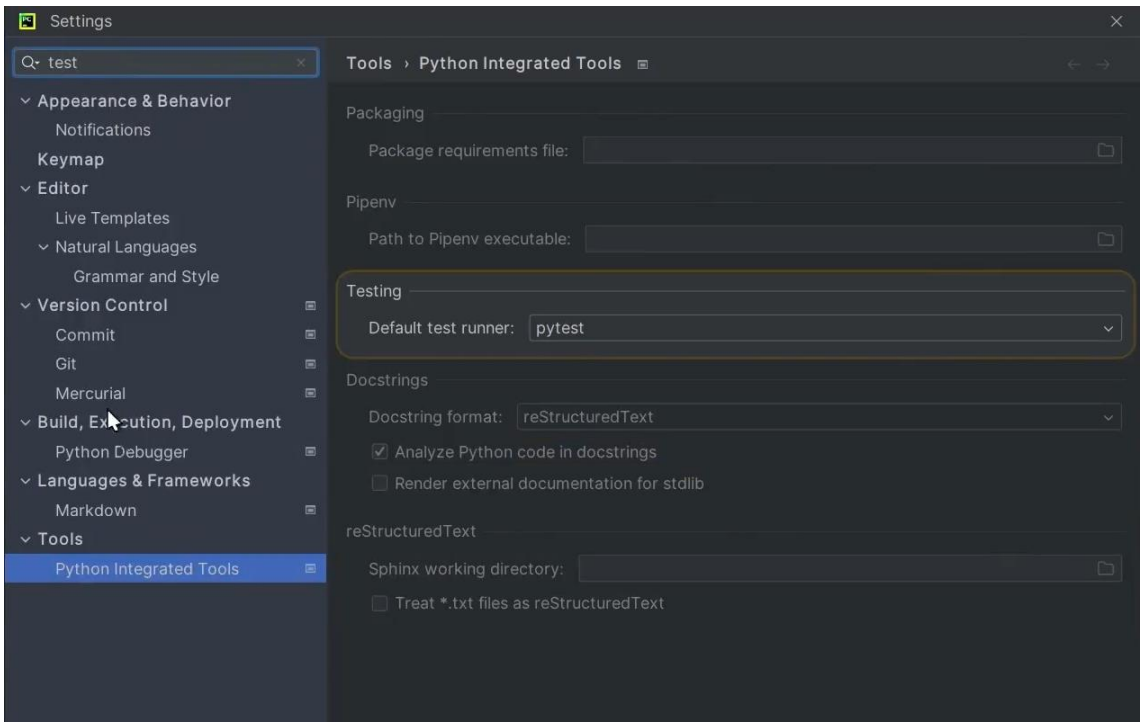
- Dentro de cada método de teste, usamos assertEqual() para verificar se o resultado retornado pelo método é o esperado.

Para executar estes testes, você pode salvar o código em um arquivo Python (por exemplo, test_calculadora.py) e executar o arquivo. Os resultados dos testes serão exibidos no console.

TEMA 4 – PYTEST

O unittest que vimos no bloco anterior é um framework de teste integrado ao Python. Todavia, existe também o PyTest, que é uma estrutura de teste de terceiros, mas que acaba sendo bastante poderosa e flexível. Para ativar o PyTest no PyCharm, primeiro é necessário instalar a ferramenta. Para isso, faça: pip install pytest.

Uma vez instalado, abra o menu de configurações (Settings) e pesquise por test:



Você irá localizar a ferramenta Python Integrated Tools > Testing. Altere de unittest para pytest. Mas quais as vantagens de se trabalhar com o PyTest?

- Sintaxe mais limpa e concisa: o pytest tem uma sintaxe mais limpa e concisa para escrever testes, resultando em código mais legível e fácil de manter.

- Suporte nativo para asserts simplificados: o pytest permite o uso de assert simples sem a necessidade de métodos específicos de asserção, o que torna os testes mais claros e diretos.
- Descoberta automática de testes: o pytest pode descobrir automaticamente todos os testes em um diretório e executá-los, eliminando a necessidade de estruturar seus arquivos de teste de uma maneira específica.
- Parametrização de testes integrada: o pytest oferece suporte integrado para testes parametrizados, permitindo que você execute o mesmo teste com diferentes conjuntos de dados de entrada com facilidade.
- Suporte a plugins: o pytest tem uma vasta coleção de plugins disponíveis que estendem suas funcionalidades, como cobertura de código, testes parametrizados, geração de relatórios, entre outros.
- Integração com outras ferramentas: o pytest se integra facilmente com outras ferramentas de desenvolvimento, como IDEs, sistemas de integração contínua e cobertura de código, tornando-o uma escolha popular em muitos ambientes de desenvolvimento.

Vejamos novamente a Calculadora que faz soma e subtração:

```
# Classe que queremos testar
class Calculadora:
    def soma(self, a, b):
        return a + b

    def subtracao(self, a, b):
        return a - b
```

Agora queremos testar seus métodos com PyTest:

```
# test_calculadora.py
import pytest
from calculadora import Calculadora

# Teste para o método soma
def test_soma():
    calc = Calculadora()
    resultado = calc.soma(2, 3)
```

```
    assert resultado == 5

# Teste para o método subtração
def test_subtracao():
    calc = Calculadora()
    resultado = calc.subtracao(5, 3)
    assert resultado == 2
```

Explicando o teste para o método soma:

- Definimos uma função chamada test_soma(). Essa função é um teste unitário para o método soma da classe Calculadora.
- Criamos uma instância da classe Calculadora chamada calc.
- Chamamos o método soma com os argumentos 2 e 3 e armazenamos o resultado em resultado.
- Usamos a asserção assert para verificar se o resultado é igual a 5.

TEMA 5 – GERANDO EXECUTÁVEL DO PYTHON PARA WINDOWS

Gerar um executável a partir de um código em Python pode simplificar a distribuição, melhorar a experiência do usuário e, em certos casos, proteger o código-fonte. Isso torna a prática valiosa em uma variedade de contextos, incluindo desenvolvimento de software comercial, distribuição de aplicativos e criação de ferramentas de linha de comando. Vamos agora aprender a gerar o executável de um código Python de duas maneiras distintas:

- PyInstaller;
- cx_Freeze.

Para gerar um instalador de um script Python usando o PyInstaller, você pode seguir estes passos:

1. Instalação do PyInstaller: se você ainda não tiver o PyInstaller instalado, pode instalá-lo usando o pip: pip install pyinstaller.
2. Navegue até o diretório do seu projeto: no terminal ou prompt de comando, navegue até o diretório que contém seu script Python ou seus arquivos.
3. Execução do PyInstaller: execute o PyInstaller, fornecendo o caminho para o seu script Python como argumento: pyinstaller seu_script.py

O PyInstaller irá então criar uma pasta chamada dist no diretório atual, que conterá o instalador ou os arquivos do executável, dependendo do sistema operacional.

Se você estiver no Windows, o PyInstaller criará um arquivo .exe na pasta dist. Se estiver no Linux, criará um arquivo binário executável, e se estiver no macOS, criará um aplicativo .app.

Para gerar um instalador do Python usando o cx_Freeze, você precisa seguir alguns passos. O cx_Freeze é uma ferramenta de criação de pacotes que transforma aplicativos Python em executáveis independentes. Aqui está um exemplo de como usar o cx_Freeze para criar um instalador:

1. Instalação: certifique-se de ter o cx_Freeze instalado. Você pode instalar usando pip: pip install cx-Freeze
2. Criar um arquivo de configuração: crie um arquivo de configuração chamado setup.py. Este arquivo irá conter informações sobre o seu projeto, como o nome do arquivo de script principal e quais módulos externos devem ser incluídos no pacote. Aqui está um exemplo simples de setup.py:

```
from cx_Freeze import setup, Executable

setup(
    name="meu_programa",
    version="1.0",
    description="Descrição do meu programa",
    executables=[Executable("seu_script.py")]
)
```

Substitua “meu_programa” pelo nome do seu programa, “1.0” pela versão e “Descrição do meu programa” por uma breve descrição do que seu programa faz. Certifique-se de substituir “seu_script.py” pelo nome do arquivo de script Python principal do seu programa.

3. Executando o cx_Freeze: abra um terminal na mesma pasta onde está o arquivo setup.py e execute o seguinte comando: `python setup.py build`.
4. Opcional: criando um instalador: Se você deseja criar um instalador para o seu aplicativo, você pode usar ferramentas adicionais como Inno Setup,

NSIS ou WiX Toolset. Essas ferramentas permitem criar instaladores personalizados para o seu aplicativo.

5. Testando o aplicativo: após a criação do pacote, você pode testar o aplicativo executando o arquivo executável criado na pasta build.

Este é um procedimento básico para criar um instalador usando o cx_Freeze. Certifique-se de ler a documentação oficial do cx_Freeze para mais informações e opções avançadas.

FINALIZANDO

Ao longo desta etapa, você aprendeu sobre:

- ferramentas de suporte ao desenvolvedor: code completion e refactorings;
- ferramentas de suporte: code assistance e debug;
- começando com testes unitários;
- Pytest;
- gerando executável do Python para Windows.

REFERÊNCIAS

PUGA, S.; RISSETI, G. **Lógica de Programação e Estrutura de Dados**. 3. ed. São Paulo: Pearson, 2016.

ROSEWOOD, E. **Programação Orientada a Objetos em Python**: dos fundamentos às técnicas avançadas. Edição Independente.

ZHART, D. **Design Patterns**. Refactoring Guru. Disponível em: <https://refactoring.guru/design-patterns>. Acesso em: 14 mar. 2024.