openpyxl - Documentation

2025-02-09

On this page

What is openpyxl?

openpyxl is a Python library for reading and writing Excel 2010 xlsx/xlsm/xltx/xltm files. It provides a robust and efficient way to interact with Excel spreadsheets programmatically, allowing developers to create, modify, and analyze data within Python applications. Unlike older libraries that only support older .xls file formats, openpyxl fully supports the more modern and feature-rich .xlsx format, including features like charts, images, styles, and formulas. It's a powerful tool for automating tasks involving spreadsheet data, integrating Excel functionalities into larger applications, and building data-driven tools.

Installation and Setup

The easiest way to install openpyxl is using pip, the Python package installer:

```
pip install openpyxl
```

This command will download and install the latest stable version of openpyxl and its dependencies. You'll need Python 3.7 or higher. If you encounter issues, ensure your pip is up-to-date using `pip install --upgrade pip`. For specific versions, consult the openpyxl documentation

for compatibility information and alternative installation methods. After successful installation, you can verify it by importing the library in a Python interpreter:

```
import openpyxl
```

If no errors occur, openpyxl is correctly installed.

Discover more Microsoft Excel Writing write Application software Installation spreadsheet Excel Macro applications Python

Basic Concepts and Terminology

openpyxl represents an Excel file as a workbook, which contains one or more worksheets. A worksheet is a grid of cells organized into rows and columns. Each cell can contain different data types: numbers, text, formulas, dates, booleans, etc. Key terminology includes:

- **Workbook:** The entire Excel file. It's represented by the `Workbook` class.
- **Worksheet:** A single sheet within the workbook. Represented by the `Worksheet` class.
- **Cell:** An individual entry in the worksheet, identified by its column and row coordinates (e.g., "A1", "B2"). Accessed using `worksheet["A1"]` or `worksheet.cell(row=1, column=1)`.
- **Row:** A horizontal sequence of cells.
- **Column:** A vertical sequence of cells.
- **Cell Value:** The content of a cell (string, number, formula, etc.).
- **Cell Style:** Formatting applied to a cell (font, alignment, number format, etc.).

Example: Reading and Writing a Simple Spreadsheet

Discover more Writing Application programming interface Libraries spreadsheets Spreadsheet write Python Process Excel Installation

This example demonstrates reading data from an existing Excel file and writing new data to a new file:

```
from openpyxl import load_workbook, Workbook

# Reading from an existing file
```

```
workbook = load_workbook('example.xlsx') # Replace 'example.xlsx' with
         your file
sheet = workbook.active  # Get the active worksheet

# Accessing cell values
cell_value = sheet['A1'].value
print(f"Value of A1: {cell_value}")

# Iterating through rows
for row in sheet.iter_rows(min_row=1, max_row=5, min_col=1, max_col=2):
         # Adjust ranges as needed
    for cell in row:
        print(f"Cell value: {cell.value}")


# Writing to a new file
new_workbook = Workbook()
new_sheet = new_workbook.active
new_sheet['A1'] = "Hello, openpyxl!"
new_sheet['B1'] = 123

# Save the new workbook
new_workbook.save("new_example.xlsx")
```

Remember to replace `'example.xlsx'` with the actual path to your Excel file. This example showcases basic read and write operations. For more advanced features (charts, formulas, styles), refer to the comprehensive openpyxl documentation.

Workbook and Worksheet Objects

Creating Workbooks

To create a new Excel workbook, use the `Workbook()` class constructor:

```
from openpyxl import Workbook

workbook = Workbook()  # Creates a new workbook with a single, empty
         worksheet

# Access the active worksheet (the first one by default)
sheet = workbook.active
```

```
# ... add data to the worksheet ...

workbook.save("my_new_workbook.xlsx") # Save the workbook to a file.
```

By default, a new workbook is created with a single, empty worksheet. You can add more worksheets as needed (see below).

Discover more libraries Python Spreadsheet Library Microsoft Excel Excel library Application software applications Writing

Accessing Worksheets

Several methods provide access to worksheets within a workbook. The most common is accessing the active worksheet (the one currently selected in Excel), using `workbook.active`. You can also access worksheets by name or index:

```
from openpyxl import load_workbook

workbook = load_workbook("my_workbook.xlsx")

# Accessing the active worksheet:
active_sheet = workbook.active

# Accessing a worksheet by name:
sheet_by_name = workbook["Sheet1"]  # Replace "Sheet1" with the actual
          sheet name

# Accessing a worksheet by index (index starts from 0):
sheet_by_index = workbook.worksheets[0] # Accesses the first worksheet


# Check if a worksheet exists before accessing it (avoid errors)
if "Sheet2" in workbook.sheetnames:
  sheet2 = workbook["Sheet2"]
```

Creating Worksheets

Discover more APIs Macro write library Python Macros libraries Process Writing Application software

Adding new worksheets is straightforward using the `create_sheet()` method. This method takes the sheet name as the first argument, and optional arguments for `index` (position within the workbook) and `before` (insert before another sheet):

```
from openpyxl import Workbook

workbook = Workbook()

# Create a new worksheet named "Data" at the end
new_sheet = workbook.create_sheet("Data")

# Create a new worksheet named "Summary" before the sheet named "Data"
summary_sheet = workbook.create_sheet("Summary", index=1,
         before=new_sheet)


workbook.save("workbook_with_sheets.xlsx")
```

Deleting Worksheets

To remove a worksheet, use the `remove()` method:

```
from openpyxl import load_workbook

workbook = load_workbook("my_workbook.xlsx")

# Delete a worksheet by name
sheet_to_delete = workbook["Sheet3"]  # Replace "Sheet3" with the sheet
         to delete
workbook.remove(sheet_to_delete)


#Delete a worksheet by index
workbook.remove(workbook.worksheets[1]) #Deletes the second sheet

workbook.save("my_workbook_modified.xlsx")
```

Remember to save the workbook after deleting a sheet.

Discover more spreadsheet write Application programming interface Macros Microsoft Excel Library Spreadsheet writing Excel APIs

Renaming Worksheets

Change a worksheet's name using the `title` attribute:

```
from openpyxl import load_workbook

workbook = load_workbook("my_workbook.xlsx")
sheet = workbook["Sheet1"]
sheet.title = "New Sheet Name"

workbook.save("my_workbook_renamed.xlsx")
```

Working with Multiple Workbooks

openpyxl allows you to work with multiple workbooks simultaneously. Just load or create multiple `Workbook` objects:

```
from openpyxl import load_workbook, Workbook

workbook1 = load_workbook("file1.xlsx")
workbook2 = Workbook()

# ... process both workbooks ...

workbook1.save("file1_modified.xlsx")
workbook2.save("file2.xlsx")
```

Workbook Properties

Discover more spreadsheet Installation Microsoft Excel Writing Macro Process spreadsheets macros Spreadsheet library

Workbooks have several properties, including:

- `active`: The currently active worksheet.
- `worksheets`: A list of all worksheets in the workbook.
- `sheetnames`: A list of the names of all worksheets.

- `properties`: Access to file properties using the `Workbook.properties` attribute, allowing setting things like author, last modified by, etc. See the openpyxl documentation for details.

Remember to always save your changes using `workbook.save("filename.xlsx")` to persist modifications to the Excel file. Handle potential `FileNotFoundError` exceptions if the specified file doesn't exist when loading a workbook. Also, it's good practice to close the workbook using `workbook.close()` after you're finished working with it to release resources, though this is automatically handled by Python's garbage collection.

Cells and Cell Values

Accessing Cell Values

Cell values are accessed using several methods. The most common is indexing using the cell's coordinates as a string (e.g., "A1") or using the `cell()` method, which takes row and column numbers:

```
from openpyxl import load_workbook

workbook = load_workbook("my_workbook.xlsx")
sheet = workbook.active

# Accessing using string index
cell_value = sheet["A1"].value
print(f"Value of A1: {cell_value}")

# Accessing using cell() method (row, column) - 1-based indexing
cell_value = sheet.cell(row=2, column=3).value # Accesses cell C2
print(f"Value of C2: {cell_value}")

#Checking for a None value (empty cell)
if sheet["B5"].value is None:
    print("Cell B5 is empty")
```

Remember that cell coordinates are 1-based (the top-left cell is A1). An empty cell will have a `value` of `None`.

Discover more Macro Application software writing Installation Excel applications Process library Macros libraries

Setting Cell Values

Setting a cell's value is equally simple:

```
from openpyxl import Workbook

workbook = Workbook()
sheet = workbook.active

sheet["A1"] = "Hello"
sheet["B1"] = 42
sheet["C1"] = 3.14159
sheet.cell(row=2, column=1).value = True # Setting a boolean value

workbook.save("output.xlsx")
```

You can assign various data types to cells (see below for more details).

Discover more library spreadsheets macros write Process Microsoft Excel Libraries Writing Python Spreadsheet

Data Types

openpyxl handles several data types:

- **Numbers:** Integers, floats, etc. are stored as numbers.
- **Strings:** Text values are stored as strings.
- **Booleans:** `True` and `False` are supported.
- **Dates and Times:** These are stored as Python `datetime` objects.
- **Formulas:** Formulas are stored as strings, but openpyxl can evaluate some simple formulas (see below).
- **None:** Represents an empty cell.

Formulas and Calculations

openpyxl can handle formulas in cells:

```
from openpyxl import Workbook

workbook = Workbook()
sheet = workbook.active

sheet["A1"] = 10
sheet["A2"] = 20
sheet["A3"] = "=SUM(A1:A2)" # Formula to sum A1 and A2

workbook.save("formula_example.xlsx")
```

While openpyxl doesn't inherently *calculate* formulas (that's Excel's job), it stores and saves them correctly, allowing Excel to perform the calculation upon opening. More complex formula evaluation might require external libraries or using Excel's calculation engine through COM (for Windows).

Number Formatting

To format numbers, use the `number_format` property of the cell:

```
from openpyxl import Workbook
from openpyxl.styles import numbers

workbook = Workbook()
sheet = workbook.active

sheet["A1"] = 1234.56

# Apply different number formats
sheet["A1"].number_format = numbers.FORMAT_PERCENTAGE #Percentage
sheet["A2"].number_format = numbers.FORMAT_CURRENCY_USD #US Dollar
sheet["A3"].number_format = '0.00' #Two decimal places
sheet["A4"].number_format = '#,##0' #Comma separated thousands

workbook.save("number_formats.xlsx")
```

Consult the openpyxl documentation for a complete list of number formats. You can use custom number formats as well.

Dates and Times

Dates and times are represented using Python's `datetime` objects:

```
from openpyxl import Workbook
from datetime import datetime
```

```
workbook = Workbook()
sheet = workbook.active

sheet["A1"] = datetime(2024, 3, 15, 10, 30, 0) # Date and Time

workbook.save("date_example.xlsx")
```

openpyxl handles the conversion between Python's `datetime` objects and Excel's date system automatically.

Working with Cell Ranges

You can efficiently work with ranges of cells using `sheet.iter_rows()` and `sheet.iter_cols()`. These methods provide iterators to traverse ranges efficiently:

```
from openpyxl import load_workbook

workbook = load_workbook("my_workbook.xlsx")
sheet = workbook.active

# Iterate through rows 1 to 5 and columns A to C
for row in sheet.iter_rows(min_row=1, max_row=5, min_col=1, max_col=3):
    for cell in row:
        print(cell.value)

#Iterate through columns 1 to 3
for col in sheet.iter_cols(min_row=1, max_row=5, min_col=1, max_col=3):
    for cell in col:
        print(cell.value)
```

These iterators are significantly more memory-efficient than retrieving all cells at once when dealing with large spreadsheets. Remember that `min_row`, `max_row`, `min_col`, and `max_col` use 1-based indexing.

Styles and Formatting

Fonts

You can control font attributes like name, size, bold, italic, color, etc., using the `Font` class from `openpyxl.styles`:

```
from openpyxl import Workbook
from openpyxl.styles import Font
```

```
workbook = Workbook()
sheet = workbook.active

font = Font(name='Arial', size=14, bold=True, italic=True,
         color="FF0000") #Red color using hex code

sheet["A1"].font = font
sheet["A1"] = "Styled Text"


workbook.save("font_styles.xlsx")
```

Remember that color can be specified using RGB hex codes (e.g., “FF0000” for red) or named colors (check the openpyxl documentation for available named colors).

Fill Colors

Cell background colors are controlled with the `PatternFill` class:

```
from openpyxl import Workbook
from openpyxl.styles import PatternFill

workbook = Workbook()
sheet = workbook.active

#Solid Fill
fill = PatternFill(start_color="FFFF00", end_color="FFFF00",
         fill_type="solid") #Yellow

sheet["B1"].fill = fill
sheet["B1"] = "Yellow Fill"

#Other fill types are available such as:
#fill = PatternFill(start_color="0000FF", end_color="FFFFFF",
         fill_type="lightUp") #Light up blue
#fill = PatternFill(start_color="00FF00", end_color="FFFFFF",
         fill_type="lightDown") #Light down green
#fill = PatternFill(start_color="FF0000", end_color="00FF00",
         fill_type="lightGrid") #Light grid red-green
#fill = PatternFill(start_color="FFFF00", end_color="0000FF",
         fill_type="darkDown") #Dark down yellow-blue
#fill = PatternFill(start_color="00FFFF", end_color="FF0000",
         fill_type="darkGrid") #Dark grid cyan-red
#fill = PatternFill(start_color="FF00FF", end_color="FFFF00",
         fill_type="darkUp") #Dark up magenta-yellow
#fill = PatternFill(start_color="000000", end_color="FFFFFF",
         fill_type="gray125") #Gray 125

sheet["C1"].fill = PatternFill(start_color="0000FF", end_color="FFFFFF",
         fill_type="lightUp")
```

```
sheet["C1"] = "LightUp Fill"

workbook.save("fill_styles.xlsx")
```

`fill_type` can be “solid”, “darkDown”, “darkGrid”, “darkUp”, “gray0625”, “gray125”, “lightDown”, “lightGrid”, “lightUp”, “lightGray”, “mediumGray”, or “patterned”. Start and end colors are specified using hex codes.

Borders

Borders are defined using the `Border`, `Side` classes:

```
from openpyxl import Workbook
from openpyxl.styles import Border, Side

workbook = Workbook()
sheet = workbook.active

thin_border = Border(left=Side(style='thin'),
                     right=Side(style='thin'),
                     top=Side(style='thin'),
                     bottom=Side(style='thin'))

sheet["D1"].border = thin_border
sheet["D1"] = "Thin Border"

#Other border styles: dashDot, dashDotDot, dashed, dotted, double, hair,
         medium, mediumDashDot, mediumDashDotDot, mediumDashed,
         slantDashDot

thick_border = Border(left=Side(style='thick', color="FF0000"),
         right=Side(style='thick'), top=Side(style='thick'),
         bottom=Side(style='thick'))
sheet["E1"].border = thick_border
sheet["E1"] = "Thick Border"

workbook.save("border_styles.xlsx")
```

`Side` styles include ‘thin’, ‘medium’, ‘thick’, ‘double’, ‘dotted’, ‘dashed’, ‘dashDot’, ‘dashDotDot’, ‘slantDashDot’, and ‘hair’. You can specify border colors as well.

Alignment

Cell content alignment (horizontal, vertical, text wrapping, etc.) is controlled with the `Alignment` class:

```
from openpyxl import Workbook
from openpyxl.styles import Alignment

workbook = Workbook()
sheet = workbook.active

alignment = Alignment(horizontal='center', vertical='center',
         wrap_text=True)

sheet["F1"].alignment = alignment
sheet["F1"] = "Center Aligned Text"

workbook.save("alignment_styles.xlsx")
```

`horizontal` can be 'general', 'left', 'center', 'right', 'fill', 'justify', 'centerContinuous', 'distributed'. `vertical` can be 'top', 'center', 'bottom', 'justify', 'distributed'.

Number Formats

Number formats were covered in the previous section ("Cells and Cell Values"). Refer to that section for details on formatting numbers, dates, and currency.

Conditional Formatting

Conditional formatting involves applying styles based on cell values or formulas. openpyxl provides support for this, but it's more complex; refer to the openpyxl documentation for detailed examples and API usage. It involves using the `DifferentialStyle` and `Rule` classes and working with the `conditional_formatting` feature of the worksheet.

Styles and Themes

openpyxl allows working with styles and themes, but the specifics are advanced and are best explored in the library's documentation. Themes govern the overall look and feel, while styles provide more fine-grained control over individual elements.

Applying Styles to Cells and Ranges

Styles are applied to cells using the appropriate style properties (e.g., `cell.font`, `cell.fill`, `cell.alignment`, `cell.number_format`, `cell.border`). To apply styles to ranges, iterate through the cells in the range and apply the styles to each cell individually, or explore using `Conditional`

`Formatting` which can apply styles to ranges based on conditions. Creating and applying a custom `Style` object can also be helpful for consistently applying multiple formatting elements.

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

workbook = Workbook()
sheet = workbook.active

# Define styles once
bold_red_font = Font(bold=True, color="FF0000")
yellow_fill = PatternFill(start_color="FFFF00", end_color="FFFF00",
          fill_type="solid")
thin_border = Border(left=Side(style='thin'), right=Side(style='thin'),
          top=Side(style='thin'), bottom=Side(style='thin'))
center_alignment = Alignment(horizontal="center")

#Apply styles to a range
for row in sheet.iter_rows(min_row=1, max_row=3, min_col=1, max_col=3):
    for cell in row:
        cell.font = bold_red_font
        cell.fill = yellow_fill
        cell.border = thin_border
        cell.alignment = center_alignment

workbook.save("range_styles.xlsx")
```

Charts and Graphs

Creating Charts

Creating charts in openpyxl involves several steps: first, define the chart data, then create a chart object specifying the chart type, and finally add the chart to a worksheet. The process uses classes from the `openpyxl.chart` module.

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

workbook = Workbook()
sheet = workbook.active

# Sample data
data = [
    ['Category', 'Value'],
    ['A', 10],
```

```
    ['B', 15],
    ['C', 20],
]

for row in data:
    sheet.append(row)

# Create chart data reference
chart_data = Reference(sheet, min_col=2, min_row=1, max_col=2,
         max_row=4)

# Create bar chart
chart = BarChart()
chart.add_data(chart_data)

# Customize chart (optional - see below)
# ...

# Add chart to worksheet
sheet.add_chart(chart, "E1") # Add chart at cell E1

workbook.save("chart_example.xlsx")
```

This creates a simple bar chart from the data. Remember that the `Reference` object specifies the cell range containing your chart data. The `add_chart()` method adds the chart to the worksheet at a given cell location.

Chart Types

openpyxl supports various chart types, including:

- `BarChart` (bar chart)
- `LineChart` (line chart)
- `PieChart` (pie chart)
- `ScatterChart` (scatter chart)
- `AreaChart` (area chart)
- `RadarChart` (radar chart)
- `DoughnutChart` (doughnut chart)
- `StockChart` (stock chart)

To create a different chart type, simply replace `BarChart` with the desired chart class in the code above. Each chart type has specific configuration options; refer to the openpyxl documentation.

Customizing Charts

Charts can be customized extensively:

```
from openpyxl import Workbook
from openpyxl.chart import BarChart, Reference, Series
from openpyxl.chart.axis import CategoryAxis, ValueAxis

workbook = Workbook()
sheet = workbook.active

# ... (data and chart creation as above) ...

# Customize chart title
chart.title = "Sales Data"

# Customize axes
chart.x_axis = CategoryAxis(title='Categories')
chart.y_axis = ValueAxis(title='Sales')

# Add series labels (for multiple data series)
chart.series[0].name = 'Sales Figures'

# Add a legend
chart.legend = None # Remove legend (default is shown)

# Adjust chart layout
chart.width = 20 #Adjust Width
chart.height = 10 #Adjust Height


# ... (add chart to worksheet) ...

workbook.save("chart_custom.xlsx")
```

This adds a title, labels to the axes, and removes the chart legend. Numerous other customization options exist, including data label formatting, styling the chart elements, changing colors, adding gridlines, and much more (consult the openpyxl documentation for a complete overview). Consider the use of `Series` objects to add additional series to a chart.

Adding Charts to Worksheets

Charts are added using the `sheet.add_chart()` method. The first argument is the chart object, and the second is the top-left cell location where the chart will be placed on the worksheet. For example:

```
sheet.add_chart(chart, "A10")  # Places the chart starting at cell A10
```

The chart will automatically adjust its size to fit the data it represents, but you can manually adjust chart size using its `width` and `height` attributes before adding it to the worksheet. Avoid placing charts over data that you intend to use; they will overlap.

Images and Drawings

Adding Images to Worksheets

Adding images to worksheets involves using the `openpyxl.drawing.image` module. You'll need the path to your image file.

```
from openpyxl import Workbook
from openpyxl.drawing.image import Image

workbook = Workbook()
sheet = workbook.active

img = Image("my_image.png") # Replace "my_image.png" with your image
          path

sheet.add_image(img, "A1") # Add image starting at cell A1

workbook.save("image_example.xlsx")
```

This adds the image to the worksheet, anchored at cell A1. The image will be scaled to fit within the cell's bounds. Ensure that the image file exists in a location accessible to your Python script.

Working with Drawing Objects

Images and other drawing objects in openpyxl are represented as `Drawing` objects. You can get a reference to these objects to manipulate their properties after adding them to the worksheet. For complex scenarios involving multiple images or other drawing objects in precise locations and with fine-grained control over their positioning and overlay, you will need to work directly with the `Drawing` object and its contained elements. This requires a deeper understanding of how openpyxl handles drawings which is best explored through detailed examples in the official documentation.

```
from openpyxl import Workbook
from openpyxl.drawing.image import Image

workbook = Workbook()
sheet = workbook.active
```

```
img = Image("my_image.png")
drawing = sheet.add_image(img, "A1") # drawing now holds reference to
          the added image

# Accessing and modifying drawing properties (see below for details)
# ...

workbook.save("drawing_example.xlsx")
```

The specific properties and methods of the `drawing` object for detailed manipulation are best consulted in the openpyxl documentation.

Resizing and Positioning Images

While `add_image` initially places and sizes the image based on the cell, you can manually adjust the size and position using the `width`, `height`, `anchor`, and `left`, `top` attributes of the underlying `Drawing` object. Note: Directly changing cell anchors might lead to unexpected layout issues. For precise control, work directly with the pixel coordinates (`left`, `top`) and dimensions (`width`, `height`) of the `Drawing` object.

```
from openpyxl import Workbook
from openpyxl.drawing.image import Image
from openpyxl.utils.units import pixels_to_EMU

workbook = Workbook()
sheet = workbook.active

img = Image("my_image.png")
drawing = sheet.add_image(img, "A1")

# Resize the image (in EMUs - English Metric Units)
drawing.width = pixels_to_EMU(200)  # Width in pixels, converted to EMUs
drawing.height = pixels_to_EMU(150) # Height in pixels, converted to
          EMUs

# Reposition the image (in EMUs) - Top-left corner
drawing.left = pixels_to_EMU(100)  # Left offset in pixels, converted to
          EMUs
drawing.top = pixels_to_EMU(50)  # Top offset in pixels, converted to
          EMUs


workbook.save("resized_image.xlsx")
```

The `pixels_to_EMU` function from `openpyxl.utils.units` is crucial for correct conversion from pixel units to EMUs, which are the units used by open-pyxl for drawing objects.

Image Formats

openpyxl supports various image formats, including PNG, JPEG, GIF, and others. The specific formats supported depend on the underlying libraries used by openpyxl (usually Pillow). If you encounter issues with a specific format, check your Pillow installation and ensure it supports that format. PNG is generally recommended for lossless image quality.

Working with Tables

Creating Tables

openpyxl provides functionality to work with Excel tables (also known as ListObjects). Creating a table involves specifying a range of cells and optionally providing a table name.

```
from openpyxl import Workbook
from openpyxl.worksheet.table import Table, TableStyleInfo

workbook = Workbook()
sheet = workbook.active

# Sample data
data = [
    ['Name', 'Age', 'City'],
    ['Alice', 30, 'New York'],
    ['Bob', 25, 'London'],
    ['Charlie', 35, 'Paris'],
]

for row in data:
    sheet.append(row)

# Create a table from cells A1 to C4
table = Table(displayName="MyTable", ref="A1:C4")

# Define a table style (optional)
style = TableStyleInfo(name="TableStyleMedium9", showFirstColumn=False,
                       showLastColumn=False, showRowStripes=True,
         showColumnStripes=False)
table.tableStyleInfo = style

# Add the table to the worksheet
sheet.add_table(table)

workbook.save("table_example.xlsx")
```

This creates a table named “MyTable” encompassing the data from A1 to C4, applying a predefined style. You can find a list of available table styles in the openpyxl documentation.

Formatting Tables

Table formatting involves applying styles to the table’s appearance, such as header row styles, banded rows, total row, etc. This is done by modifying the `tableStyleInfo` attribute of the `Table` object. You can choose from predefined styles or create custom styles.

```
from openpyxl import Workbook
from openpyxl.worksheet.table import Table, TableStyleInfo

# ... (table creation as above) ...

#Custom Style Example
custom_style = TableStyleInfo(name="CustomTableStyle",
         showFirstColumn=True,
                               showLastColumn=False, showRowStripes=True,
         showColumnStripes=True,
                               pivotButton=True) # Enables Pivot Button.
custom_style.font = "Arial,12"
custom_style.fill = {"type": "solid", "fgColor": "FFFFCC"}

table.tableStyleInfo = custom_style

# ... (add table and save workbook) ...
```

Refer to the openpyxl documentation for details on customizing table styles. Note that not all style elements are fully customizable, and some properties might not be reflected in all Excel versions.

Working with Table Data

Accessing and modifying data within a table is similar to working with regular cells. You can directly access cells using their coordinates or iterate through rows and columns.

```
from openpyxl import load_workbook

workbook = load_workbook("table_example.xlsx")
sheet = workbook.active
table = sheet["MyTable"]

# Accessing a specific cell
cell_value = table["A2"].value
print(f"Value of A2: {cell_value}")
```

```
# Iterating through rows
for row in table.rows:
    for cell in row:
        print(cell.value)

# Adding a new row to the table
table.add_row(['David', 40, 'Tokyo'])

workbook.save("updated_table.xlsx")
```

Adding rows usually appends to the end of the table. Deleting rows or columns requires careful handling of the table's structure and might need updating the table's reference range using `table.ref`.

Filtering and Sorting

openpyxl's support for directly manipulating Excel filters and sorts is limited. You can set filter criteria through manual manipulation of the XML structure, which is highly advanced and not recommended unless you're dealing with low-level XML manipulation of the `.xlsx` file. If you need robust filtering and sorting capabilities, it's generally better to leverage Excel's built-in functionality or use other libraries designed specifically for data manipulation and analysis. These external tools (e.g., Pandas) may be better suited for filtering and sorting before writing data back to the spreadsheet using openpyxl.

Advanced Features

Working with Pivot Tables

Creating and modifying pivot tables directly within openpyxl is currently not fully supported. The structure of pivot tables in the `.xlsx` file is complex, requiring detailed manipulation of the underlying XML. While you can technically access and potentially modify some aspects of existing pivot tables through low-level XML manipulation, this approach is highly discouraged due to its complexity, fragility, and lack of robustness. The official openpyxl documentation explicitly cautions against this. For creating and managing pivot tables, it's generally far more practical and reliable to use Excel's built-in pivot table functionality or to process the data externally using libraries like Pandas before writing the results back to the spreadsheet using openpyxl.

Data Validation

Data validation rules in Excel can be added and modified using openpyxl, but it's an advanced feature that involves manipulating the XML structure representing the validation rules. Direct programmatic creation and modification of data validation rules is not explicitly straightforward. The process often involves creating `DataValidation` objects with properties that specify the validation type (e.g., whole number, list, date), criteria, input message, and error message. These objects are then added to the worksheet using methods associated with the `worksheet` object. Consult the openpyxl documentation for detailed examples and API information on the `DataValidation` class and its various properties and options, as the process is complex and requires careful attention to the XML structure of the validation rules.

Macros and VBA (if applicable)

openpyxl's core functionality focuses on spreadsheet data and structure; it doesn't directly support creating or manipulating VBA macros. You cannot create or edit VBA macros using openpyxl. If your workflow involves macros, you'll likely need to use external tools or libraries specialized in handling VBA code within Excel files.

External Links

openpyxl can handle external links within spreadsheets. When loading a workbook, openpyxl will detect and retain information about external links, even if it doesn't actively resolve them. Modifying or adding external links usually involves low-level XML manipulation of the workbook's relationships, which is a complex task best left to advanced users well-versed in the XML structure of `.xlsx` files. The openpyxl documentation provides limited guidance on this, typically suggesting the use of external tools or libraries for tasks heavily involving linked workbooks.

Protecting Worksheets and Workbooks

openpyxl supports protecting worksheets and workbooks to restrict access and modifications. This is done by setting appropriate properties on the worksheet or workbook object.

```
from openpyxl import Workbook
from openpyxl.worksheet.protection import WorksheetProtection
from openpyxl.workbook.protection import WorkbookProtection

workbook = Workbook()
sheet = workbook.active
```

```
# Worksheet protection
sheet_protection = WorksheetProtection()
sheet_protection.sheet = True  # Protect the entire sheet
sheet_protection.objects = True  # Prevent modification of objects
sheet.protection = sheet_protection

# Workbook protection (password protected)
workbook_protection = WorkbookProtection()
workbook_protection.set_password('mypassword') #Set a password (Note:
         security limitations exist)
workbook.protection = workbook_protection

workbook.save("protected_workbook.xlsx")
```

Remember that the password protection provided by openpyxl is relatively basic and may not be sufficient for high-security applications. Advanced protection mechanisms are best handled by Excel's built-in security features or through external encryption methods. The password used here is not cryptographically secure.

Error Handling and Best Practices

Common Errors and Troubleshooting

Several common errors arise when using openpyxl:

- **`FileNotFoundError`:** Occurs when trying to load a workbook that doesn't exist. Always check file existence before loading.
- **`KeyError`:** Happens when accessing a worksheet or cell that doesn't exist. Verify worksheet names and cell coordinates.
- **`ValueError`:** Can be caused by incorrect data types, invalid cell references, or other data inconsistencies. Carefully check your input data and indexing.
- **`TypeError`:** Arises from using incorrect data types or calling methods inappropriately. Ensure data types match expected types and review method signatures.

Troubleshooting involves carefully examining error messages, checking input data, and using debugging tools to step through your code and identify the source of the problem. The openpyxl documentation and online forums often contain solutions to common issues.

Performance Optimization

For optimal performance:

- **Iterators:** Use iterators (`iter_rows`, `iter_cols`) to process large worksheets efficiently instead of loading the entire worksheet into memory at once.
- **Data Chunking:** If dealing with extremely large files, consider processing data in smaller chunks to reduce memory consumption.
- **Avoid Unnecessary Operations:** Minimize repeated accesses to cells or unnecessary calculations.
- **Profiling:** Use profiling tools to identify performance bottlenecks in your code.

Memory Management

Memory management is crucial when working with large Excel files. Openpyxl uses generators and iterators where possible to minimize memory usage, but for very large files, additional strategies are necessary:

- **Iterators (again!):** Emphasize the use of iterators to process data incrementally instead of loading everything at once.
- **Garbage Collection:** Python's garbage collection will automatically reclaim memory, but you can occasionally force garbage collection using `gc.collect()` (though this is usually unnecessary).
- **Context Managers:** Use context managers (`with openpyxl.load_workbook(...) as wb: ...`) to ensure files are properly closed and resources released.
- **Chunking (again!):** Process data in logical chunks, saving intermediate results if needed to reduce memory demands for any single operation.

Large Files and Efficient Processing

For very large Excel files (>100MB), efficient processing is vital. Employ these techniques:

- **Memory Mapping:** Consider using memory-mapped files to reduce memory overhead (though this adds complexity).
- **Incremental Processing:** Process data in stages, saving progress to avoid restarting from the beginning if an error occurs or if processing needs to be interrupted.
- **Parallel Processing:** If appropriate, explore using multiprocessing to distribute the workload across multiple CPU cores.

- **External Libraries:** For extensive data manipulation, consider using optimized libraries like Pandas which are designed for handling large datasets efficiently. Use openpyxl to read/write the data to the Excel file, using Pandas for in-memory processing.

Security Considerations

- **Input Validation:** Always validate user inputs to prevent malicious code injection or data corruption.
- **Password Protection (Limitations):** Be aware that the password protection provided by openpyxl is relatively weak; it's not suitable for highly sensitive data.
- **External Links:** Exercise caution when working with workbooks containing external links to prevent unintended access to external resources.
- **File Permissions:** Control file permissions to restrict access to your Excel files.
- **Sanitize Data:** Sanitize any data read from an Excel file before using it in other parts of your application to avoid unexpected issues or security risks. This includes checking for unexpected data types or values.
- **Do not use openpyxl to handle untrusted files**: Always verify the integrity and source of any Excel files you work with before opening them or processing their contents. Untrusted files can easily introduce security vulnerabilities into your application.

Appendix: Reference Tables and APIs

This appendix provides a concise overview of openpyxl's key components. For detailed and comprehensive information, always refer to the official openpyxl documentation. This section offers a starting point and highlights key areas.

List of Functions and Methods

openpyxl offers a wide range of functions and methods. A complete list is beyond the scope of this manual, but key examples include:

- `load_workbook(filename)`: Loads an existing Excel workbook.
- `Workbook()`: Creates a new workbook.
- `workbook.active`: Returns the active worksheet.
- `workbook.create_sheet(title, index=None, before=None)`: Creates a new worksheet.
- `workbook.save(filename)`: Saves the workbook to a file.
- `worksheet['A1']`: Accesses a cell by its coordinates.

- `worksheet.cell(row, column)`: Accesses a cell using row and column numbers.
- `worksheet.iter_rows()`: Iterates through rows in a range.
- `worksheet.iter_cols()`: Iterates through columns in a range.
- `cell.value`: Gets or sets the cell's value.
- `cell.font`: Gets or sets the cell's font style.
- `cell.fill`: Gets or sets the cell's fill color.
- `cell.alignment`: Gets or sets the cell's alignment.
- `cell.number_format`: Gets or sets the cell's number format.
- `chart.add_data()`: Adds data to a chart.
- `sheet.add_chart()`: Adds a chart to a worksheet.
- `Image(filename)`: Creates an image object.
- `sheet.add_image()`: Adds an image to a worksheet.
- `Table(displayName, ref)`: Creates an Excel table.
- `sheet.add_table()`: Adds a table to a worksheet.

This is not exhaustive; explore the official API documentation for the complete list.

Data Structures

Key data structures in openpyxl include:

- `Workbook`: Represents the entire Excel file.
- `Worksheet`: Represents a single sheet within a workbook.
- `Cell`: Represents an individual cell in a worksheet.
- `Reference`: Defines a range of cells.
- `Table`: Represents an Excel table (ListObject).
- `Chart`: Base class for various chart types. Specific chart types (like `BarChart`, `LineChart`) inherit from this.
- `Font`, `PatternFill`, `Alignment`, `Border`, `Side`: Style classes for formatting cells.
- `Image`: Represents an image to be added to a worksheet.
- `Drawing`: A container for drawings and images within the worksheet.

Understanding these core structures is fundamental to effectively using openpyxl.

Constants and Enumerations

openpyxl uses constants and enumerations to represent various options and settings (e.g., alignment types, fill types, border styles, chart types). Consult the official documentation for a comprehensive list. Examples include constants representing different alignment types within the `Alignment` class or constants specifying different fill types in `PatternFill`.

API Reference

The most complete API reference is always found in the official openpyxl documentation. This documentation provides detailed descriptions of all classes, methods, functions, and attributes within the library. It's the definitive source for understanding the capabilities and usage of openpyxl. The API reference typically includes examples, parameters, return values, and explanations of each component. This allows developers to quickly look up the specifics of functions and classes to help them write efficient and correct code.