Automatizando Excel

Asimov Academy

ASIMOV

Conteúdo

01. Instalando o Openpyxl

Bem-vindo ao módulo **Automatizando Excel com Python**. Neste curso, vamos utilizar a biblioteca `openpyxl` para automatizar a criação, leitura e edição de planilhas Excel diretamente com código Python.

Se você já trabalhou com o `pandas` antes, pode estar se perguntando: *qual a diferença entre `pandas` e `openpyxl` para lidar com planilhas?* Enquanto o `pandas` foca na manipulação de dados e usa o Excel apenas como meio de entrada ou saída, o `openpyxl` permite **acesso completo** à estrutura e estilo das planilhas Excel. Com ele, é possível:

- Inserir fórmulas
- Mudar formatações (cores, fontes)
- Inserir imagens e gráficos
- Criar ou editar várias abas (sheets)

Ou seja, é ideal para gerar relatórios Excel programaticamente e compartilhá-los com outras pessoas da sua empresa.

Preparando o ambiente

Deixe na sua pasta de trabalho apenas os seguintes arquivos:

- `logo_gato.png` – para inserir imagens futuramente
- `exemplo.xlsx` – planilha para manipulação
- `CardioData.xlsx` – dataset com informações de pessoas com doenças cardíacas.

Estes arquivos estarão disponíveis na pasta zipada que também contem essa apostila.

Nomeando seu script

No VS Code, crie um arquivo chamado `openpyxl_.py`.

Evite dar o mesmo nome da biblioteca (`openpyxl.py`) para não causar conflitos na importação. Para contornar isso, adicionamos o _ no final do nome do arquivo.

Instalando a biblioteca

Para instalar o `openpyxl`, abra o terminal e execute:

```
pip install openpyxl
```

Lendo um arquivo Excel com OpenPyXL

Vamos começar importando a função `load_workbook`, responsável por carregar um arquivo `.xlsx`.

```
from openpyxl import load_workbook

wb = load_workbook("exemplo.xlsx")
```

Observação: arquivos antigos `.xls` podem não ser lidos corretamente. Use `.xlsx` (Excel 2010 ou superior) para evitar erros.

Se o script estiver na mesma pasta que o arquivo, não é necessário passar o caminho completo. Mas, caso esteja em outro local, você pode usar o módulo `os` para gerar o caminho:

```
import os
caminho = os.path.join("pasta", "exemplo.xlsx")
wb = load_workbook(caminho)
```

Entendendo o objeto Workbook

Depois de carregar o arquivo, o objeto retornado é uma instância de `Workbook`, que representa o arquivo Excel inteiro:

```
print(type(wb))
# <class 'openpyxl.workbook.workbook.Workbook'>
```

Listando as abas (sheets)

Para listar todas as abas (ou *sheets*) de um arquivo Excel, use o atributo `sheetnames`:

```
sheetnames = wb.sheetnames
print(sheetnames)
# ['Sheet1', 'Sheet2', 'Sheet3']
```

Importante: a função `get_sheet_names()` era usada em versões antigas do `openpyxl`, mas está obsoleta. O uso correto atualmente é com `wb.sheetnames`.

Para selecionar uma aba, use o nome diretamente como se fosse um dicionário:

```
sheet = wb["Sheet1"]
```

Selecionando células

Agora que temos a aba selecionada, podemos acessar células específicas:

```
celula = sheet["A3"]
print(celula.value)
# Exibe o valor da célula A3, no caso do nosso arquivo  "06/04/2015 12:46"
```

Conclusão

Neste primeiro capítulo, aprendemos a instalar a biblioteca `openpyxl`, carregar arquivos do Excel utilizando a função `load_workbook`, listar e acessar as abas das planilhas, além de selecionar e ler valores de células específicas. Se tudo funcionou corretamente no seu ambiente, parabéns! Você está pronto para seguir. Seguiremos explorando como editar e formatar as planilhas.

02. Conceitos básicos

Neste capítulo, vamos nos aprofundar nas funcionalidades básicas da biblioteca `openpyxl`, com foco em seleção e manipulação de células dentro de uma planilha Excel.

Acessando Células

Depois de carregar um `workbook` e selecionar uma aba (ou *sheet*), podemos acessar células individuais usando duas abordagens:

Notação padrão do Excel:

```
celula = sheet["B2"]
print(celula.value)  # Exemplo: 'Cherries'
```

Essa forma é útil quando você já sabe o endereço da célula (como “A1”, “B2”, etc).

Método `.cell(row, column)`:

```
celula = sheet.cell(row=2, column=2)  # Equivalente a B2
print(celula.value)
```

Esse método é mais prático para automações, pois permite trabalhar com índices numéricos diretamente (linha e coluna), facilitando a criação de loops. Também vale ressaltar que o índice numérico sempre começa em 1.

Navegando por Células com Loops

Para iterar sobre uma coluna (por exemplo, a coluna B), podemos usar o atributo `max_row`, que retorna o número máximo de linhas preenchidas:

```
for i in range(1, sheet.max_row + 1):
    print(sheet.cell(row=i, column=2).value)
```

Atenção: O método `.cell()` começa a contagem em 1, por isso usamos `range(1, ...)`.

Editando Valores de Células

Também é possível alterar valores:

```
sheet.cell(row=2, column=3).value = 75
# Substitui o valor da célula C2 por 75
```

No entanto, essas alterações ficam apenas na memória. Para que as mudanças sejam salvas no arquivo, é necessário usar:

```
wb.save("exemplo.xlsx")
```

Mesclando e Desfazendo Mesclagem de Células

Mesclar (merge) células:

```
sheet.merge_cells("A1:D1")
wb.save("exemplo.xlsx")
```

Desfazer a mesclagem:

```
sheet.unmerge_cells("A1:D1")
wb.save("exemplo.xlsx")
```

Inserindo e Deletando Linhas e Colunas

Inserir uma linha na posição 4:

```
sheet.insert_rows(4)
wb.save("exemplo.xlsx")
```

Deletar uma linha:

```
sheet.delete_rows(4)
wb.save("exemplo.xlsx")
```

Deletar colunas (por exemplo, da coluna 2 até a 6):

```
sheet.delete_cols(2, 5)  # Deleta da B até a F
wb.save("exemplo2.xlsx")
```

Adicionando Imagens

É possível adicionar imagens à planilha utilizando o módulo `drawing.image`:

```
from openpyxl.drawing.image import Image

img = Image("cat_logo.png")
sheet.add_image(img, "A1")
wb.save("exemplo2.xlsx")
```

A imagem será inserida na célula especificada. Para redimensioná-la ou reposicioná-la, consulte a documentação oficial.

Consulte a Documentação

A biblioteca `openpyxl` é extensa e possui muitos outros recursos, como fórmulas, gráficos e formatação avançada. Caso precise de funcionalidades adicionais, acesse a documentação oficial ou pesquise exemplos no Stack Overflow.

03. Fórmulas

Nesta aula, vamos aprender a trabalhar com **fórmulas no Excel usando a biblioteca `openpyxl`**. Até aqui, manipulamos células, planilhas, inserimos linhas e editamos valores. Agora, vamos adicionar **cálculos automáticos** nas planilhas usando fórmulas.

Criando um Arquivo Novo com `Workbook`

Em vez de abrir um arquivo existente, vamos criar um novo do zero:

```
from openpyxl import Workbook

wb = Workbook()
sheet = wb.active  # seleciona a aba ativa (padrão)
```

Agora que temos uma planilha nova, podemos começar a inserir valores e fórmulas.

Inserindo Valores e Fórmulas

Vamos adicionar dois valores e depois usar uma fórmula para somá-los:

```
sheet["A1"].value = 100
sheet["A2"].value = 200
sheet["A3"].value = "=SUM(A1:A2)"  # fórmula como string
```

> **Importante**: As fórmulas devem ser escritas em **inglês**, com **vírgulas** como separadores, exatamente como o Excel espera.

Salve o arquivo para visualizar no Excel:

```
wb.save("formulas.xlsx")
```

Abrindo o arquivo, você verá que a célula A3 soma automaticamente A1 e A2.

Traduzindo Fórmulas para Outras Células

No Excel, quando copiamos e colamos células com fórmulas, as referências são ajustadas automaticamente. Para fazer o mesmo no `openpyxl`, usamos o **tradutor de fórmulas**:

```
from openpyxl.formula.translate import Translator
```

Vamos adicionar novos valores e aplicar a tradução da fórmula:

```
sheet["B1"] = 300
sheet["B2"] = 250

# Fórmula original
formula = "=SUM(A1:A2)"

# Traduz a fórmula da célula A3 para B3
translated_formula = Translator(formula, origin="A3").translate_formula("B3")
sheet["B3"] = translated_formula
```

Salvando e abrindo novamente, veremos que a célula B3 soma corretamente B1 e B2.

Verificando Funções Suportadas

O `openpyxl` também permite consultar quais **funções do Excel são suportadas** pela biblioteca. Para isso:

```
from openpyxl.utils import FORMULAE

print(type(FORMULAE))  # <class 'frozenset'>
print("SUM" in FORMULAE)  # True
```

A variável `FORMULAE` é um conjunto congelado (`frozenset`) contendo todas as funções válidas. Você pode usar isso para verificar se uma função como `AVERAGEIF` está presente:

```
print("AVERAGEIF" in FORMULAE)  # Exemplo
```

Conclusão

Este capítulo abordou como criar um novo arquivo do Excel com a biblioteca `openpyxl`, inserir fórmulas utilizando strings, traduzir fórmulas automaticamente para outras células e verificar se determinadas funções do Excel são suportadas. Com esse conjunto de recursos, é possível automatizar planilhas complexas com fórmulas, trazendo a flexibilidade do Excel para dentro de scripts Python.

04. Estilização

Neste capítulo, vamos aprender como aplicar estilos às células de uma planilha Excel utilizando o `openpyxl`. Isso inclui trabalhar com fontes, preenchimentos, bordas, alinhamentos, cores e outros elementos visuais.

Como funciona a estilização no `openpyxl`

Cada célula do Excel, quando manipulada via `openpyxl`, é representada por um objeto que contém diversas informações: valor, tipo (número, texto, fórmula), estilo de fonte, cor, bordas, preenchimento, entre outros. Para realizar estilizações, utilizamos classes do módulo `openpyxl.styles`.

```
from openpyxl import Workbook
from openpyxl.styles import Font, PatternFill, Border, Side, Alignment, Protection
```

Estilizando fontes

A classe `Font` permite definir propriedades como:

- `name`: nome da fonte (ex: `'Calibri'`)
- `size`: tamanho da fonte
- `bold`: negrito (`True` ou `False`)
- `italic`: itálico
- `underline`: sublinhado (`'single'`, `'double'`, etc.)
- `strike`: tachado
- `color`: cor da fonte (em hexadecimal, ex: `'FFFF0000'` para vermelho)

Exemplo: Estilizando fonte da célula

```
wb = Workbook()
ws = wb.active
# Valor padrão na célula
ws["A1"] = "Texto em negrito"
ws["A1"].font = Font(bold=True)
# Outro exemplo com mais parâmetros
ws["A2"] = "Fonte grande e vermelha"
ws["A2"].font = Font(size=20, underline='single', color='FFFF0000')

wb.save("estilizado.xlsx")
```

Cores em RGB

A cor hexadecimal é composta por 8 dígitos: os dois primeiros representam o canal Alpha (transparência), seguidos pelos canais de cor RGB. Exemplo de vermelho:

```
color='FFFF0000'
```

Preenchimento da célula (Background)

A classe `PatternFill` define o fundo da célula:

```
fill = PatternFill(start_color='FFFFFF00', end_color='FF000000', fill_type='solid')
ws["A3"] = "Célula com fundo"
ws["A3"].fill = fill
```

Bordas

Para trabalhar com bordas, usamos as classes `Border` e `Side`. É possível definir bordas para cada lado da célula:

```
side = Side(style='double', color='FF0000FF')  # Azul
border = Border(right=side)

ws["A4"] = "Com borda à direita"
ws["A4"].border = border
```

Estilos disponíveis para bordas:

- `'thin'`
- `'medium'`
- `'dashed'`
- `'dotted'`
- `'double'`
- `'thick'`
- `'hair'`
- Entre outros.

Alinhamento

A classe `Alignment` permite configurar o alinhamento horizontal e vertical, rotação do texto e quebra automática:

```
ws["A5"] = "Texto centralizado"
ws["A5"].alignment = Alignment(horizontal='center', vertical='center')
```

Formatação de número

Você pode aplicar formatações como moeda, porcentagem, datas, etc., usando o atributo `number_format`:

```
from datetime import datetime

ws["A6"] = datetime.today()
ws["A6"].number_format = 'DD/MM/YYYY'
```

Proteção

Com a classe `Protection`, é possível configurar se a célula será protegida contra edição:

```
ws["A7"] = "Protegida"
ws["A7"].protection = Protection(locked=True)
```

Dica: Para explorar todas as opções possíveis de cada classe (como `Font`, `Alignment`, etc.), consulte a documentação oficial do openpyxl.

Exemplo visual

Abaixo, segue uma imagem que ilustra algumas das personalizações que fizemos com os exemplos acima:

Figure 1: Ilustração das edições

Conclusão

Neste capítulo, vimos como o Python, através da biblioteca `openpyxl`, oferece controle completo sobre a formatação de células em uma planilha do Excel. Aprendemos a modificar fontes, aplicar cores

de fundo, definir bordas personalizadas, ajustar alinhamentos e até mesmo configurar proteções em células específicas.

Com essas ferramentas, você já é capaz de criar planilhas não apenas funcionais, mas também visualmente organizadas e agradáveis — o que é essencial para destacar informações importantes e facilitar a leitura de dados. Além disso, essa personalização abre caminho para a criação de relatórios automatizados com um nível profissional de apresentação.

No próximo capítulo, vamos explorar a criação de gráficos dentro do Excel via Python — um recurso poderoso para visualizar dados diretamente das suas planilhas automatizadas.

05. Gráfico de Linhas

Introdução aos gráficos no OpenPyXL

Para fechar a parte teórica do nosso curso, vamos abordar a criação de gráficos no Excel utilizando a biblioteca `openpyxl`. A ideia é apresentar os conceitos fundamentais que permitirão a você construir gráficos simples e, a partir deles, adaptar para versões mais avançadas.

Vamos organizar esse conteúdo em uma pasta chamada `charts`, onde incluiremos exemplos de diferentes tipos de gráficos. Neste capítulo, vamos focar nos gráficos de **linha** (`LineChart`), que cobrem a maioria das situações práticas em visualização de dados no Excel.

A biblioteca `openpyxl` possui uma documentação bem completa sobre gráficos. Por isso, nosso foco será em entender a estrutura básica e a lógica necessária para criar um gráfico. Assim, quando você precisar de algo mais avançado, já saberá como consultar e aplicar.

Estrutura básica para criação de um gráfico de linha

Primeiro, vamos para a criação da planilha e o preenchimento dos dados. Abaixo, segue o código que usaremos para preencher a tabela.

```
from openpyxl import Workbook
from openpyxl.chart import LineChart, Reference
import datetime

wb = Workbook()
ws = wb.active

# Dados exemplo: datas + três séries numéricas
rows = [
    ["Data", "Batch 1", "Batch 2", "Batch 3"],
    [datetime.date(2023, 7, 1), 40, 30, 25],
    [datetime.date(2023, 7, 2), 42, 32, 27],
    [datetime.date(2023, 7, 3), 44, 34, 30],
    [datetime.date(2023, 7, 4), 46, 36, 32],
    [datetime.date(2023, 7, 5), 48, 38, 35],
    [datetime.date(2023, 7, 6), 50, 40, 37],
]

for row in rows:
    ws.append(row)

wb.save("line.xlsx")
```

Criação do gráfico de linha

Agora, criaremos de fato o gráfico, consumindo os dados da planilha criada anteriormente.

```
c1 = LineChart()
c1.title = "Line Chart"
c1.x_axis.title = "Data"
c1.y_axis.title = "Valores"

# Definindo a área de dados
data = Reference(ws, min_col=2, min_row=1, max_col=4, max_row=7)
c1.add_data(data, titles_from_data=True)

# Adicionando o gráfico à planilha
ws.add_chart(c1, "A10")
wb.save("line.xlsx")
```

Após abrir o arquivo `line.xlsx`, você verá um gráfico de linha gerado a partir dos dados que preenchemos na planilha.

Personalização do gráfico

A partir desse ponto, podemos customizar o gráfico, como trocar símbolos das séries, aplicar cores, modificar preenchimentos, entre outros. Cada série do gráfico é acessada via `c1.series[i]`, onde `i` representa o índice da série.

Exemplo: Personalizar marcadores

```
s1 = c1.series[0]
s1.marker.symbol = "triangle"
s1.marker.graphicalProperties.solidFill = "FF0000"  # Cor vermelha do marcador
s1.graphicalProperties.line.noFill = True  # Remove a linha conectando os pontos
```

Essas alterações devem ser feitas antes de adicionar o gráfico com `ws.add_chart()` e salvar o arquivo.

Observações importantes

- Sempre use a classe `Reference` para informar ao gráfico quais dados devem ser utilizados.
- O parâmetro `titles_from_data=True` instrui o gráfico a usar a primeira linha da seleção como título das séries.
- Após salvar e reabrir o arquivo, você verá todas as alterações refletidas no gráfico.
- Caso precise redesenhar o gráfico, recrie a instância do `LineChart()` para evitar conflitos.

Resultado final

Abaixo, segue o gráfico que geramos através do código que desenvolvemos ao longo do capítulo:

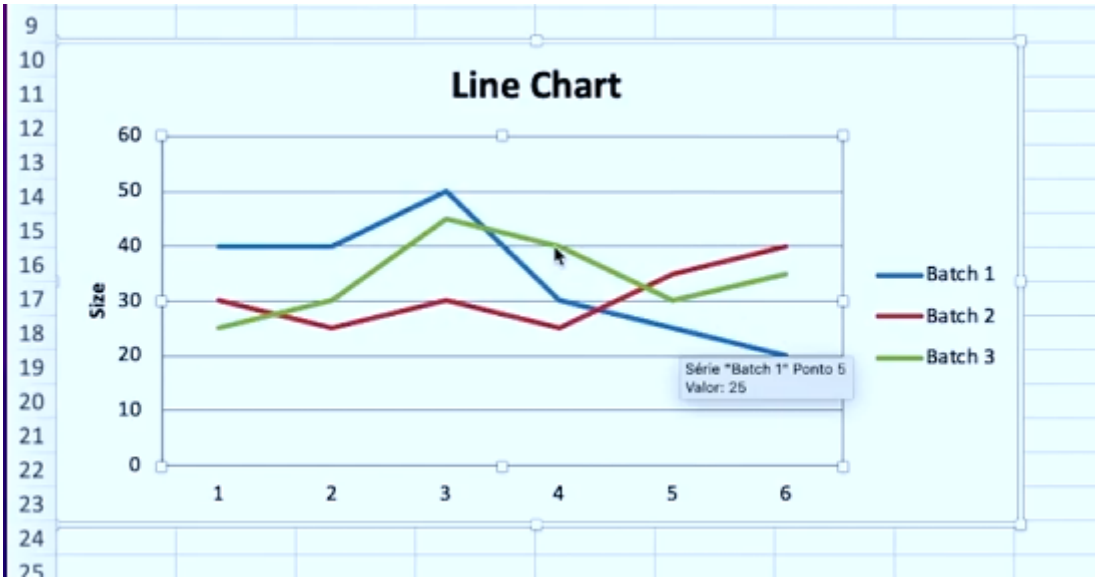

Figure 2: Gráfico gerado com Openpyxl

Conclusão

Neste capítulo, aprendemos a criar gráficos de linha no Excel utilizando a biblioteca `openpyxl`. Compreendemos a lógica por trás da construção de um gráfico: desde a preparação dos dados, passando pela definição de áreas de referência, até a inserção e estilização do gráfico na planilha.

Você viu que, com apenas algumas etapas, é possível transformar dados em representações visuais que facilitam a interpretação e a análise. Além disso, foi mostrado como personalizar o gráfico, incluindo títulos, eixos, marcadores e até cores — ampliando o potencial de apresentação dos dados.

A principal lição aqui é que, uma vez entendido o processo básico de criação de um gráfico simples, você estará apto a adaptar esse conhecimento para construir visualizações mais complexas, inclusive com outros tipos de gráficos como barras, pizza ou áreas. E sempre que surgir a necessidade de personalizações mais específicas, a documentação oficial da biblioteca será sua maior aliada.

No próximo capítulo, veremos como criar gráficos de barras — uma variação simples, mas muito útil, que segue a mesma lógica que acabamos de aprender.

06. Gráficos de barras

Neste capítulo, vamos aprender a criar **gráficos de barras** utilizando a biblioteca `openpyxl`, seguindo uma lógica muito semelhante à dos gráficos de linha. A ideia é construir um gráfico de barras vertical (tipo `col`) e também um horizontal (tipo `bar`). Além disso, veremos como clonar gráficos com o `deepcopy` para fazer variações rapidamente.

Estrutura básica para criação de um gráfico de barras

Assim como no capítulo anterior, primeiro, vamos para a criação da planilha e o preenchimento dos dados. Abaixo, segue o código que usaremos para preencher a tabela.

```
from openpyxl import Workbook
from openpyxl.chart import BarChart, Reference

wb = Workbook()
ws = wb.active

# Dados de exemplo
rows = [
    ("Number", "Batch 1", "Batch 2"),
    (1, 10, 30),
    (2, 40, 60),
    (3, 50, 70),
    (4, 20, 40),
    (5, 10, 20),
    (6, 50, 60),
]

for row in rows:
    ws.append(row)

wb.save("bar.xlsx")
```

Criação do gráfico de barras vertical

Agora, criaremos de fato o gráfico, consumindo os dados da planilha criada anteriormente.

```
chart1 = BarChart()
chart1.type = "col"
chart1.title = "Bar Chart"
chart1.x_axis.title = "Test Number"
chart1.y_axis.title = "Sample Length"

# Referência aos dados (Batch 1 e 2)
data = Reference(ws, min_col=2, min_row=1, max_col=3, max_row=7)
chart1.add_data(data, titles_from_data=True)
```

```
# Referência às categorias (coluna "Number")
cats = Reference(ws, min_col=1, min_row=2, max_row=7)
chart1.set_categories(cats)

# Inserção do gráfico
ws.add_chart(chart1, "E5")
wb.save("bar.xlsx")
```

Após abrir o arquivo `bar.xlsx`, você verá um gráfico de barras com as colunas verticais para as categorias de número e seus respectivos valores em Batch 1 e Batch 2.

Resultado final

Abaixo, segue o gráfico que geramos através do código que desenvolvemos ao longo do capítulo:

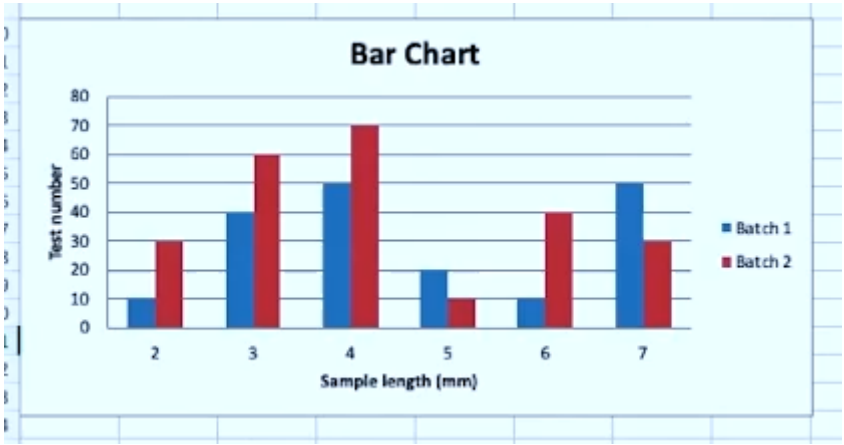

Figure 3: Gráfico em barras gerado com Openpyxl

Clonando o gráfico com deepcopy

Para criar uma nova versão do gráfico com alterações específicas, como mudar de vertical para horizontal, podemos usar o `deepcopy`:

```
from copy import deepcopy

chart2 = deepcopy(chart1)
chart2.type = "bar"
chart2.title = "Horizontal Bar Chart"

# Adicionando o novo gráfico à planilha
ws.add_chart(chart2, "G10")
wb.save("bar.xlsx")
```

> Evite fazer `chart2 = chart1`, pois isso apenas aponta para o mesmo objeto na memória. Com `deepcopy`, garantimos que são gráficos independentes.

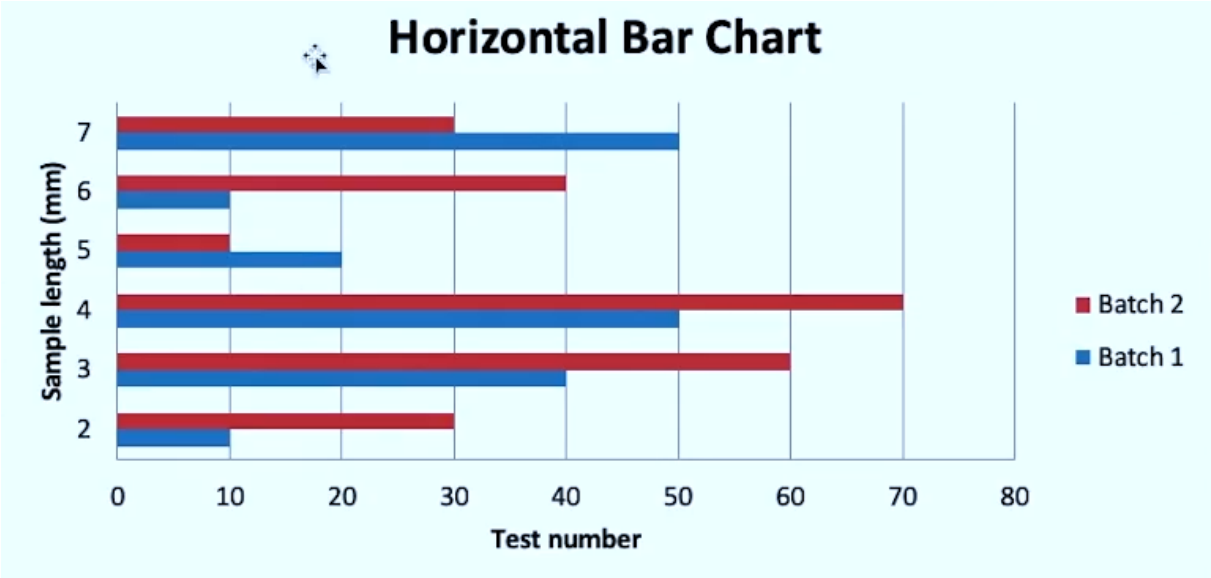

Figure 4: Gráfico em barra invertido

Conclusão

Neste capítulo, aprendemos a construir gráficos de barras verticais e horizontais no Excel com Python, utilizando a biblioteca `openpyxl`. Vimos como preparar os dados, definir as referências e personalizar o gráfico com títulos e categorias.

Também exploramos como duplicar gráficos usando o `deepcopy`, o que facilita muito na hora de criar variações sem reescrever o código. Esse recurso é essencial para manter a produtividade quando você precisa gerar múltiplos gráficos similares com pequenas diferenças.

A partir daqui, você já tem um bom domínio para criar representações visuais eficazes no Excel, automatizando relatórios e melhorando a análise de dados nas suas rotinas de trabalho.

No próximo capítulo, vamos explorar como construir gráficos de **pizza**, ideais para mostrar proporções e distribuições relativas de forma clara e intuitiva.