Print

Introduction

The requirement: to print

This is probably the most wide-ranging question I'll have to address here, and the one with the greatest disparity between the number and complexity of solutions and the simplicity of the requirement. The answer is: it all depends what you're trying to print, what tools you have at your disposal, and how much control you need.

- If you simply have a "document" (read: file of a well-known type, associated with one application) you wish to print, and aren't too fussy about controlling, then you can use the ShellExecute approach. This works (assuming you have the corresponding applications installed) for Microsoft Office documents, PDF files, text files, and pretty much any major application. Try it and see.
- The next most general case is where you have something, for example a text file or raw PCL, which you know you can send directly to a printer. In that case, you can use the win32print functions directly.
- If you have an image to print, you can combine the power of the Python Imaging Library with the win32ui module to do a rough-and-ready but useful print to any printer.
- If you have a fair amount of text to print, your best bet is to use the Reportlab PDF Toolkit and its Platypus document system to generate readable PDFs from any amount of text, and then use the ShellExecute technique to print it.

Standard document: use ShellExecute

Make use of the fact that within Win32, file types (in effect, extensions) can be associated with applications via command verbs. Typically the same application will handle all verbs (and typically those verbs are Open and Print) but that's not strictly necessary. This means that you can tell the OS to take your file and call whatever's necessary to print it.

- Takes care of standard file types
- No need to mess around with printer lists
- Gives you no control
- Only works for well-defined document-application pairings.
- ~~Only prints to default printer~~

UPDATE: Kudos to Chris Curvey for pointing out that you can specify a printer by including it with a d: switch in the params section. Don't know if it works for every file type.

```
import tempfile
import win32api
import win32print

filename = tempfile.mktemp (".txt")
open (filename, "w").write ("This is a test")
win32api.ShellExecute (
  0,
  "print",
  filename,
```

```
  #
  # If this is None, the default printer will
  # be used anyway.
  #
  '/d:"%s"' % win32print.GetDefaultPrinter (),
  ".",
  0
)
```

UPDATE 2: Mat Baker & Michael "micolous" both point out that there's an underdocumented printto verb which takes the printer name as a parameter, enclosed in quotes if it contains spaces. I haven't got this to work but they both report success for at least some file types.

```
import tempfile
import win32api
import win32print

filename = tempfile.mktemp (".txt")
open (filename, "w").write ("This is a test")
win32api.ShellExecute (
  0,
  "printto",
  filename,
  '"%s"' % win32print.GetDefaultPrinter (),
  ".",
  0
)
```

Raw printable data: use win32print directly

The win32print module offers (almost) all the printing primitives you'll need to take some data and throw it at a printer which has already been defined on your system. The data must be in a form which the printer will happily swallow, usually something like text or raw PCL.

- Quick and easy
- You can decide which printer to use
- Data must be printer-ready

```
import os, sys
import win32print
printer_name = win32print.GetDefaultPrinter ()
#
# raw_data could equally be raw PCL/PS read from
#  some print-to-file operation
#
if sys.version_info >= (3,):
  raw_data = bytes ("This is a test", "utf-8")
else:
  raw_data = "This is a test"

hPrinter = win32print.OpenPrinter (printer_name)
try:
  hJob = win32print.StartDocPrinter (hPrinter, 1, ("test of raw data", None, "RAW"))
  try:
    win32print.StartPagePrinter (hPrinter)
    win32print.WritePrinter (hPrinter, raw_data)
    win32print.EndPagePrinter (hPrinter)
  finally:
    win32print.EndDocPrinter (hPrinter)
finally:
```

```
  win32print.ClosePrinter (hPrinter)
```

Single image: use PIL and win32ui

Without any extra tools, printing an image on a Windows machine is almost insanely difficult, involving at least three device contexts all related to each other at different levels and a fair amount of trial-and-error. Fortunately, there is such a thing as a device-independent bitmap (DIB) which lets you cut the Gordian knot -- or at least some of it. Even more fortunately, the Python Imaging Library supports the beast. The following code does a quick job of taking an image file and a printer and printing the image as large as possible on the page without losing the aspect ratio, which is what you want most of the time.

- It works
- You can decide which printer to use
- (Thanks to the PIL) You can select loads of image formats
- If you're not up on Windows device contexts, it's not the most intelligble of techniques.

```
import win32print
import win32ui
from PIL import Image, ImageWin

#
# Constants for GetDeviceCaps
#
#
# HORZRES / VERTRES = printable area
#
HORZRES = 8
VERTRES = 10
#
# LOGPIXELS = dots per inch
#
LOGPIXELSX = 88
LOGPIXELSY = 90
#
# PHYSICALWIDTH/HEIGHT = total area
#
PHYSICALWIDTH = 110
PHYSICALHEIGHT = 111
#
# PHYSICALOFFSETX/Y = left / top margin
#
PHYSICALOFFSETX = 112
PHYSICALOFFSETY = 113

printer_name = win32print.GetDefaultPrinter ()
file_name = "test.jpg"

#
# You can only write a Device-independent bitmap
#  directly to a Windows device context; therefore
#  we need (for ease) to use the Python Imaging
#  Library to manipulate the image.
#
# Create a device context from a named printer
#  and assess the printable size of the paper.
#
hDC = win32ui.CreateDC ()
hDC.CreatePrinterDC (printer_name)
printable_area = hDC.GetDeviceCaps (HORZRES), hDC.GetDeviceCaps (VERTRES)
printer_size = hDC.GetDeviceCaps (PHYSICALWIDTH), hDC.GetDeviceCaps (PHYSICALHEIGHT)
```

```
printer_margins = hDC.GetDeviceCaps (PHYSICALOFFSETX), hDC.GetDeviceCaps
(PHYSICALOFFSETY)

#
# Open the image, rotate it if it's wider than
#  it is high, and work out how much to multiply
#  each pixel by to get it as big as possible on
#  the page without distorting.
#
bmp = Image.open (file_name)
if bmp.size[0] > bmp.size[1]:
  bmp = bmp.rotate (90)

ratios = [1.0 * printable_area[0] / bmp.size[0], 1.0 * printable_area[1] /
bmp.size[1]]
scale = min (ratios)

#
# Start the print job, and draw the bitmap to
#  the printer device at the scaled size.
#
hDC.StartDoc (file_name)
hDC.StartPage ()

dib = ImageWin.Dib (bmp)
scaled_width, scaled_height = [int (scale * i) for i in bmp.size]
x1 = int ((printer_size[0] - scaled_width) / 2)
y1 = int ((printer_size[1] - scaled_height) / 2)
x2 = x1 + scaled_width
y2 = y1 + scaled_height
dib.draw (hDC.GetHandleOutput (), (x1, y1, x2, y2))

hDC.EndPage ()
hDC.EndDoc ()
hDC.DeleteDC ()
```

Given the technique (creating a device context) you could use any of the standard Windows functions on it, such as DrawText, BitBlt &c. eg (example from pywin32 documentation):

```
import win32ui
import win32print
import win32con

INCH = 1440

hDC = win32ui.CreateDC ()
hDC.CreatePrinterDC (win32print.GetDefaultPrinter ())
hDC.StartDoc ("Test doc")
hDC.StartPage ()
hDC.SetMapMode (win32con.MM_TWIPS)
hDC.DrawText ("TEST", (0, INCH * -1, INCH * 8, INCH * -2), win32con.DT_CENTER)
hDC.EndPage ()
hDC.EndDoc ()
```

Lots of text: generate a PDF

You could just send text directly to the printer, but you're at the mercy of whatever fonts and margins and what-have-you the printer has defined. Rather than start emitting raw PCL codes you can generate PDFs and let Acrobat look after printing. The Reportlab toolkit does this supremely well, and especially its Platypus document framework, which gives you the ability to generate pretty much arbitrarily complex documents. The example below hardly scratches the surface of the toolkit, but shows that you don't need two pages of setup code to generate a perfectly usable PDF. Once this is generated, you can use the ShellExecute technique outlined

above to print.

```
from reportlab.platypus import SimpleDocTemplate, Paragraph, Spacer
from reportlab.lib.styles import getSampleStyleSheet
from reportlab.lib.units import inch

import cgi
import tempfile
import win32api

source_file_name = "c:/temp/temp.txt"
pdf_file_name = tempfile.mktemp (".pdf")

styles = getSampleStyleSheet ()
h1 = styles["h1"]
normal = styles["Normal"]

doc = SimpleDocTemplate (pdf_file_name)
#
# reportlab expects to see XML-compliant
#  data; need to escape ampersands &c.
#
text = cgi.escape (open (source_file_name).read ()).splitlines ()

#
# Take the first line of the document as a
#  header; the rest are treated as body text.
#
story = [Paragraph (text[0], h1)]
for line in text[1:]:
  story.append (Paragraph (line, normal))
  story.append (Spacer (1, 0.2 * inch))

doc.build (story)
w
i
n
3
2
a
p
i
.
S
h
e
l
l
E
x
e
c
u
t
e

(
0
,

"
p
r
i
n
t
"
,

n
```