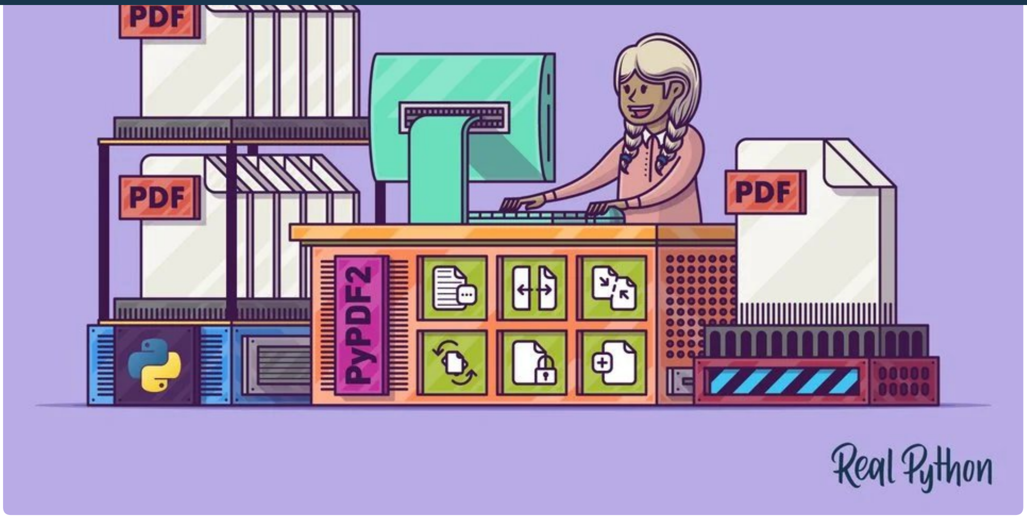


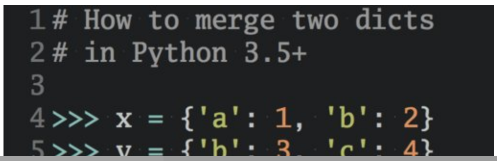
Create and Modify PDF Files in Python

by David Amos 14 Comments intermediate python

Mark as Completed

Tweet Share Email

Table of Contents



```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}
```

It’s really useful to know how to create and modify PDF files in Python. The **PDF**, or **P**ortable **D**ocument **F**ormat, is one of the most common formats for sharing documents over the Internet. PDFs can contain text, images, tables, forms, and rich media like videos and animations, all in a single file.

This abundance of content types can make working with PDFs difficult. There are a lot of different kinds of data to decode when opening a PDF file! Fortunately, the Python ecosystem has some great packages for reading, manipulating, and creating PDF files.

In this tutorial, you’ll learn how to:

- **Read** text from a PDF
- **Split** a PDF into multiple files

- **Concatenate** and **merge** PDF files
- **Rotate** and **crop** pages in a PDF file
- **Encrypt** and **decrypt** PDF files with passwords
- **Create** a PDF file from scratch

Note: This tutorial is adapted from the chapter “Creating and Modifying PDF Files” in *Python Basics: A Practical Introduction to Python 3*.

The book uses Python’s built-in IDLE editor to create and edit Python files and interact with the Python shell, so you will see occasional references to IDLE throughout this tutorial. However, you should have no problems running the example code from the editor and environment of your choice.

Along the way, you’ll have several opportunities to deepen your understanding by following along with the examples. You can download the materials used in the examples by clicking on the link below:

Download the sample materials: Click here to get the materials you’ll use to learn about creating and modifying PDF files in this tutorial.

Extracting Text From a PDF

In this section, you’ll learn how to read a PDF file and extract the text using the `PyPDF2` package. Before you can do that, though, you need to install it with `pip`:

Shell

```
$ python3 -m pip install PyPDF2
```

Verify the installation by running the following command in your terminal:

Shell

```
$ python3 -m pip show PyPDF2
Name: PyPDF2
Version: 1.26.0
Summary: PDF toolkit
Home-page: http://mstamy2.github.com/PyPDF2
Author: Mathieu Fenniak
Author-email: biziqe@mathieu.fenniak.net
License: UNKNOWN
Location: c:\\users\\david\\python38-32\\lib\\site-packages
Requires:
Required-by:
```

Pay particular attention to the version information. At the time of writing, the latest version of `PyPDF2` was `1.26.0`. If you have IDLE open, then you'll need to restart it before you can use the `PyPDF2` package.

Opening a PDF File

Let's get started by opening a PDF and reading some information about it. You'll use the `Pride_and_Prejudice.pdf` file located in the `practice_files/` folder in the companion repository.

Open IDLE's interactive window and import the `PdfFileReader` class from the `PyPDF2` package:

Python >>>

```python
>>> from PyPDF2 import PdfFileReader
```

To create a new instance of the `PdfFileReader` class, you'll need the path to the PDF file that you want to open. Let's get that now using the `pathlib` module:

Python >>>

```python
>>> from pathlib import Path
>>> pdf_path = (
...     Path.home()
...     / "creating-and-modifying-pdfs"
...     / "practice_files"
...     / "Pride_and_Prejudice.pdf"
... )
```

The `pdf_path` variable now contains the path to a PDF version of Jane Austen's *Pride and Prejudice*.

> **Note:** You may need to change `pdf_path` so that it corresponds to the location of the `creating-and-modifying-pdfs/` folder on your computer.

Now create the `PdfFileReader` instance:

```
Python                                                          >>>

>>> pdf = PdfFileReader(str(pdf_path))
```

You convert `pdf_path` to a string because `PdfFileReader` doesn't know how to read from a `pathlib.Path` object.

Recall from chapter 12, "File Input and Output," that all open files should be closed before a program terminates. The `PdfFileReader` object does all of this for you, so you don't need to worry about opening or closing the PDF file!

Now that you've created a `PdfFileReader` instance, you can use it to gather information about the PDF. For example, `.getNumPages()` returns the number of pages contained in the PDF file:

```
Python                                                          >>>

>>> pdf.getNumPages()
234
```

Notice that `.getNumPages()` is written in mixedCase, not lower_case_with_underscores as recommended in PEP 8. Remember, PEP 8 is a set of guidelines, not rules. As far as Python is concerned, mixedCase is perfectly acceptable.

Note: `PyPDF2` was adapted from the `pyPdf` package. `pyPdf` was written in 2005, only four years after PEP 8 was published.

At that time, many Python programmers were migrating from languages in which mixedCase was more common.

You can also access some document information using the `.documentInfo` attribute:

Python >>>

```
>>> pdf.documentInfo
{'/Title': 'Pride and Prejudice, by Jane Austen', '/Author': 'C
'/Creator': 'Microsoft® Office Word 2007',
'/CreationDate': 'D:20110812174208', '/ModDate': 'D:2011081217
'/Producer': 'Microsoft® Office Word 2007'}
```

The object returned by `.documentInfo` looks like a dictionary, but it's not really the same thing. You can access each item in `.documentInfo` as an attribute.

For example, to get the title, use the `.title` attribute:

Python >>>

```
>>> pdf.documentInfo.title
'Pride and Prejudice, by Jane Austen'
```

The `.documentInfo` object contains the PDF **metadata**, which is set when a PDF is created.

The `PdfFileReader` class provides all the necessary methods and attributes that you need to access data in a PDF file. Let's explore what you can do with a PDF file and how you can do it!

Extracting Text From a Page

PDF pages are represented in `PyPDF2` with the `PageObject` class. You use `PageObject` instances to interact with pages in a PDF file. You don’t need to create your own `PageObject` instances directly. Instead, you can access them through the `PdfFileReader` object’s `.getPage()` method.

There are two steps to extracting text from a single PDF page:

1. Get a `PageObject` with `PdfFileReader.getPage()`.
2. Extract the text as a string with the `PageObject` instance’s `.extractText()` method.

`Pride_and_Prejudice.pdf` has `234` pages. Each page has an index between `0` and `233`. You can get the `PageObject` representing a specific page by passing the page’s index to `PdfFileReader.getPage()`:

```python
>>> first_page = pdf.getPage(0)
```

`.getPage()` returns a `PageObject`:

```python
>>> type(first_page)
<class 'PyPDF2.pdf.PageObject'>
```

You can extract the page’s text with `PageObject.extractText()`:

```python
>>> first_page.extractText()
'\\n \\nThe Project Gutenberg EBook of Pride and Prejudice, by
Austen\\n \\n\\nThis eBook is for the use of anyone anywhere at
and with\\n \\nalmost no restrictions whatsoever.  You may copy
give it away or\\n \\nre\\n-\\nuse it under the terms of the Pr
Gutenberg License included\\n \\nwith this eBook or online at
www.gutenberg.org\\n \\n \\n \\nTitle: Pride and Prejudice\\n \
\\nAuthor: Jane Austen\\n \\n \\nRelease Date: August 26, 2008
[EBook #1342]\\n\\n[Last updated: August 11, 2011]\\n \\n \\nLa
```

```
Eng\\nlish\\n \\n \\nCharacter set encoding: ASCII\\n \\n \\n**
START OF THIS PROJECT GUTENBERG EBOOK PRIDE AND PREJUDICE ***\\
\\n \\n \\n \\nProduced by Anonymous Volunteers, and David Widg
\\n \\n \\n \\n \\n \\n \\nPRIDE AND PREJUDICE \\n \\n \\nBy Ja
Austen \\n \\n\\n \\n \\nContents\\n \\n'
```

Note that the output displayed here has been formatted to fit better on this page. The output you see on your computer may be formatted differently.

Every `PdfFileReader` object has a `.pages` attribute that you can use to iterate over all of the pages in the PDF in order.

For example, the following `for` loop prints the text from every page in the *Pride and Prejudice* PDF:

Python >>>

```python
>>> for page in pdf.pages:
...     print(page.extractText())
...
```

Let’s combine everything you’ve learned and write a program that extracts all of the text from the `Pride_and_Prejudice.pdf` file and saves it to a `.txt` file.

Putting It All Together

Open a new editor window in IDLE and type in the following code:

Python

```python
from pathlib import Path
from PyPDF2 import PdfFileReader

# Change the path below to the correct path for your computer.
pdf_path = (
    Path.home()
    / "creating-and-modifying-pdfs"
    / "practice-files"
    / "Pride_and_Prejudice.pdf"
)

# 1
```

```python
pdf_reader = PdfFileReader(str(pdf_path))
output_file_path = Path.home() / "Pride_and_Prejudice.txt"

# 2
with output_file_path.open(mode="w") as output_file:
    # 3
    title = pdf_reader.documentInfo.title
    num_pages = pdf_reader.getNumPages()
    output_file.write(f"{title}\\nNumber of pages: {num_pages}\

    # 4
    for page in pdf_reader.pages:
        text = page.extractText()
        output_file.write(text)
```

Let's break that down:

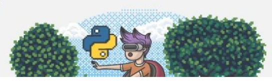
1. First, you assign a new `PdfFileReader` instance to the `pdf_reader` variable. You also create a new `Path` object that points to the file `Pride_and_Prejudice.txt` in your home directory and assign it to the `output_file_path` variable.

2. Next, you open `output_file_path` in write mode and assign the file object returned by `.open()` to the variable `output_file`. The `with` statement, which you learned about in chapter 12, "File Input and Output," ensures that the file is closed when the `with` block exits.

3. Then, inside the `with` block, you write the PDF title and number of pages to the text file using `output_file.write()`.

4. Finally, you use a `for` loop to iterate over all the pages in the PDF. At each step in the loop, the next `PageObject` is assigned to the `page` variable. The text from each page is extracted with `page.extractText()` and is written to the `output_file`.

When you save and run the program, it will create a new file in your home directory called `Pride_and_Prejudice.txt` containing the full text of the `Pride_and_Prejudice.pdf` document. Open it up and check it out!



Check Your Understanding

Expand the block below to check your understanding:

Exercise: Print Text From a PDF Show/Hide

You can expand the block below to see a solution:

Solution: Print Text From a PDF Show/Hide

When you're ready, you can move on to the next section.

Extracting Pages From a PDF

In the previous section, you learned how to extract all of the text from a PDF file and save it to a `.txt` file. Now you'll learn how to extract a page or range of pages from an existing PDF and save them to a new PDF.

You can use the `PdfFileWriter` to create a new PDF file. Let's explore this class and learn the steps needed to create a PDF using `PyPDF2`.

Using the `PdfFileWriter` Class

The `PdfFileWriter` class creates new PDF files. In IDLE's interactive window, import the `PdfFileWriter` class and create a new instance called `pdf_writer`:

Python >>>

```
>>> from PyPDF2 import PdfFileWriter
>>> pdf_writer = PdfFileWriter()
```

`PdfFileWriter` objects are like blank PDF files. You need to add some pages to them before you can save them to a file.

Go ahead and add a blank page to `pdf_writer`:

```python
>>> page = pdf_writer.addBlankPage(width=72, height=72)
```

The `width` and `height` parameters are required and determine the dimensions of the page in units called **points**. One point equals 1/72 of an inch, so the above code adds a one-inch-square blank page to `pdf_writer`.

`.addBlankPage()` returns a new `PageObject` instance representing the page that you added to the `PdfFileWriter`:

```python
>>> type(page)
<class 'PyPDF2.pdf.PageObject'>
```

In this example, you've assigned the `PageObject` instance returned by `.addBlankPage()` to the `page` variable, but in practice you don't usually need to do this. That is, you usually call `.addBlankPage()` without assigning the return value to anything:

```python
>>> pdf_writer.addBlankPage(width=72, height=72)
```

To write the contents of `pdf_writer` to a PDF file, pass a file object in binary write mode to `pdf_writer.write()`:

```python
>>> from pathlib import Path
>>> with Path("blank.pdf").open(mode="wb") as output_file:
...     pdf_writer.write(output_file)
...
```

This creates a new file in your current working directory called `blank.pdf`. If you open the file with a PDF reader, such as Adobe

Acrobat, then you'll see a document with a single blank one-inch-square page.

Technical detail: Notice that you save the PDF file by passing the file object to the `PdfFileWriter` object's `.write()` method, *not* to the file object's `.write()` method.

In particular, the following code will not work:

Python

```python
>>> with Path("blank.pdf").open(mode="wb") as output_file
...     output_file.write(pdf_writer)
```

This approach seems backwards to many new programmers, so make sure you avoid this mistake!

`PdfFileWriter` objects can write to new PDF files, but they can't create new content from scratch other than blank pages.

This might seem like a big problem, but in many situations, you don't need to create new content. Often, you'll work with pages extracted from PDF files that you've opened with a `PdfFileReader` instance.

Note: You'll learn how to create PDF files from scratch below, in the section "Creating a PDF File From Scratch."

In the example you saw above, there were three steps to create a new PDF file using `PyPDF2`:

1. Create a `PdfFileWriter` instance.
2. Add one or more pages to the `PdfFileWriter` instance.
3. Write to a file using `PdfFileWriter.write()`.

You'll see this pattern over and over as you learn various ways to add pages to a `PdfFileWriter` instance.

Extracting a Single Page From a PDF

Let’s revisit the *Pride and Prejudice* PDF that you worked with in the previous section. You’ll open the PDF, extract the first page, and create a new PDF file containing just the single extracted page.

Open IDLE’s interactive window and import `PdfFileReader` and `PdfFileWriter` from `PyPDF2` as well as the `Path` class from the `pathlib` module:

Python >>>

```
>>> from pathlib import Path
>>> from PyPDF2 import PdfFileReader, PdfFileWriter
```

Now open the `Pride_and_Prejudice.pdf` file with a `PdfFileReader` instance:

Python >>>

```
>>> # Change the path to work on your computer if necessary
>>> pdf_path = (
...     Path.home()
...     / "creating-and-modifying-pdfs"
...     / "practice_files"
...     / "Pride_and_Prejudice.pdf"
... )
>>> input_pdf = PdfFileReader(str(pdf_path))
```

Pass the index `0` to `.getPage()` to get a `PageObject` representing the first page of the PDF:

Python >>>

```
>>> first_page = input_pdf.getPage(0)
```

Now create a new `PdfFileWriter` instance and add `first_page` to it with `.addPage()`:

```python
>>> pdf_writer = PdfFileWriter()
>>> pdf_writer.addPage(first_page)
```

The `.addPage()` method adds a page to the set of pages in the `pdf_writer` object, just like `.addBlankPage()`. The difference is that it requires an existing `PageObject`.

Now write the contents of `pdf_writer` to a new file:

```python
>>> with Path("first_page.pdf").open(mode="wb") as output_file:
...     pdf_writer.write(output_file)
...
```

You now have a new PDF file saved in your current working directory called `first_page.pdf`, which contains the cover page of the `Pride_and_Prejudice.pdf` file. Pretty neat!

Extracting Multiple Pages From a PDF

Let's extract the first chapter from `Pride_and_Prejudice.pdf` and save it to a new PDF.

If you open `Pride_and_Prejudice.pdf` with a PDF viewer, then you can see that the first chapter is on the second, third, and fourth pages of the PDF. Since pages are indexed starting with `0`, you'll need to extract the pages at the indices `1`, `2`, and `3`.

You can set everything up by importing the classes you need and opening the PDF file:

```python
>>> from PyPDF2 import PdfFileReader, PdfFileWriter
>>> from pathlib import Path
>>> pdf_path = (
...     Path.home()
...     / "creating-and-modifying-pdfs"
...     / "practice_files"
...     / "Pride_and_Prejudice.pdf"
... )
>>> input_pdf = PdfFileReader(str(pdf_path))
```

Your goal is to extract the pages at indices `1`, `2`, and `3`, add these to a new `PdfFileWriter` instance, and then write them to a new PDF file.

One way to do this is to loop over the range of numbers starting at `1` and ending at `3`, extracting the page at each step of the loop and adding it to the `PdfFileWriter` instance:

Python >>>

```
>>> pdf_writer = PdfFileWriter()
>>> for n in range(1, 4):
...     page = input_pdf.getPage(n)
...     pdf_writer.addPage(page)
...
```

The loop iterates over the numbers `1`, `2`, and `3` since `range(1, 4)` doesn't include the right-hand endpoint. At each step in the loop, the page at the current index is extracted with `.getPage()` and added to the `pdf_writer` using `.addPage()`.

Now `pdf_writer` has three pages, which you can check with `.getNumPages()`:

Python >>>

```
>>> pdf_writer.getNumPages()
3
```

Finally, you can write the extracted pages to a new PDF file:

Python >>>

```
>>> with Path("chapter1.pdf").open(mode="wb") as output_file:
...     pdf_writer.write(output_file)
...
```

Now you can open the `chapter1.pdf` file in your current working directory to read just the first chapter of *Pride and Prejudice*.

Another way to extract multiple pages from a PDF is to take advantage of the fact that `PdfFileReader.pages` supports slice notation. Let's redo the previous example using `.pages` instead of looping over a `range` object.

Start by initializing a new `PdfFileWriter` object:

Python >>>

```
>>> pdf_writer = PdfFileWriter()
```

Now loop over a slice of `.pages` from indices starting at `1` and ending at `4`:

Python >>>

```
>>> for page in input_pdf.pages[1:4]:
...     pdf_writer.addPage(page)
...
```

Remember that the values in a slice range from the item at the first index in the slice up to, but not including, the item at the second index in the slice. So `.pages[1:4]` returns an iterable containing the pages with indices `1`, `2`, and `3`.

Finally, write the contents of `pdf_writer` to the output file:

Python >>>

```
>>> with Path("chapter1_slice.pdf").open(mode="wb") as output_f
...     pdf_writer.write(output_file)
...
```

Now open the `chapter1_slice.pdf` file in your current working directory and compare it to the `chapter1.pdf` file you made by looping over the `range` object. They contain the same pages!

Sometimes you need to extract every page from a PDF. You can use the methods illustrated above to do this, but `PyPDF2` provides a shortcut. `PdfFileWriter` instances have an `.appendPagesFromReader()` method that you can use to append pages from a `PdfFileReader` instance.

To use `.appendPagesFromReader()`, pass a `PdfFileReader` instance to the method's `reader` parameter. For example, the following code copies every page from the *Pride and Prejudice* PDF to a `PdfFileWriter` instance:

```python
>>> pdf_writer = PdfFileWriter()
>>> pdf_writer.appendPagesFromReader(pdf_reader)
```

`pdf_writer` now contains every page in `pdf_reader`!

Check Your Understanding

Expand the block below to check your understanding:

Exercise: Extract The Last Page of a PDF Show/Hide

You can expand the block below to see a solution:

Solution: Extract The Last Page of a PDF Show/Hide

When you're ready, you can move on to the next section.

Concatenating and Merging PDFs

Two common tasks when working with PDF files are concatenating and merging several PDFs into a single file.

When you **concatenate** two or more PDFs, you join the files one after another into a single document. For example, a company may concatenate several daily reports into one monthly report at the end of a month.

Merging two PDFs also joins the PDFs into a single file. But instead of joining the second PDF to the end of the first, merging allows you to insert it after a specific page in the first PDF. Then it pushes all of the first PDF's pages after the insertion point to the end of the second PDF.

In this section, you'll learn how to concatenate and merge PDFs using the `PyPDF2` package's `PdfFileMerger`.

Using the `PdfFileMerger` Class

The `PdfFileMerger` class is a lot like the `PdfFileWriter` class that you learned about in the previous section. You can use both classes to write PDF files. In both cases, you add pages to instances of the class and then write them to a file.

The main difference between the two is that `PdfFileWriter` can only append, or concatenate, pages to the end of the list of pages already contained in the writer, whereas `PdfFileMerger` can insert, or merge, pages at any location.

Go ahead and create your first `PdfFileMerger` instance. In IDLE's interactive window, type the following code to import the `PdfFileMerger` class and create a new instance:

```python
>>> from PyPDF2 import PdfFileMerger
>>> pdf_merger = PdfFileMerger()
```

`PdfFileMerger` objects are empty when they're first instantiated. You'll need to add some pages to your object before you can do anything with it.

There are a couple of ways to add pages to the `pdf_merger` object, and which one you use depends on what you need to accomplish:

- **`.append()`** concatenates every page in an existing PDF document to the end of the pages currently in the `PdfFileMerger`.

- `.merge()` inserts all of the pages in an existing PDF document after a specific page in the `PdfFileMerger`.

You'll look at both methods in this section, starting with `.append()`.

Concatenating PDFs With `.append()`

The `practice_files/` folder has a subdirectory called `expense_reports` that contains three expense reports for an employee named Peter Python.

Peter needs to concatenate these three PDFs and submit them to his employer as a single PDF file so that he can get reimbursed for some work-related expenses.

You can start by using the `pathlib` module to get a list of `Path` objects for each of the three expense reports in the `expense_reports/` folder:

Python >>>

```python
>>> from pathlib import Path
>>> reports_dir = (
...     Path.home()
...     / "creating-and-modifying-pdfs"
...     / "practice_files"
...     / "expense_reports"
... )
```

After you import the `Path` class, you need to build the path to the `expense_reports/` directory. Note that you may need to alter the code above to get the correct path on your computer.

Once you have the path to the `expense_reports/` directory assigned to the `reports_dir` variable, you can use `.glob()` to get an iterable of paths to PDF files in the directory.

Take a look at what's in the directory:

Python >>>

```
>>> for path in reports_dir.glob("*.pdf"):
...     print(path.name)
...
Expense report 1.pdf
Expense report 3.pdf
Expense report 2.pdf
```

The names of the three files are listed, but they aren't in order. Furthermore, the order of the files you see in the output on your computer may not match the output shown here.

In general, the order of paths returned by `.glob()` is not guaranteed, so you'll need to order them yourself. You can do this by creating a list containing the three file paths and then calling `.sort()` on that list:

Python >>>

```
>>> expense_reports = list(reports_dir.glob("*.pdf"))
>>> expense_reports.sort()
```

Remember that `.sort()` sorts a list in place, so you don't need to assign the return value to a variable. The `expense_reports` list will be sorted alphabetically by filename after `.list()` is called.

To confirm that the sorting worked, loop over `expense_reports` again and print out the filenames:

Python >>>

```
>>> for path in expense_reports:
...     print(path.name)
...
Expense report 1.pdf
Expense report 2.pdf
Expense report 3.pdf
```

That looks good!

Now you can concatenate the three PDFs. To do that, you'll use `PdfFileMerger.append()`, which requires a single string argument representing the path to a PDF file. When you call `.append()`, all of the pages in the PDF file are appended to the set of pages in the `PdfFileMerger` object.

Let's see this in action. First, import the `PdfFileMerger` class and create a new instance:

```python
>>> from PyPDF2 import PdfFileMerger
>>> pdf_merger = PdfFileMerger()
```

Now loop over the paths in the sorted `expense_reports` list and append them to `pdf_merger`:

```python
>>> for path in expense_reports:
...     pdf_merger.append(str(path))
...
```

Notice that each `Path` object in `expense_reports/` is converted to a string with `str()` before being passed to `pdf_merger.append()`.

With all of the PDF files in the `expense_reports/` directory concatenated in the `pdf_merger` object, the last thing you need to do is write everything to an output PDF file. `PdfFileMerger` instances have a `.write()` method that works just like the `PdfFileWriter.write()`.

Open a new file in binary write mode, then pass the file object to the `pdf_merge.write()` method:

```python
>>> with Path("expense_reports.pdf").open(mode="wb") as output_
...     pdf_merger.write(output_file)
...
```

You now have a PDF file in your current working directory called `expense_reports.pdf`. Open it up with a PDF reader and you'll find all three expense reports together in the same PDF file.

Merging PDFs With `.merge()`

To merge two or more PDFs, use `PdfFileMerger.merge()`. This method is similar to `.append()`, except that you must specify where in the output PDF to insert all of the content from the PDF you are merging.

Take a look at an example. Goggle, Inc. prepared a quarterly report but forgot to include a table of contents. Peter Python noticed the mistake and quickly created a PDF with the missing table of contents. Now he needs to merge that PDF into the original report.

Both the report PDF and the table of contents PDF can be found in the `quarterly_report/` subfolder of the `practice_files` folder. The report is in a file called `report.pdf`, and the table of contents is in a file called `toc.pdf`.

In IDLE's interactive window, import the `PdfFileMerger` class and create the `Path` objects for the `report.pdf` and `toc.pdf` files:

Python >>>

```python
>>> from pathlib import Path
>>> from PyPDF2 import PdfFileMerger
>>> report_dir = (
...     Path.home()
...     / "creating-and-modifying-pdfs"
...     / "practice_files"
...     / "quarterly_report"
... )
>>> report_path = report_dir / "report.pdf"
>>> toc_path = report_dir / "toc.pdf"
```

The first thing you'll do is append the report PDF to a new `PdfFileMerger` instance using `.append()`:

```python
>>> pdf_merger = PdfFileMerger()
>>> pdf_merger.append(str(report_path))
```

Now that `pdf_merger` has some pages in it, you can merge the table of contents PDF into it at the correct location. If you open the `report.pdf` file with a PDF reader, then you'll see that the first page of the report is a title page. The second is an introduction, and the remaining pages contain different report sections.

You want to insert the table of contents after the title page and just before the introduction section. Since PDF page indices start with `0` in `PyPDF2`, you need to insert the table of contents after the page at index `0` and before the page at index `1`.

To do that, call `pdf_merger.merge()` with two arguments:

1. The integer `1`, indicating the index of the page at which the table of contents should be inserted
2. A string containing the path of the PDF file for the table of contents

Here's what that looks like:

```python
>>> pdf_merger.merge(1, str(toc_path))
```

Every page in the table of contents PDF is inserted *before* the page at index `1`. Since the table of contents PDF is only one page, it gets inserted at index `1`. The page currently at index `1` then gets shifted to index `2`. The page currently at index `2` gets shifted to index `3`, and so on.

Now write the merged PDF to an output file:

```python
```

```
>>> with Path("full_report.pdf").open(mode="wb") as output_file
...     pdf_merger.write(output_file)
...
```

You now have a `full_report.pdf` file in your current working directory. Open it up with a PDF reader and check that the table of contents was inserted at the correct spot.

Concatenating and merging PDFs are common operations. While the examples in this section are admittedly somewhat contrived, you can imagine how useful a program would be for merging thousands of PDFs or for automating routine tasks that would otherwise take a human lots of time to complete.

Check Your Understanding

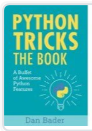
Expand the block below to check your understanding:

Exercise: Concatenate Two PDFs Show/Hide

You can expand the block below to see a solution:

Solution: Concatenate Two PDFs Show/Hide

When you're ready, you can move on to the next section.



Rotating and Cropping PDF Pages

So far, you've learned how to extract text and pages from PDFs and how to and concatenate and merge two or more PDF files. These are all common operations with PDFs, but `PyPDF2` has many other useful features.

Note: This tutorial is adapted from the chapter “Creating and Modifying PDF Files” in *Python Basics: A Practical Introduction to Python 3*. If you enjoy what you’re reading, then be sure to check out the rest of the book.

In this section, you’ll learn how to rotate and crop pages in a PDF file.

Rotating Pages

You’ll start by learning how to rotate pages. For this example, you’ll use the `ugly.pdf` file in the `practice_files` folder. The `ugly.pdf` file contains a lovely version of Hans Christian Andersen’s *The Ugly Duckling*, except that every odd-numbered page is rotated counterclockwise by ninety degrees.

Let’s fix that. In a new IDLE interactive window, start by importing the `PdfFileReader` and `PdfFileWriter` classes from `PyPDF2`, as well as the `Path` class from the `pathlib` module:

Python >>>

```
>>> from pathlib import Path
>>> from PyPDF2 import PdfFileReader, PdfFileWriter
```

Now create a `Path` object for the `ugly.pdf` file:

Python >>>

```
>>> pdf_path = (
...     Path.home()
...     / "creating-and-modifying-pdfs"
...     / "practice_files"
...     / "ugly.pdf"
... )
```

Finally, create new `PdfFileReader` and `PdfFileWriter` instances:

Python >>>

```
>>> pdf_reader = PdfFileReader(str(pdf_path))
>>> pdf_writer = PdfFileWriter()
```

Your goal is to use `pdf_writer` to create a new PDF file in which all of the pages have the correct orientation. The even-numbered pages in the PDF are already properly oriented, but the odd-numbered pages are rotated counterclockwise by ninety degrees.

To correct the problem, you'll use `PageObject.rotateClockwise()`. This method takes an integer argument, in degrees, and rotates a page clockwise by that many degrees. For example, `.rotateClockwise(90)` rotates a PDF page clockwise by ninety degrees.

> **Note:** In addition to `.rotateClockwise()`, the `PageObject` class also has `.rotateCounterClockwise()` for rotating pages counterclockwise.

There are several ways you can go about rotating pages in the PDF. We'll discuss two different ways of doing it. Both of them rely on `.rotateClockwise()`, but they take different approaches to determine which pages get rotated.

The first technique is to loop over the indices of the pages in the PDF and check if each index corresponds to a page that needs to be rotated. If so, then you'll call `.rotateClockwise()` to rotate the page and then add the page to `pdf_writer`.

Here's what that looks like:

Python >>>

```python
>>> for n in range(pdf_reader.getNumPages()):
...     page = pdf_reader.getPage(n)
...     if n % 2 == 0:
...         page.rotateClockwise(90)
...     pdf_writer.addPage(page)
...
```

Notice that the page gets rotated if the index is even. That might seem strange since the odd-numbered pages in the PDF are the ones that are rotated incorrectly. However, the page numbers in the PDF start with `1`, whereas the page indices start with `0`. That means odd-numbered PDF pages have even indices.

If that makes your head spin, don't worry! Even after years of dealing with stuff like this, professional programmers still get tripped up by these sorts of things!

> **Note:** When you execute the `for` loop above, you'll see a bunch of output in IDLE's interactive window. That's because `.rotateClockwise()` returns a `PageObject` instance.
>
> You can ignore this output for now. When you execute programs from IDLE's editor window, this output won't be visible.

Now that you've rotated all the pages in the PDF, you can write the content of `pdf_writer` to a new file and check that everything worked:

Python >>>

```
>>> with Path("ugly_rotated.pdf").open(mode="wb") as output_fil
...     pdf_writer.write(output_file)
...
```

You should now have a file in your current working directory called `ugly_rotated.pdf`, with the pages from the `ugly.pdf` file all rotated correctly.

The problem with the approach you just used to rotate the pages in the `ugly.pdf` file is that it depends on knowing ahead of time which pages need to be rotated. In a real-world scenario, it isn't practical to go through an entire PDF taking note of which pages to rotate.

In fact, you can determine which pages need to be rotated without prior knowledge. Well, *sometimes* you can.

Let's see how, starting with a new `PdfFileReader` instance:

Python >>>

```
>>> pdf_reader = PdfFileReader(str(pdf_path))
```

You need to do this because you altered the pages in the old `PdfFileReader` instance by rotating them. So, by creating a new instance, you're starting fresh.

`PageObject` instances maintain a dictionary of values containing information about the page:

Python >>>

```
>>> pdf_reader.getPage(0)
{'/Contents': [IndirectObject(11, 0), IndirectObject(12, 0),
IndirectObject(13, 0), IndirectObject(14, 0), IndirectObject(15
IndirectObject(16, 0), IndirectObject(17, 0), IndirectObject(18
'/Rotate': -90, '/Resources': {'/ColorSpace': {'/CS1':
IndirectObject(19, 0), '/CS0': IndirectObject(19, 0)}, '/XObjec
{'/Im0': IndirectObject(21, 0)}, '/Font': {'/TT1':
IndirectObject(23, 0), '/TT0': IndirectObject(25, 0)}, '/ExtGSt
{'/GS0': IndirectObject(27, 0)}}, '/CropBox': [0, 0, 612, 792],
'/Parent': IndirectObject(1, 0), '/MediaBox': [0, 0, 612, 792],
'/Type': '/Page', '/StructParents': 0}
```

Yikes! Mixed in with all that nonsensical-looking stuff is a key called `/Rotate`, which you can see on the fourth line of output above. The value of this key is `-90`.

You can access the `/Rotate` key on a `PageObject` using subscript notation, just like you can on a Python `dict` object:

Python >>>

```
>>> page = pdf_reader.getPage(0)
>>> page["/Rotate"]
-90
```

If you look at the `/Rotate` key for the second page in `pdf_reader`, you'll see that it has a value of `0`:

Python >>>

```
>>> page = pdf_reader.getPage(1)
>>> page["/Rotate"]
0
```

What all this means is that the page at index `0` has a rotation value of `-90` degrees. In other words, it's been rotated counterclockwise by ninety degrees. The page at index `1` has a rotation value of `0`, so it has not been rotated at all.

If you rotate the first page using `.rotateClockwise()`, then the value of `/Rotate` changes from `-90` to `0`:

```python
>>> page = pdf_reader.getPage(0)
>>> page["/Rotate"]
-90
>>> page.rotateClockwise(90)
>>> page["/Rotate"]
0
```

Now that you know how to inspect the `/Rotate` key, you can use it to rotate the pages in the `ugly.pdf` file.

The first thing you need to do is reinitialize your `pdf_reader` and `pdf_writer` objects so that you get a fresh start:

```python
>>> pdf_reader = PdfFileReader(str(pdf_path))
>>> pdf_writer = PdfFileWriter()
```

Now write a loop that loops over the pages in the `pdf_reader.pages` iterable, checks the value of `/Rotate`, and rotates the page if that value is `-90`:

```python
>>> for page in pdf_reader.pages:
...     if page["/Rotate"] == -90:
...         page.rotateClockwise(90)
...     pdf_writer.addPage(page)
...
```

Not only is this loop slightly shorter than the loop in the first solution, but it doesn't rely on any prior knowledge of which pages need to be rotated. You could use a loop like this to rotate pages in any PDF without ever having to open it up and look at it.

To finish out the solution, write the contents of `pdf_writer` to a new file:

```python
>>> with Path("ugly_rotated2.pdf").open(mode="wb") as output_fi
...     pdf_writer.write(output_file)
...
```

Now you can open the `ugly_rotated2.pdf` file in your current working directory and compare it to the `ugly_rotated.pdf` file you generated earlier. They should look identical.

> **Note:** One word of warning about the `/Rotate` key: it's not guaranteed to exist on a page.
>
> If the `/Rotate` key doesn't exist, then that usually means that the page has not been rotated. However, that isn't always a safe assumption.
>
> If a `PageObject` has no `/Rotate` key, then a `KeyError` will be raised when you try to access it. You can catch this exception with a `try...except` block.

The value of `/Rotate` may not always be what you expect. For example, if you scan a paper document with the page rotated ninety degrees counterclockwise, then the contents of the PDF will appear rotated. However, the `/Rotate` key may have the value `0`.

This is one of many quirks that can make working with PDF files frustrating. Sometimes you'll just need to open a PDF in a PDF reader program and manually figure things out.

Cropping Pages

Another common operation with PDFs is cropping pages. You might need to do this to split a single page into multiple pages or to extract just a small portion of a page, such as a signature or a figure.

For example, the `practice_files` folder includes a file called `half_and_half.pdf`. This PDF contains a portion of Hans Christian Andersen's *The Little Mermaid*.

Each page in this PDF has two columns. Let's split each page into two pages, one for each column.

To get started, import the `PdfFileReader` and `PdfFileWriter` classes from `PyPDF2` and the `Path` class from the `pathlib` module:

Python >>>

```python
>>> from pathlib import Path
>>> from PyPDF2 import PdfFileReader, PdfFileWriter
```

Now create a `Path` object for the `half_and_half.pdf` file:

Python >>>

```python
>>> pdf_path = (
...     Path.home()
...     / "creating-and-modifying-pdfs"
...     / "practice_files"
...     / "half_and_half.pdf"
... )
```

Next, create a new `PdfFileReader` object and get the first page of the PDF:

Python >>>

```python
>>> pdf_reader = PdfFileReader(str(pdf_path))
>>> first_page = pdf_reader.getPage(0)
```

To crop the page, you first need to know a little bit more about how pages are structured. `PageObject` instances like `first_page` have a `.mediaBox` attribute that represents a rectangular area defining the boundaries of the page.

You can use IDLE's interactive window to explore the `.mediaBox` before using it crop the page:

Python >>>

```
>>> first_page.mediaBox
RectangleObject([0, 0, 792, 612])
```

The `.mediaBox` attribute returns a `RectangleObject`. This object is defined in the `PyPDF2` package and represents a rectangular area on the page.

The list `[0, 0, 792, 612]` in the output defines the rectangular area. The first two numbers are the x- and y-coordinates of the lower-left corner of the rectangle. The third and fourth numbers represent the width and height of the rectangle, respectively. The units of all of the values are points, which are equal to 1/72 of an inch.

`RectangleObject([0, 0, 792, 612])` represents a rectangular region with the lower-left corner at the origin, a width of `792` points, or 11 inches, and a height of 612 points, or 8.5 inches. Those are the dimensions of a standard letter-sized page in landscape orientation, which is used for the example PDF of *The Little Mermaid*. A letter-sized PDF page in portrait orientation would return the output `RectangleObject([0, 0, 612, 792])`.

A `RectangleObject` has four attributes that return the coordinates of the rectangle's corners: `.lowerLeft`, `.lowerRight`, `.upperLeft`, and `.upperRight`. Just like the width and height values, these coordinates are given in points.

You can use these four properties to get the coordinates of each corner of the `RectangleObject`:

Python >>>

```
>>> first_page.mediaBox.lowerLeft
(0, 0)
>>> first_page.mediaBox.lowerRight
(792, 0)
>>> first_page.mediaBox.upperLeft
(0, 612)
>>> first_page.mediaBox.upperRight
(792, 612)
```

Each property returns a `tuple` containing the coordinates of the specified corner. You can access individual coordinates with square brackets just like you would any other Python tuple:

Python >>>

```python
>>> first_page.mediaBox.upperRight[0]
792
>>> first_page.mediaBox.upperRight[1]
612
```

You can alter the coordinates of a `mediaBox` by assigning a new tuple to one of its properties:

Python >>>

```python
>>> first_page.mediaBox.upperLeft = (0, 480)
>>> first_page.mediaBox.upperLeft
(0, 480)
```

When you change the `.upperLeft` coordinates, the `.upperRight` attribute automatically adjusts to preserve a rectangular shape:

Python >>>

```python
>>> first_page.mediaBox.upperRight
(792, 480)
```

When you alter the coordinates of the `RectangleObject` returned by `.mediaBox`, you effectively crop the page. The `first_page` object now contains only the information present within the boundaries of the new `RectangleObject`.

Go ahead and write the cropped page to a new PDF file:

```
Python                                                          >>>
>>> pdf_writer = PdfFileWriter()
>>> pdf_writer.addPage(first_page)
>>> with Path("cropped_page.pdf").open(mode="wb") as output_fil
...     pdf_writer.write(output_file)
...
```

If you open the `cropped_page.pdf` file in your current working directory, then you'll see that the top portion of the page has been removed.

How would you crop the page so that just the text on the left side of the page is visible? You would need to cut the horizontal dimensions of the page in half. You can achieve this by altering the `.upperRight` coordinates of the `.mediaBox` object. Let's see how that works.

First, you need to get new `PdfFileReader` and `PdfFileWriter` objects since you've just altered the first page in `pdf_reader` and added it to `pdf_writer`:

```
Python                                                          >>>
>>> pdf_reader = PdfFileReader(str(pdf_path))
>>> pdf_writer = PdfFileWriter()
```

Now get the first page of the PDF:

```
Python                                                          >>>
>>> first_page = pdf_reader.getPage(0)
```

This time, let's work with a copy of the first page so that the page you just extracted stays intact. You can do that by importing the `copy` module from Python's standard library and using `deepcopy()` to make a copy of the page:

```
Python                                                          >>>
>>> import copy
>>> left_side = copy.deepcopy(first_page)
```

Now you can alter `left_side` without changing the properties of `first_page`. That way, you can use `first_page` later to extract the text on the right side of the page.

Now you need to do a little bit of math. You already worked out that you need to move the upper right-hand corner of the `.mediaBox` to the top center of the page. To do that, you'll create a new `tuple` with the first component equal to half the original value and assign it to the `.upperRight` property.

First, get the current coordinates of the upper-right corner of the `.mediaBox`.

Python >>>

```
>>> current_coords = left_side.mediaBox.upperRight
```

Then create a new `tuple` whose first coordinate is half the value of the current coordinate and second coordinate is the same as the original:

Python >>>

```
>>> new_coords = (current_coords[0] / 2, current_coords[1])
```

Finally, assign the new coordinates to the `.upperRight` property:

Python >>>

```
>>> left_side.mediaBox.upperRight = new_coords
```

You've now cropped the original page to contain only the text on the left side! Let's extract the right side of the page next.

First get a new copy of `first_page`:

Python >>>

```
>>> right_side = copy.deepcopy(first_page)
```

Move the `.upperLeft` corner instead of the `.upperRight` corner:

Python >>>

```
>>> right_side.mediaBox.upperLeft = new_coords
```

This sets the upper-left corner to the same coordinates that you moved the upper-right corner to when extracting the left side of the page. So, `right_side.mediaBox` is now a rectangle whose upper-left corner is at the top center of the page and whose upper-right corner is at the top right of the page.

Finally, add the `left_side` and `right_side` pages to `pdf_writer` and write them to a new PDF file:

Python >>>

```
>>> pdf_writer.addPage(left_side)
>>> pdf_writer.addPage(right_side)
>>> with Path("cropped_pages.pdf").open(mode="wb") as output_fi
...     pdf_writer.write(output_file)
...
```

Now open the `cropped_pages.pdf` file with a PDF reader. You should see a file with two pages, the first containing the text from the left-hand side of the original first page, and the second containing the text from the original right-hand side.

Check Your Understanding

Expand the block below to check your understanding:

Exercise: Rotate Pages in a PDF Show/Hide

You can expand the block below to see a solution:

Solution: Rotate Pages in a PDF Show/Hide

Encrypting and Decrypting PDFs

Sometimes PDF files are password protected. With the `PyPDF2` package, you can work with encrypted PDF files as well as add password protection to existing PDFs.

Note: This tutorial is adapted from the chapter "Creating and Modifying PDF Files" in *Python Basics: A Practical Introduction to Python 3*. If you enjoy what you're reading, then be sure to check out the rest of the book.

Encrypting PDFs

You can add password protection to a PDF file using the `.encrypt()` method of a `PdfFileWriter()` instance. It has two main parameters:

1. **`user_pwd`** sets the user password. This allows for opening and reading the PDF file.
2. **`owner_pwd`** sets the owner password. This allows for opening the PDF without any restrictions, including editing.

Let's use `.encrypt()` to add a password to a PDF file. First, open the `newsletter.pdf` file in the `practice_files` directory:

Python >>>

```python
>>> from pathlib import Path
>>> from PyPDF2 import PdfFileReader, PdfFileWriter
>>> pdf_path = (
...     Path.home()
...     / "creating-and-modifying-pdfs"
...     / "practice_files"
...     / "newsletter.pdf"
... )
>>> pdf_reader = PdfFileReader(str(pdf_path))
```

Now create a new `PdfFileWriter` instance and add the pages from `pdf_reader` to it:

Python >>>

```
>>> pdf_writer = PdfFileWriter()
>>> pdf_writer.appendPagesFromReader(pdf_reader)
```

Next, add the password `"SuperSecret"` with `pdf_writer.encrypt()`:

Python >>>

```
>>> pdf_writer.encrypt(user_pwd="SuperSecret")
```

When you set only `user_pwd`, the `owner_pwd` argument defaults to the same string. So, the above line of code sets both the user and owner passwords.

Finally, write the encrypted PDF to an output file in your home directory called `newsletter_protected.pdf`:

Python >>>

```
>>> output_path = Path.home() / "newsletter_protected.pdf"
>>> with output_path.open(mode="wb") as output_file:
...     pdf_writer.write(output_file)
```

When you open the PDF with a PDF reader, you'll be prompted to enter a password. Enter `"SuperSecret"` to open the PDF.

If you need to set a separate owner password for the PDF, then pass a second string to the `owner_pwd` parameter:

Python >>>

```
>>> user_pwd = "SuperSecret"
>>> owner_pwd = "ReallySuperSecret"
>>> pdf_writer.encrypt(user_pwd=user_pwd, owner_pwd=owner_pwd)
```

In this example, the user password is `"SuperSecret"` and the owner password is `"ReallySuperSecret"`.

When you encrypt a PDF file with a password and attempt to open it, you must provide the password before you can view its contents. This protection extends to reading from the PDF in a Python program. Next, let's see how to decrypt PDF files with `PyPDF2`.

Decrypting PDFs

To decrypt an encrypted PDF file, use the `.decrypt()` method of a `PdfFileReader` instance.

`.decrypt()` has a single parameter called `password` that you can use to provide the password for decryption. The privileges you have when opening the PDF depend on the argument you passed to the `password` parameter.

Let's open the encrypted `newsletter_protected.pdf` file that you created in the previous section and use `PyPDF2` to decrypt it.

First, create a new `PdfFileReader` instance with the path to the protected PDF:

Python >>>

```
>>> from pathlib import Path
>>> from PyPDF2 import PdfFileReader, PdfFileWriter
>>> pdf_path = Path.home() / "newsletter_protected.pdf"
>>> pdf_reader = PdfFileReader(str(pdf_path))
```

Before you decrypt the PDF, check out what happens if you try to get the first page:

Python >>>

```
>>> pdf_reader.getPage(0)
Traceback (most recent call last):
  File "/Users/damos/github/realpython/python-basics-exercises/
  lib/python38-32/site-packages/PyPDF2/pdf.py", line 1617, in g
    raise utils.PdfReadError("file has not been decrypted")
PyPDF2.utils.PdfReadError: file has not been decrypted
```

A `PdfReadError` exception is raised, informing you that the PDF file has not been decrypted.

Note: The above traceback has been shortened to highlight the important parts. The traceback you see on your computer will be much longer.

Go ahead and decrypt the file now:

Python >>>

```
>>> pdf_reader.decrypt(password="SuperSecret")
1
```

`.decrypt()` returns an integer representing the success of the decryption:

- `0` indicates that the password is incorrect.
- `1` indicates that the user password was matched.
- `2` indicates that the owner password was matched.

Once you've decrypted the file, you can access the contents of the PDF:

Python >>>

```
>>> pdf_reader.getPage(0)
{'/Contents': IndirectObject(7, 0), '/CropBox': [0, 0, 612, 792
'/MediaBox': [0, 0, 612, 792], '/Parent': IndirectObject(1, 0),
'/Resources': IndirectObject(8, 0), '/Rotate': 0, '/Type': '/Pa
```

Now you can extract text and crop or rotate pages to your heart's content!

Check Your Understanding

Expand the block below to check your understanding:

Exercise: Encrypt a PDF Show/Hide

You can expand the block below to see a solution:

Solution: Encrypt a PDF Show/Hide

Creating a PDF File From Scratch

The `PyPDF2` package is great for reading and modifying existing PDF files, but it has a major limitation: you can't use it to create a new PDF file. In this section, you'll use the ReportLab Toolkit to generate PDF files from scratch.

ReportLab is a full-featured solution for creating PDFs. There is a commercial version that costs money to use, but a limited-feature open source version is also available.

> **Note:** This section is not meant to be an exhaustive introduction to ReportLab, but rather a sample of what is possible.
>
> For more examples, check out the ReportLab's code snippet page.

Installing `reportlab`

To get started, you need to install `reportlab` with `pip`:

Shell

```
$ python3 -m pip install reportlab
```

You can verify the installation with `pip show`:

Shell

```
$ python3 -m pip show reportlab
Name: reportlab
Version: 3.5.34
Summary: The Reportlab Toolkit
Home-page: http://www.reportlab.com/
Author: Andy Robinson, Robin Becker, the ReportLab team
        and the community
Author-email: reportlab-users@lists2.reportlab.com
License: BSD license (see license.txt for details),
         Copyright (c) 2000-2018, ReportLab Inc.
Location: c:\users\davea\venv\lib\site-packages
```

```
Requires: pillow
Required-by:
```

At the time of writing, the latest version of `reportlab` was 3.5.34. If you have IDLE open, then you'll need to restart it before you can use the `reportlab` package.

Using the `Canvas` Class

The main interface for creating PDFs with `reportlab` is the `Canvas` class, which is located in the `reportlab.pdfgen.canvas` module.

Open a new IDLE interactive window and type the following to import the `Canvas` class:

Python >>>

```python
>>> from reportlab.pdfgen.canvas import Canvas
```

When you make a new `Canvas` instance, you need to provide a string with the filename of the PDF you're creating. Go ahead and create a new `Canvas` instance for the file `hello.pdf`:

Python >>>

```python
>>> canvas = Canvas("hello.pdf")
```

You now have a `Canvas` instance that you've assigned to the variable name `canvas` and that is associated with a file in your current working directory called `hello.pdf`. The file `hello.pdf` does not exist yet, though.

Let's add some text to the PDF. To do that, you use `.drawString()`:

Python >>>

```python
>>> canvas.drawString(72, 72, "Hello, World")
```

The first two arguments passed to `.drawString()` determine the location on the canvas where the text is written. The first specifies the distance from the left edge of the canvas, and the second specifies the distance from the bottom edge.

The values passed to `.drawString()` are measured in points. Since a point equals 1/72 of an inch, `.drawString(72, 72, "Hello, World")` draws the string `"Hello, World"` one inch from the left and one inch from the bottom of the page.

To save the PDF to a file, use `.save()`:

```python
>>> canvas.save()
```

You now have a PDF file in your current working directory called `hello.pdf`. You can open it with a PDF reader and see the text `Hello, World` at the bottom of the page!

There are a few things to notice about the PDF you just created:

1. The default page size is A4, which is not the same as the standard US letter page size.
2. The font defaults to Helvetica with a font size of 12 points.

You're not stuck with these settings.

Setting the Page Size

When you instantiate a `Canvas` object, you can change the page size with the optional `pagesize` parameter. This parameter accepts a tuple of floating-point values representing the width and height of the page in points.

For example, to set the page size to `8.5` inches wide by `11` inches tall, you would create the following `Canvas`:

```python
canvas = Canvas("hello.pdf", pagesize=(612.0, 792.0))
```

`(612, 792)` represents a letter-sized paper because `8.5` times `72` is `612`, and `11` times `72` is `792`.

If doing the math to convert points to inches or centimeters isn't your cup of tea, then you can use the `reportlab.lib.units` module to help you with the conversions. The `.units` module contains several helper objects, such as `inch` and `cm`, that simplify your conversions.

Go ahead and import the `inch` and `cm` objects from the `reportlab.lib.units` module:

Python >>>

```python
>>> from reportlab.lib.units import inch, cm
```

Now you can inspect each object to see what they are:

Python >>>

```python
>>> cm
28.346456692913385
>>> inch
72.0
```

Both `cm` and `inch` are floating-point values. They represent the number of points contained in each unit. `inch` is `72.0` points and `cm` is `28.346456692913385` points.

To use the units, multiply the unit name by the number of units that you want to convert to points. For example, here's how to use `inch` to set the page size to `8.5` inches wide by `11` inches tall:

Python >>>

```python
>>> canvas = Canvas("hello.pdf", pagesize=(8.5 * inch, 11 * inc
```

By passing a tuple to `pagesize`, you can create any size of page that you want. However, the `reportlab` package has some standard built-in page sizes that are easier to work with.

The page sizes are located in the `reportlab.lib.pagesizes` module. For example, to set the page size to letter, you can import the `LETTER` object from the `pagesizes` module and pass it to the `pagesize` parameter when instantiating your `Canvas`:

```python
>>> from reportlab.lib.pagesizes import LETTER
>>> canvas = Canvas("hello.pdf", pagesize=LETTER)
```

If you inspect the `LETTER` object, then you'll see that it's a tuple of floats:

```python
>>> LETTER
(612.0, 792.0)
```

The `reportlab.lib.pagesize` module contains many standard page sizes. Here are a few with their dimensions:

Page Size	Dimensions
`A4`	210 mm x 297 mm
`LETTER`	8.5 in x 11 in
`LEGAL`	8.5 in x 14 in
`TABLOID`	11 in x 17 in

In addition to these, the module contains definitions for all of the ISO 216 standard paper sizes.

Setting Font Properties

You can also change the font, font size, and font color when you write text to the `Canvas`.

To change the font and font size, you can use `.setFont()`. First, create a new `Canvas` instance with the filename `font-example.pdf` and a letter page size:

Python >>>

```python
>>> canvas = Canvas("font-example.pdf", pagesize=LETTER)
```

Then set the font to Times New Roman with a size of `18` points:

Python >>>

```python
>>> canvas.setFont("Times-Roman", 18)
```

Finally, write the string `"Times New Roman (18 pt)"` to the canvas and save it:

Python >>>

```python
>>> canvas.drawString(1 * inch, 10 * inch, "Times New Roman (18
>>> canvas.save()
```

With these settings, the text will be written one inch from the left side of the page and ten inches from the bottom. Open up the `font-example.pdf` file in your current working directory and check it out!

There are three fonts available by default:

1. `"Courier"`
2. `"Helvetica"`
3. `"Times-Roman"`

Each font has bolded and italicized variants. Here's a list of all the font variations available in `reportlab`:

- `"Courier"`
- `"Courier-Bold`
- `"Courier-BoldOblique"`
- `"Courier-Oblique"`
- `"Helvetica"`
- `"Helvetica-Bold"`
- `"Helvetica-BoldOblique"`

- `"Helvetica-Oblique"`
- `"Times-Bold"`
- `"Times-BoldItalic`
- `"Times-Italic"`
- `"Times-Roman"`

You can also set the font color using `.setFillColor()`. In the following example, you create a PDF file with blue text named `font-colors.pdf`:

Python

```python
from reportlab.lib.colors import blue
from reportlab.lib.pagesizes import LETTER
from reportlab.lib.units import inch
from reportlab.pdfgen.canvas import Canvas

canvas = Canvas("font-colors.pdf", pagesize=LETTER)

# Set font to Times New Roman with 12-point size
canvas.setFont("Times-Roman", 12)

# Draw blue text one inch from the left and ten
# inches from the bottom
canvas.setFillColor(blue)
canvas.drawString(1 * inch, 10 * inch, "Blue text")

# Save the PDF file
canvas.save()
```

`blue` is an object imported from the `reportlab.lib.colors` module. This module contains several common colors. A full list of colors can be found in the `reportlab` source code.

The examples in this section highlight the basics of working with the `Canvas` object. But you've only scratched the surface. With `reportlab`, you can create tables, forms, and even high-quality graphics from scratch!

The ReportLab User Guide contains a plethora of examples of how to generate PDF documents from scratch. It's a great place to start if you're interested in learning more about creating PDFs with Python.

Check Your Understanding

Expand the block below to check your understanding:

Exercise: Create a PDF From Scratch Show/Hide

You can expand the block below to see a solution:

Solution: Create a PDF From Scratch Show/Hide

When you're ready, you can move on to the next section.

Conclusion: Create and Modify PDF Files in Python

In this tutorial, you learned how to create and modify PDF files with the `PyPDF2` and `reportlab` packages. If you want to follow along with the examples you just saw, then be sure to download the materials by clicking the link below:

Download the sample materials: Click here to get the materials you'll use to learn about creating and modifying PDF files in this tutorial.

With `PyPDF2`, you learned how to:

- **Read** PDF files and **extract** text using the `PdfFileReader` class
- **Write** new PDF files using the `PdfFileWriter` class
- **Concatenate** and **merge** PDF files using the `PdfFileMerger` class
- **Rotate** and **crop** PDF pages
- **Encrypt** and **decrypt** PDF files with passwords

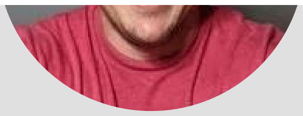
You also had an introduction to creating PDF files from scratch with the `reportlab` package. **You learned how to:**

- Use the `Canvas` class
- **Write** text to a `Canvas` with `.drawString()`
- Set the **font** and **font size** with `.setFont()`
- Change the **font color** with `.setFillColor()`

`reportlab` is a powerful PDF creation tool, and you only scratched the surface of what's possible. If you enjoyed what you learned in this sample from *Python Basics: A Practical Introduction to Python 3*, then be sure to check out the rest of the book.

Happy coding!

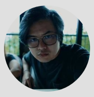
About **David Amos**



David is a writer, programmer, and mathematician passionate about exploring mathematics through code.

» More about David

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

 Aldren Joanna Jacob

What Do You Think?

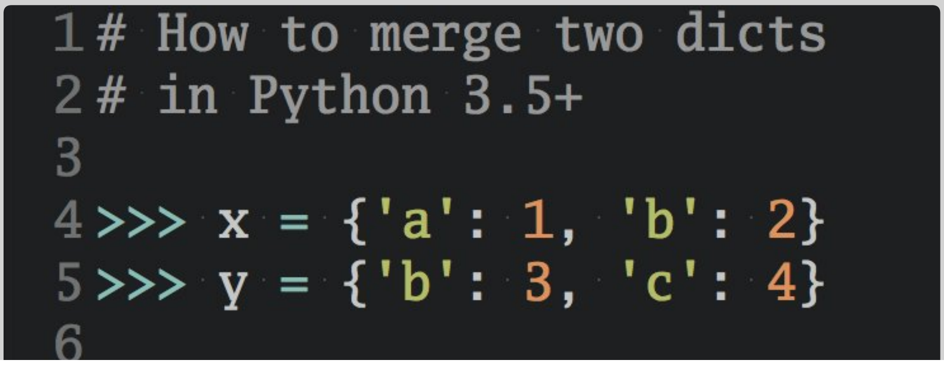
What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

Looking for a real-time conversation? Visit the Real Python Community Chat or join the next "Office Hours" Live Q&A Session. Happy Pythoning!

Keep Learning

Related Tutorial Categories: intermediate python



```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email…

Get Python Tricks »

No spam. Unsubscribe any time.

All Tutorial Topics

advanced api basics best-practices community databases data-science devops django docker flask front-end gamedev gui intermediate machine-learning projects python testing tools web-dev web-scraping

redis

NoSQL Data Modeling Patterns and Practices Using Redis

Read Now

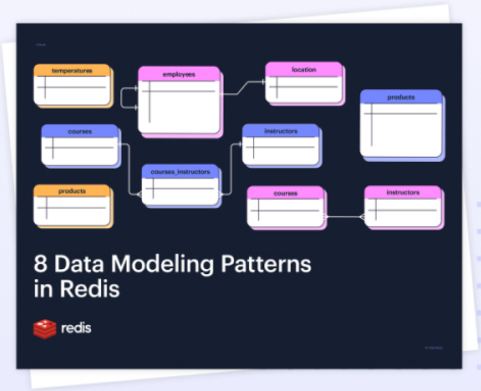


Table of Contents

Mark as Completed

Tweet Share Email