IvyPdf

Version 1.61

IvyPdf helps you to extract valuable information from unstructured PDF documents in a quick and easy way. It can extract unlimited number of individual values and tables and provides powerful post-processing mechanism to further clean and format the data.

While PDFs are the main target of the library, it can be also used to parse Excel, Text, HTML and other file formats, thus allowing you to use a single tool for all your data processing needs.

How to use IvyPdf

IvyPdf can be used in a few different ways:

- In your .Net projects: Add reference to `IvyPdf.dll` and use `PdfParser` object to load and parse PDF files.

```
PdfParser p = new PdfParser(IvyDocumentReader.ReadPdf("mydoc.pdf"));
string text = p.Find("Revenue").Find("Total").Right().Text;
```

- As a stand-alone application: Use Ivy Template Editor to write extraction logic, organize it by templates, preview and validate the results. Use "Bulk File Processing" menu option or command-line utility (IvyTemplate.exe) to run the extraction templates on your files.

- Hybrid approach: Use Ivy Template Editor to write the extraction logic and store it as .tl file. Then load .tl file from your .Net code and run it on PDF files. See Using TemplateLib in .Net section below for more details.

Getting Started

Download IvyPdf.zip and unzip it to a folder of your choice. If you have a license key please start IvyTemplateEditor.exe, go to Help/About and enter the key there. It will register Ivy on your machine. If you don't have a license key, the 30 days trial will begin from the day of the first use.

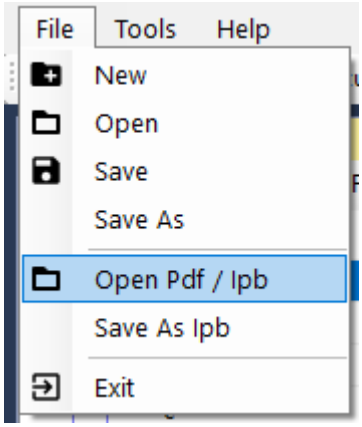
Check out Ivy examples (sample.tl and Disney.ipb) in the Samples folder.

Read Tutorial and Quick Hints sections below, or use API Reference to familiarize yourself with Ivy commands and syntax.

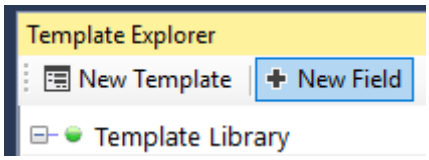
Ivy Template Editor

Even if you plan to use IvyPdf strictly from your program code, it's a good idea to use Ivy Template Editor to test the code and the extraction logic first.

1. Start IvyTemplateEditor.exe
2. Open a PDF document from the top menu, or drag-drop it on the editor pane



3. Highlight a template and create a new field



4. Go to the *Code* window and try a few commands. For example:

```
p.Reset()
.Find("Net Income")
.Right()
.Text
```

Click *Evaluate* to test your code.

Use *Toolbox* window to test different commands, for example `Find`, `Right`, `Down`. Try extracting tables with various parameters. The history of commands is shown there for your reference and can be copied to the code definition.

API Reference

PdfParser

PdfParser class is used for parsing token collection, extracted from PDF or IPB files.

Create parser

`PdfParser()` – default constructor.

`PdfParser(IvyDocument ivyDocument)` – create a parser from `IvyDocument` instance.

Open document

`PdfParser(IvyDocumentReader.ReadPdf(string filename))` – open PDF file.

`PdfParser(IvyDocumentReader.ReadIpb(string filename))` – open IPB file.

IPB files contain only text information from PDFs and therefore are smaller and faster to use. Consider converting PDF files to IPB if you are doing multiple extractions, working on extraction template and so on.

Documents can be open from memory:

`PdfParser(IvyDocumentReader.ReadPdf(byte[] fileContent))`

`PdfParser(IvyDocumentReader.ReadIpb(byte[] fileContent))`

Search text

`Find(params string[] text)` - search for next token containing text (partial match).

`Find(Regex regex)` - search for next token matching specified regular expression.

`Find(Predicate<Token> predicate)` - search for next token matching specified properties.

`FindPattern(string pattern)` - search for next token matching specified pattern. See Pattern Matching

You can search backwards:

`FindPrev(params string text)`

`FindPrev(Regex regex)`

`FindPrev(Predicate<Token> predicate)`

`FindPrevPattern(string pattern)`

Search by page number

`FindPage(int pageNumber)` - find first token on specified page.

Find a token relative to current

`Left(float offsetY = 0, float deviation = 0)` - find first token on the left of the current token. To be found the token should reside on the imaginary line that starts from the center of the current token. Optional `offsetY` parameter moves the line down (negative value moves up). Token will be captured if it's located `deviation` points away from the line.

`Down(float offsetX = 0, float deviation = 0)` - find first token located right below from the current token. To be found the token should reside on the imaginary line that starts from the center of the current token. Optional offsetX parameter moves the line right (negative value moves left). Token will be captured if it's located `deviation` points away from the line.

`Right(float offsetY = 0, float deviation = 0)` - find first token on the right of the current token.

`Up(float offsetX = 0, float deviation = 0)` - find first token located right above the current token

Find first token in line above or below the current

`Above()` - find first (left-most) token located in the line above the current token.

`Below()` - find first (left-most) token located in the line below the current token.

Define search region

All search is done within a "Region". Initial region is the whole document.

Here are methods to *FILTER* a region:

`FilterWindow(float x1, float y1, float x2, float y2)` - set region to the rectangular defined by X,Y coordinates on the current token page. Only tokens that completely fit into the window are included.

`FilterCWindow(float x1, float y1, float x2, float y2)` - set region to the rectangular defined by X,Y coordinates on the current token page. Tokens that partially overlap are included too (cross-window tool).

`FilterOffset(float left, float up, float right, float down)` - set region to the window located relative to top-left corner of the current token (within the same page).

`FilterCOffset(float left, float up, float right, float down)` - set region to the window located relative to top-left corner of the current token (within the same page). Tokens that partially overlap are included too.

`FilterIndex(int fromIndex, int toIndex)` - set region to range of tokens by their index.

`FilterCurrentPage()` - set region to the page of current token.

`FilterPage(int fromPage, int toPage = -1)` – set region to page range.

`FilterText(params string[] text)` – include only tokens that contain specified text.

`FilterTextPattern(string pattern)` - include only tokens matching specified pattern. See Pattern Matching

`FilterRegex(Regex regex)` – include only tokens that match provided regular expression.

`Filter(Predicate<Token> predicate)` – region will include only tokens matching the predicate conditions.

`FilterEnclosedArea()` - set region to an area enclosed by `Line` objects around the current token.

`FilterSections(Section section)` - filter to specific section.

`FilterSections(IEnumerable<Section> sections)` - filter to a list of sections.

`FilterSections(Predicate<Section> sectionPredicate)` - filter to sections chosen based on a predicate condition.

`FilterSections()` - filter to all sections in the `Sections` collection.

`FilterClear()` – remove filter. Region will be set to the whole document.

`Reset()` – remove filter and move to the first token in the document.

`Reset(string sequenceName)` - optionally set `SequenceName` that can be used by Exception handler to log errors.

Parser properties and methods

`IvyDocument IvyDocument` – reference to underline document used by Parser.

`IvyDocument` properties are also referenced by Parser object:

- `List Tokens` – collection of all tokens (including filtered)
- `List Lines` – collection of line objects
- `double[] PageSizesX` – contains width of each page
- `double[] PageSizesY` – contains height of each page
- `void SaveIpb(string filename)` – save current document as IPB file for quick loading later
- `IvyDocument LoadIpb(string filename)` – load from IPB file

`string SequenceName` – current search Sequence, set by a call to Reset(string sequenceName)

`List<string> ParsingHistory` - history of commands since the last Reset

`Token Token` - current token (set by one of Find methods or Filter)

`double GetPageWidth(int page)` – returns page width

`double GetPageHeight(int page)` – returns page height

`int GetPageCount()` – returns number of pages in the document

`string Version` – returns version of the IvyPdf library

`Clone()` – returns an exact copy of the PdfParser object that can be used for sub-searches.

`Subset(int StartIndex, int EndIndex)` - returns a copy of PdfParser object, including only tokens in the provided range (by token index).

`MergePages(int startPage, int endPage)` - merge document pages together, based on the provided range.

`MergePages()` - merge all document pages into one long page.

`RegionContainsLines(int pageNumber, float x1, float y1, float x2, float y2, LineFType lineFType)` - check whether the specified region contains any line objects. `lineFType` can have the following values: `All`, `Horizontal`, `Vertical`, `Slanted`

The following properties are referencing the current Token and can be used as shortcuts (for example, you can use `Text` instead of `Token.Text`):

`string Text` - token text

`string Font` - name of the font used to print this token in the PDF document

`bool Bold` - font bold flag

`bool Italic` - font italic flag

`int Page` - token page number

`float Width` - width of token bounding box

`float Height` - height of token bounding box (essentially font height)

`float X` - coordinate of top-level corner of token bounding box

`float Y` - coordinate of top-level corner of token bounding box

`IvyColor Color` - text color, represented as (R, G, B) tuple

`bool Black` - flag, indicating that token `Color` is black

`bool White` - flag, indicating that token `Color` is white

`bool Grey` - flag, indicating that token `Color` is grey (R = G = B)

`int Index` - index of the token in the `Tokens` collection

`string DataType` - data type of the token, guessed from text. Can be "String", "Number" or "DateTime".

`double? ToNumber()` – auto-convert current token text to Double

`DateTime? ToDate()` – auto-convert current token text to DateTime

`Token Next()` - reference to the next token (by Index)

`Token Prev()` - reference to the previous token (by Index)

Text extraction

`string ExtractText(TextPosition textPosition, bool removeBlankLines, bool trimSpaces, double linePixelDeviation)`

Parameters:

- Text Position:

    - `GeometricCompact` - uses token coordinates to combine them into text. Ignores space between tokens.
    - `GeometricSpaced` - uses token coordinates to combine them into text, adding spaces between tokens according to their position. (Default option)
    - `TokenOrder` - uses order of tokens to prepare text.
- `removeBlankLines` - when set the text won't have any empty lines. Default value = true
- `trimSpaces` - will trim spaces around text. Default value = true
- `linePixelDeviation` - allowed vertical deviation of tokens to belong to the same line. Default value = 5.0

`string ExtractText()` - extract text using default parameters.

Bookmarks

Bookmarks can be used for repetitive tasks. For example find a token “X”, filter and search in the filtered region, then come back to the token “X” and search again.

`SetBookmark(string name)` – set a bookmark with specified name

`GoBookmark(string name)` – set current Token to the bookmark

`DeleteBookmark(string name)` – delete a bookmark

`DeleteAllBookmarks()` – delete all bookmarks

Conditions and Loops

You can use `If` method to add conditional logic, checking for specific condition or successful code execution. You can conditionally execute an action or return a value.

`PdfParser If(condition, thenAction, [elseAction])`

`PdfParser If(action, thenAction, [elseAction])`

`dynamic If(condition, thenValue, elseValue)`

`dynamic If(action, thenValue, elseValue)`

Examples:

Go up or down, depending on a condition:

```
p.If(myvariable == 42, x=>x.Down(), x=>x.Up());
```

Version without “else”:

```
p.If(myvariable == 42, x=>x.Down());
```

Check if a word “test” exists, then search something else:

```
p.If(x=>x.Find(“test”), x=>x.Find(“word1”), x=>x.Find(“word2”));
```

If a word “test” exists return a string:

```
p.If(x=>x.Find(“test”), “Found”, “Not Found”);
```

Return a string based on condition:

`p.If(myVariable == true, “Yes”, “No”);`

In a similar way you can use `While` loop to test for a condition, or run a code until it succeeds:

`While(condition, action)`

`While(testAction)`

`While(testAction, action)`

Move down until a bold token is found:

`p.While(x => !x.Bold, x => x.Down());`

Find right-most token starting from your position:

`p.While(x => x.Right());`

Count number of occurrences of word “test”:

`int counter=0; p.While(x=>x.Find(*"test"*), ()=>counter++);`

You can test for successful code execution using `Try` method:

`bool Try(action)`

Count number of occurrences of word “test”:

`while(p.Try(x=>x.Find(“test”))) counter++;`

Table extraction

`DataTable Table(PdfTableOptions tableOptions)` - Use for tables with column headers. To start table extraction the current token should be a header token of one of the columns.

`DataTable Grid(PdfTableOptions tableOptions)` - Use for tables that don't have a header. Returns a table with generic columns (Field0, Field1, Field2...). Current token should be in the first row.

Parameters:

- `WhiteSpaceLimit` – amount of white space that is used to determine the end of table, as ratio of table row height. (Default value is 2.5)
- `MaxRowHeight` – determine end of table using absolute distance between rows. (Default value 0)
- `MultiPage` – attempt to find the table on the next page(s). Table should have header row on every page. (Default true)
- `IncludeUnmatchedCells` – extra cells that do not match to header will be added into a new column. (Default true)
- `ColumnBorders` – location of every column that will be used to position tokens into columns. Starts with left border, up to right border (should have [number of columns] + 1). (Default null)
- `HeaderSeparatedByLine` – use lines in the table header to determine column border locations. (Default false)

- `ColumnsSeparatedByLines` – use lines in the table body to assign tokens to columns. (Default false)
- `RowsSeparatedByLines` – use lines in the table body to assign tokens to rows. (Default false)
- `TableCellType` – defines whether returned DataTable contains Token objects or String objects. Possible values are `Token`, `String` or `ParserDefault`.

  `ParserDefault` is using global value in `PdfParser.Options.TableCellType`.

In Ivy Template Editor the settings can be defined on the template level, by adding this code to Template Settings:

```
protected override void Init() { p.Options.TableCellType = TableCellType.Token; }
```

Using `Token` objects allows you to get location information from PDFs, but makes coding more complicated. All `Token` properties will be included in JSON or XML output in Ivy Template Editor.

Sections

Using `Sections` feature you can split the document into logical parts, making data extraction more reliable. For example, you can specify that you want to parse only specific sections or subsections, or loop through sections of your choice and extract some data from each one.

`Sections` collection is a `List` of `Section` objects. Each `Section` has StartIndex and EndIndex, which refer to the indexes of tokens that belong to that section. Initially `Sections` collection is empty. You need to run `Split` method to build the collection (based on the rules that you provide).

Syntax of `Split` function:

`Split(Predicate<Token> startSectionPredicate)` - split on every occurrence of condition.

`Split(Predicate<Token> startSectionPredicate, Predicate<Token> endSectionPredicate)` - create sections between tokens that suffice start/end condition.

`Split(Func<Token, double> ScoringFunction)` - split the document based on token score. Sections will be split iteratively, by tokens having MAX score.

`Split()` - split based on default scoring function.

> Split functions can be chained. In this case every section will be split into subsections.

Examples:

- Create sections based on font size, with larger font being a parent section and smaller font being a subsection.

  `Split()`

  or

```
Split(x => {
    double score = Math.Round(x.Height); // use text height as the main
scoring point
    if (x.Height < 8) return 0;          // ignore small text
    if (x.Bold) score++;                 // add one point for bold
    if (x.Y - x.Prev().Y > 40) score++;  // add one point for tokens that
have 40px of white space before them
    return score;
})");
```

- Split documents on every occurrence of a word.

  `Split(x=>x.Text == "Chapter")`
- Split on token of specific font and location. `Split(x=>x.Height==14 && x.Y < 100)`
- Split by substantial amount of whitespace. `Split(x=>x.Y - x.Prev().Y > 50)`
- Split by chapters first, then by font size `Split(x=>x.Text == "Chapter").Split()`

> Note: You can see all current sections in the Ivy Template Editor using "View Sections" button in the Toolbox.

Once sections are created you can use them to filter the document:

`FilterSections(Section section)` - filter to specific section.

`FilterSections(IEnumerable<Section> sections)` - filter to a list of sections.

`FilterSections(Predicate<Section> sectionPredicate)` - filter to sections chosen based on a predicate condition.

`FilterSections()` - filter to all sections in the `Sections` collection.

Examples:

```
p.FilterSections(Sections[0]);
p.FilterSections(Sections.Where(x=>x.Name.Contains("Chapter")));
p.FilterSections(Sections[0].Children);
p.FilterSections(x=>x.Name.Contains("Chapter"));
```

`SectionsClear()` - remove `Sections`

Section` object contains the following properties and methods:

`int ID` - section number

`int ParentID` - parent section number

`Token Token` - first token in the section. All `Token` properties are also all available on the section level. So instead of `Section.Token.Text` you can use `Section.Text` and so on.

`int StartIndex` - index of the first token in the section

`int EndIndex` - index of the last token in the section

`int Level` - section level. Top level is 1, sublevels are 2, 3, 4...

`Section Next` - next section

`Section Prev` - previous section

`Section Parent` - parent section

`List<Section> Children` - subsections of the current section

`List<Section> Siblings` - all sections that belong to the same parent (including current section)

`Filter()` - filter the PdfParser to the current section

`Parse()` - return a new PdfParser object (full copy) that contains only objects in the current section

DataSetParser

To parse Excel, CSV and other structured formats you can use DataSetParser class.

Create parser

`DataSetParser(DataSet dataSet)` – create from existing `DataSet`

`DataSetParser(DataTable dataTable)` – create from existing `DataTable`

Open document

`DataSetParser(DataSetReader.ReadExcel(string filename))` – open xls, xlsx, xlsm or csv file.

`DataSetParser(DataSetReader.ReadExcel(Stream stream))` - open from a stream.

Search text

`Find(params string[] text)` - search for next cell containing text (partial match)

`Find(Regex regex)` - search for next cell matching specified pattern.

`FindPattern(string pattern)` - search for next token matching specified pattern. See Pattern Matching

You can search backwards:

`FindPrev(params string text)`

`FindPrev(Regex regex)`

`FindPrevPattern(string pattern)` - search for next token matching specified pattern. See Pattern Matching

Search sheet (tab) by number or name

`FindSheet(int sheet)` – move to first cell on the specified sheet

`FindSheet(string sheetName)` – find a sheet where name contains provided text

`FindSheetPattern(string pattern)` - find a sheet where name matches specified pattern. See Pattern Matching

Find a cell relative to current

`Left(int steps = 0)` – move current position to the left. If non-zero number is specified then move exactly that numbers of cells. Otherwise, move until non-empty cell is found.

`Right(int steps = 0)`

`Up(int steps = 0)`

`Down(int steps = 0)`

Find first non-empty cell in line above or below the current

`Above()` - finds left-most non-empty cell in the line above.

`Below()` - finds left-most non-empty cell in the line below.

Select table area

`DataTable Table(int headerRows = 1)` – auto-grow table from current position, in left, right and down directions until empty columns/rows encountered. The top *headerRows* rows will be used as a header (default = 1)

`DataTable Table(bool left, bool up, bool right, bool down, int emptyColumnLimit, int emptyRowLimit, int headerRows = 1)` – auto-grow in specific directions only, allow limited number of empty columns/rows on the way.

`DataTable Table(int left, int top, int width, int height, int headerRows = 1)` – select area relative to current position.

`DataTable Table(int width, int height, int headerRows = 1)` – select area starting from current position.

`DataTable Grid()` – auto-grow table from current position, in left, right and down directions until empty columns/rows encountered. The header will be Field1, Field2, ...

`DataTable Grid(bool left, bool up, bool right, bool down, int emptyColumnLimit, int emptyRowLimit)` – auto-grow in specific directions only, allow limited number of empty columns/rows on the way.

`DataTable Grid(int left, int top, int width, int height)` – select area relative to current position.

`DataTable Grid(int width, int height)` – select area starting from current position.

DataSetParser properties

`DataSet DataSet` – reference to underline `DataSet` object

`int Sheet` – current sheet number (zero-based)

`string SheetName` – current sheet name

`int X` – current column

`int Y` – current row

`string Text` – text value of the current cell

IvyDocument

Ivy converts PDF files to collection of `Tokens` and `Lines`. PdfParser class can be used to search this collection and extract useful information. In addition the collection can be stored as IPB file - a special format that can be loaded much quicker than PDF.

Properties

`List<Token> Tokens` - collection of tokens extracted from PDF. (Tokens are text objects with location and size information.)

`List<LineF> Lines` - collection of lines extracted from PDF.

`double[] PageSizesX` - pages width

`double[] PageSizesY` - pages height

Methods

`LoadIpb(string filename) - read IPB file`

`LoadIpb(byte[] fileContents)` - read IPB file from in-memory byte array

`SaveIpb(string filename)` - save to IPB file

`PerformTokenLayout(PdfReadOptions pdfReadOptions)` - after tokens are loaded from PDF file Ivy performs some additional steps to make data easier to use. The following logic is applied:

- Long tokens with whitespace characters are being split. For example token "A.....B" would be split into tokens "A" and "B".
- Adjacent tokens with same font height are merged (combined) together.
- Pdf documents often have tokens with no text, used as a whitespace. By default Ivy uses these tokens to properly position the tokens that contain actual text. However, in some cases it may be beneficial to ignore them completely. `TokenLayoutType` parameter specifies the required behavior:
  - `UseWhitespaceTokens` - whitespace tokens are being used (default)
  - `IgnoreWhitespaceTokens` - whitespace tokens are not used
  - `Follow Pdf Sequence` - only tokens from same PDF print sequence (BT/ET) are merged
  - `None` - no layout logic applied
- Tokens that are intersected by lines are split into two (if `SplitTokenByLines` flag is set)

You have an option to ignore default Ivy layout logic and apply your own instead. You can write your own logic completely, or you can use the following pre-defined methods:

`CombineTokens(Comparison<Token> predicate)` - combine tokens based on predicate logic

`CombineTokens(Comparison<Token> predicate, string character)` - combine tokens, adding a character in between (e.g. space)

`SplitTokensByCharSequence(char[] chars, int minLength)` - split tokens on provided characters, making sure the resulting tokens are not small than `minLength`

`Append(IvyDocument anotherIvyDocument)` - append another `IvyDocument` at the end of the current one. Can be used to combine multiple PDF files together.

IvyDocumentReader

This class is used to read PDF documents and convert them to token collection representation.

`IvyDocument ReadPdf(string filename)` - read PDF document from file.

`IvyDocument ReadPdf(byte[] fileContents)` - read PDF document from memory.

`IvyDocument ReadIpb(string filename)` - read IPB document from file.

`IvyDocument ReadIpb(byte[] fileContents)` - read IPB document from memory.

Open document with specified reading and post-processing logic

`IvyDocument ReadPdf(string filename, PdfReadOptions pdfReadOptions)`

`IvyDocument ReadPdf(byte[] fileContent, PdfReadOptions pdfReadOptions)`

Optional parameter `pdfReadOptions` specifies layout logic applied to token collection and reading options. Available parameters are

- TokenLayoutType - specifies layout logic applied to tokens after the document is read
  - `UseWhitespaceTokens` - whitespace tokens are being used (default)
  - `IgnoreWhitespaceTokens` - whitespace tokens are not used
  - `Follow Pdf Sequence` - only tokens from same PDF print sequence (BT/ET) are merged
  - `None` - no layout logic applied
- ReadLines - read graphical objects from PDF. Default: true
- ReadCurves - read curve objects (ReadLines should be set too). Default: true
- ReadAnnotations - read annotations objects (PDF forms, and other content). Default: true
- IgnoreWhiteLines - ignore lines/curves of white color, as they are usually invisible. Default: false
- RemoveShadows - tries to remove shadows (duplicate tokens printed on top). Default: true
- SplitTokensByLines - split text tokens at the points they are intersected by lines. Default: false

In Ivy Template Editor you can use "File\Options" form to specify PDF reading options. These options will be applied the next time you open a document.

In addition you can specify these options on the Template object. When you specify a template to be used for extraction, the corresponding `pdfReadOptions` parameters are being use.

DataSetReader

This class is used to read Excel and CSV documents into a DataSet object.

`DataSet ReadExcel(string filename)` - read Excel or CSV document.

`DataSet ReadExcel(Stream stream)` - read from a stream, automatically recognizing file format.

DataTable extensions

Ivy Library includes many extension methods that can be used to join and filter `DataTable` objects to get the data you need.

The methods below return `DataTable` object. We will skip the return data type for readability:

Filter columns

- `SelectColumns(bool allowMissingColumns, params int[] indexes)` - select subset of columns by index.

  `SelectColumns(bool allowMissingColumns, params string[] names)` - select subset of columns by name.
- `SelectColumns(Predicate ColumnPredicate)` - select subset of columns by predicate condition.
- `SelectColumnsPattern(params string[] patterns)` - select subset of columns matching specified patterns. See Pattern Matching
- `SelectColumnsAs(bool allowMissingColumns, bool addMissingColumnAsEmpty, params string[] names)` - select subset of columns by name and rename to provided new names.
- `SelectColumnsAsPattern(bool allowMissingColumns, bool addMissingColumnAsEmpty, params string[] names)` - select subset of columns by patterns and rename to provided new names. See Pattern Matching

`allowMissingColumns` - optional parameter (default = false). If not set and the table doesn't contain specified column the exception is thrown.

`addMissingColumnAsEmpty` - optional parameter (default = false). If set and the table doesn't contain specified column, the column is added as empty. Otherwise this column is ignored.

- `DeleteColumns(bool allowMissingColumns, params int[] indexes)` - remove columns by index.
- `DeleteColumns(bool allowMissingColumns, params string[] names)` - remove columns by name.
- `DeleteColumns(Predicate columnPredicate)` - remove columns by condition.

- `DeleteColumnsPattern(bool allowMissingColumns, params string[] patterns)` - remove columns matching specified patterns. See Pattern Matching
- `NameColumns(params string[] names)` - rename columns, providing the names in order.
- `AddColumn(string name, dynamic value = null, Type columnType)` - add new column with specified name, default value and data type.

Example:

```
  mytable.SelectColumns(0,2,3)   //select first, third and fourth columns
  mytable.SelectColumns("name","price","quantity")  // select by name
  mytable.SelectColumns(x=>x.ColumnHeader.Contains("Amount"))  //select by
condition

  mytable.DeleteColumns(0,2,3)   //delete first, third and fourth columns
  mytable.DeleteColumns("name","price","quantity")  // delete by name
  mytable.DeleteColumns(x=>ColumnHeader.Contains("Amount"))  //delete by
condition

  mytable.AddColumn("Amount", "0.00", typeof(string))
  mytable.NameColumns("Price", "Amount", "Total")


  //Let's assume mytable contains columns "Field1", "Field2", "Field3"
  mytable.SelectColumns("Field1", "Field4") // throws exception "Cannot find
Field4"
  mytable.SelectColumns(true, "Field1", "Field4") // returns Field1

  mytable.SelectColumnsAs("Field1", "Col1", "Field4", "Col2") // throws
exception
  mytable.SelectColumnsAs(true, "Field1", "Col1", "Field4", "Col2") // returns
Field1 as Col1
  mytable.SelectColumnsAs(true, true, "Field1", "Col1", "Field4", "Col2") //
returns Field1 as Col1 and empty column Col2
```

Filter rows

- `SelectRows(string RowFilter)` - select subset of rows by row filter.
- `SelectRows(Predicate RowPredicate)` - select subset of rows by predicate condition.
- `SelectRowsContainingText(params string[] text)` - select rows that contain specified text in one of the columns.

  `SelectRowsContainingTextPattern(string pattern)` - select rows that contain text matching specified pattern. See Pattern Matching

- `DeleteRows(Predicate rowPredicate)` - remove rows by predicate condition
- `DeleteRowsRange(Predicate fromRowPredicate, Predicate toRowPredicate)` - remove rows between those found by predicate condition.
- `DeleteRowsContainingText(params string[] text)` - delete rows that contain specified text in one of the columns.

- `DeleteRowsContainingTextPattern(string pattern)` - delete rows that contain text matching specified pattern. See Pattern Matching

Example:

```
mytable.SelectRows("amount > 0")
mytable.SelectRows(x=>x[0].ToString().Length > 0) //rows where first column is
not empty
mytable.SelectRows(x=>x[0].ToString().Contains("Total")) //rows where first
column contains some text

mytable.DeleteRowsContainingText("Total", "total") //delete by specific text
mytable.DeleteRows(x=>x[0].ToString() == "") //rows where first column is empty
mytable.DeleteRowsRange(x=>x[0].ToString().Contains("start"),
     x=>x[0].ToString().Contains("end")) //delete rows between text "start" and
"end"
mytable.Rows.RemoveAt(0) //delete the first row
```

Relational operations

- `Join(DataTable Second, string FJC, string SJC)` - join two tables based on specified columns (inner join).
- `DataTable LeftJoin(DataTable Second, string FJC, string SJC)` - left outer join on specified columns.
- `DataTable FullOuterJoin(DataTable Second, string FJC, string SJC)` - full outer join on specified columns.
- `Union(DataTable Second)` - union (append) two tables together.
- `Union(DataTable Second, bool UseColumnNames)` - union two tables, but use column names to match them, even if columns are in different order.

  Example:

  ```
  mytable.Join(mytable2, "name", "name") //inner join
  mytable.LeftJoin(mytable2, "item", "product") //left join
  mytable.Union(mytable2) //union in column order
  mytable.Union(mytable2, true) //union using column names
  ```

Group by, rollup, transpose and reverse

- `Distinct(string Column)` - return unique values from specified column.

  Example: `myTable.Distinct("name")`
- `GroupBy(DataColumn[] Grouping, string[] AggregateExpressions, string[] ExpressionNames, Type[] Types)` - group by specified columns
- `Rollup(string nonEmptyColumn)` - if specified column has empty value in any row the data in this row is "rolled" (concatenated) into previous row. This is useful for cases when table has a column with a long text that extends to second row.

- `Rollup(bool fromTopToBottom, params int[] nonEmptyColumnIndexes)` - rollup values with the row above or below the empty one, based on `fromTopToBottom` parameter.
- `Rollup(float minRolledUpLength, int nonEmptyColumnIndex )` - rollup only if total text length in the concatenated column is over `minRolledUpLength`
- `Transpose(this DataTable dt)` - transpose table using first column values are column headers
- `Reverse(this DataTable dt)` - reverse the table row order
- `TableFromArray(object[,] DataArray)` - create table object from a data array

Formatting and cleanup

- `ToNumber(params string[] names)` - convert text in the specified columns to the numeric representation, removing currency symbols, percentages, etc. Non-numeric values are changed to empty string.

  This function uses global `IvyOptions.ToNumberCultureInfo` settings for number parsing.
- `ToNumber(params int[] indexes)` - convert text in the specified columns to the numeric representation.
- `ToDate(params string[] names)` - convert text in the specified columns to dates, removing dates that cannot be recognized.

  This function uses global `IvyOptions.ToDateFormatMonthFirst` settings for date parsing.
- `ToDate(params int[] indexes)` - convert text in the specified columns to dates.
- `TokenToString(this DataTable dt)` - converts all `Token` objects in the table to their string representation.

  Example:

```
mytable.ToNumber(0,1,4) //format text in the specified columns as numbers
mytable.ToDate("maturity", "current date") //format text as date
```

Update values

- `Update(Action action)` - update table values according to `action` logic.
- `Update(Action action, Predicate where)` - update table values according to `action` logic and `where` condition.
- `ReplaceText(int columnIndex, string find, string replaceTo)` - search and replace text in the specified column.
- `ReplaceText(string find, string replaceTo)` - search and replace text across the whole table.

  Example:

```
mytable.Update(x=>x[2]=x[0].ToString() + x[1].ToSTring()) //concatenate first
and second    columns and store the result in the third column
mytable.Update(x=>x[1]="", x=>x[0].ToString().Contains("Total")) //remove text
from second column when first column contains specific word
mytable.ReplaceText("Total", "") //replace text in all table cells
```

Various functions

- `double GetSum(string colNameToSum, string whereClause)` - get total value in one column.
- `bool HasText(string text)` - check whether the table contains specified text.
- `bool ContainsColumns(params string[] names)` - check whether table header contains all provided columns.
- `bool ContainsColumnsPattern(params string[] patterns)` - check whether table header contains all columns matching provided patterns. See Pattern Matching
- `bool HeaderMatches(params string[] names)` - check whether table header matches provided list (all columns exist in the same order)
- `bool HeaderMatchesPattern(params string[] patterns)` - check whether table header matches provided patterns. See Pattern Matching
- `bool HasEmptyRows()` - checks whether the table has empty rows
- `bool HasEmptyColumns()` - checks whether the table has empty columns
- `DataSetParser Parse()` – returns `DataSetParser` that can be used to further parse the data

  Example:

```
bool correctTable = mytable.HeaderMatches("name", "quantity", "amount");
bool empty = mytable.HasEmptyColumns();

double sum = mytable.GetSum("amount","quantity>0");

//using DataSetParser
double total = mytable.Parse().Find("Total").Right().Text.ToNumber();
```

String extensions

- `double? ToNumber()` - convert string to number.

  This function uses global `IvyOptions.ToNumberCultureInfo` settings for number parsing.
- `DateTime? ToDate()` - convert string to DateTime.

  This function uses global `IvyOptions.ToDateFormatMonthFirst` settings for date parsing.
- `string TextBetween(string matchFrom, string matchTo)` - returns text between two strings.
- `string[] TextsBetween(string matchFrom, string matchTo, bool includeTags = false)` - returns all occurrences of text between two strings. Both beginning and ending strings should exist in the text and not overlap.
- `bool MatchesPattern(string pattern)` - checks whether string matches provided pattern. See Pattern Matching

  Example:

```
double value = "150%".ToNumber();
DateTime date = "On the July 1, 2015".ToDate();

string text = "this is text".TextBetween("this", "text");
string[] texts = "<td>value1</td><td>value2</td>".TextsBetween("<td>","
</td>");
```

Global options

`IvyOptions` class can be used to specify some global options. It has the following properties:

- `TableCellType TableCellType` – specifies default value for `PdfParser.Options.TableCellType`
- `CultureInfo ToNumberCultureInfo` – specifies culture settings used by `ToNumber()` function.
- `bool ToDateFormatMonthFirst` – used by `ToDate()` function.

`CultureInfo` and `ToDateFormatMonthFirst` are set according to local machine settings by default.

In Ivy Template Editor the settings can be changed on the template level, by adding this code to Template Settings:

```
protected override void Init()
{
    IvyOptions.TableCellType = TableCellType.String;
    IvyOptions.ToNumberCultureInfo = new CultureInfo("fr-FR");
    IvyOptions.ToDateFormatMonthFirst = true;
}
```

Reserved words

Ivy templates have the following pre-defined objects (please refer to definition of `TemplateBase` class, which can be found in Template Editor: Template Library \ Custom Code Modules collection):

`p` – `PdfParser` object that is loaded from PDF or IPB files.

`d` – `DataSetParser` object loaded from xls, xlsx, xlsm or csv files.

`s` – `string` object loaded from txt, htm, html, xml or json files.

In addition, there are the following public fields:

`filename` – contains the full path of the loaded file.

`args[]` – additional parameters that can be provided via command-line.

`_Init()` method initializes the objects above.

`Init()` method is meant to be overridden by code in templates, providing custom initialization when needed.

Command-line parameters

Ivy Template

-e Extract data and save into Excel, Json or XML:

`IvyTemplate.exe -e InputFile OutputFile TemplateLibrary TemplateName|Auto [Parameters]`

For the Auto-template selection the template should have a field called `AutoTemplateSelectionCriteria`, containing logic that returns "true" if the template should be selected.

`OutputFile` should have extension .xlsx, .json or .xml to determine output format

-v Validate extraction – run the template, but do not save the results:

`IvyTemplate.exe -v InputFile TemplateLibrary TemplateName [Parameters]`

-i Convert a PDF file to IPB:

`IvyTemplate.exe -i InputFile OutputFile`

Ivy Template Editor

-o Open for preview in GUI mode:

`IvyTemplateEditor.exe -o InputFile TemplateLibrary`

Returned %%ERRORLEVEL%% values:

0 - success

1 - error

2 - template validation failed

Examples

PdfParser

- Load PDF file:

```
PdfParser p = new PdfParser(IvyDocumentReader.ReadPdf("mydoc.pdf"));
```

- Get text from section "Revenue", on the right of word "Total":

```
string text = p.Find("Revenue").Find("Total").Right().Text;
```

- Extract full text from page 15:

```
string text = p.FindPage(15).FilterCurrentPage().ExtractText();
```

- Extract full text from page containing word "summary":

```
string text = p.Find("summary").FilterCurrentPage().ExtractText();
```

- Find page containing word "summary" and extract text in the left upper corner:

```
string text = p.Find("summary").Window(0, 0, 100, 100).ExtractText();
```

- Extract text between words "summary" and "total":

```
int fromToken = p.Find("summary").Index;
int toToken = p.Find("total").Index;
string text = p.FilterIndex(fromToken, toToken).ExtractText();
```

- Extract full document text:

```
string text = p.ExtractText();
```

- Extract table using absolute border location:

```
PdfTableOptions options = new PdfTableOptions();
options.ColumnBorders   = new double[5]{0,235,293,366,430};
DataTable myTable = p.Table(options);
```

Don't forget to call `p.Reset()` between the calls (if needed).

DataSetParser

- Load Excel file:

```
DataSetParser d = new
DataSetParser(DataSetReader.ReadExcel("myspreadsheet.xls"));
```

- Get text from section "Revenue", on the right of word "Total":

```
string text = d.Find("Revenue").Find("Total").Right().Text;
```

- Capture table on tab "Sheet2" that starts from word "Price":

```
DataTable price = d.FindSheet("Sheet2").Find("Price").Table();
```

Pattern Matching

Functions that end with "Pattern" are accepting wildcards for text matching. The syntax is:

`*` - any sequence of characters

`?` - any character

`|` - divider between multiple patterns

All pattern searches are case insensitive.

Examples:

```
p.FindPattern("*amount*") //text that contains "amount"
p.FindPattern("amount*") //text that starts with "amount"
p.FindPattern("*a??unt") //text that ends with "a", followed by two characters,
then "unt"
p.FindPattern("amount*|value*") //text that ends with "amount" or "value"
p.FindPattern("amount|value") //text that equals to "amount" or "value"

mytable.SelectColumnsPattern("exactname1|exactname2", "exactname3")
mytable.SelectColumnsPattern("*category*|*type*", "*amount*|*value*")
mytable.SelectColumnsAsPattern("*category*|*type*", "name" "*amount*|*value*",
"amount")

mytable.DeleteColumns(x=>x.ColumnName.MatchesPattern("Field*"))
```

Using TemplateLib in .Net

Requires .Net Framework 4.5

You need to add references to the following DLLs:

- `IvyPdf.dll` – to use PdfParser functionality
- `IvyDocumentReader.dll` – to read PDF files
- `IvyDataSet.dll` – to read Excel and CSV files
- `IvyTemplateLib.dll` – to use templates created via Ivy Template Editor

```
using IvyPdf;
using IvyTemplateLib;

//Open template library file
TemplateLibrary tl = TemplateLibrary.LoadTemplateLibrary("sample_library.tl");
tl.StaticCodeMode = true; //to prevent recompile on every run


//Open PDF file
PdfParser p = new PdfParser(IvyDocumentReader.ReadPdf("sample_document.pdf"));

//Run a template and get the results
List results;
tl.RunTemplate("Template1", p, out results);
```

Opening a document with specified `pdfReadOptions` parameter

```
using IvyPdf;
using IvyTemplateLib;

//The very first call to RunTemplate() may incur 2-3 second delay, due to
initialization of Roslyn engine.
//There's no way to avoid it completely, but we can start initialization on the
background before it's being used. This is optional and needs to be done only
once.
TemplateLibrary.WarmUp();

//Open template library file
TemplateLibrary tl = TemplateLibrary.LoadTemplateLibrary("sample_library.tl");
tl.StaticCodeMode = true; //to prevent recompile on every run

//Find template to run
Template t = Templates.First<Template>(x => x.TemplateName == "Template1");

//Open PDF file using template-specific PdfReadOptions
PdfParser p = new PdfParser(IvyDocumentReader.ReadPdf("sample_document.pdf",
t.PdfReadOptions));

//Run a template and get the results
List results;
tl.RunTemplate(t, p, out results);
```

Custom layout

In some cases default layout logic doesn't work and you may have to replace it with custom code. For your reference below is standard logic used to combine token collection.

```
IvyDocument doc = IvyDocumentReader.ReadPdf("abc.pdf", TokenLayoutType.None);

//this is standard logic used to combine tokens. You may want to adjust it.

//Remove shadow tokens
doc.RemoveTokens((t1, t2) => (
       (t1.OverlapRatio(t2) > 0.95)
    && (t1.Text == t2.Text)
    && (t1.Text.Trim() != "")
    ) ? 0 : -1);

//filter out vertical lines only, to speed up comparison
//also note start of each page
Dictionary<int, int> linePageStart = new Dictionary<int, int>();
int p = 0;
List<LineF> lines = new List<LineF>();
foreach (LineF l in Lines)
{
    if (Math.Abs(l.X1 - l.X2) > 5) continue;
    lines.Add(l);
    if (l.Page != p) {
        linePageStart.Add(l.Page, lines.Count - 1);
```

```
            p = l.Page;
        }
    }
    lines.Sort((a, b) => (a.Page > b.Page)?1:(a.Page<b.Page)?-1:
    a.X1.CompareTo(b.X1));

    //combine adjacent tokens
    doc.CombineTokens((t1, t2) => (
        t1.Overlap(t2) //adjacent/overlap and on the same Y (+/- 1/3 of height)
        && (t1.Font == t2.Font) //same font properties
        && (t1.Bold == t2.Bold)
        && (t1.Italic == t2.Italic)
        && (Math.Abs(t1.Height - t2.Height) < 0.1) //same height
        && (!SeparatedByVerticalLine(t1, t2, lines, linePageStart)) //do not have a
    line between
        ) ? 0 : -1);

    //re-sort in case some tokens moved
    doc.Tokens.Sort(new TokenComparer());

    //split long tokens by sequence of whitespace characters
    doc.SplitTokensByCharSequence(new char[] { ' ', '.' }, 4);
    doc.SplitTokensByCharSequence(new char[] { '…' }, 1);

    //remove all whitespace tokens that haven't been combined with anything else,
    including standalone periods
    doc.Tokens.RemoveAll(x => string.IsNullOrWhiteSpace(x.Text.Trim(new char[] {
    '…', ' ', '.' })));

    //trim text
    foreach (Token token in doc.Tokens) token.Text = token.Text.Trim();
```

Tutorial

PDF documents can be tricky. They range from simple and clean reports to extremely convoluted ones, with random artifacts and structural errors. The task of extracting specific values may be daunting, especially if you need to do it for large number of documents on multiple occasions. A computer can help with repeating tasks, but it's up to you to define the parameters and the kind of information you are looking for. Every document is different, however, there are some common scenarios, so let's try to break them down.

The data you need is always in the same exact location

This is pretty common for various receipts, financial statements and so on. And it's easy to filter out:

First get the page you need, e.g. it's on page 3.

`P.FilterPage(3)`

(In Ivy Template Editor `p` is a predefined `PdfParser` object)

Then filter for exact location:

`.FilterWindow(10,10,50,20)`

(Use Filter button in the template editor to create a window, then move it to the required location to get coordinates)

This will get you the first token in the selected "window". If you want all text that fits in there - just add `.ExtractText()`

The data you need follows a specific word

It may move to different places in the document, so exact position is not known. Let's say you want to get a number that follows a word "Total":

`P.Find("Total").Right().Text`

That was easy, wasn't it? Please be aware that `Find()` uses case-sensitive search and it will find any token that contains your string. You can also search for occurrence of multiple strings, like this:

`Find("Total", "total")`

You probably expect to find a number there, but it may have extra characters like currency signs, commas, percentage symbols. Simply use `.ToNumber()` function to clean it up. Another handy function is `ToDate()`. It can recognize dates in most formats, even surrounded by other text.

Hint: You don't necessarily need to type the search text yourself. In the Template Editor right-click the token you want to get and click "Suggest". You may need to clean up some code that is generated (the logic there is to search for a specific section, which would usually have larger font, then subsections, then the word next to the token you need. Some of these steps may be omitted in your case)

Extracting tables

Let's say there is a table in PDF and it's a tedious task to get it out manually, so let's see what we can do. Tables can quickly become tricky. There's no way to deal with every table out there, since there are way too many variations. Let's start with a simple one first.

Simple, rectangular table with a header

First you need to find a header. Right-click any header token and click "Suggest" - you should get a logic that brings you to that token. Let's say this:

`P.Find("Field1")`

Now just add `.Table()` and preview the results. In many cases this works right away and is really that simple.

By default IvyPdf does not use any graphical objects, like lines that surround table cells. Instead, the table is built using positions of the text tokens relative to each other. Due to this, some cells may get shifted to a wrong column or row. Also, you may get some unwanted data. You can try to fix this post-factum, using table extension methods like `.Rollup`, `DeleteRows`, `DeleteColumns` and so on.

Table without header

Just find any token in the top row. (For example it may be `Below` some specific text). Then use `Grid()` function.

Table with subtotals between rows

You have two options: you can get the whole table, then delete specific rows using `DeleteRows` method, or you can filter out unwanted tokens first. Let's assume the subtotals are in bold:

```
p.Filter(x=>x.Bold).Find("header1").Table()
```

Table with sub-headings that you want to add as a data column

This one is tricky. You would need some sort of loop for this. First create an empty `DataTable` object. Then find a sub-header and store it into a variable. Then move `Below` and call `Grid()`. `Union` the results with the empty table. Add new column, providing the stored header as a default value. Repeat. It may need a lot of tweaking to make this work, but you have a few helpful tools at your disposal:

- *Bookmarks* are handy when you need to go back and forth, especially if you applied a filter and need to move to previous position.
- *Conditional If and While loop* - you can write a regular C# code instead, but these two provide a concise syntax that can make your code easier to read. (Just don't forget to add comments.)
- Table and Grid functions have one very important option - *Whitespace limit*. You already pointed to the table header, but how does it know where the table ends? Well, usually there's some gap between end of the table and the following text, which is larger than spacing between rows. By default the limit is *2.5* which means the gap should be 2.5 times bigger than average row height. You can change that number according to your needs.

Various collections

Let's say you want to find all telephone numbers in the document. First, `Filter` by a regular expression, then use resulting `Tokens` collection - loop or convert to a `DataTable`. IvyPdf is using DataTables extensively. We prefer DataTables over other collections for their flexibility. We extended their functionality, so you can use `Join`, `Union`, and many other handy functions. However, you can create any collections you like, use Linq, add your own extensions and so on.

Connect to a database, read a text file, get data from a web service

In the "Template Library Settings" you can add as many "Modules" as you want. The modules are C# classes that you can call from your expressions. In addition you can reference any .Net assembly and use its methods.

Quick Hints

Here are some quick hints on Ivy Template Editor (in no particular order):

- Ivy is using C#, therefore everything is case-sensitive. Commands, variable names, fields and search strings are all case-sensitive.
- Template and field names should follow C# naming conventions. Spaces are not allowed and names cannot start with a number.
- When you run one field, all templates and fields are compiled as one piece. This means if you have a syntax error in one field the others will fail as well. It can be hard to figure out where the issue is, so the best approach is to change one field at a time and test it after each change.
- Field *Data Type* is very important. It defaults to `dynamic`, but it's a good idea to change it to proper type instead. Otherwise you may get errors when you try to reuse this field in other fields.

  For example, if field `a` is a table, but defined as `dynamic` you cannot use `a.Rows.Count` property. But if you change it to `DataTable` it works as expected.
- In most cases you should use `.Text` to get string values of tokens. For example:

  Instead of:

  `P.Find("total").Right()`

  Use:

  `P.Find("total").Right().Text`

  The exception is when you want to find a token and then move in different directions from it's position. For example you may want to use:

  `b=a.Up()` and `c=a.Down()`
- The example above - `b =a.Up()` - leads to another problem. Your token object is a reference and if you move `Up()` in one field it changes the position of the referenced object and the other fields will be changed too. The correct way to do this is `a.Copy().Up()` and `a.Copy().Down()` - this creates a full copy of the original object. (You can also use `Clone()`, which is same thing)
- Use "toolbox" to test various functions, e.g. search, filters, tables, then copy the resulting pieces of code into your definition.
- Use "suggest" command (on the right-click) to quickly write search sequences.
- Instead of creating new fields manually use Ctrl+drag-and-drop to make a copy of existing field, along with code and validation logic. Then refine the code.
- You can reuse code by creating an intermediate field. Simply uncheck "include in the output" and use it in other fields. For example, capture a table once, then select different rows/cells from it.
- If you are not familiar with C# DataTable object it's helpful to spend some time and go over the docs. You can use `Rows` and `Columns` collections, loop through them and use many extension methods provided by IvyPdf to do filters, joins, and so on.
- We often get questions on what is the best way to organize the templates. The answer really depends on your use case. In simple cases you may get away with one template and a few fields. In more complicated ones we suggest using template inheritance to cover a few similar formats. If you have very large number of formats along with many fields/sections that need to be captured, you may consider the following approach: create a separate template for every item that needs to be captured (e.g. Accounts, Payments, Ledger, etc) and have variables like Accounts1, Accounts2, ... for every version of document format. This way you can create a mapping between your documents and the correct variable that should be

used for that document (outside of Ivy) as well as test these variables with multiple documents. It is a complicated structure that requires additional code and a database to store the results, but some users found it to be the best approach for complicated scenarios.

FAQ

General questions

- What is IvyPdf?
  - IvyPdf is a software that extracts data from PDF documents using the logic you define.
- What are system requirements?
  - .Net 4.5
- Can it read any PDF?
  - No. Scanned PDFs don't have text information and cannot be used. Some PDFs may have inconsistent structure and various artifacts, making them hard to process. We recommend downloading a trial version and test your documents.
- How does it know which data to extract?
  - You write instructions using simple commands, like "Find text", "Left", "Right". The commands can be stored as a template and applied to multiple files.
- Do I need to know programming to use it?
  - Programming background is helpful, but you can accomplish a lot using our user interface with minimal coding. Try it yourself!
- When should I use IvyPdf library and when Ivy Template Editor?
  - You can use IvyPdf library from your .Net solution. Template Editor allows you to do everything in UI and is a standalone product.
- Why would I use Template Editor? Can't I do the same in Visual Studio instead?
  - You can use either one, or even both mixed together. Visual Studio provides better debugging capabilities. On the other hand Ivy Template Editor has a dedicated UI, helps to preview expression results, and organizes expressions into templates.
- How do I get the results from Template Editor?
  - The extraction results can be saved as Excel, XML or Json files.
- How do I run Template Editor on multiple files? How do I call it from my process?
  - You can use "Bulk file processing" menu option in the Template Editor.
  - You can use command line interface.
  - You can run it from .Net code.
- Can I add my own functions to Template Editor?
  - You can add your own code modules, written in C# or reference any .Net library and use it's functions.
- What is template inheritance?
  - Inheritance is handy when you have sets of PDFs with small structural differences. Instead of creating a brand new template you can use inheritance to add another variable or change the logic for existing one, without changing anything else.
- I have multiple templates, how does it know which one to use?
  - You can provide specific template name as a parameter, or you can write a rule for automatic template selection.

- Can I read files other than PDFs using Template Editor?
    - Yes. It can parse Excel/CSV files, has basic support for text documents and ability to plug-in custom logic for everything else.
    - If you need to combine data from a PDF file with other sources in one template (for example another PDF, excel, text, CSV, database, web service) you can do it as well, though it's a bit tricky. You would need to write a C# code or reference appropriate library.
- Can it read password-protected PDFs or PDFs protected from printing/data extraction?
    - No, you need to remove protection first.
- Why some PDFs look fine in Adobe Reader, but have junk/garbage characters in IvyPdf?
    - The issue is with PDF fonts. Every font should contain proper "ToUnicode" table that specifies how the characters are converted to text. If this table is missing or incorrectly set in the PDF file, the IvyPdf cannot extract text property. Try to select and copy text from Adobe Reader and paste it in Notepad - if you cannot get correct text this way then IvyPdf will have problems too.

PDF Parsing

- What are those "tokens" and why the text is split like this in my document?
    - These are pieces of text contained in the PDF. Their appearance and location is set in the PDF document itself. A token can be a paragraph of text or a single letter - depends on how the PDF was created. IvyPdf is reading the tokens from your document and does some minor post-processing to make them easier to extract.
- Why would I need filters?
    - Filters are used to ignore tokens that should be ignored. For example you can filter a document for a specific page or area of the page, ignore text of a specified size or font, filter between tokens and so on.
- How do I extract table data?
    - You have to get to a header token first, then use Table() or Grid() functions.
- What is the difference between Table and Grid?
    - Grid doesn't have header. The columns will be named as Field1, Field2,...
- Why my table is not recognized properly?
    - Table recognition is a very complex task. IvyPdf attempts to reconstruct table structure using tokens size and position. It's easy for well-structured tables, but becomes tricky for tables with subgroups, subtotals, shifted columns, columns without header and so on.
- My table is not recognized properly. What can I do?
    - There are a few parameters that you can play with. See API Reference.
    - Try to use Filter() first to remove unwanted text.
    - Try to use post-processing functions to fix the data (Rollup, Delete rows/columns).
    - If nothing works you may have to to write your own logic that works for your case (using Tokens collection).
- Why do I need bookmarks?
    - Bookmarks are used when you need to do repeating tasks. For example you can store your current position, apply required filters and get data. Then reset, move to previously stored position and continue from there.

Licensing

IvyPdf is licensed per installation. Depending on how you use Ivy different types of licenses are required.

- If you are using IvyPdf for personal or non-for-profit work you can obtain the license key by clicking the link on Downloads page. The limitation of the free license is that it expires every two months and you will need to obtain new license key and re-register the program when it expires.
- Commercial use within your organization:
    - If you are using Ivy Template Editor to write extraction logic you need to purchase developer license for every machine where IvyPdf is installed.
    - If you are a developer using IvyPdf in your .Net projects you need to purchase developer license for every machine where IvyPdf is installed.
    - If you develop an internal application that uses IvyPdf and it is deployed to other machines you need:
        - Developer license for every developer machine.
        - If developed application is used interactively, i.e. in manual mode, where users work with one document at a time then the end users also need developer license.
        - If the developed application is used in automatic (batch) mode, without user intervention then you need server license for every machine where the application is installed.
    - Enterprise license covers all installations within the organization.
- Commercial use outside of your organization:
    - If you create application for use outside of your organization then there are two options:
        - Your end users purchase IvyPdf license according to number of machines that require IvyPdf installations (developer license if application is used interactively, one document at a time, or server license if application is used in automatic (batch) mode).
        - You purchase OEM licenses and include them along with your software. Please contact us for OEM pricing.
    - Developer license is required for every developer machine in your organization.

License code can be set programmatically.

`License.SetLicense("your license code")` - sets the license and registers IvyPdf on the current machine.

`License.SetSessionLicense("your license code")` - sets the license for the currently running program, but does not register IvyPdf on the machine. This is preferable way for OEM-distributed software.