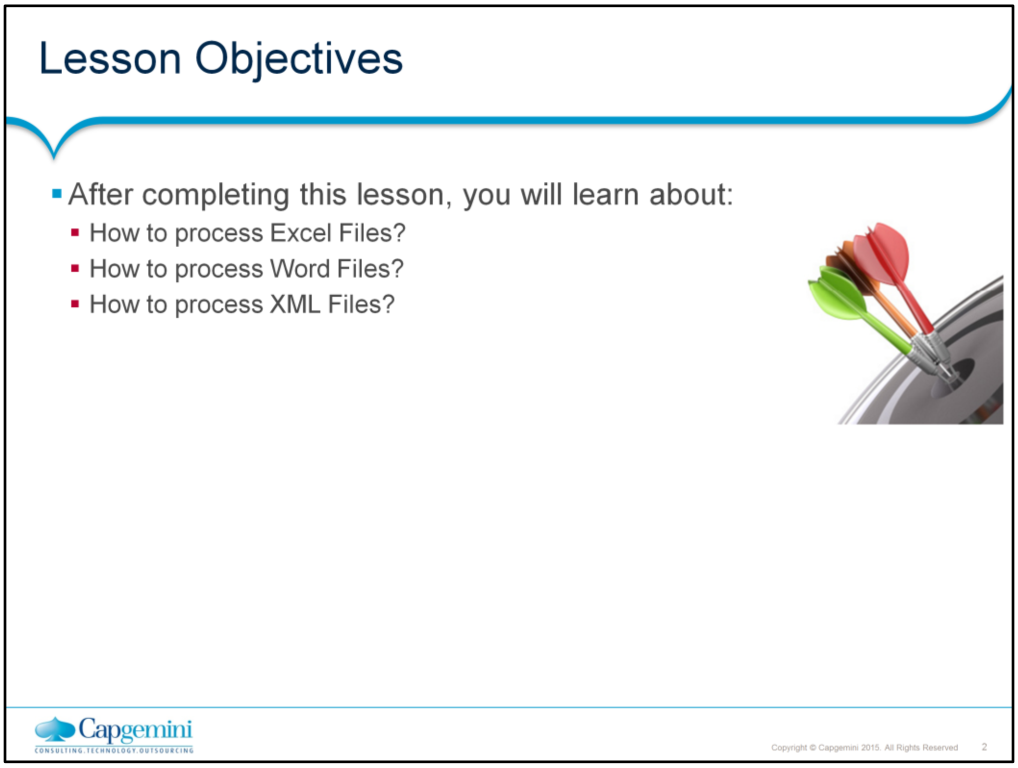
# Python Training

Lesson 06: Working with Files

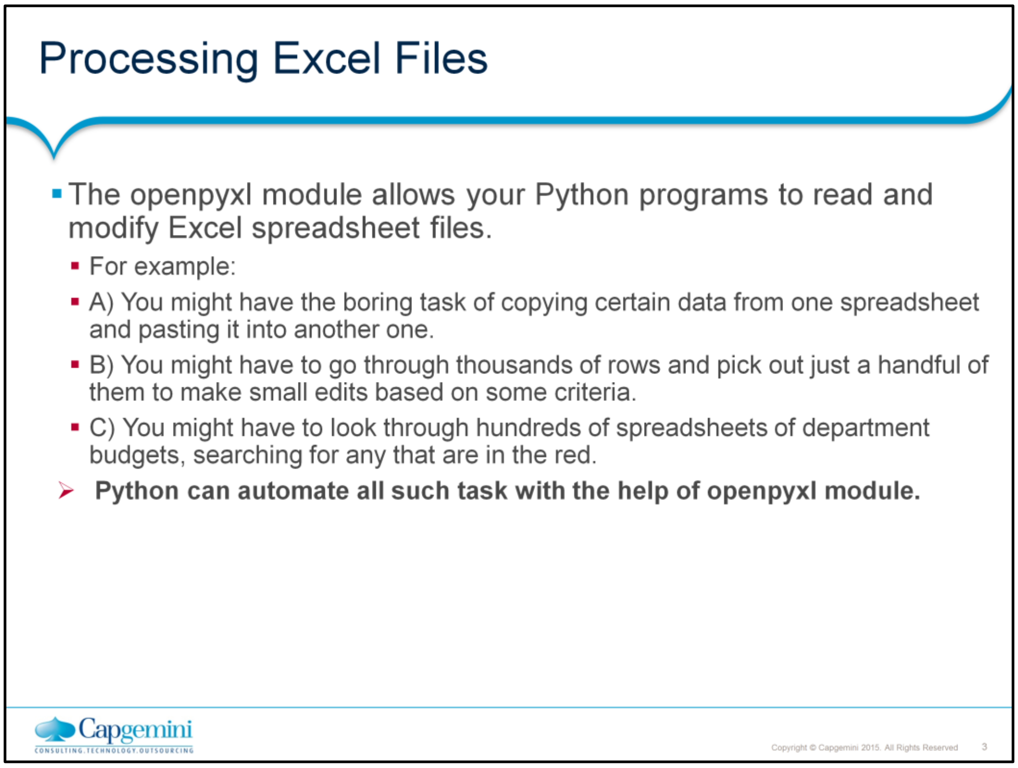
People matter, results count.

1



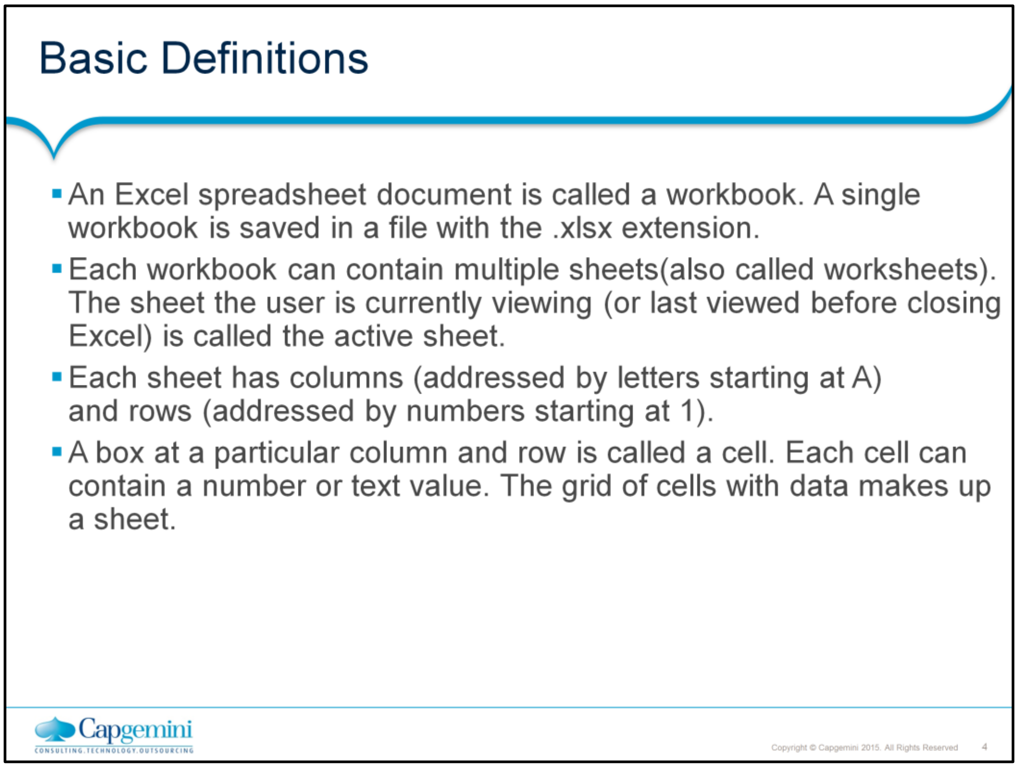
## Lesson Objectives

- After completing this lesson, you will learn about:
  - How to process Excel Files?
  - How to process Word Files?
  - How to process XML Files?



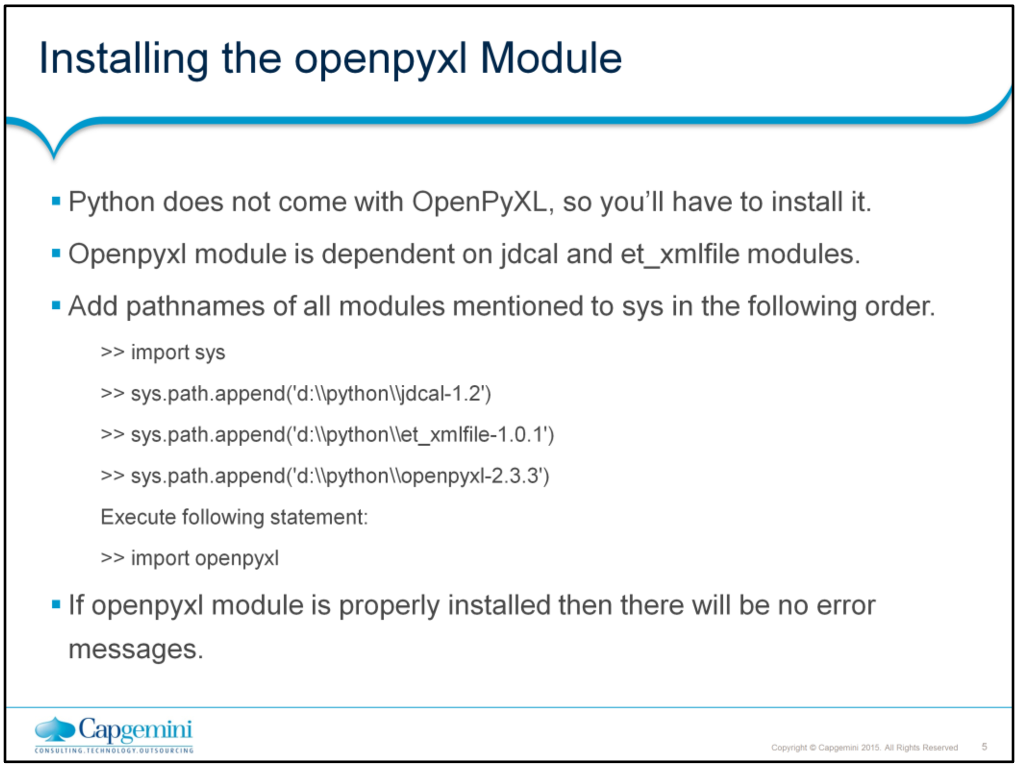
## Processing Excel Files

- The openpyxl module allows your Python programs to read and modify Excel spreadsheet files.
  - For example:
  - A) You might have the boring task of copying certain data from one spreadsheet and pasting it into another one.
  - B) You might have to go through thousands of rows and pick out just a handful of them to make small edits based on some criteria.
  - C) You might have to look through hundreds of spreadsheets of department budgets, searching for any that are in the red.
  - ➢ **Python can automate all such task with the help of openpyxl module.**



## Basic Definitions

- An Excel spreadsheet document is called a workbook. A single workbook is saved in a file with the .xlsx extension.
- Each workbook can contain multiple sheets(also called worksheets). The sheet the user is currently viewing (or last viewed before closing Excel) is called the active sheet.
- Each sheet has columns (addressed by letters starting at A) and rows (addressed by numbers starting at 1).
- A box at a particular column and row is called a cell. Each cell can contain a number or text value. The grid of cells with data makes up a sheet.



## Installing the openpyxl Module

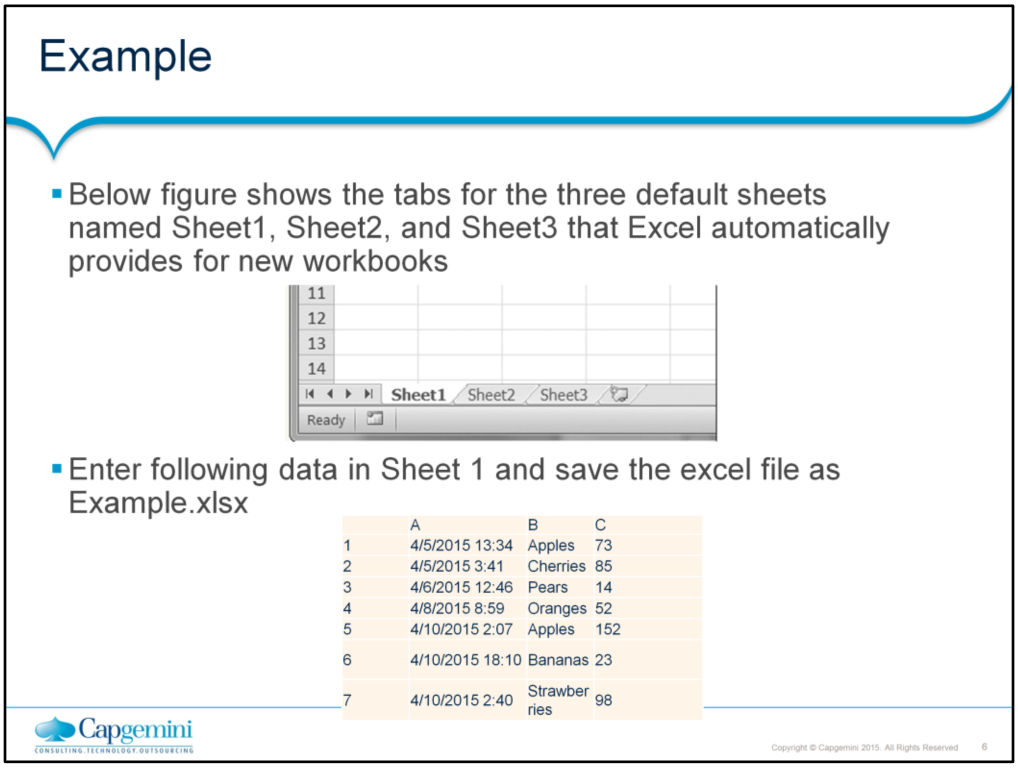
- Python does not come with OpenPyXL, so you'll have to install it.
- Openpyxl module is dependent on jdcal and et_xmlfile modules.
- Add pathnames of all modules mentioned to sys in the following order.

```
>> import sys
>> sys.path.append('d:\\python\\jdcal-1.2')
>> sys.path.append('d:\\python\\et_xmlfile-1.0.1')
>> sys.path.append('d:\\python\\openpyxl-2.3.3')
```

Execute following statement:

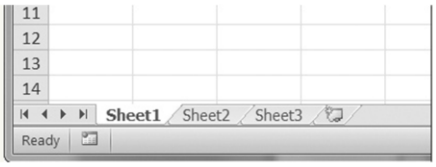
```
>> import openpyxl
```

- If openpyxl module is properly installed then there will be no error messages.



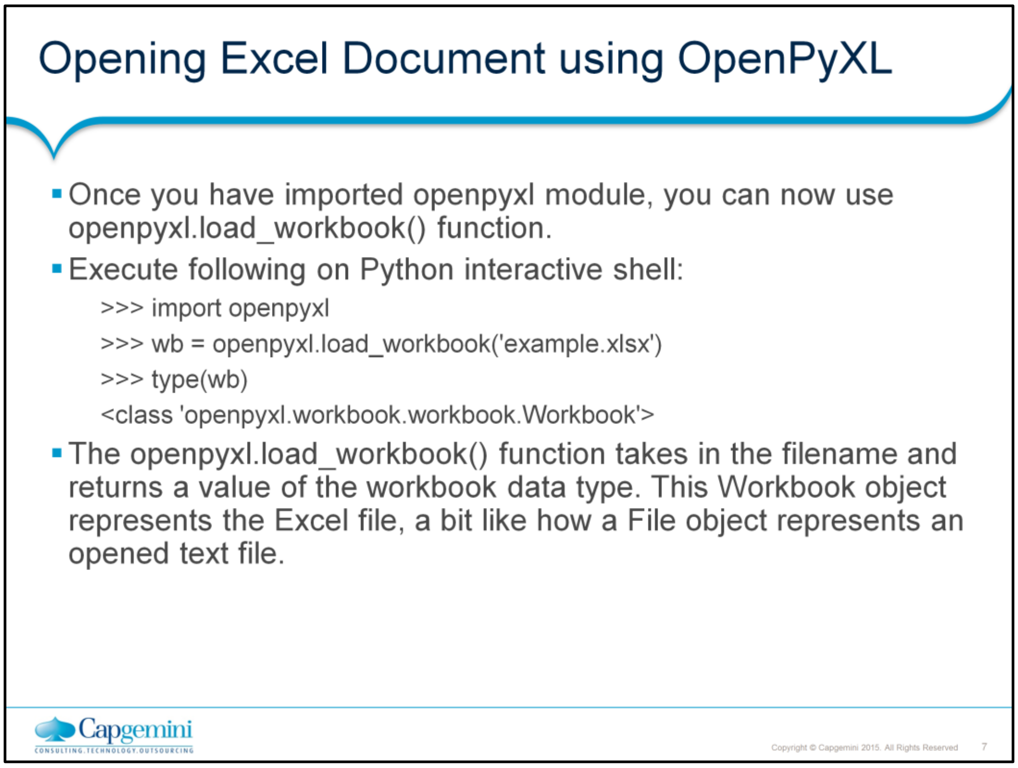
## Example

- Below figure shows the tabs for the three default sheets named Sheet1, Sheet2, and Sheet3 that Excel automatically provides for new workbooks



- Enter following data in Sheet 1 and save the excel file as Example.xlsx

| | A | B | C |
|---|---|---|---|
| 1 | 4/5/2015 13:34 | Apples | 73 |
| 2 | 4/5/2015 3:41 | Cherries | 85 |
| 3 | 4/6/2015 12:46 | Pears | 14 |
| 4 | 4/8/2015 8:59 | Oranges | 52 |
| 5 | 4/10/2015 2:07 | Apples | 152 |
| 6 | 4/10/2015 18:10 | Bananas | 23 |
| 7 | 4/10/2015 2:40 | Strawberries | 98 |



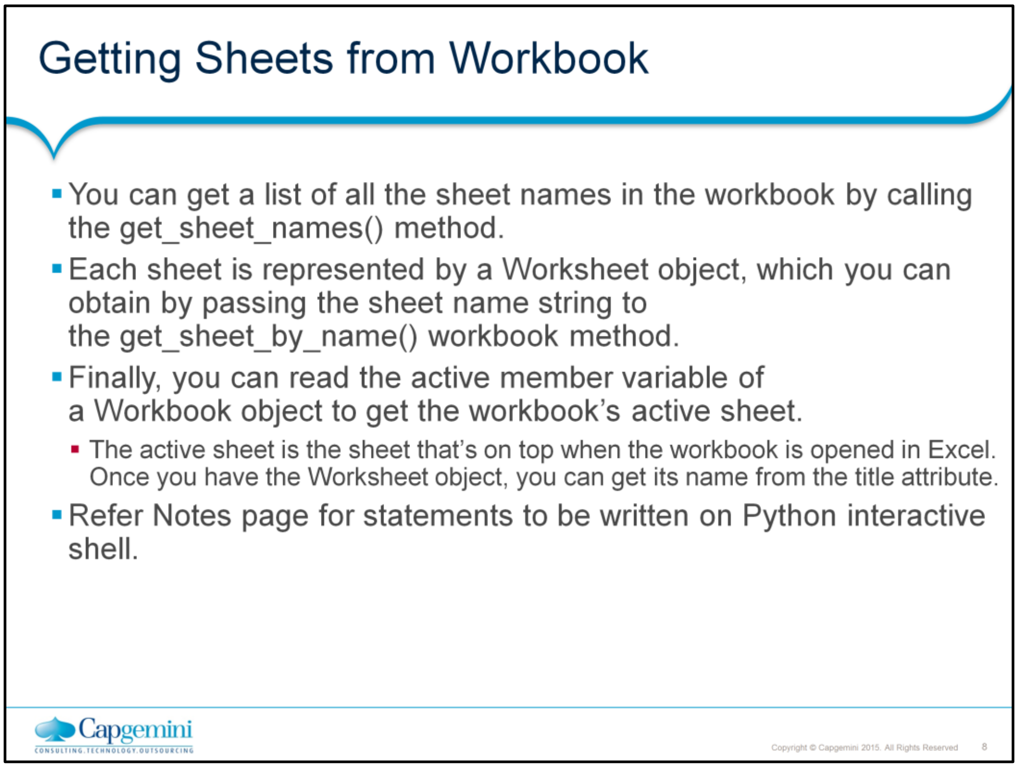
## Opening Excel Document using OpenPyXL

- Once you have imported openpyxl module, you can now use openpyxl.load_workbook() function.
- Execute following on Python interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> type(wb)
<class 'openpyxl.workbook.workbook.Workbook'>
```

- The openpyxl.load_workbook() function takes in the filename and returns a value of the workbook data type. This Workbook object represents the Excel file, a bit like how a File object represents an opened text file.

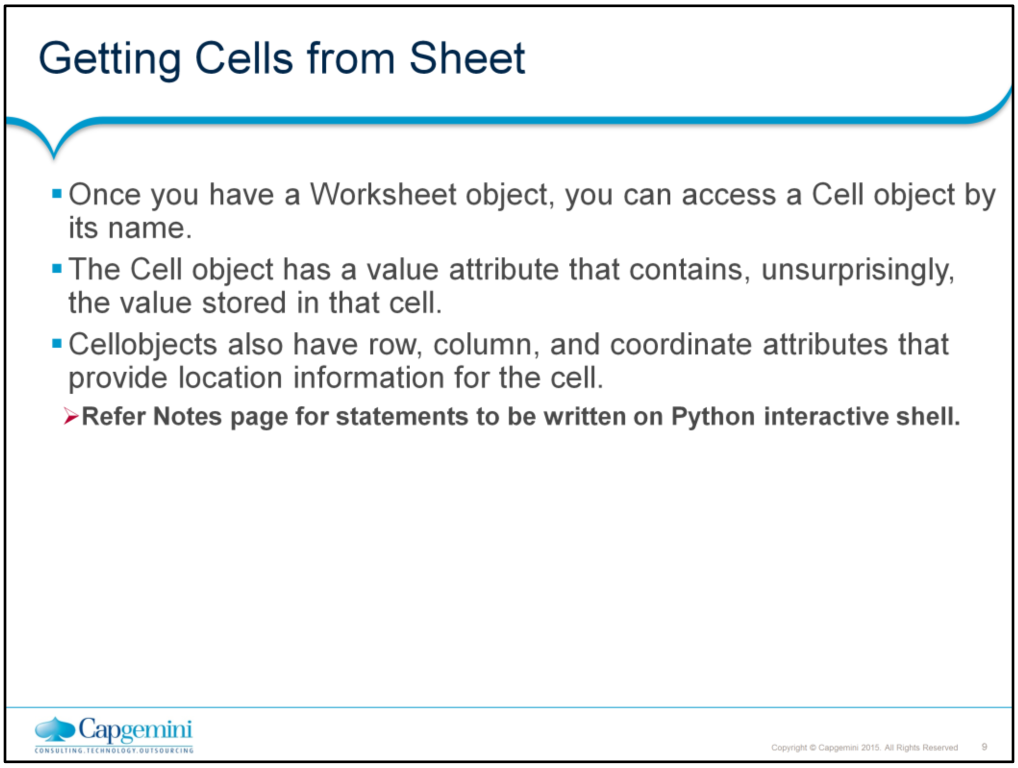
Remember that *example.xlsx* needs to be in the current working directory in order for you to work with it. You can find out what the current working directory is by importing os and using os.getcwd(), and you can change the current working directory using os.chdir().



## Getting Sheets from Workbook

- You can get a list of all the sheet names in the workbook by calling the get_sheet_names() method.
- Each sheet is represented by a Worksheet object, which you can obtain by passing the sheet name string to the get_sheet_by_name() workbook method.
- Finally, you can read the active member variable of a Workbook object to get the workbook's active sheet.
  - The active sheet is the sheet that's on top when the workbook is opened in Excel. Once you have the Worksheet object, you can get its name from the title attribute.
- Refer Notes page for statements to be written on Python interactive shell.

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> wb.get_sheet_names()
['Sheet1', 'Sheet2', 'Sheet3']
>>> sheet = wb.get_sheet_by_name('Sheet3')
>>> sheet <Worksheet "Sheet3">
>>> type(sheet)
<class 'openpyxl.worksheet.worksheet.Worksheet'>
>>> sheet.title
 'Sheet3'
>>> anotherSheet = wb.active
>>> anotherSheet
 <Worksheet "Sheet1">
```

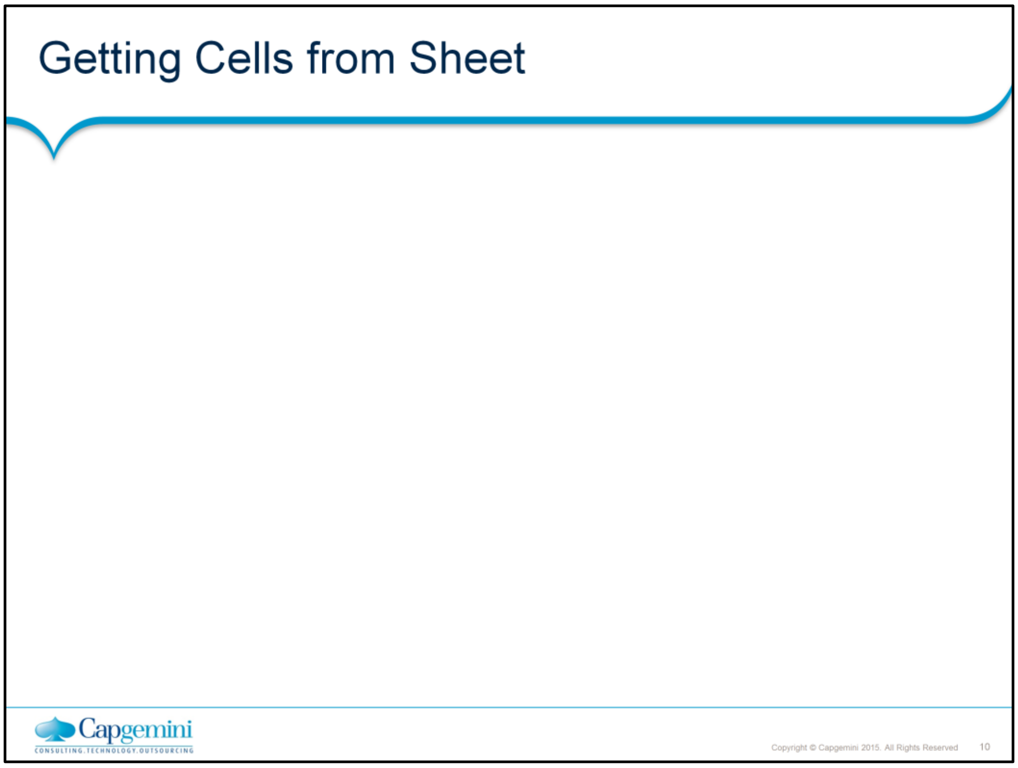


## Getting Cells from Sheet

- Once you have a Worksheet object, you can access a Cell object by its name.
- The Cell object has a value attribute that contains, unsurprisingly, the value stored in that cell.
- Cellobjects also have row, column, and coordinate attributes that provide location information for the cell.
  - **Refer Notes page for statements to be written on Python interactive shell.**

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.get_sheet_by_name('Sheet1')
>>> sheet['A1']
<Cell Sheet1.A1>
>>> sheet['A1'].value
datetime.datetime(2015, 4, 5, 13, 34, 2)
>>> c = sheet['B1']
>>> c.value
'Apples'
>>> 'Row ' + str(c.row) + ', Column ' + c.column + ' is ' + c.value
'Row 1, Column B is Apples'
>>> 'Cell ' + c.coordinate + ' is ' + c.value
'Cell B1 is Apples'
>>> sheet['C1'].value
73
```

Here, accessing the value attribute of our Cell object for cell B1 gives us the string 'Apples'. The row attribute gives us the integer 1, the column attribute gives us 'B', and the coordinate attribute gives us 'B1'.
OpenPyXL will automatically interpret the dates in column A and return them as datetime values rather than strings.



## Getting Cells from Sheet

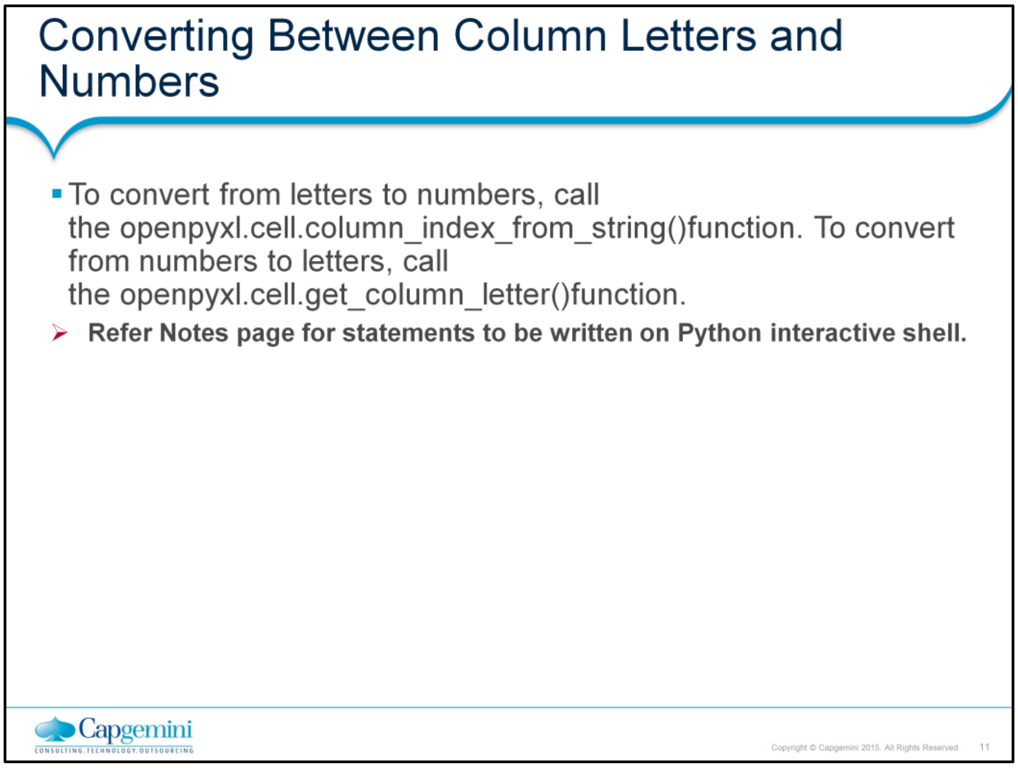
Specifying a column by letter can be tricky to program, especially because after column Z, the columns start by using two letters: AA, AB, AC, and so on. As an alternative, you can also get a cell using the sheet's cell() method and passing integers for its row and column keyword arguments. The first row or column integer is 1, not 0. Continue the interactive shell example by entering the following:

```
>>> sheet.cell(row=1, column=2)
<Cell Sheet1.B1>
>>> sheet.cell(row=1, column=2).value
'Apples'
>>> for i in range(1, 8, 2):
        print(i, sheet.cell(row=i, column=2).value)

1 Apples
3 Pears
5 Apples
7 Strawberries
```

Continue adding more statements on the shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.get_sheet_by_name('Sheet1')
>>> sheet.max_row
7
>>> sheet.max_column
3
```

Note that the max_column method returns an integer rather than the letter that appears in Excel.



## Converting Between Column Letters and Numbers

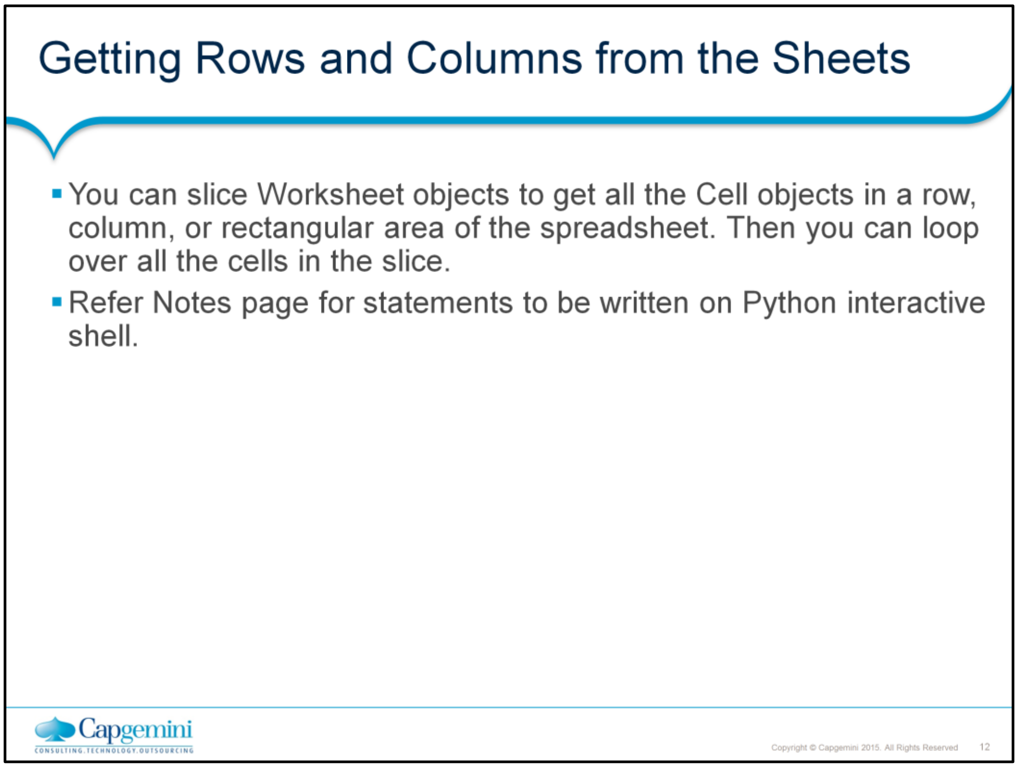
- To convert from letters to numbers, call the openpyxl.cell.column_index_from_string()function. To convert from numbers to letters, call the openpyxl.cell.get_column_letter()function.
  - **Refer Notes page for statements to be written on Python interactive shell.**

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

```
>>> import openpyxl
>>> from openpyxl.cell import get_column_letter, column_index_from_string
>>> get_column_letter(1)
'A'
>>> get_column_letter(2)
'B'
>>> get_column_letter(27)
'AA'
>>> get_column_letter(900)
'AHP'
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.get_sheet_by_name('Sheet1')
>>> get_column_letter(sheet.max_column)
'C'
>>> column_index_from_string('A')
1
>>> column_index_from_string('AA')
27
```

After you import these two functions from the openpyxl.cell module, you can callget_column_letter() and pass it an integer like 27 to figure out what the letter name of the 27th column is.

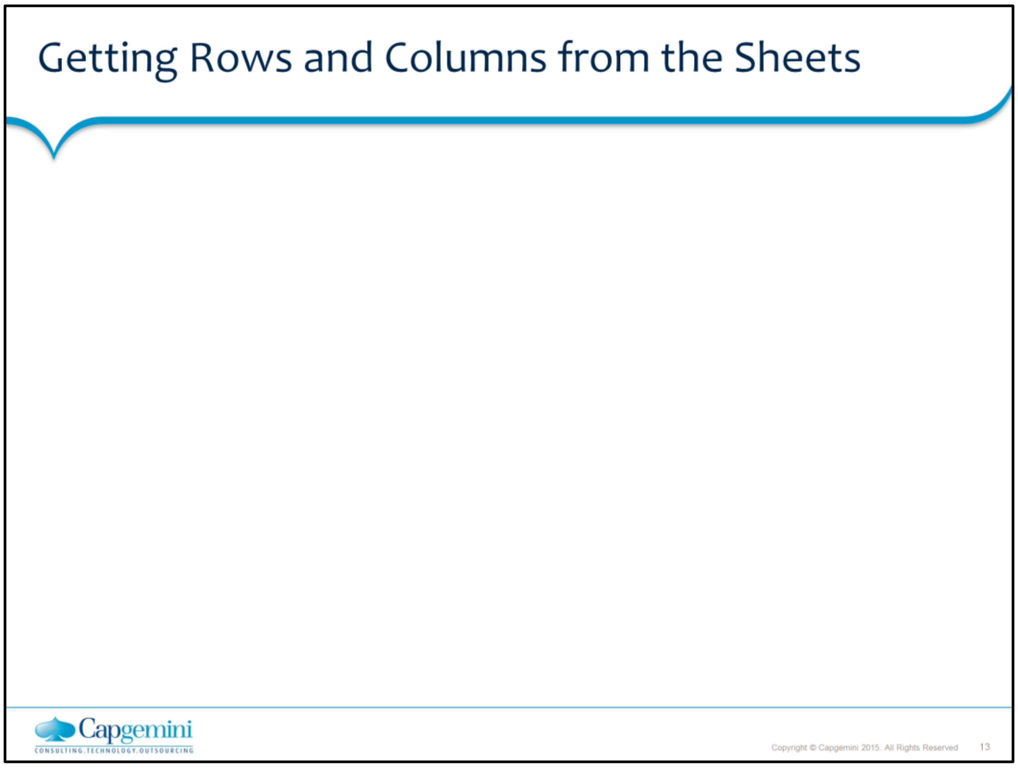
The function column_index_string() does the reverse: You pass it the letter name of a column, and it tells you what number that column is.



## Getting Rows and Columns from the Sheets

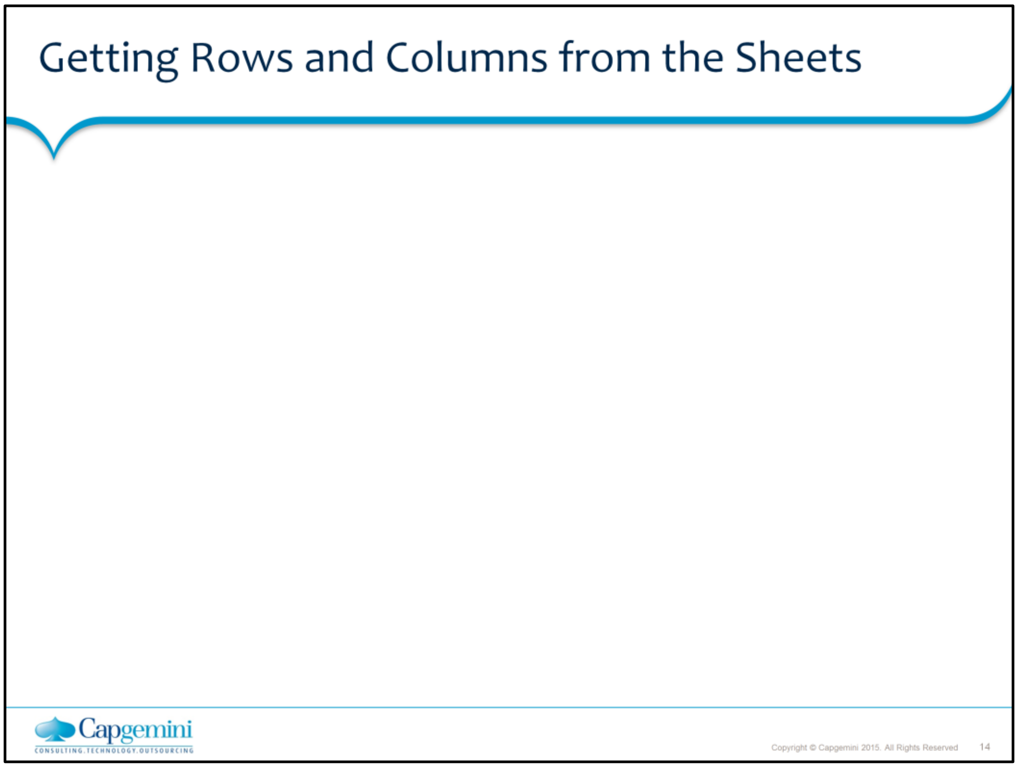
- You can slice Worksheet objects to get all the Cell objects in a row, column, or rectangular area of the spreadsheet. Then you can loop over all the cells in the slice.
- Refer Notes page for statements to be written on Python interactive shell.

```
>>> import openpyxl
  >>> wb = openpyxl.load_workbook('example.xlsx')
  >>> sheet = wb.get_sheet_by_name('Sheet1')
  >>> tuple(sheet['A1':'C3'])
  ((<Cell Sheet1.A1>, <Cell Sheet1.B1>, <Cell Sheet1.C1>), (<Cell Sheet1.A2>,
  <Cell Sheet1.B2>, <Cell Sheet1.C2>), (<Cell Sheet1.A3>, <Cell Sheet1.B3>,
  <Cell Sheet1.C3>))
  >>> for rowOfCellObjects in sheet['A1':'C3']:
         for cellObj in rowOfCellObjects:
           print(cellObj.coordinate, cellObj.value)
       print('--- END OF ROW ---')
  A1 2015-04-05 13:34:02
  B1 Apples
  C1 73
  --- END OF ROW ---
  A2 2015-04-05 03:41:23
  B2 Cherries
  C2 85
  --- END OF ROW ---
  A3 2015-04-06 12:46:51
  B3 Pears
  C3 14
  --- END OF ROW ---
```



## Getting Rows and Columns from the Sheets

Here, we specify that we want the Cell objects in the rectangular area from A1 to C3, and we get aGenerator object containing the Cell objects in that area. To help us visualize this Generatorobject, we can use tuple() on it to display its Cell objects in a tuple.
This tuple contains three tuples: one for each row, from the top of the desired area to the bottom. Each of these three inner tuples contains the Cell objects in one row of our desired area, from the leftmost cell to the right. So overall, our slice of the sheet contains all the Cell objects in the area from A1 to C3, starting from the top-left cell and ending with the bottom-right cell.
To print the values of each cell in the area, we use two for loops. The outer for loop goes over each row in the slice . Then, for each row, the nested for loop goes through each cell in that row .



## Getting Rows and Columns from the Sheets

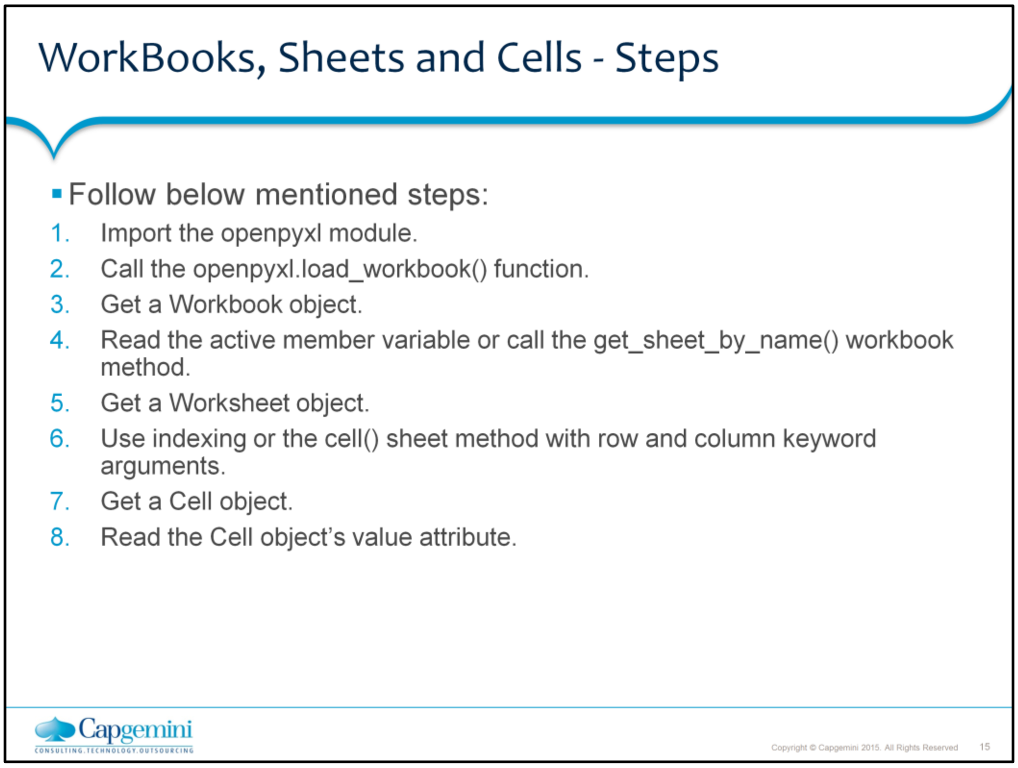
To access the values of cells in a particular row or column, you can also use a Worksheet object'srows and columns attribute. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.active
>>> sheet.columns[1]
(<Cell Sheet1.B1>, <Cell Sheet1.B2>, <Cell Sheet1.B3>, <Cell Sheet1.B4>,
<Cell Sheet1.B5>, <Cell Sheet1.B6>, <Cell Sheet1.B7>)
>>> for cellObj in sheet.columns[1]:
        print(cellObj.value)

Apples
Cherries
Pears
Oranges
Apples
Bananas
Strawberries
```

Using the rows attribute on a Worksheet object will give you a tuple of tuples. Each of these inner tuples represents a row, and contains the Cell objects in that row.

The columns attribute also gives you a tuple of tuples, with each of the inner tuples containing the Cell objects in a particular column. For *example.xlsx*, since there are 7 rows and 3 columns, rows gives us a tuple of 7 tuples (each containing 3 Cell objects), and columns gives us a tuple of 3 tuples (each containing 7 Cellobjects).

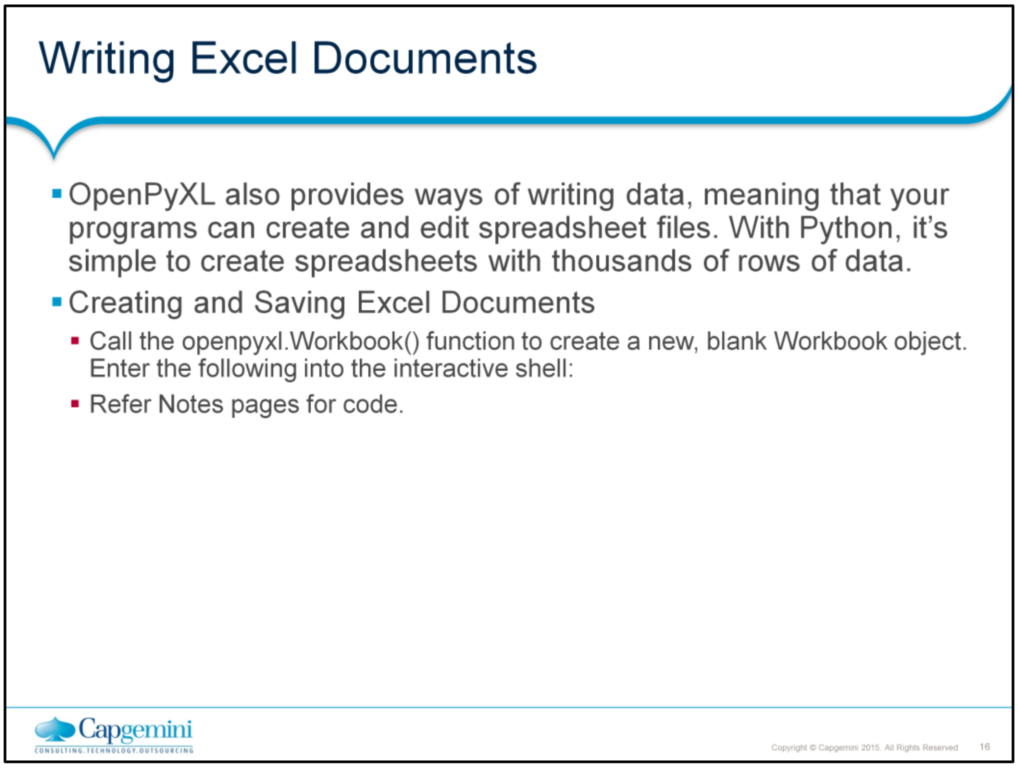
To access one particular tuple, you can refer to it by its index in the larger tuple. For example, to get the tuple that represents column B, you use sheet.columns[1]. To get the tuple containing the Cellobjects in column A, you'd use sheet.columns[0]. Once you have a tuple representing one row or column, you can loop through its Cell objects and print their values.



## WorkBooks, Sheets and Cells - Steps

- Follow below mentioned steps:

1. Import the openpyxl module.
2. Call the openpyxl.load_workbook() function.
3. Get a Workbook object.
4. Read the active member variable or call the get_sheet_by_name() workbook method.
5. Get a Worksheet object.
6. Use indexing or the cell() sheet method with row and column keyword arguments.
7. Get a Cell object.
8. Read the Cell object's value attribute.



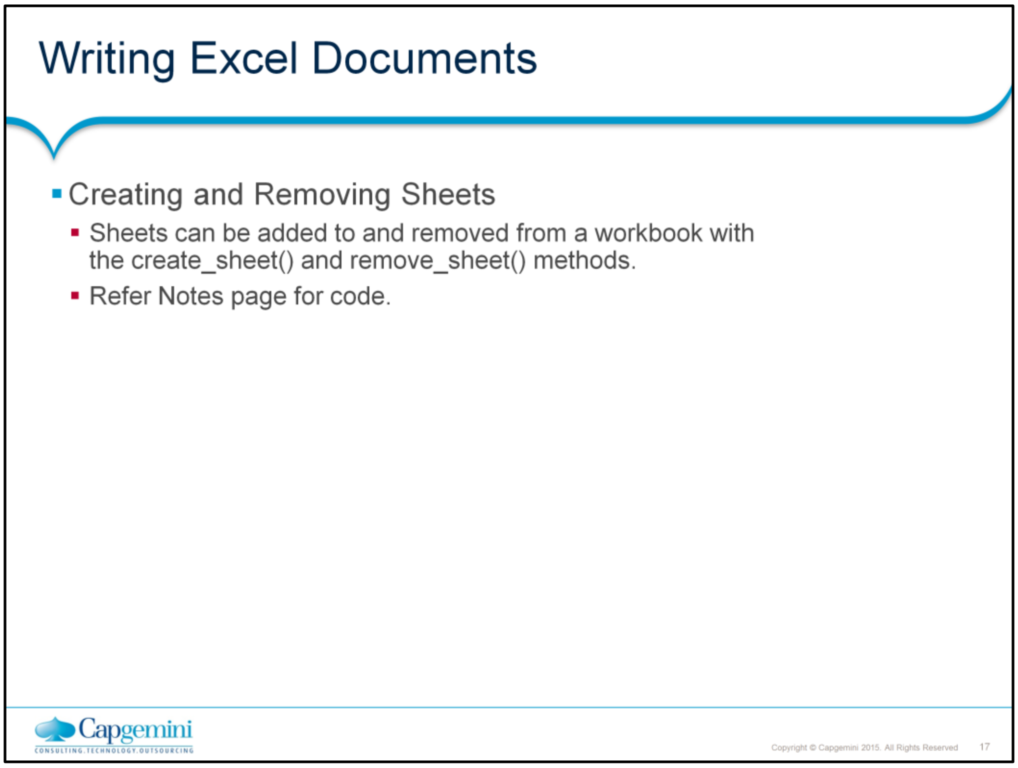
## Writing Excel Documents

- OpenPyXL also provides ways of writing data, meaning that your programs can create and edit spreadsheet files. With Python, it’s simple to create spreadsheets with thousands of rows of data.
- Creating and Saving Excel Documents
  - Call the openpyxl.Workbook() function to create a new, blank Workbook object. Enter the following into the interactive shell:
  - Refer Notes pages for code.

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> wb.get_sheet_names()
['Sheet']
>>> sheet = wb.active
>>> sheet.title
'Sheet'
>>> sheet.title = 'Spam Bacon Eggs Sheet'
>>> wb.get_sheet_names()
['Spam Bacon Eggs Sheet']
```

The workbook will start off with a single sheet named *Sheet*. You can change the name of the sheet by storing a new string in its title attribute.
Any time you modify the Workbook object or its sheets and cells, the spreadsheet file will not be saved until you call the save() workbook method. Enter the following into the interactive shell (with*example.xlsx* in the current working directory):

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.active
>>> sheet.title = 'Spam Spam Spam'
>>> wb.save('example_copy.xlsx')
```



## Writing Excel Documents

- Creating and Removing Sheets
  - Sheets can be added to and removed from a workbook with the create_sheet() and remove_sheet() methods.
  - Refer Notes page for code.

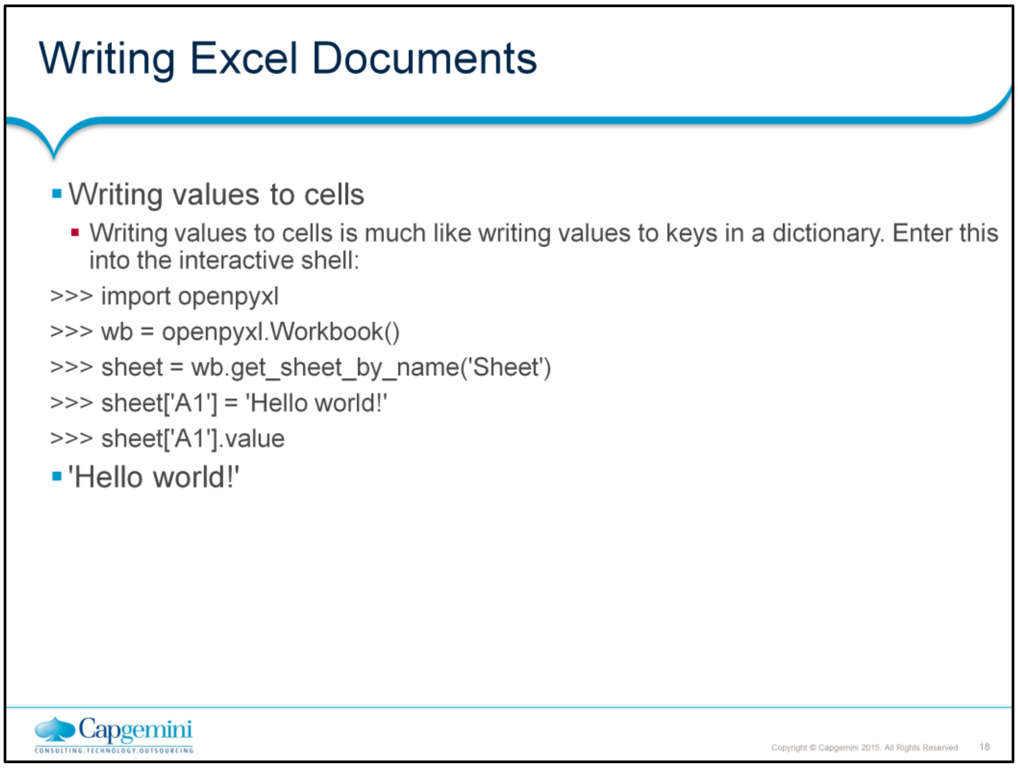
```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> wb.get_sheet_names()
['Sheet']
>>> wb.create_sheet()
<Worksheet "Sheet1">
>>> wb.get_sheet_names()
['Sheet', 'Sheet1']
>>> wb.create_sheet(index=0, title='First Sheet')
<Worksheet "First Sheet">
>>> wb.get_sheet_names()
['First Sheet', 'Sheet', 'Sheet1']
>>> wb.create_sheet(index=2, title='Middle Sheet')
<Worksheet "Middle Sheet">
>>> wb.get_sheet_names()
['First Sheet', 'Sheet', 'Middle Sheet', 'Sheet1']
```

The create_sheet() method returns a new Worksheet object named Sheet*X*, which by default is set to be the last sheet in the workbook. Optionally, the index and name of the new sheet can be specified with the index and title keyword arguments.

Continue the previous example by entering the following:

```
>>> wb.get_sheet_names()
 ['First Sheet', 'Sheet', 'Middle Sheet', 'Sheet1']
>>> wb.remove_sheet(wb.get_sheet_by_name('Middle Sheet'))
 >>> wb.remove_sheet(wb.get_sheet_by_name('Sheet1'))
 >>> wb.get_sheet_names()
['First Sheet', 'Sheet']
```

The remove_sheet() method takes a Worksheet object, not a string of the sheet name, as its argument. If you know only the name of a sheet you want to remove, call get_sheet_by_name()and pass its return value into remove_sheet().
Remember to call the save() method to save the changes after adding sheets to or removing sheets from the workbook.



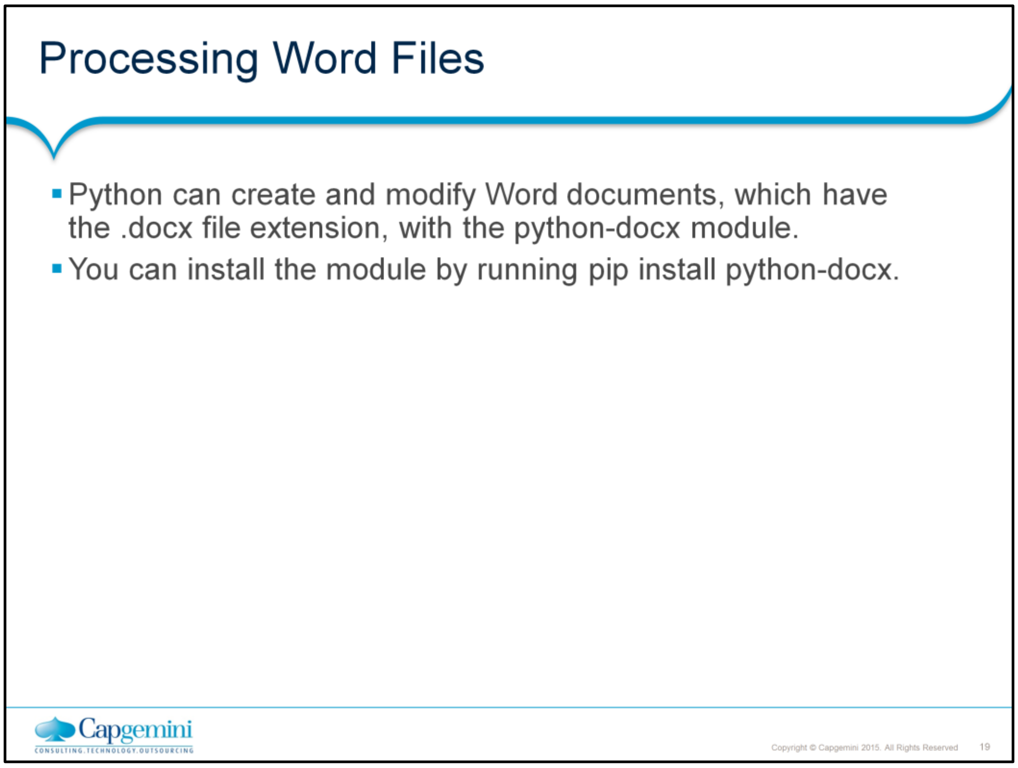
## Writing Excel Documents

- Writing values to cells
  - Writing values to cells is much like writing values to keys in a dictionary. Enter this into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb.get_sheet_by_name('Sheet')
>>> sheet['A1'] = 'Hello world!'
>>> sheet['A1'].value
```

- 'Hello world!'

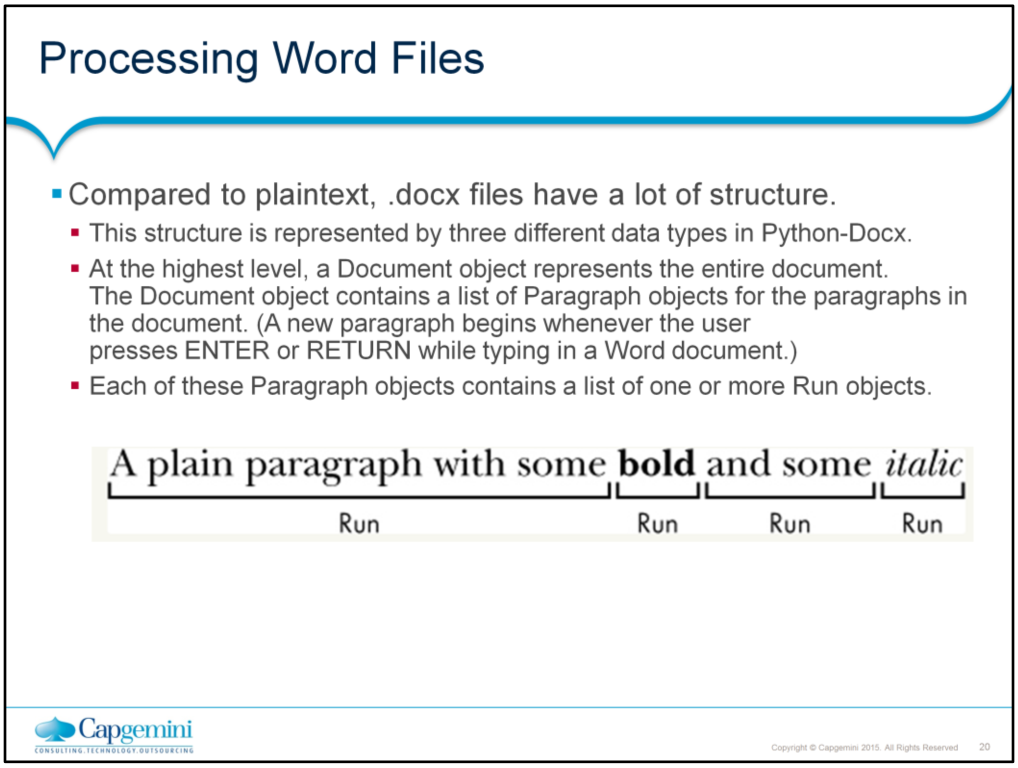
If you have the cell's coordinate as a string, you can use it just like a dictionary key on the Worksheetobject to specify which cell to write to.



## Processing Word Files

- Python can create and modify Word documents, which have the .docx file extension, with the python-docx module.
- You can install the module by running pip install python-docx.

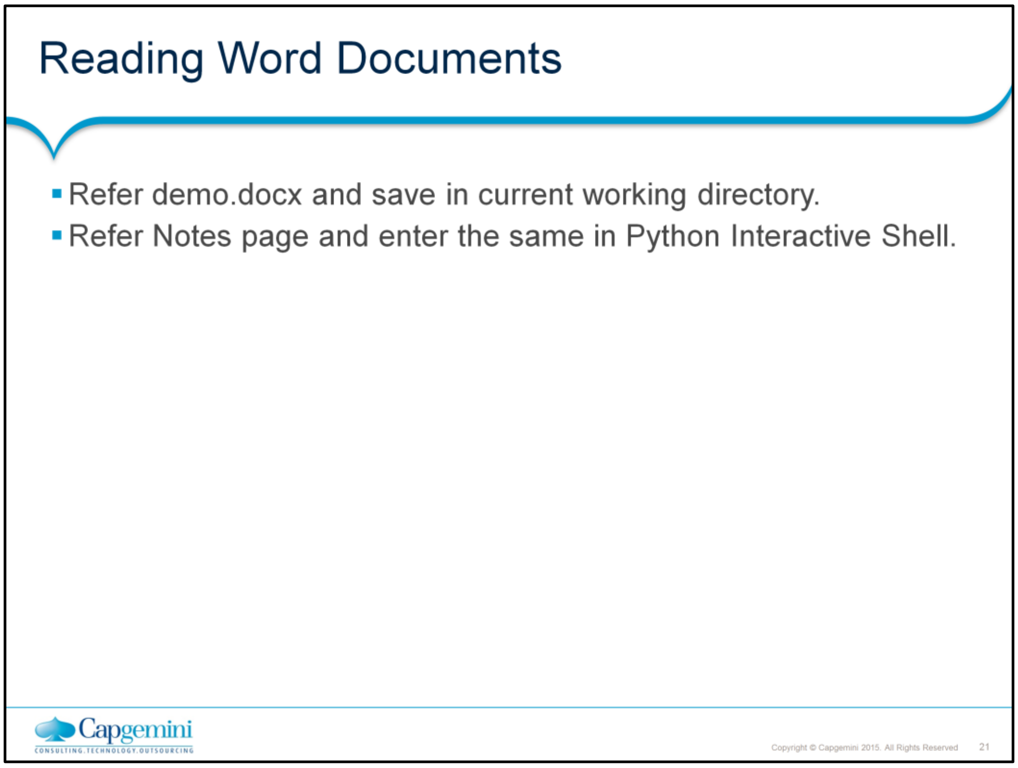
The full documentation for Python-Docx is available at *https://python-docx.readthedocs.org/*



## Processing Word Files

- Compared to plaintext, .docx files have a lot of structure.
  - This structure is represented by three different data types in Python-Docx.
  - At the highest level, a Document object represents the entire document. The Document object contains a list of Paragraph objects for the paragraphs in the document. (A new paragraph begins whenever the user presses ENTER or RETURN while typing in a Word document.)
  - Each of these Paragraph objects contains a list of one or more Run objects.

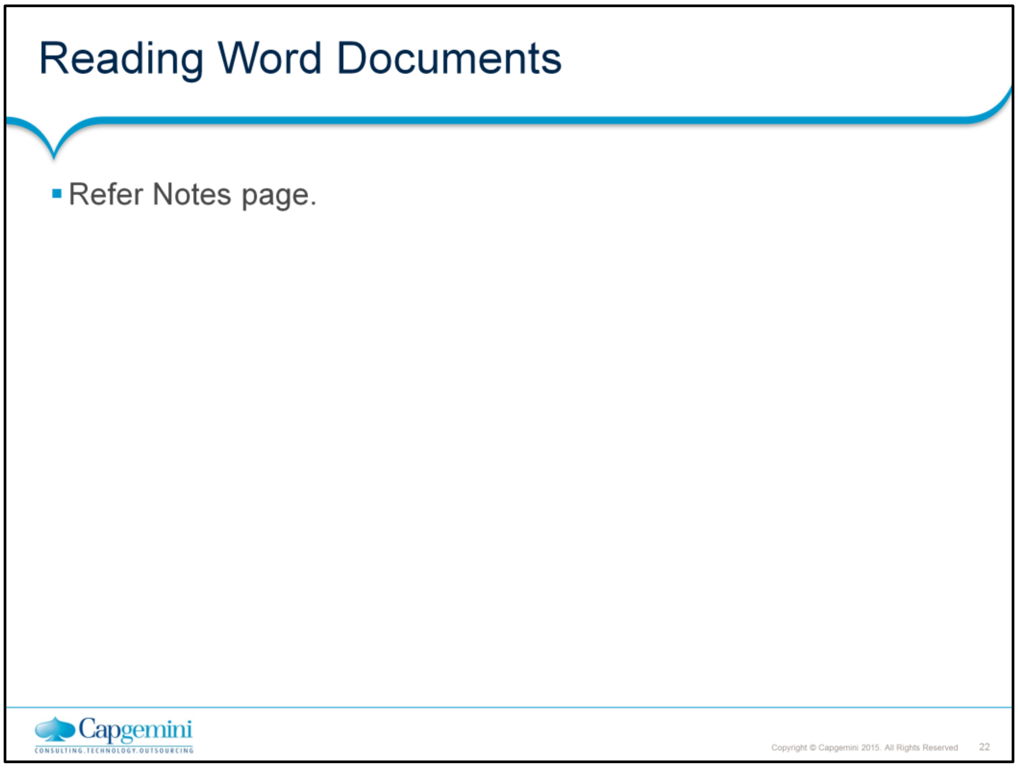
- The text in a Word document is more than just a string. It has font, size, color, and other styling information associated with it.
- A *style* in Word is a collection of these attributes.
- A Run object is a contiguous run of text with the same style. A new Run object is needed whenever the text style changes.



## Reading Word Documents

- Refer demo.docx and save in current working directory.
- Refer Notes page and enter the same in Python Interactive Shell.

```
>>> import docx
❶ >>> doc = docx.Document('demo.docx')
❷ >>> len(doc.paragraphs)
  7
❸ >>> doc.paragraphs[0].text
  'Document Title'
❹ >>> doc.paragraphs[1].text
  'A plain paragraph with some bold and some italic'
❺ >>> len(doc.paragraphs[1].runs)
  4
❻ >>> doc.paragraphs[1].runs[0].text
  'A plain paragraph with some '
❼ >>> doc.paragraphs[1].runs[1].text
  'bold'
❽ >>> doc.paragraphs[1].runs[2].text
  ' and some '
❾ >>> doc.paragraphs[1].runs[3].text
  'italic'
```

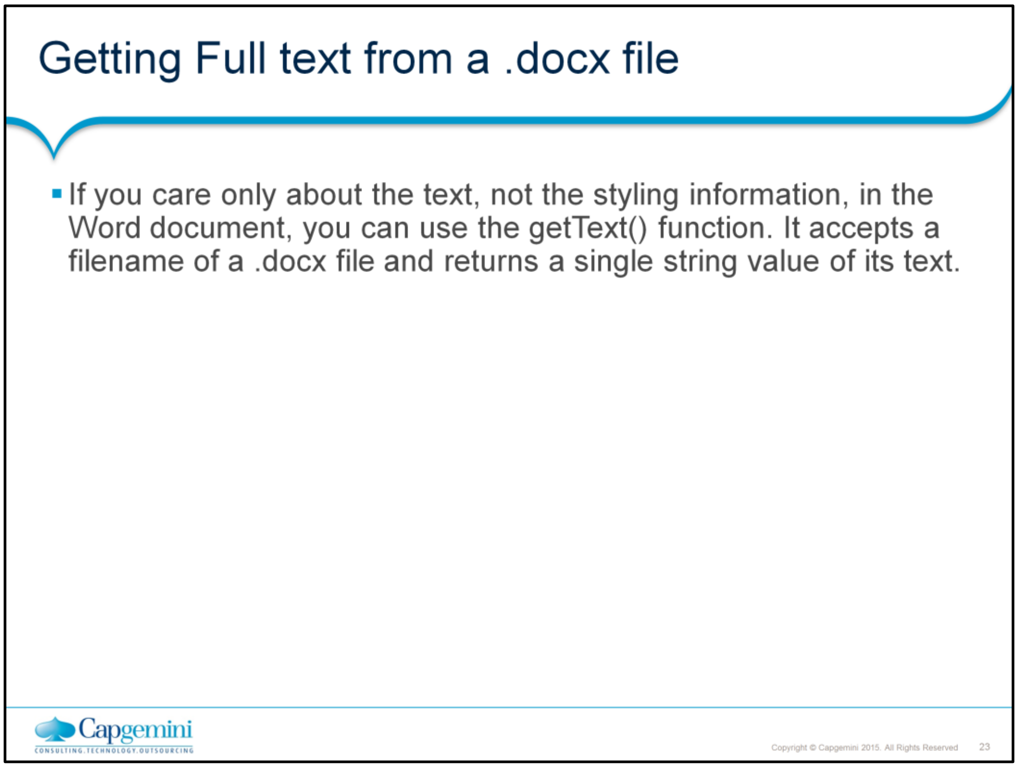


## Reading Word Documents

- Refer Notes page.

Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING

1. At ❶, we open a *.docx* file in Python, call docx.Document(), and pass the filename *demo.docx*. This will return a Document object, which has a paragraphs attribute that is a list of Paragraph objects. When we call len() on doc.paragraphs, it returns 7, which tells us that there are sevenParagraph objects in this document ❷.
2. Each of these Paragraph objects has a text attribute that contains a string of the text in that paragraph (without the style information). Here, the first textattribute contains 'DocumentTitle' ❸,
3. The second contains 'A plain paragraph with some bold and some italic' ❹.
4. Each Paragraph object also has a runs attribute that is a list of Run objects. Run objects also have atext attribute, containing just the text in that particular run. Let's look at the text attributes in the second Paragraph object, 'A plain paragraph with some bold and some italic'. Calling len() on this Paragraph object tells us that there are four Run objects ❺. 5. The first run object contains 'A plain paragraph with some ' ❻.
6. Then, the text change to a bold style, so 'bold'starts a new Run object ❼.
7. The text returns to an unbolded style after that, which results in a third Runobject, ' and some ' ❽.
8. Finally, the fourth and last Run object contains 'italic' in an italic style ❾.



## Getting Full text from a .docx file

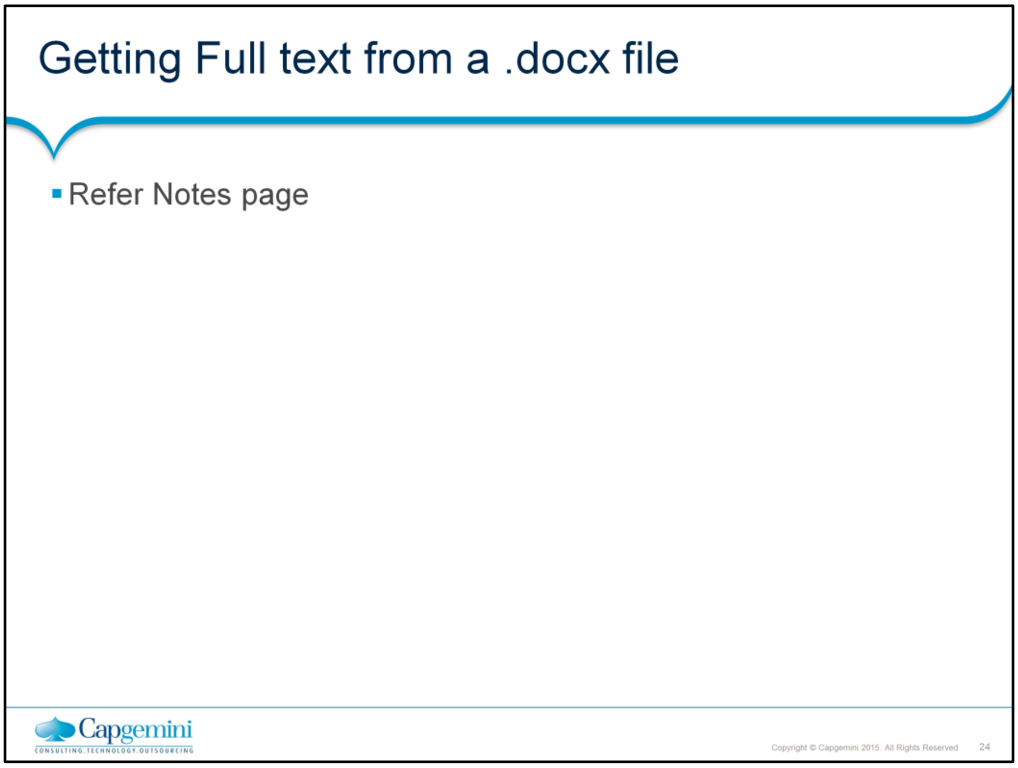
- If you care only about the text, not the styling information, in the Word document, you can use the getText() function. It accepts a filename of a .docx file and returns a single string value of its text.

Open a new file editor window and enter the following code, saving it as readDocx.py:

```
#! python3

import docx

def getText(filename):
    doc = docx.Document(filename)
    fullText = []
    for para in doc.paragraphs:
        fullText.append(para.text)
    return '\n'.join(fullText)
```

The getText() function opens the Word document, loops over all the Paragraph objects in the paragraphs list, and then appends their text to the list in fullText. After the loop, the strings in fullText are joined together with newline characters.
The *readDocx.py* program can be imported like any other module. Now if you just need the text from a Word document, you can enter the following:

```
>>> import readDocx
>>> print(readDocx.getText('demo.docx'))


Document Title
A plain paragraph with some bold and some italic
Heading, level 1
Intense quote
first item in unordered list
first item in ordered list
```



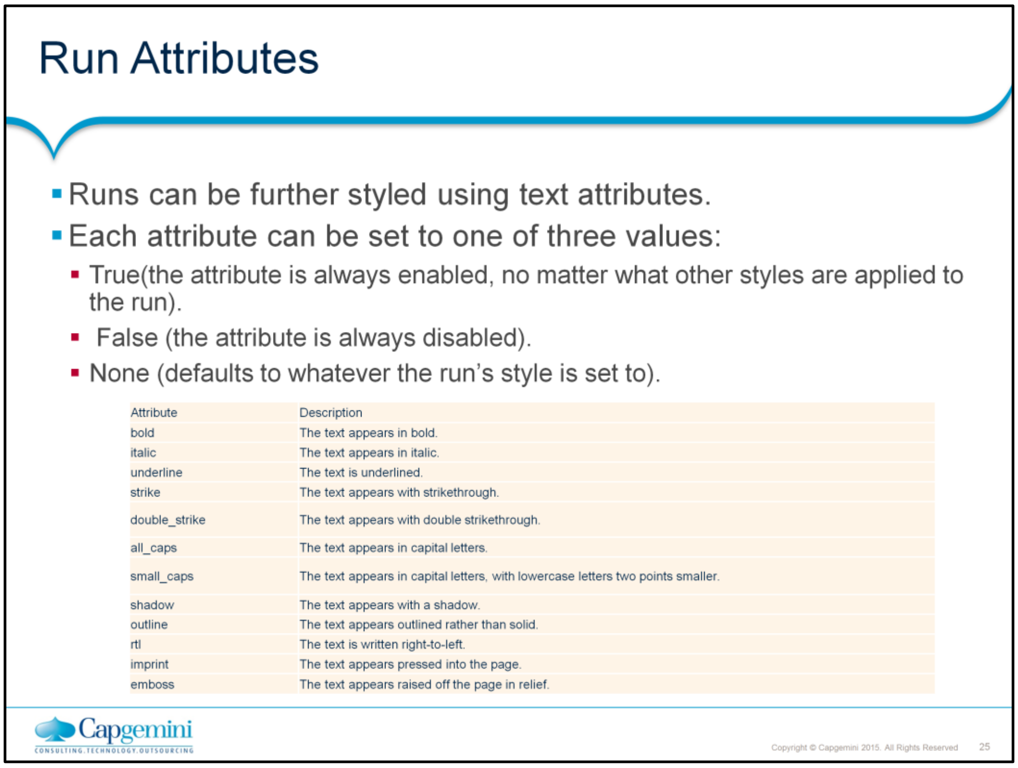
## Getting Full text from a .docx file

- Refer Notes page

You can also adjust getText() to modify the string before returning it. For example, to indent each paragraph, replace the append() call in *readDocx.py* with this:
fullText.append(**' ' +** para.text)

To add a double space in between paragraphs, change the join() call code to this:
return '\n**\n**'.join(fullText)

As you can see, it takes only a few lines of code to write functions that will read a *.docx* file and return a string of its content



## Run Attributes

- Runs can be further styled using text attributes.
- Each attribute can be set to one of three values:
  - True(the attribute is always enabled, no matter what other styles are applied to the run).
  - False (the attribute is always disabled).
  - None (defaults to whatever the run's style is set to).

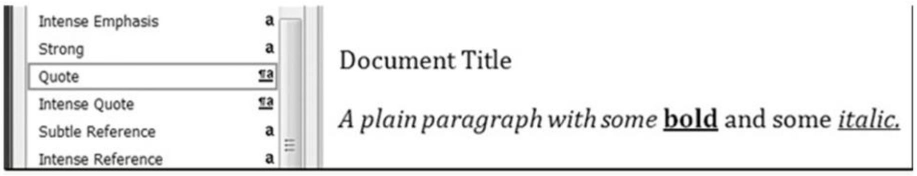
| Attribute | Description |
|---|---|
| bold | The text appears in bold. |
| italic | The text appears in italic. |
| underline | The text is underlined. |
| strike | The text appears with strikethrough. |
| double_strike | The text appears with double strikethrough. |
| all_caps | The text appears in capital letters. |
| small_caps | The text appears in capital letters, with lowercase letters two points smaller. |
| shadow | The text appears with a shadow. |
| outline | The text appears outlined rather than solid. |
| rtl | The text is written right-to-left. |
| imprint | The text appears pressed into the page. |
| emboss | The text appears raised off the page in relief. |

For example, to change the styles of demo.docx, enter the following into the interactive shell:

```
>>> doc = docx.Document('demo.docx')
>>> doc.paragraphs[0].text
'Document Title'
>>> doc.paragraphs[0].style
'Title'
>>> doc.paragraphs[0].style = 'Normal'
>>> doc.paragraphs[1].text
'A plain paragraph with some bold and some italic'
>>> (doc.paragraphs[1].runs[0].text, doc.paragraphs[1].runs[1].text, doc.
paragraphs[1].runs[2].text, doc.paragraphs[1].runs[3].text)
('A plain paragraph with some ', 'bold', ' and some ', 'italic')
>>> doc.paragraphs[1].runs[0].style = 'QuoteChar'
>>> doc.paragraphs[1].runs[1].underline = True
>>> doc.paragraphs[1].runs[3].underline = True
>>> doc.save('restyled.docx')
```

## Run Attributes

- Here we show how the styles of paragraphs and runs look in restyled.docx.

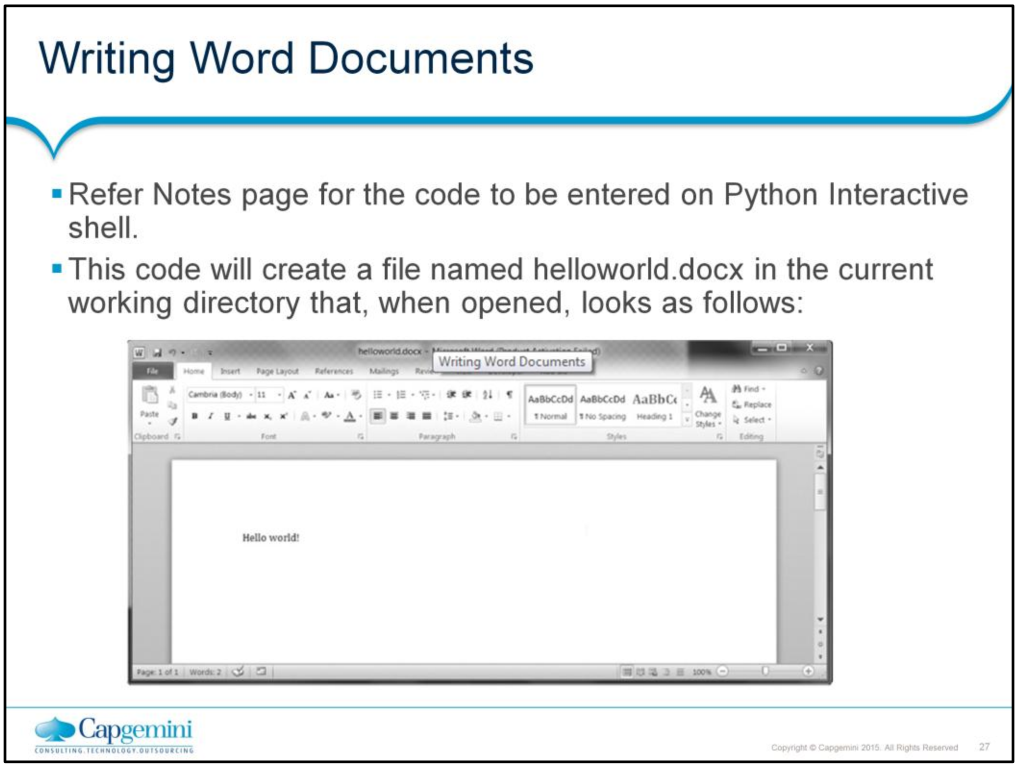


Document Title

*A plain paragraph with some* **bold** and some *italic.*

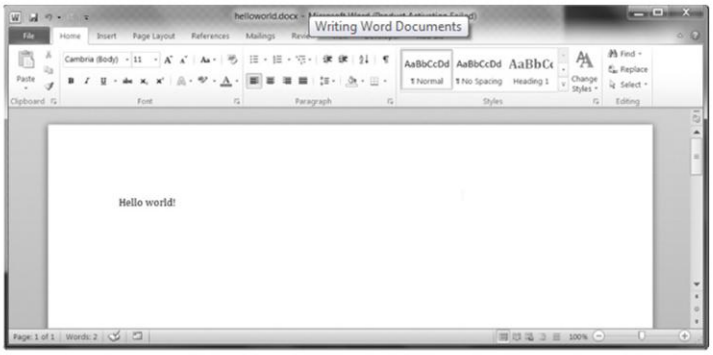
In code in notes page of previous slide, we use the text and style attributes to easily see what's in the paragraphs in our document.
We can see that it's simple to divide a paragraph into runs and access each run individually. So we get the first, second, and fourth runs in the second paragraph, style each run, and save the results to a new document.

The words *Document Title* at the top of *restyled.docx* will have the Normal style instead of the Title style, the Run object for the text *A plain paragraph with some* will have the QuoteChar style, and the two Run objects for the words *bold* and *italic* will have their underline attributes set to True.



## Writing Word Documents

- Refer Notes page for the code to be entered on Python Interactive shell.
- This code will create a file named helloworld.docx in the current working directory that, when opened, looks as follows:

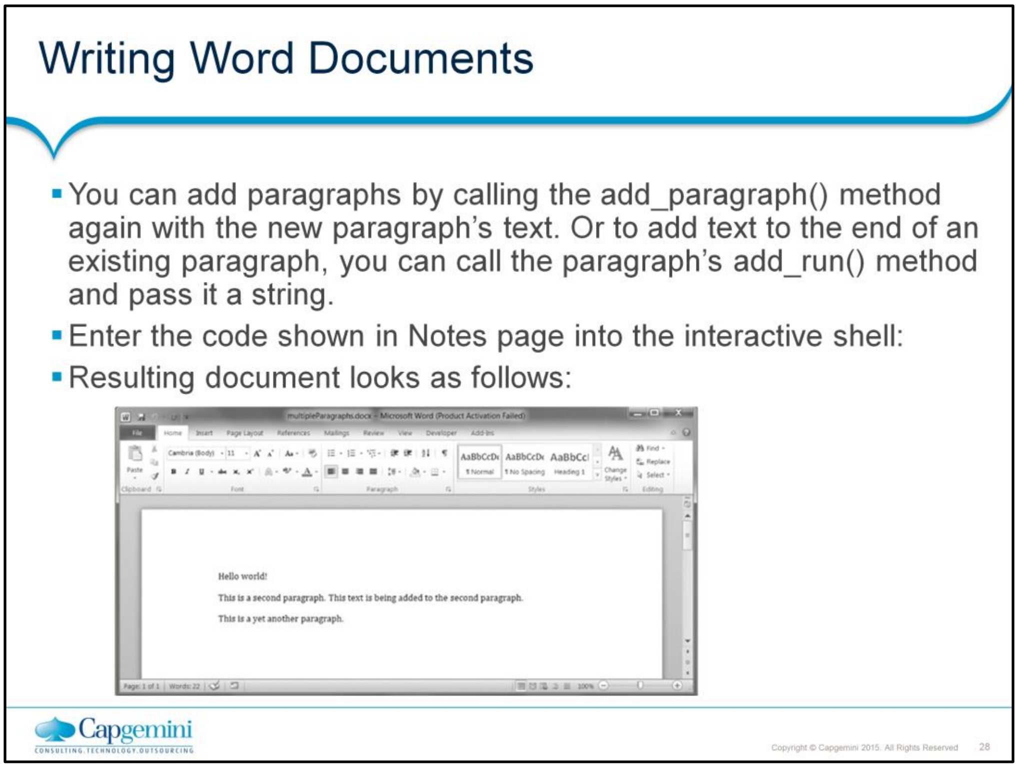


Writing Word Documents

Hello world!

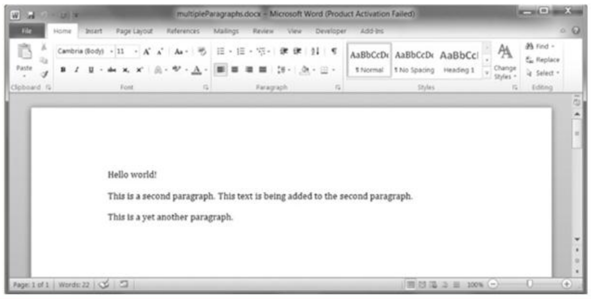
```
>>> import docx
>>> doc = docx.Document()
>>> doc.add_paragraph('Hello world!')

<docx.text.Paragraph object at 0x0000000003B56F60>

>>> doc.save('helloworld.docx')
```

To create your own *.docx* file, call docx.Document() to return a new, blank Word Document object. The add_paragraph() document method adds a new paragraph of text to the document and returns a reference to the Paragraph object that was added. When you’re done adding text, pass a filename string to the save() document method to save the Document object to a file.



## Writing Word Documents

- You can add paragraphs by calling the add_paragraph() method again with the new paragraph’s text. Or to add text to the end of an existing paragraph, you can call the paragraph’s add_run() method and pass it a string.
- Enter the code shown in Notes page into the interactive shell:
- Resulting document looks as follows:



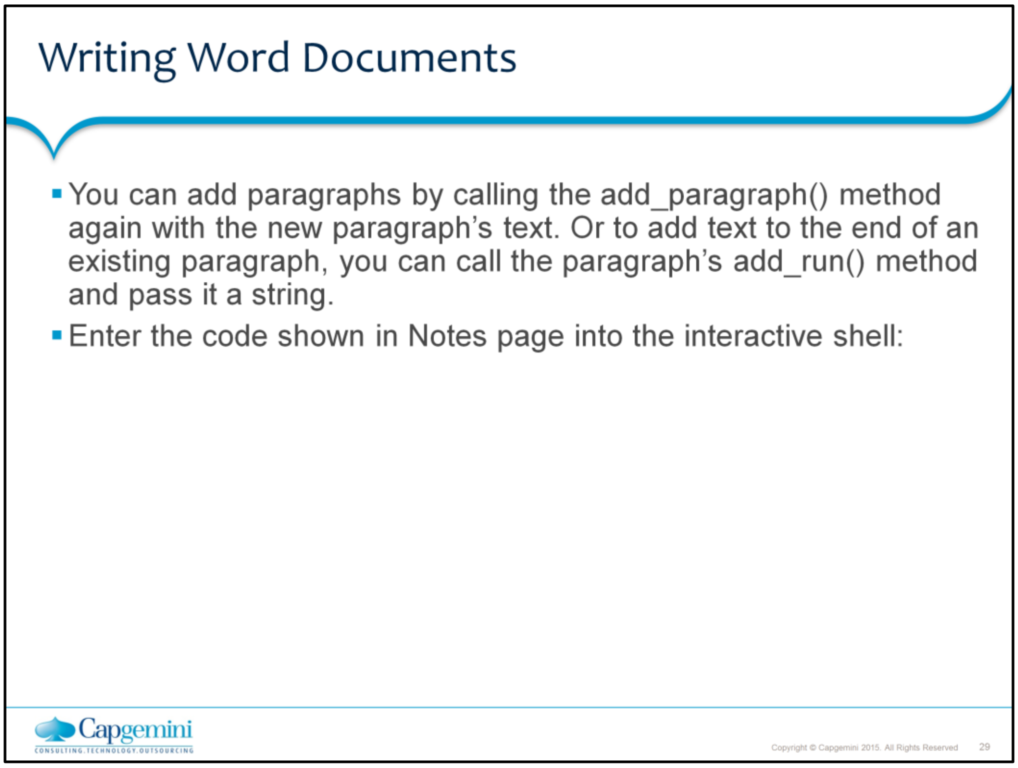
Hello world!

This is a second paragraph. This text is being added to the second paragraph.

This is a yet another paragraph.

```
>>> import docx
>>> doc = docx.Document()
>>> doc.add_paragraph('Hello world!')

<docx.text.Paragraph object at 0x000000000366AD30>

>>> paraObj1 = doc.add_paragraph('This is a second paragraph.')
>>> paraObj2 = doc.add_paragraph('This is a yet another paragraph.')
>>> paraObj1.add_run(' This text is being added to the second paragraph.')

<docx.text.Run object at 0x0000000003A2C860>

>>> doc.save('multipleParagraphs.docx')
```

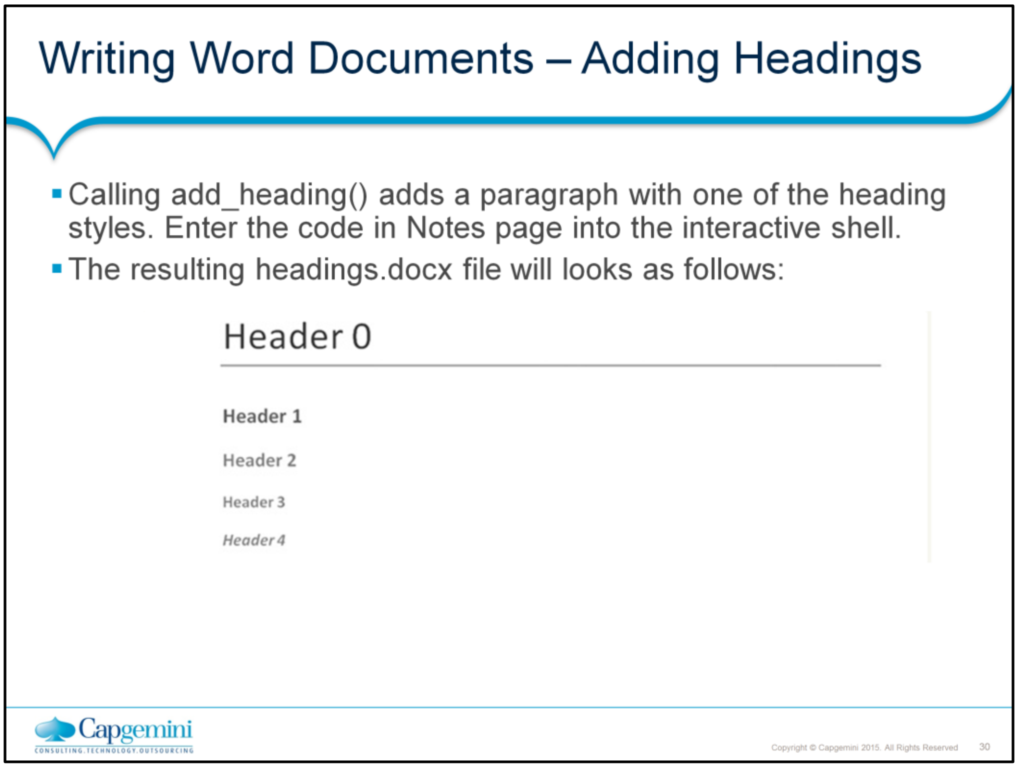
Note that the text *This text is being added to the second paragraph.* was added to the Paragraph object in paraObj1, which was the second paragraph added to doc. The add_paragraph() and add_run() functions return paragraph and Runobjects, respectively, to save you the trouble of extracting them as a separate step.
Keep in mind that as of Python-Docx version 0.5.3, new Paragraph objects can be added only to the end of the document, and new Run objects can be added only to the end of a Paragraph object.
The save() method can be called again to save the additional changes you’ve made.



## Writing Word Documents

- You can add paragraphs by calling the add_paragraph() method again with the new paragraph's text. Or to add text to the end of an existing paragraph, you can call the paragraph's add_run() method and pass it a string.
- Enter the code shown in Notes page into the interactive shell:

```
>>> import docx
>>> doc = docx.Document()
>>> doc.add_paragraph('Hello world!')

<docx.text.Paragraph object at 0x000000000366AD30>

>>> paraObj1 = doc.add_paragraph('This is a second paragraph.')
>>> paraObj2 = doc.add_paragraph('This is a yet another paragraph.')
>>> paraObj1.add_run(' This text is being added to the second paragraph.')

<docx.text.Run object at 0x0000000003A2C860>

>>> doc.save('multipleParagraphs.docx')
```



## Writing Word Documents – Adding Headings

- Calling add_heading() adds a paragraph with one of the heading styles. Enter the code in Notes page into the interactive shell.
- The resulting headings.docx file will looks as follows:

Header 0

**Header 1**

Header 2

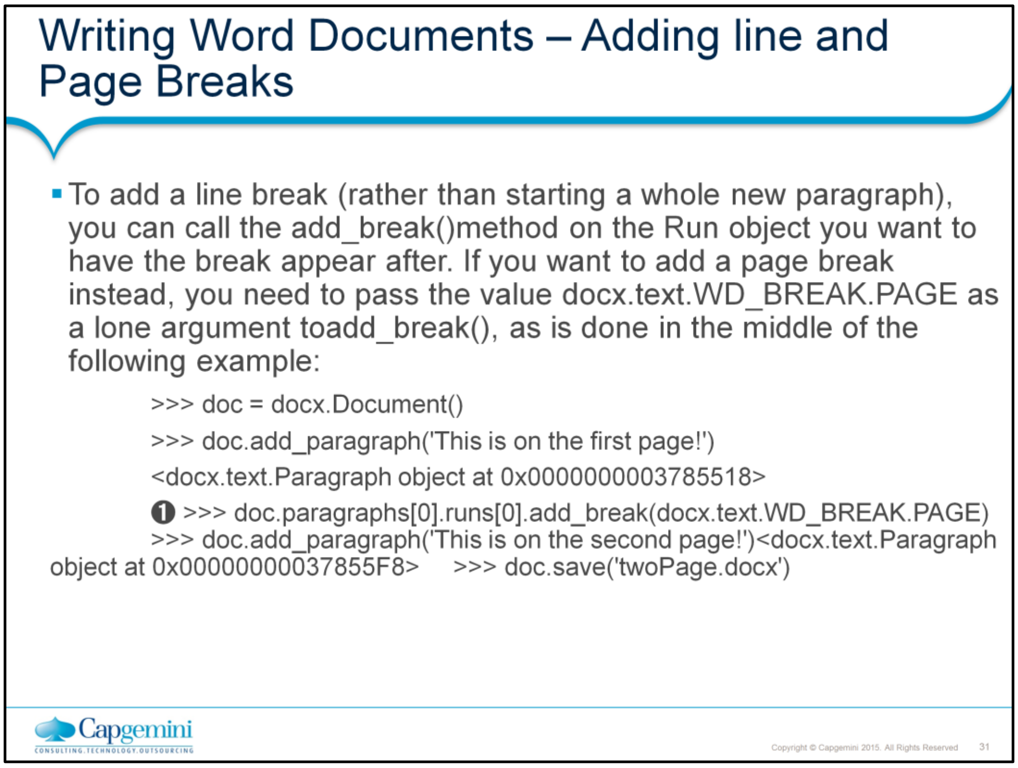
Header 3

*Header 4*

Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING

```
>>> doc = docx.Document()
>>> doc.add_heading('Header 0', 0)

<docx.text.Paragraph object at 0x00000000036CB3C8>
>>> doc.add_heading('Header 1', 1)

<docx.text.Paragraph object at 0x00000000036CB630>
>>> doc.add_heading('Header 2', 2)

<docx.text.Paragraph object at 0x00000000036CB828>
>>> doc.add_heading('Header 3', 3)

<docx.text.Paragraph object at 0x00000000036CB2E8>
>>> doc.add_heading('Header 4', 4)

<docx.text.Paragraph object at 0x00000000036CB3C8>

>>> doc.save('headings.docx')
```

The arguments to add_heading() are a string of the heading text and an integer from 0 to 4. The integer 0 makes the heading the Title style, which is used for the top of the document. Integers 1 to 4are for various heading levels, with 1 being the main heading and 4 the lowest subheading. Theadd_heading() function returns a Paragraph object to save you the step of extracting it from the Document object as a separate step.

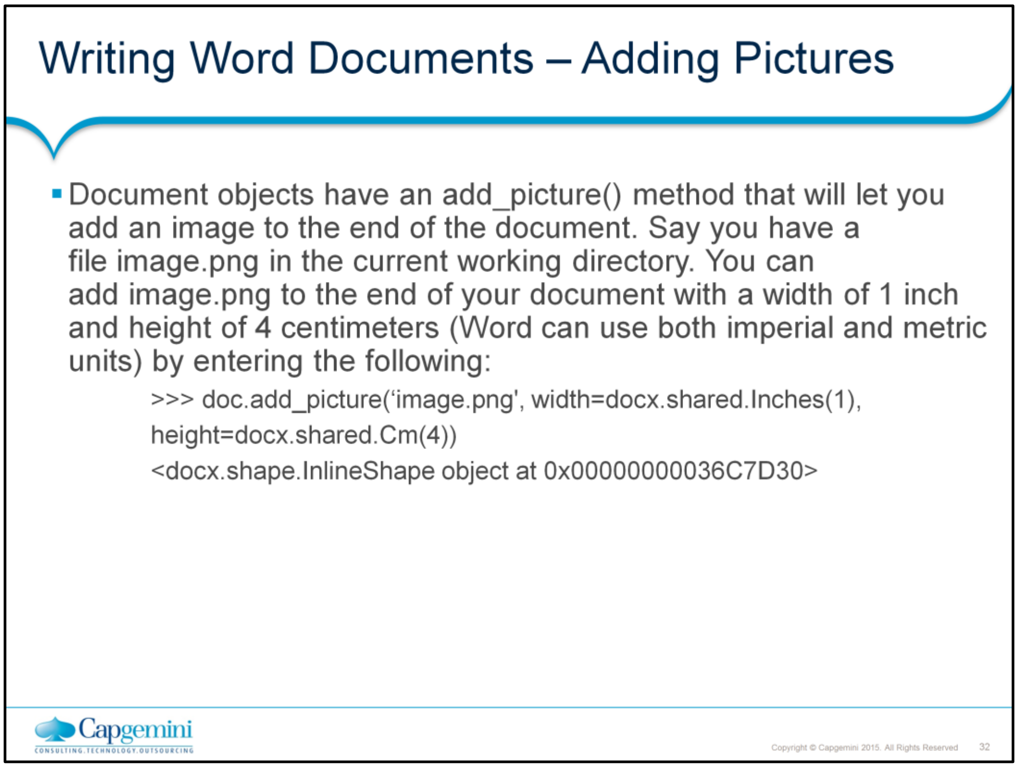


## Writing Word Documents – Adding line and Page Breaks

- To add a line break (rather than starting a whole new paragraph), you can call the add_break()method on the Run object you want to have the break appear after. If you want to add a page break instead, you need to pass the value docx.text.WD_BREAK.PAGE as a lone argument toadd_break(), as is done in the middle of the following example:

```
>>> doc = docx.Document()
>>> doc.add_paragraph('This is on the first page!')
<docx.text.Paragraph object at 0x0000000003785518>
❶ >>> doc.paragraphs[0].runs[0].add_break(docx.text.WD_BREAK.PAGE)
>>> doc.add_paragraph('This is on the second page!')<docx.text.Paragraph object at 0x00000000037855F8>     >>> doc.save('twoPage.docx')
```

This creates a two-page Word document with *This is on the first page!* on the first page and *This is on the second page!* on the second. Even though there was still plenty of space on the first page after the text *This is on the first page!*, we forced the next paragraph to begin on a new page by inserting a page break after the first run of the first paragraph ❶.

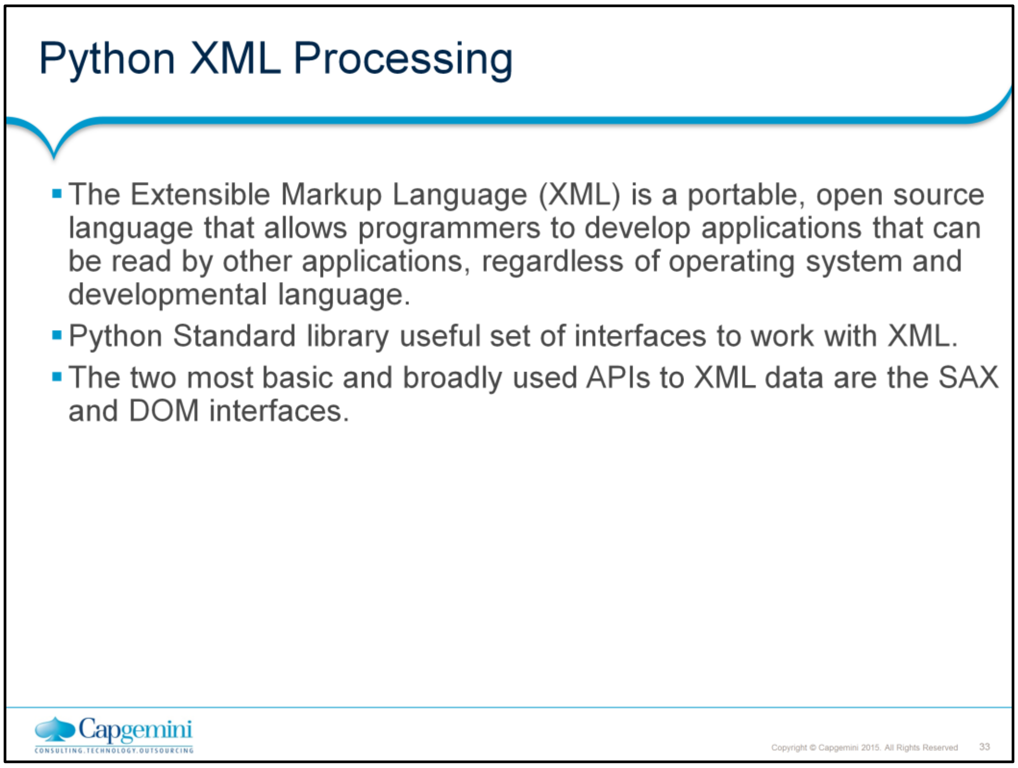


## Writing Word Documents – Adding Pictures

- Document objects have an add_picture() method that will let you add an image to the end of the document. Say you have a file image.png in the current working directory. You can add image.png to the end of your document with a width of 1 inch and height of 4 centimeters (Word can use both imperial and metric units) by entering the following:

```
>>> doc.add_picture('image.png', width=docx.shared.Inches(1),
height=docx.shared.Cm(4))
<docx.shape.InlineShape object at 0x00000000036C7D30>
```

The first argument is a string of the image's filename. The optional width and height keyword arguments will set the width and height of the image in the document. If left out, the width and height will default to the normal size of the image.
You'll probably prefer to specify an image's height and width in familiar units such as inches and centimeters, so you can use the docx.shared.Inches() and docx.shared.Cm() functions when you're specifying the width and height keyword arguments.



## Python XML Processing

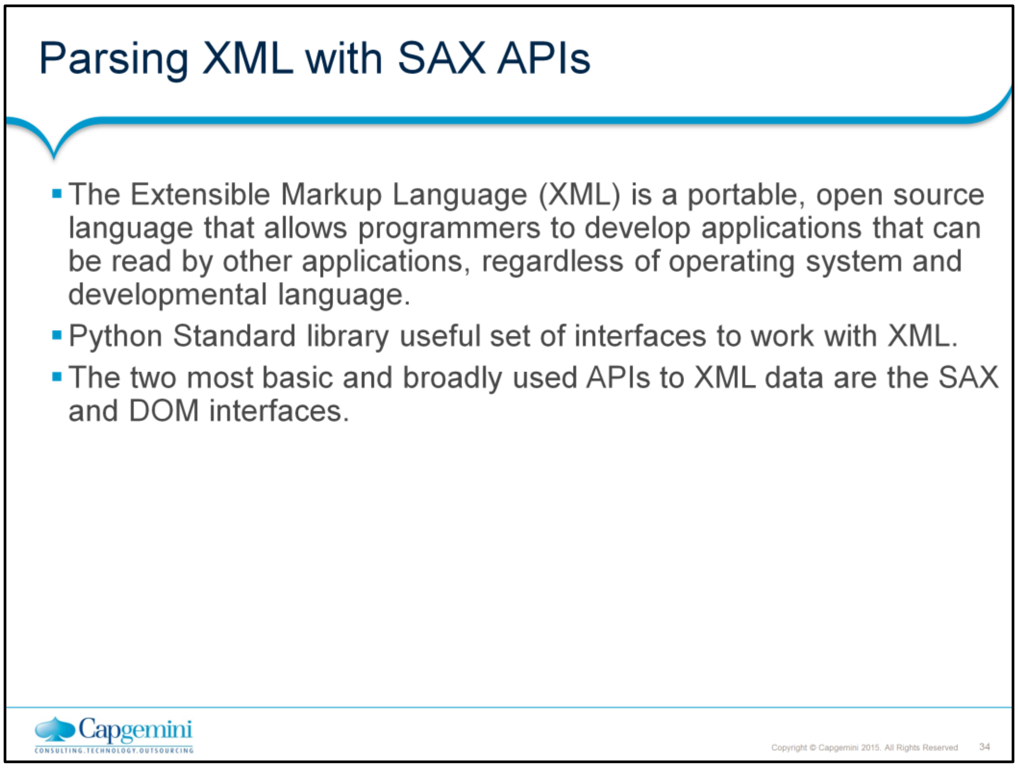
- The Extensible Markup Language (XML) is a portable, open source language that allows programmers to develop applications that can be read by other applications, regardless of operating system and developmental language.
- Python Standard library useful set of interfaces to work with XML.
- The two most basic and broadly used APIs to XML data are the SAX and DOM interfaces.

- Simple API for XML (SAX) : Here, you register callbacks for events of interest and then let the parser proceed through the document. This is useful when your documents are large or you have memory limitations, it parses the file as it reads it from disk and the entire file is never stored in memory.

- Document Object Model (DOM) API : This is a World Wide Web Consortium recommendation wherein the entire file is read into memory and stored in a hierarchical (tree-based) form to represent all the features of an XML document.

- SAX obviously cannot process information as fast as DOM can when working with large files. On the other hand, using DOM exclusively can really kill your resources, especially if used on a lot of small files.

- SAX is read-only, while DOM allows changes to the XML file.



## Parsing XML with SAX APIs

- The Extensible Markup Language (XML) is a portable, open source language that allows programmers to develop applications that can be read by other applications, regardless of operating system and developmental language.
- Python Standard library useful set of interfaces to work with XML.
- The two most basic and broadly used APIs to XML data are the SAX and DOM interfaces.

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

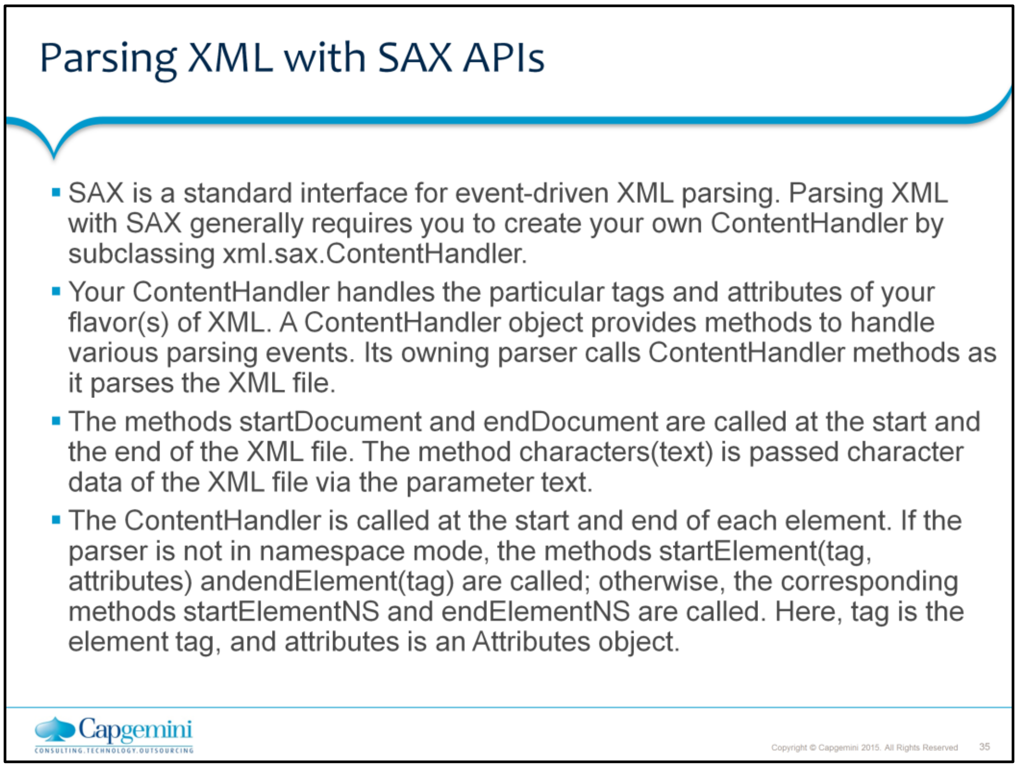
34

- Simple API for XML (SAX) : Here, you register callbacks for events of interest and then let the parser proceed through the document. This is useful when your documents are large or you have memory limitations, it parses the file as it reads it from disk and the entire file is never stored in memory.

- Document Object Model (DOM) API : This is a World Wide Web Consortium recommendation wherein the entire file is read into memory and stored in a hierarchical (tree-based) form to represent all the features of an XML document.

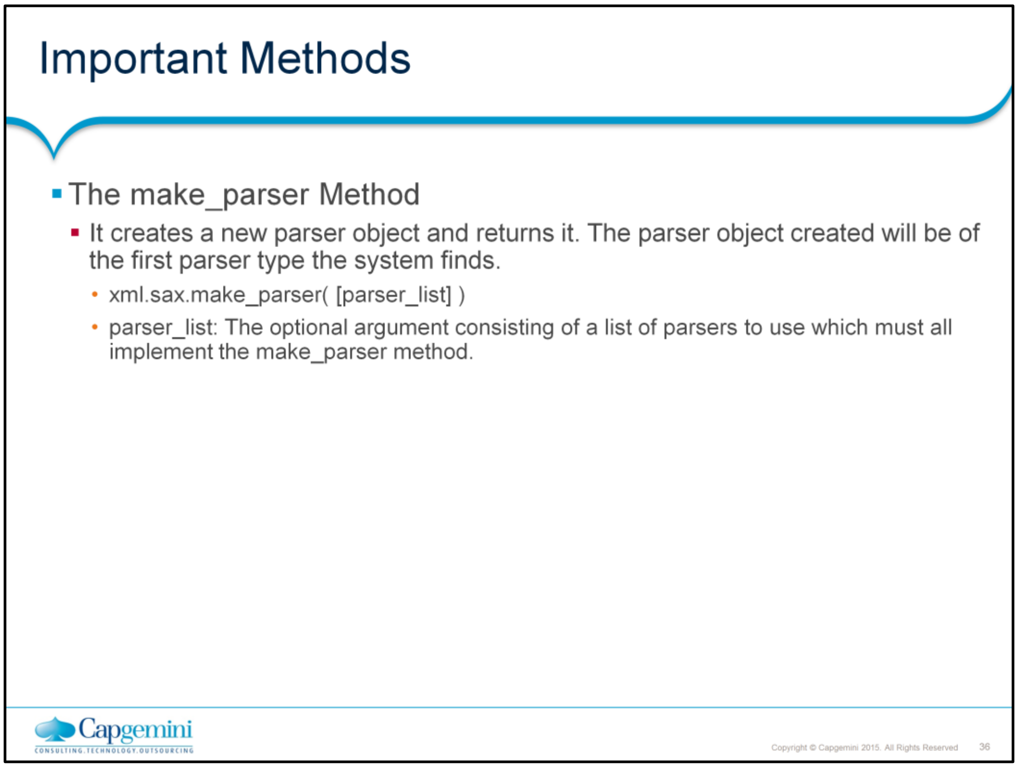
- SAX obviously cannot process information as fast as DOM can when working with large files. On the other hand, using DOM exclusively can really kill your resources, especially if used on a lot of small files.

- SAX is read-only, while DOM allows changes to the XML file.



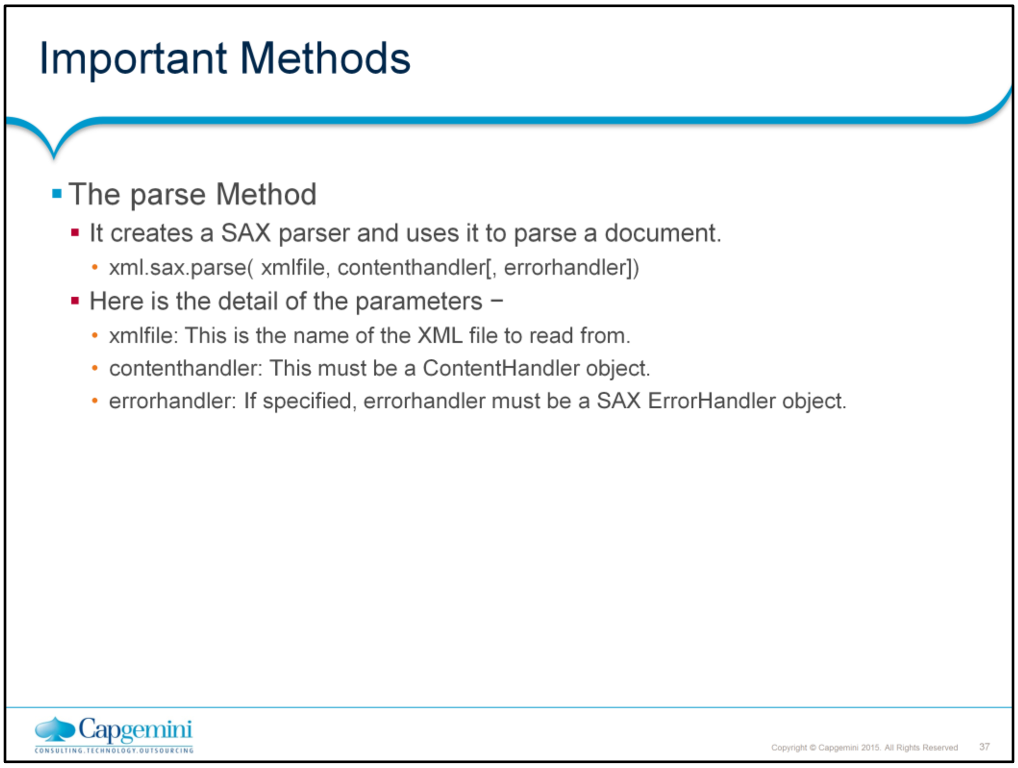
## Parsing XML with SAX APIs

- SAX is a standard interface for event-driven XML parsing. Parsing XML with SAX generally requires you to create your own ContentHandler by subclassing xml.sax.ContentHandler.
- Your ContentHandler handles the particular tags and attributes of your flavor(s) of XML. A ContentHandler object provides methods to handle various parsing events. Its owning parser calls ContentHandler methods as it parses the XML file.
- The methods startDocument and endDocument are called at the start and the end of the XML file. The method characters(text) is passed character data of the XML file via the parameter text.
- The ContentHandler is called at the start and end of each element. If the parser is not in namespace mode, the methods startElement(tag, attributes) andendElement(tag) are called; otherwise, the corresponding methods startElementNS and endElementNS are called. Here, tag is the element tag, and attributes is an Attributes object.



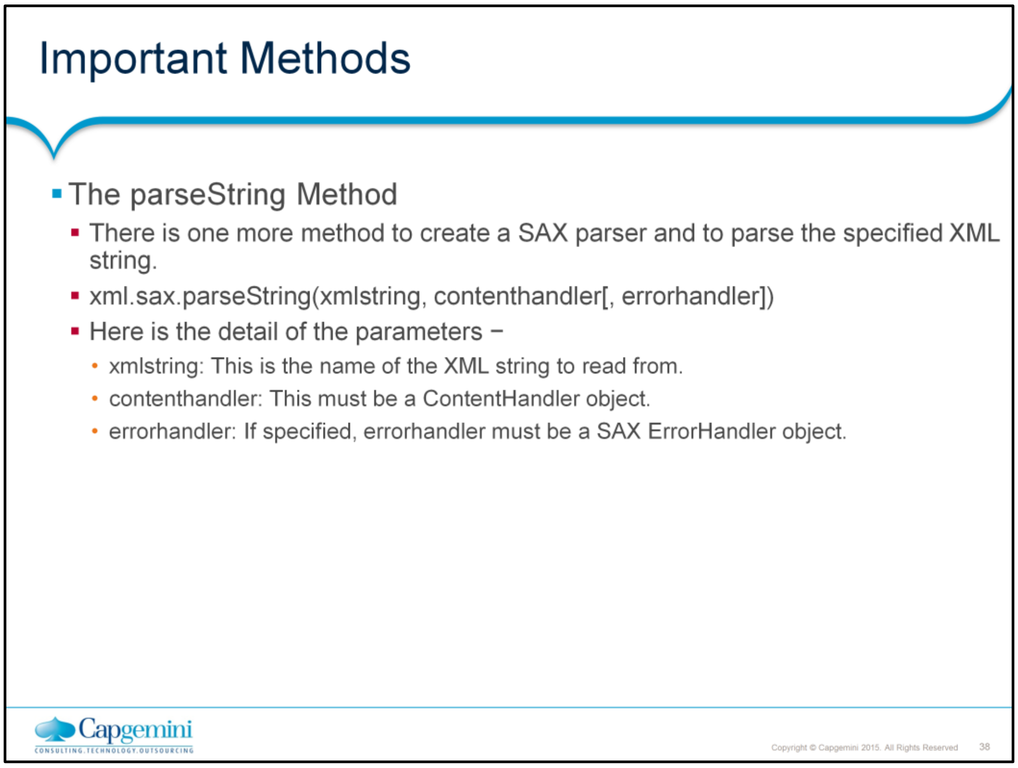
## Important Methods

- The make_parser Method
  - It creates a new parser object and returns it. The parser object created will be of the first parser type the system finds.
    - xml.sax.make_parser( [parser_list] )
    - parser_list: The optional argument consisting of a list of parsers to use which must all implement the make_parser method.



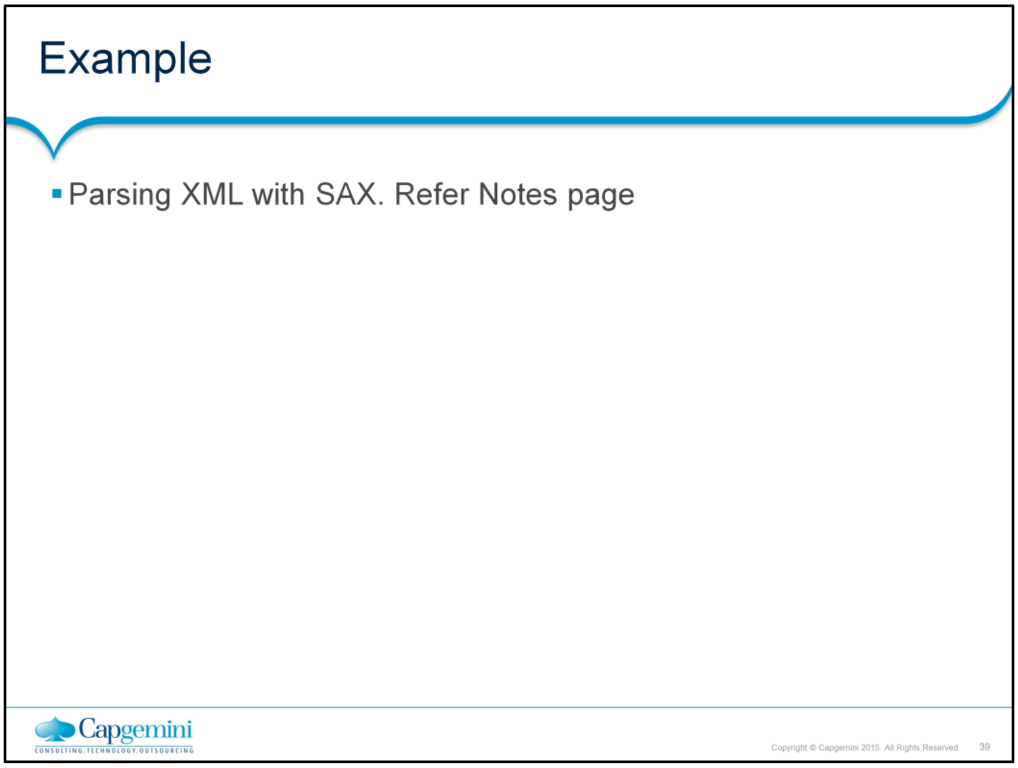
## Important Methods

- The parse Method
  - It creates a SAX parser and uses it to parse a document.
    - xml.sax.parse( xmlfile, contenthandler[, errorhandler])
  - Here is the detail of the parameters −
    - xmlfile: This is the name of the XML file to read from.
    - contenthandler: This must be a ContentHandler object.
    - errorhandler: If specified, errorhandler must be a SAX ErrorHandler object.



## Important Methods

- The parseString Method
  - There is one more method to create a SAX parser and to parse the specified XML string.
  - xml.sax.parseString(xmlstring, contenthandler[, errorhandler])
  - Here is the detail of the parameters −
    - xmlstring: This is the name of the XML string to read from.
    - contenthandler: This must be a ContentHandler object.
    - errorhandler: If specified, errorhandler must be a SAX ErrorHandler object.



## Example

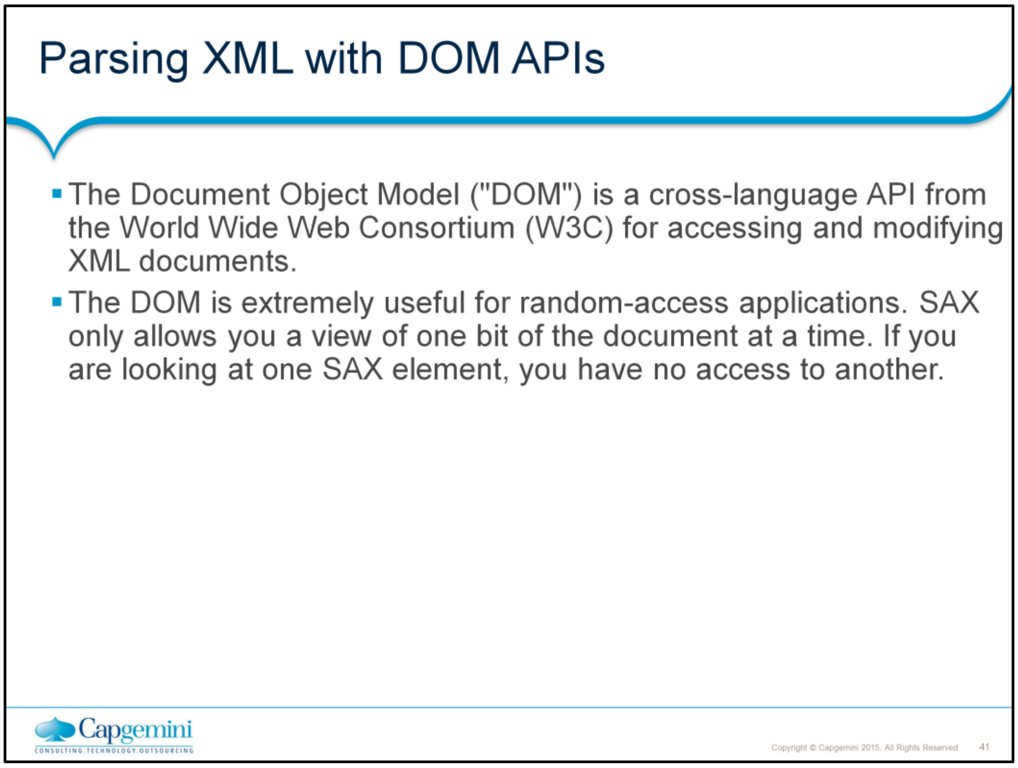
- Parsing XML with SAX. Refer Notes page

```
#!/usr/bin/python

import xml.sax

class MovieHandler( xml.sax.ContentHandler ):
   def __init__(self):
      self.CurrentData = ""
      self.type = ""
      self.format = ""
      self.year = ""
      self.rating = ""
      self.stars = ""
      self.description = ""

   # Call when an element starts
   def startElement(self, tag, attributes):
      self.CurrentData = tag
      if tag == "movie":
         print "*****Movie*****"
         title = attributes["title"]
         print "Title:", title

   # Call when an elements ends
   def endElement(self, tag):
      if self.CurrentData == "type":
         print "Type:", self.type
      elif self.CurrentData == "format":
         print "Format:", self.format
      elif self.CurrentData == "year":
         print "Year:", self.year
      elif self.CurrentData == "rating":
         print "Rating:", self.rating
      elif self.CurrentData == "stars":
         print "Stars:", self.stars
      elif self.CurrentData == "description":
         print "Description:", self.description
      self.CurrentData = ""
```

```
# Call when a character is read
   def characters(self, content):
      if self.CurrentData == "type":
         self.type = content
      elif self.CurrentData == "format":
         self.format = content
      elif self.CurrentData == "year":
         self.year = content
      elif self.CurrentData == "rating":
         self.rating = content
      elif self.CurrentData == "stars":
         self.stars = content
      elif self.CurrentData == "description":
         self.description = content

if ( __name__ == "__main__"):

   # create an XMLReader
   parser = xml.sax.make_parser()
   # turn off namepsaces
   parser.setFeature(xml.sax.handler.feature_namespaces, 0)

   # override the default ContextHandler
   Handler = MovieHandler()
   parser.setContentHandler( Handler )

   parser.parse("movies.xml")
```

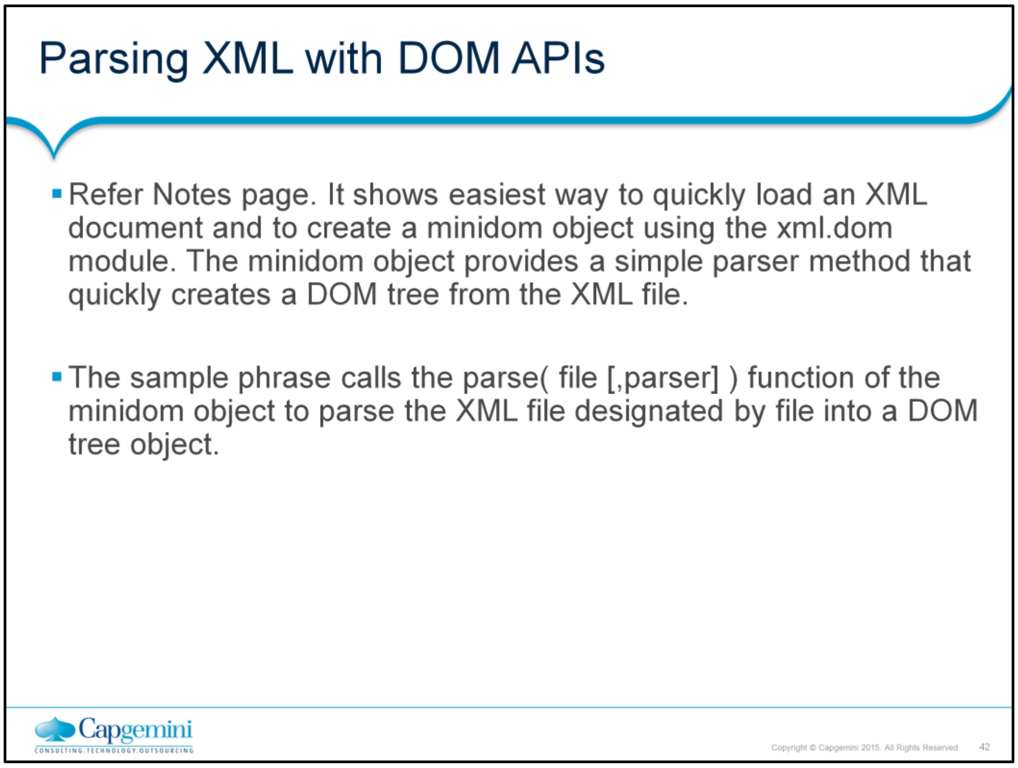
This would produce following result −

```
*****Movie*****
Title: Enemy Behind
Type: War, Thriller
Format: DVD
Year: 2003
Rating: PG
Stars: 10
Description: Talk about a US-Japan war
*****Movie*****
Title: Transformers
Type: Anime, Science Fiction
Format: DVD
Year: 1989
Rating: R
Stars: 8
Description: A schientific fiction
*****Movie*****
Title: Trigun
Type: Anime, Action
Format: DVD
Rating: PG
Stars: 10
Description: Vash the Stampede!
*****Movie*****
Title: Ishtar
Type: Comedy
Format: VHS
Rating: PG
Stars: 2
Description: Viewable boredom
```



## Parsing XML with DOM APIs

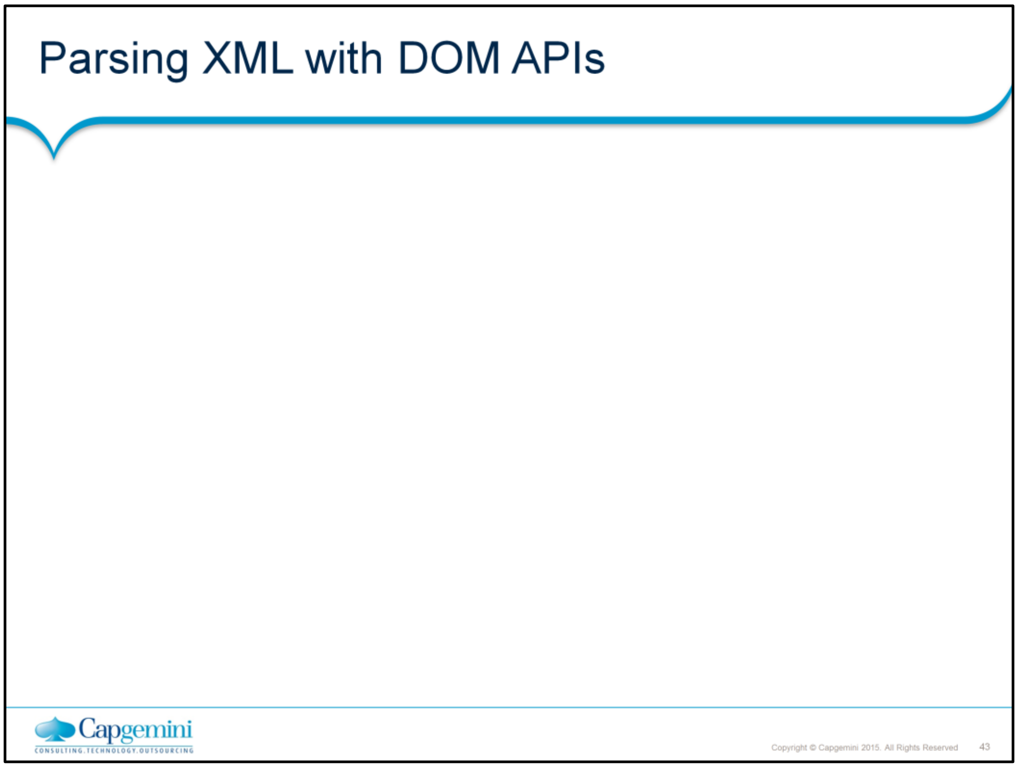
- The Document Object Model ("DOM") is a cross-language API from the World Wide Web Consortium (W3C) for accessing and modifying XML documents.
- The DOM is extremely useful for random-access applications. SAX only allows you a view of one bit of the document at a time. If you are looking at one SAX element, you have no access to another.



## Parsing XML with DOM APIs

- Refer Notes page. It shows easiest way to quickly load an XML document and to create a minidom object using the xml.dom module. The minidom object provides a simple parser method that quickly creates a DOM tree from the XML file.
- The sample phrase calls the parse( file [,parser] ) function of the minidom object to parse the XML file designated by file into a DOM tree object.

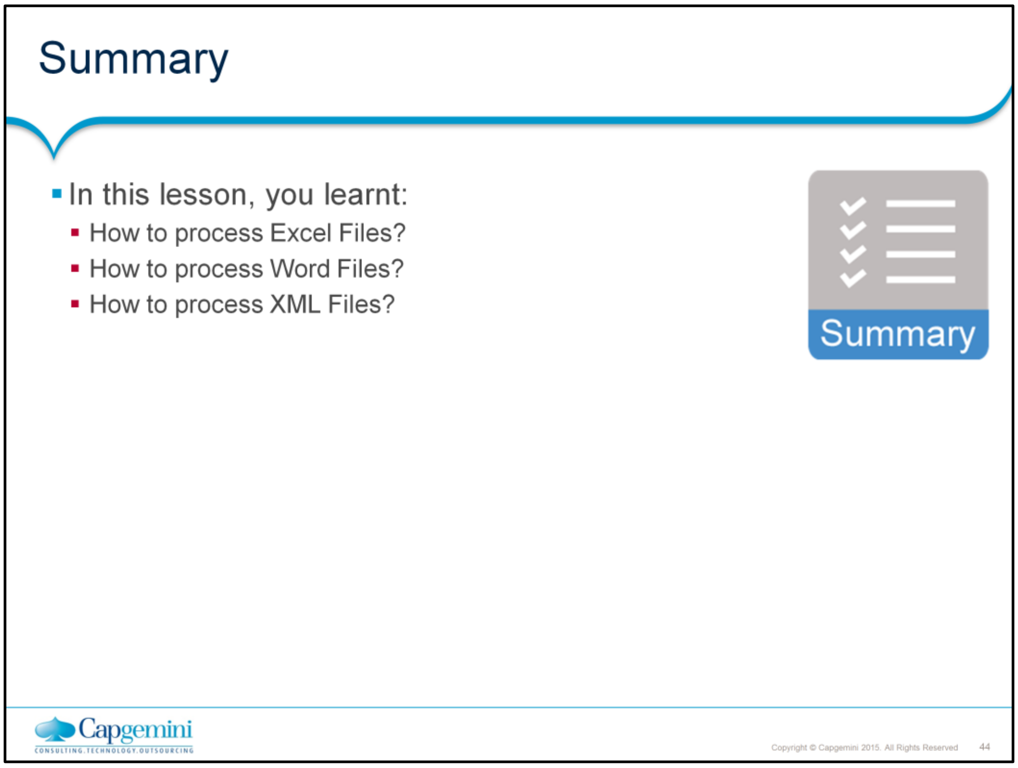
```
# Open XML document using minidom parser
DOMTree = xml.dom.minidom.parse("movies.xml")
collection = DOMTree.documentElement
if collection.hasAttribute("shelf"):
   print "Root element : %s" % collection.getAttribute("shelf")

# Get all the movies in the collection
movies = collection.getElementsByTagName("movie")

# Print detail of each movie.
for movie in movies:
   print "*****Movie*****"
   if movie.hasAttribute("title"):
      print "Title: %s" % movie.getAttribute("title")

   type = movie.getElementsByTagName('type')[0]
   print "Type: %s" % type.childNodes[0].data
   format = movie.getElementsByTagName('format')[0]
   print "Format: %s" % format.childNodes[0].data
   rating = movie.getElementsByTagName('rating')[0]
   print "Rating: %s" % rating.childNodes[0].data
   description = movie.getElementsByTagName('description')[0]
   print "Description: %s" % description.childNodes[0].data
```



## Parsing XML with DOM APIs

This would produce the following result −

```
Root element : New Arrivals
*****Movie*****
Title: Enemy Behind
Type: War, Thriller
Format: DVD
Rating: PG
Description: Talk about a US-Japan war
*****Movie*****
Title: Transformers
Type: Anime, Science Fiction
Format: DVD
Rating: R
Description: A schientific fiction
*****Movie*****
Title: Trigun
Type: Anime, Action
Format: DVD
Rating: PG
Description: Vash the Stampede!
*****Movie*****
Title: Ishtar
Type: Comedy
Format: VHS
Rating: PG
Description: Viewable boredom
```



## Summary

- In this lesson, you learnt:
  - How to process Excel Files?
  - How to process Word Files?
  - How to process XML Files?

Summary