# pyprototypr

# Contents

# Dedication

**pyprototypr** is dedicated to my Mom who, like many pre-WorldWar mothers, perhaps never really understood what her son was doing with that "strange little box" but nonetheless supported all my efforts!

# Introduction

**pyprototypr** is a program for designing & creating simple graphical outputs.

**pyprototypr** allows you to easily write a *script* i.e. a recipe or list of instructions in a text file, that defines a game board, or a set of cards or tiles, or any other, similar, regular graphical designs. When the script is "run" (or executed) it will create a PDF document containing this design.

In more technical terms, **pyprototypr** is a Python module which makes use of third-party graphical and font modules - primarily the excellent ReportLab library (see http://www.reportlab.com/opensource/) - to provide reusable classes, fronted by a wrapper script to simplify graphics creation.

**pyprototypr** is useful for anyone that needs to work on a design in an incremental fashion, tweaking and changing as they go along. Doing this with a regular graphics package can be tedious; especially when common changes need to be made across many elements. Simple designs that make use of regular-shaped symbols or fonts are straightforward to implement in **pyprototypr**; but more complex pictures or background images should be made, as usual, in your regular graphics design package and then imported via your script.

**pyprototypr** is *NOT* designed to be a graphics editor - like the Adobe Photoshop suite, or GIMP, or Inkscape packages - so it does not attempt in any way to replicate their functionality.

Some more 'background' to the program is provided in APPENDIX I: Design Approach.

# Installing pyprototypr

**pyprototypr** requires a computing device that already has the correct version of Python (version 3.11) installed. If your device does not have Python installed, it can be obtained from http://www.python.org/download/.

It is recommended that you work with Python in a *virtualenv*; see http://docs.python-guide.org/en/latest/dev/virtualenvs/ for a useful overview.

You can also use *pyenv*, if needed, to install the correct version of Python. See also APPENDIX VI: Working with pyenv

## Testing that Python is installed

In order to test that Python is installed, start a **command-line window**. The way you do this depends on your operating system.

- For Windows users

  Go to "Start -> Run" (On Windows 7 to 10, press "WindowsKey+R" or use the search box at the bottom of the Start menu)

- For Mac OS X users

  Go to your Applications/Utilities folder and choose "Terminal".

  There is also a helpful guide on working with Python from pyLadies; see: http://www.pyladies.com/blog/Get-Your-Mac-Ready-for-Python-Programming/

- For Linux users, you should already know how to do this!

When the command-line window appears, type:

```
python
```

You should see something like:

```
Python 3.11.7 (main, Dec 15 2023, 18:12:31) [GCC 11.2.0]
Type "help", "copyright", "credits" or "license" for more information.
```

You can now close the command-line window.

## After Python is installed

The simplest way to install **pyprototypr** itself, in the virtualenv, is via:

```
python setup.py install
```

Alternatively, for a manual approach, you can install these graphics and font modules (a **module** is an add-on to Python that gives it extra or specialised functionality):

```
reportlab - see https://pypi.python.org/pypi/reportlab
xlrd - see https://pypi.python.org/pypi/xlrd
boardgamegeek - see https://github.com/SukiCZ/boardgamegeek
```

If you are working with a virtualenv as recommended, then running:

```
pip install -r requirements.txt
```

will install all of these for you.

## Other Software Installs

You will also need a program that can display PDF files; for example, **Adobe Acrobat** (cross-platform), or **evince** (Linux), or **Preview** (Mac), or **foxit** (windows).

For Linux users, it is recommended that you install Microsoft's Core Fonts - see http://mscorefonts2.sourceforge.net/ Ubuntu users can install these via:

```
sudo apt-get install ttf-mscorefonts-installer
```

## Checking pyprototypr usage

To now check that *pyprototypr* works, then you can run one (or more) of the scripts (files) from any of the *examples* sub-directories.

By "run", its meant that you open a command-line window (see previous section) and type something like:

```
python example1.py
```

where you would replace the *example1.py* with the name of the example file.

For example, the *example1.py* file in the *examples/manual* directory contains these lines; each line in this Python script that does not start with a # is a called an **instruction**:

```
# `example1` script for pyprototypr
# Written by: Derek Hohls
# Created on: 29 February 2016
from pyprototypr.draw import *
Create()
PageBreak()
Save()
```

and is designed to produce a single blank A4-sized page! If you run this script, it will create an output file called *example1.pdf*, which will appear in the same directory as the script. You should be able to open and view this PDF file via a PDF viewer (see Other Software Installs).

The script also shows the two key instructions - *Create()* and *Save()* - that **must** appear near the start and at the end - respectively - of every script .

# Learning pyprototypr by example

Assuming you have all the software installed and ready to use (see Installing pyprototypr), you're now ready to start using it!

This manual includes a set of simple examples, for creating card designs, that you can work through (see APPENDIX II: Card Examples). These examples start off very simply and each new one adds new options. It's therefore best to try and work through these in order; and check that **pyprototypr** produces the expected results as you follow along.

In general, what you are doing is typing a set of instructions into a text file (a Python script), saving that file, and then using Python to process the file to create your output (a PDF file) containing the results of the instructions.

If the examples make sense, and you are ready to learn the ins-and-outs of *pyprototypr* to tackle your own projects, please keep reading ...

# Basic Concepts

## Page Layout

When using **pyprototypr** what you are doing is defining where and how various things appear on a page. Each page has an imaginary x-y (Cartesian) grid whose "zero" points for the axes appear on the lower-left of the page.

So, if you take an A4 page of about 21cm wide and 30cm tall; then a point in the middle of the page would have an **x-position** of 10.5cm - the distance from the left edge of the page; and a **y-position** of about 15cm - the distance from the bottom edge of the page. Similarly, for a letter-sized page of 8.5" by 11", a point in the middle of the page would have an **x-position** of 4.25" and a **y-position** of 5.5".

## What's on a Page?

Almost everything in **pyprototypr** that appears in the output is an element of some sort. Elements are often geometrical **shapes**, such lines, circles or rectangles, but can also be text or images.

## Other Properties

Elements have other properties apart from their position.

For example, the rectangle which represents the outline of a card has a *size*. The rectangle size is measured in terms of its *height* and *width*. The line used to draw the rectangle also has a *thickness* and a *color*. A circle will have a *radius* property, and so on. The script elements section describes these in more detail.

## Units

All positions, distances, line thicknesses and sizes need to be measured in a particular set of **units**.

In the USA, people tend to use the so-called Imperial System. In **pyprototypr** this means that distances could be measured in units of inches (abbreviated as *in*). In the rest of the world, everyone uses the Metric System. In **pyprototypr** this means that distances could be measured in units of centimetres (abbreviated as *cm*). For conversion purposes, 1 inch equals 25.4 centimetres.

**pyprototypr** also allows units of *points*, which are measurement units traditionally used in the printing industry. There are 72 points in 1 inch.

## Colors

Everything we see has color. Color, for the purposes of **pyprototypr**, is defined the same way as it is in pages appearing on the World Wide Web i.e. in RGB (red-green-blue) hexadecimal format - for example, *#A0522D* represents a shade of the color we would call "brown". For more details on this format, please refer to http://www.w3.org/TR/css3-color.

Colors in **pyprototypr** can also make use of names from a pre-defined list of colors - for example *#A0522D* is defined as the color *sienna*. A PDF called *colorset.pdf*, which shows all the names of all the colors that are available, can be found in the *examples* directory.

# How pyprototypr works

## Summary

To work with **pyprototypr**, you first create a text file containing a number of lines, each with an instruction, which, when taken together, will create your design "prototype". Such a file is referred to in this manual as a *script*.

You then use Python to *"run"* (process) that script, or file, to create an output file (in PDF format) with your resulting design. If you want to make changes to the design, then you add to, delete or change the instructions in your script and then use Python to re-process it and update the PDF.

## The script concept

The script is similar to the process of building a house; in the sense that the instructions which come first create layers that are "deeper down", in the same way that a foundation is below a floor, which is below the walls, which are below the ceiling, which is below the roof. The lower layers are often not "visible", even if they are there.

So, for example, a page may contain rectangles representing cards. Each card may then have additional rectangles placed on it, representing some aspect that is part of your card design. Those rectangles, in turn, could have images or icons placed on them. So, each item can "obscure" part or all of the item it is placed on.

In summary - the *order* of instructions in a script is important!

## Instruction

In general, one line in the script will contain one instruction.

The first line of every script **must** contain the phrase:

```
from pyprototypr.draw import *
```

which is a special instruction, telling Python that it needs to use ("import") the functions available in the **pyprototypr** *draw* module when processing your instructions.

Each instruction will look something like this:

```
SomeInstruction()
```

or this:

```
SomeName(value)
```

or this:

```
SomeNames(value1, value2)
```

or this:

```
AnotherName(item = value)
```

or this:

```
YetAnotherName(item1 = something(valueA, valueB, valueC))
```

or this:

```
SomeOtherName(item1 = value1, item2 = value2)
```

or this:

```
OneMoreName(item1 = value1, item2 = something(valueA, valueB))
```

or this:

```
OneMoreName(item1 = value1, item2 = [valueA, valueB, valueC])
```

To break this down in more detail; each instruction consists of:

- an **InstructionName** - which defines the type of instruction; and possibly
- zero, one or more properties that can be chosen for that instruction; all of these need to enclosed in a pair of curved brackets - *()*
- each property in turn can be given a value, or values, via the **=** sign.

This will probably be clearer if you look at the examples!

### *Named Instruction*

Some instructions can also be assigned to a **name** (a word that you make up), by using the equal (=) sign. This creates a "shortcut" to that instruction for use elsewhere in the script.

For example:

```
myname = NewName(value1, item1 = value2)
```

### *Types of Instructions*

There are three main types of instructions:

1. The most common instruction is that used to create page elements: text; shapes or images - see Script Elements
2. The other type is used for specifying the overall card and deck designs, and directing how and where the output should appear - see Defining Cards
3. The third type are obligatory instructions; *Create* and *Save* **must** appear at the start and end of every script respectively (see below).

### *Writing an Instruction*

**NOTE:** If an instruction is split over multiple lines, make sure that the split happens directly after a comma, not in the middle of a word. For example:

```
myname = NewName(value1,
                 item1 = value2)
```

and **NOT**

> **myname = NewName(value1, ite**
> m1 = value2)

### *ProperCase vs lowercase*

**pyprototypr**, like Python, is case-sensitive (unlike some computer languages or file names used in Windows). So:

```
Create()
```

is **NOT** the same as:

```
create()
```

If a script does not seem to work, the first thing to check is your case.

# Obligatory Instructions

## Create

The *Create* instruction is the first instruction that **must** appear in your script file, if you want the output to be created.

The *Create* instruction defines the characteristics or properties of the physical page on which the design will be created, as well the details of the output file.

In **pyprototypr** it is defined by the instruction:

```
Create(filename = *filename*,
       pagesize = *pagesize*,
       margin = *margin*,
       landscape = True|False,
       fonts= * fonts*
       color = *color*,
       units = *units*)
```

where:

- *fonts* - for example, [('Steelfish', 'steelfis.ttf')]
- *filename* - an optional value for the name of the output file to be created (defaults to *output.pdf*); this name must be wrapped in a pair of "
- *pagesize* - an optional value for the size of paper; this can be, for example, one of the A- or B-series used in the Metric system, as well as the Letter or Legal sizes used in the United States. The value is **NOT** wrapped in a pair of "". The default page size is *A4*.
- *margin* - an optional enclosure of (*top*, *left*, *bottom*, *right*), representing the measurements for the margin between the page edge and the cards. Usually only the *top*, *left* are specified, and **pyprototypr** will maximise the space on the remainder of the page.
- *landscape* - this is optional; it can either be set to True or False (Note there are **NO** parentheses around the words True or False). If True then the page is rotated (i.e. the normal height/width are swopped around).
- *color* - is the color of the page (defaults to white - **#FFFFFF**)
- *units* - an optional value for *units* for all measurements on the page; if omitted; the defaults will be used

## Save

The Save instruction is the last instruction that **must** appear in your script file, if you want the PDF file to be generated. It appears just as:

```
Save()
```

## Properties

Instruction properties are used to refine various aspects of an instruction. They can be:

- a single value; which might be a number (e.g. *15*) or some text (e.g. *"Hello World"*). *NOTE*: text is always enclosed in parentheses i.e. **" "**
- a simple property: i.e. a **name**, followed by an **=** and then a value
- a complex property: i.e. a name, followed by an **=** and then a series of values inside a pair of curved brackets **- ()**; called an enclosure (or, in Python terminology, a **tuple**)
- a series of properties (simple or complex), each one separated by a comma

This might sound more complex than it actually is; so looking at the various instructions (for example, see Graphical Elements) and their options will probably make more sense. Again, the APPENDIX II: Card Examples will also help you to understand how the instructions and their properties work.

## Defaults

In order to avoid you having to specify many "obvious" things, **pyprototypr** uses a set of default values for common properties. These are as follows:

- The default units are **cm**
- The default values for most length or position measurements is **1** (which corresponds to 1cm if you are using the default units)
- The default card size is **8.8** high and **6.3** wide which, assuming the default units of cm, corresponds to a standard Poker card size.
- The default page size for output to PDF is *A4*.
- All line thicknesses default to **0.1** (which corresponds to 1mm if you are using the default units)
- All line colors default to *black* (**#000000** in hexadecimal format)
- All fill (background) colors default to *white* (**#FFFFFF** in hexadecimal format)
- The default angle of rotation is zero (0) degrees

# Script Elements

## Element

An element is the basic "thing" that you use in **pyprototypr**. You will use it to create your design, but **pyprototypr** does not require that all (or any) of the available elements are actually used for any given design.

There are simple properties that are common to most elements, including:

- **size** - an optional enclosure of (*y*, *x*, *height*, *width*)
- **stroke** - an optional enclosure of (*color*, *thickness*, *style*)
- **fill** - an optional enclosure of (*color*, *transparency*, *style*)
- **rotate** - the number of degrees to rotate the shape (in a clockwise direction)
- **units** - an optional value for *units* for lines and measurements; if omitted; the defaults will be used

## Font Element

A font refers to the way text appears when it is printed or viewed.

In **pyprototypr** it is defined by the instruction:

```
Font(name, size, color)
```

where:

- **name** - is the name of the font
- **size** - is the size, in points, of that font (defaults to 12)
- **color** - is the color of the font (defaults to black - **#000000**)

## Text Element

Text is a set of characters, or symbols, that are used to convey information.

In **pyprototypr** it is defined by the instruction:

```
text(text="text", font=*Font*)
```

where:

- **"text"** - is a set of characters that are wrapped between two parentheses
- **Font** - is an optional Font instruction (see Font Element)

## Image Element

An image is a picture, made of up of pixels, and stored in "png" or "jpeg" format in a file.

In **pyprototypr** it is defined by the instruction:

```
Text("filename", size = *size*,)
```

where:

- **"filename"** - is the name of the file where the image is stored; the name must be wrapped between two parentheses. If the file is not in the directory as the script, provide the full path. An image can also be one that is stored on the web; provide the full URL for such an image.
- **size** - an optional enclosure of (*y*, *x*, *height*, *width*, *units*)

## Graphical Elements

**pyprototypr** allows you to create many different kinds of graphics: rectangles, circles, ellipses, stars, polygons and general shapes. Each one is constructed in similar ways, but obviously each may also have additional properties that peculiar to it.

### Rectangle

A rectangle is defined by the instruction:

```
rectangle(size = *size*,
          line = *line*,
          fill = *fill*,
          rounded = *rounding*,
          rotate = *angle*,
          units = *units*,
          rounded = True|False,
          pattern = *pattern*)
```

where:

- **rounded** - this is optional; it can either be set to True or False (no "" around the word). If True, then rounded corners will be created on each rectangle, proportional to its size.
- **rounding** - this is optional; it is a number representing the radius of the rounding curve for rounded corners that will be created on each rectangle
- **pattern** - the name of a file; e.g. http://elemisfreebies.com/11/07/20-abstract-patterns/ has various *.png* formatted images designed to create a seamless, repeating pattern.

### Circle

A circle is defined by the instruction:

```
circle(size = *size*,
       line = *line*,
       fill = *fill*,
       radius = *radius* ,
       units = *units*)
```

For a circle, the centre is given by the *y* and *x* values in the **size**.

For a circle, the *width* value (in the *size* enclosure) will be ignored i.e. for a circle the *width* and the *height* are the same - corresponding to the circle's diameter - and if both are given, only the *width* is used. A circle does not, of course, have a *rotation*.

### Ellipse

An ellipse is defined by the instruction:

```
ellipse(size = *size*,
        line = *line*,
        fill = *fill*,
        rotate = *angle*,
        units = *units*,
        spec = *spec*)
```

### *Star*

A star is defined by the instruction:

```
star(size = *size*,
     line = *line*,
     fill = *fill*,
     rotate = *angle*,
     units = *units*,
     spec = *spec*)
```

The optional **spec** is defined as an enclosure of (*tips*, *angle*, *raster*), where:

- **tips** - the number of points of the star (defaults to 5)
- **angle** - the interior angle of each vertex of the star (in degrees)

### *Polygon*

A regular polygon is defined by the instruction:

```
polygon(size = *size*,
        line = *line*,
        fill = *fill*,
        rotate = *angle*,
        units = *units*,
        sides = *sides*)
```

The optional **sides** is the number of sides of the polygon (defaults to 3). If the **sides** is not provided, the default polygon will be an equilateral triangle.

If the polygon shows with the point to the top of the card, then it can be rotated, using the **rotate**, so that the flat side is parallel to the top. The angle of rotation can be calculated by: 360° divided by (sides x 2). For example, an octagon will need to be rotated by an angle of 22.5°.

### *Shape*

An irregular shape is defined by the instruction:

```
shape("shape",
      line = *line*,
      fill = *fill*,
      rotate = *angle*,
      units = *units*,
      points = *points*)
```

The required **points** is defined as an enclosure of **(x, y)** point enclosures, each point enclosure separated by commas. There must be a minimum of three such points in order to construct a shape; this would appear as a triangle. (As an aside, if all the points are in straight line, then the shape will **appear** to be a regular line.)

## Sub-Elements

There are some other elements that are not displayed directly in the output; they usually appear as "sub-elements" in many other elements, and it is useful to see how they are defined, as this will make it easier to create such elements and understand references to them.

NOTE: This section assumes you have already read and understood the Basic Concepts.

### *Line*

A line is defined with three properties: *color*, *width* and *style*.

### *Fill*

The *fill* is the nature of the area inside of a boundary of a shape (e.g the inner part of a circle).

A fill is defined with three properties: *color*, *transparency* and *style*

### *Size*

The *size* for a card element is defined as a combination of its **position** (*y* and *x* values) and **extent** (*height* and *width*).

# Putting it Altogether: The Script

## Do's and Don'ts

The order of instructions in the script is important. If you want to use a reference, then that must have been defined on a line prior to the instruction that uses it.

Also, no line may have any blank spaces at the start, unless its a continuation of an instruction, in which there **must** be one or more spaces at the start.

Don't give your script a filename like:

- pyprototypr.py
- reportlab.py
- draw.py
- shapes.py

## Reusing Instructions

It can be very useful to reuse instructions - see Re-using Properties.

# Defining Cards

## Deck

The Deck design is the key underlying pattern that determines the basic context for your set of cards. All cards will share this same design, and then individual cards (or ranges of cards) can be customized further using the Card instruction (see below).

A deck design is defined by the instruction:

```
Deck(cards = *count*,
     height = *height*,
     width = *width*,
     units = *units*,
     line = *stroke*,
     fill = *fill*,
     grid_markers = True|False)
```

where:

- **count** - an integer value for the total number of cards to be created
- **height** - an optional number for *height*; if omitted; the defaults will be used
- **width** -an optional number for *width*; if omitted; the defaults will be used
- **units** - an optional value for *units*; if omitted; the defaults will be used
- **stroke** - an optional enclosure of (*color*, *thickness*, *style*) for the card border. If the values for *thickness* and *style* are omitted; the defaults will be used, but if you need to specify *style*, you must also specify a value for *thickness*
- **fill** - usually a *color*; either a named color or a hexadecimal color.
- **grid_markers** - this is optional; it can either be set to True or False (no "" around True or False). If True then small lines will drawn extending from the edge of the page inwards for 5mm (one-fifth of an inch) in line with the tops and sides of the cards. These help when cutting the cards.

optional:

- **rounded** - this is optional; it can either be set to *True* or *False* (note that there is no "" around True or False). If True, then rounded corners will be created on each card, proportional to the card size.

## Card

Once a deck has been created, individual cards, or sets of cards, can be customized by adding shapes, text or images to them. This is done with a Card instruction . The instruction is specified as follows:

```
Card(*range*, *elements*)
```

where:

- **range** - a set of numbers, corresponding to cards in the deck. A range may be: a single number; a list of single numbers (or ranges), each separated by a comma (,); two numbers, corresponding to the first and last cards in a continuous sequence, separated by a dash ("-"); or two numbers, corresponding to the first card followed by a card count, separated with a hash sign "#".

- **elements** - one or more card elements; for example, a Shape or Text or Image - see Script Elements for how to create these. This element may already been created in a previous line in the script, in which case, you can use it via a named instruction.

# Advanced Usage

## Re-using Properties

The *Common()* instruction allows you to define a set of properties that can be re-used in more than one place.

## Doing Similar Things More than Once

A **loop** is a way to make a script do the same thing more than once.

**For** loops are used when you have one or commands (each on a separate line in your script) which you want to repeat multiple times.

The repeats are controlled by the *range()* function. This typically looks like:

```
range(1, 4)
```

i.e. the brackets contain two numbers. The *range* goes through numbers from the first to the last, BUT not including the last, counting in steps of 1 (as though it was in junior school!). So this range starts at one and stops at three; not four, as you might expect.

Below is an example of a loop that causes DoSomething() be executed three times: the first time, "x" has a value of 1, the second time "x" has a value of 2, and the third - and last - time, it has a value of 3:

```
for x in range(1, 4):
    DoSomething()
```

A more helpful example might that of drawing 20 lines, one above each other, on a page:

```
for line in range(1, 21):
    Line(x=1, x1=18, y=line, y1=line)
```

In this example, the two values of *y* are both set to the value of *line* (they are made the same because this is a horizontal line). There will be a gap of 1cm between each line. But what if you wanted a larger gap? To do this, you can add a third number to the range - called the *step*. It is the amount added each time the loop executes. For example:

```
for line in range(1, 21, 2):
    Line(x=1, x1=18, y=line, y1=line)
```

will draw only 10 lines - at position 2cm apart.

# Other Options

## But I also want to...

Clearly, if you are a Python programmer, you can alter the source code to make **pyprototypr** behave in any manner you want. I would ask that you contact me if you have ideas or specific code that you would like included in future versions, as this will help make it more useful for all. I can't promise to support every idea or feature but will do my best.

## Changing Defaults

Advanced users can change the **pyprototypr** defaults by editing the correct section of the source code (see the **DEFAULTS** dictionary).

New paper types can be added to the **PAPER_SIZES** dictionary.

## Learning Python

Maybe (just maybe?!) **pyprototypr** has intrigued you enough to decide you want to learn more about Python and how to "do stuff". A really, really quick and readable introduction is provided by Magnus Lie Hetland at http://hetland.org/writing/instant-hacking.html You can also follow the short course at ActiveState - http://docs.activestate.com/activepython/2.7/easytut/node3.html

## Coming up After the Break

Some ideas for *possible* inclusion in future versions of **pyprototypr** are:

- new basic shapes e.g. "pie slices"; "curves"
- tiling of elements
- expressing card units as percentages of a card size
- expressing element units as percentages of page size
- command-line syntax

# Contact

For more information or help with **pyprototypr**, please email Derek at gamesbook@gmail.com - I will do my best to respond timeously. If you use the word **pyprototypr** in the header, my email will send an auto-response.

I welcome all suggestions for improvements that follow the Python approach... but cannot promise that I will have the time or ability to implement them!

# APPENDIX I: Design Approach

## Why Python?

Basing this type of program on a language that is already designed for scripting makes sense; because then you (even as an amateur programmer) can readily add in your own functions and logic without having to change the **pyprototypr** code itself.

In addition, Python has numerous advantages as a primary programming language:

- Open source
- Well-documented
- One of the easiest programming languages to use
- Very portable (across all operating systems)
- Extensive built-in library, and *massive* range of third-party libraries

## Follow the Python approach...

The design approach to **pyprototypr** tries to follow the 'zen' of Python design:

*Simple is better than complex*

Adding too many options or too much functionality to a program does not make it easier. The commands to be used should be few, memorable and, hopefully, obvious. Use of defaults should enable the program to be used without "falling over". The program should provide helpful and useful feedback in case of errors.

Furthermore:

*Explicit is better than implicit*

*Readability counts*

e.g.:

```
Ellipse(5,14,15,23,2.5,tan,green)
```

vs.:

```
Ellipse(x=5, y=14, x1=15, y1=23, radius=2.5, fill=tan, stroke=green)
```

The second example is more "wordy" and takes you longer to type; but its obvious what the numbers refer to, and then its easy, when reading this script some months later, to see what is meant to happen.

# APPENDIX II: Card Examples

> *NOTE*: **If you "copy and paste" the example scripts directly from a PDF document into a text file, then they may not work, because the spaces in front of some of the lines may need to be removed!**

## A simple card deck example: Take 1

Open up a new text file with a text file editor (on Windows, use NotePad, on OS X, use TextEdit - but do **not** use a word processor such as "Word", "Pages" or "LibreOffice") and type the following, making sure that you start each line without any blank spaces at the start.

```
from pyprototypr.draw import *
Create()
Deck(cards=9)
Save()
```

Create tells **pyprototypr** to define the output PDF file in which this deck of 9 cards will be saved. Because no further information is given, the defaults for sizes and colors are used, as well as the default page size of A4. Deck means that **pyprototypr** defines a deck of 9 cards, again with default size. Save gives the go-ahead to create the resulting file.

Now save the text file, for example, as `cards1.py`. Then open a command-line window (see under Testing that Python is installed) and change to the directory where the file is installed. Type the following:

```
python cards1.py
```

An output PDF file should now have been created, in the same directory as your `cards1.py` file, called `cards1.pdf`. If you open this in a PDF reader program, you should see that it contains a set of 9 blank, poker-card sized, rectangular outlines (which we are calling "cards") laid out in a grid on an A4-sized page (*A4* being the default page size for **pyprototypr**).

## A simple card deck example: Take 2

Open up a new text file with a text file editor (on Windows, use NotePad, on OS X, use TextEdit) and type the following:

```
from pyprototypr.draw import *

Create(pagesize=A3,
       filename="example2.pdf")
Deck(cards=9)
Save()
```

You can see that the Create instruction has now been expanded with new properties (the items appearing in brackets). A property is just a name, followed by an "=" sign, and then a value of some type.

In this case, the page size property has been set to *A3* (note, no "" around the A3), and a specific file name has been chosen for the output PDF.

The Create instruction is split over multiple lines to make it easier to read; but you need to make sure that the split happens directly after a comma, and not in the middle of a word. Also make sure that there are one or spaces at the start of those continuation lines.

A blank line has been added before the Create instruction. Adding blank lines helps make your file more readable, but **pyprototypr** will not use them.

Now save this new file, for example, as `cards2.py`. Now open a command-line window (see Testing that Python is installed) and change to the directory where the file is installed. Type the following:

```
python cards2.py
```

An output PDF file should now have been created in the same directory as your `cards2.py` file - it will be called `example2.pdf`. It should contain a set of 9 blank cards appearing near the bottom corner on one A3-sized page.

## A simple card deck example: Take 3

If you have followed the above examples, you will know how to create the cards file, and how to create and display the output PDF file. This example will therefore only show the text in the file you create, and discuss what the resulting output should be.

Create this text in a file called `cards3.py`:

```
from pyprototypr.draw import *

Create(filename='example3.pdf', offset=0.5)

# deck design - a "template" that all cards will use
Deck(cards=50,
     fill="#702EB0",
     height=5,
     width=3.8)

# create the output card file, using the card 'deck'
Save()
```

A Deck instuction allows you to define the details for every card that will appear in the deck, such as its height, width and fill color.

In this script, the lines starting with a **#** are comments and will be ignored by **pyprototypr**. The comments are included to provide some more explanation as to what the next line, or lines, are doing.

The resulting `example3.pdf` will show two pages of small, blank, purple cards, approximately 2 inches by 1.5 inches, with 25 cards per page, for a total of 50 cards. Note that the **pyprototypr** will calculate how many cards will fit on page to make up the total number of cards for the deck.

## A simple card deck example: Take 4

This example will only show the text in the file you create, and discusses what the resulting output should be.

Create this text in a file called `cards4.py`:

```
from pyprototypr.draw import *

# create the output card file
Create(filename='example4.pdf', offset=0.5)

# create a deck design
Deck(cards=25,
     fill=skyblue,
     stroke=white,
     height=5,
     width=3.8)

# create some text, with the default font, and centre it at a location
mytext = text(text="25!", point=(1.9,1.0))

# customize a specific card (number 25) in the deck with 'mytext'
Card("25", mytext)

# specify a particular font; face and size and color
times = Font("Times New Roman", size=8, color="red")

# create more text, and display it using 'times' font
mytext2 = text(text="I'm on cards 1-10", font=times, x=1.9)

# specify a range of cards to contain 'mytext2'
Card("1-10", mytext2)

# save to file
Save()
```

This script also shows the use of a reference: a reference is just a name, followed by an "=" sign, and then an instruction. You can see that the *mydesign* reference is used further on when specifying the design for a Card; this requires a card number (or numbers) followed by the details of what is required for the card - in this case, the text stored in *mydesign*.

The resulting `example4.pdf` file will show a page of small, white-bordered, light-blue cards - with the same text appearing on cards one to ten, but different text on card number 25 (twenty-five). **Note** that cards are displayed from the bottom-left upwards; that is why the first cards appear on the bottom rows and the last card is on the top-right.

# APPENDIX III: Python Basics

## Letters and Numbers

In Python, "letters" are referred to as characters (or 'char' for short), while numbers can either be integers ('int' for short) which are whole numbers, or floating-point numbers ('float' for short) which are numbers that have fractional or decimal components.

What in English would be called a "word", in Python is referred to as a "string".

## Variables and Variable Names

A variable is "shortcut" name that can be used to store things such as numbers or strings; or the results of some calculation or operation. They are useful when you need to refer to the same "thing" in different places in the script.

Variable names can be arbitrarily long. They can contain both characters and numbers, but they have to begin with a character.

Python also has **keywords** that it uses to recognize the structure of a program, and these *cannot* be used as variable names.

The list of Python keywords is:

```
False               def                 if                  raise
None                del                 import              return
True                elif                in                  try
and                 else                is                  while
as                  except              lambda              with
assert              finally             nonlocal            yield
break               for                 not
class               from                or
continue            global              pass
```

For more reading, see https://www.tutorialspoint.com/python/python_variable_types.htm

## Indenting Lines

Indentation refers to the spaces at the beginning of a line.

In some programming languages the indentation is for "readability", but in in Python the indentation is very important.

Python uses indentation to indicate a block of code; that is a set of lines that all operate in sequence to achieve something.

You must use the same number of spaces in the same block of code, otherwise you get an error. Python programs typically use 4 spaces throughout.

## Functions

Writing functions requires more understanding of how Python - and programming languages in general - works. There are many online tutorials and books that cover this subject. Some examples are:

- https://www.programiz.com/python-programming/function
- https://www.tutorialspoint.com/python/python_functions.htm

- https://www.w3schools.com/python/python_functions.asp

## More Reading

- http://npppythonscript.sourceforge.net/docs/latest/pythonprimer.html
- http://www.informit.com/articles/article.aspx?p=2163338&seqNum=2
- http://greenteapress.com/thinkpython/html/thinkpython003.html#toc14

# APPENDIX IV: Key Definitions

The following definitions are used, or referred to, in various parts of this document.

## Command

A command is an ability associated with as instruction. It typically looks like:

```
Instruction.command(...)
```

where the *command* is replaced by the name of the specific command being used.

## Comments

A line in a **pyprototypr** file that starts with the **#** character is called a "comment". This line will be ignored by **pyprototypr**, but is something you want to appear, probably as a reminder or explanation to yourself or someone else.

## Enclosure

An *enclosure* is defined a set of values surrounded by curved brackets - *()*. (In Python terminology, this is called a *tuple*).

## Reference

A reference is a way of creating a "shortcut" to an instruction, thereby allowing it to be easily re-used. (In Python, such references are typically called *variables*.)

## Style

Style refers to how an element appears. Different types of things can have different types of styles. For example, a line can appear as single unbroken path; or it could be a series of dots or a series of dashes.

## Transparency

Transparency refers to how much an element is "faded" to allow an underlying area or item to "show through" it.

# APPENDIX V: Trouble Shooting

## Common Errors

Computers are fussier about the way programs are written than your strictest English grammar teacher! In some cases, the errors they complain about are not always obvious to solve...

In running a script, you may encounter errors such as following:

```
File "cards_design.py", line 22, in <module>
  Card("7-9", r1 l1)
                   ^
SyntaxError: invalid syntax
```

This is where you have left out a comma after the "r1" (the caret symbol "^" tries to point to the place where it thinks the error is happening but it may not always be accurate).

Another example:

```
File "cards_design.py", line 22, in <module>
  Card("7-9", r1, l)

NameError: name 'l' is not defined
```

Here, you may have defined a name "l1", but you have forgotten to add the "1" when referring to it - using "l" and "1" together is not really a good idea in a program, as they look too similar.

In this example:

```
File "cards_design.py", line 10, in <module>
  height=8.8, width=6.3, rounding=0.3, fill=iviry)

NameError: name 'iviry' is not defined
```

You are trying to use the color "ivory" but have mis-spelt the word.

Here:

```
File "basic.py", line 103
  ^
SyntaxError: EOF while scanning triple-quoted string literal
```

You have closed off a comment section (**'''**, which is what Python calls a triple-quoted string) without opening it first.

## Further Help

http://greenteapress.com/thinkpython/html/thinkpython002.html#toc6

# APPENDIX VI: Working with pyenv

Below is a **very** abbreviated guided to installing and using *pyenv* on an Ubuntu desktop. For more details, it is recommended that you consult the documentation for the package.

## Prerequiste

Run:

```
sudo apt-get install curl git-core gcc make zlib1g-dev libbz2-dev \
libreadline-dev libsqlite3-dev libssl-dev
```

## Install

Run:

```
cd ~

git clone https://github.com/yyuu/pyenv.git ~/.pyenv

echo 'export PYENV_ROOT="$HOME/.pyenv"' >> ~/.bashrc
echo 'export PATH="$PYENV_ROOT/bin:$PATH"' >> ~/.bashrc
echo 'eval "$(pyenv init -)"' >> ~/.bashrc

git clone https://github.com/yyuu/pyenv-virtualenv.git \
~/.pyenv/plugins/pyenv-virtualenv
echo 'eval "$(pyenv virtualenv-init -)"' >> ~/.bash_profile

exec $SHELL
source .bashrc
```

## Usage

To use *pyenv*, for example, in an environment called *myenv*:

```
cd ~
# get correct Python version
pyenv install 3.9.11
# make new virtualenv called "myvenev"
pyenv virtualenv 3.9.11 myvenev
#list existing virtualenvs
pyenv virtualenvs
#start using a virtualenv
pyenv activate myvenev
#stop using a virtualenv
pyenv deactivate myvenev
```