boxes.py Documentation

Release 0.1

Florian Festi

Aug 12, 2020

Contents

1 About Boxes.py 3

2 Installation 5

3 Using Boxes.py 15

4 Contributing to Boxes.py 19

5 Using the Boxes.py API 23

6 All Box Generators 47

7 Indices and tables 49

Index 51

Create boxes and more with a laser cutter!

Contents:

CHAPTER 1

About Boxes.py

- Boxes.py is an online box generator
  - https://www.festi.info/boxes.py/index.html
- Boxes.py is an Inkscape plug-in
- Boxes.py is library to write your own
- Boxes.py is free software licensed under GPL v3+
- Boxes.py is written in Python and runs with Python 3

Boxes.py comes with a growing set of ready-to-use, fully parametrized generators. See https://florianfesti.github.io/boxes/html/generators.html for the full list.

1.1 Features

Boxes.py generates SVG images that can be viewed directly in a web browser but also postscript and - with pstoedit as external helper - other vector formats including dxf, plt (aka hpgl) and gcode.

Of course the library and the generators allow selecting the "thickness" of the material used and automatically adjusts lengths and width of joining fingers and other elements.

The "burn" parameter compensates for the material removed by the laser. This allows fine tuning the gaps between joins up to the point where plywood can be press fitted even without any glue.

Finger Joints are the work horse of the library. They allow 90° edges and T connections. Their size is scaled up with the material "thickness" to maintain the same appearance. The library also allows putting holes and slots for screws (bed bolts) into finger joints, although this is currently not supported for the included generators.

Dovetail joints can be used to join pieces in the same plane.

Flex cuts allows bending and stretching the material in one direction. This is used for rounded edges and living hinges.

1.2 Documentation

Boxes.py comes with Sphinx based documentation for usage, installation and development.

The rendered version can be viewed at <https://florianfesti.github.io/boxes/html/index.html>.

CHAPTER 2

Installation

Boxes.py is a pure Python project that does support the regular setuptools method of shipping with `setup.py`. `setup.py --help-commands` and `setup.py CMD --help` provide the necessary documentation for building, installing or building binary formats.

2.1 Requirements

2.1.1 Affine

`Affine` (package name may be `python-affine` or `python3-affine`) is used for vector calculation.

2.1.2 Markdown

`Markdown` (package name may be `python-markdown` or `python3-markdown`) is used to format the description texts.

2.1.3 LXML

`lxml` (package name may be `python-lxml` or `python3-lxml`) is needed for the Inkscape plugin.

2.1.4 setuptools

Setup.py uses the `setuptools` library (package name may be `python*-setuptools`). You only need it if you want to build the package.

2.1.5 ps2edit

While not a hard requirement Boxes.py uses `ps2edit` to offer formats that are not supported by Cairo: DXF, gcode, PLT. Currently the location Boxes.py looks for `ps2edit` is hard coded to `/usr/bin/pstoedit` in the `boxes.formats.Formats` class.

2.1.6 Python

Boxes.py is implemented in Python 3. It used to work on Python 2.7, too. But with the Python 2 approaching end of life support has been dropped.

2.1.7 Sphinx

For building the documentation locally you need the *Sphinx* documentation generator (package name may be python-sphinx or python3-sphinx). It is not needed for anything else. Boxes.py can be run and changed just fine without.

2.2 Running from working dir

Due to lazy developer(s) Boxes.py can also run from the Git checkout. The scripts in `scripts/` are all suppossed to just work right after `git clone`. The Inkscape needs a bit manual work to get running. See below.

2.3 Inkscape

As binary

Boxes.py can be used as a set of Inkscape plugins. The package does install the necessary .inx files to `/usr/share/inkscape/extensions` on unix operating systems. The .inx files assume that the `boxes` executable is available in the path (which it is when installing the binary package)

git repository easy way

After cloning it may be most convenient to generate the .inx files right in place by executing `scripts/boxes2inkscape` with the taget path as only parameter.

- global: `scripts/boxes2inkscape /usr/share/inkscape/extensions/`
- userspace: `scripts/boxes2inkscape ~/.config/inkscape/extensions/`

On non unix operating the target directories may differ. You can look up the directories *"User extensions"* and *"Inkscape extensions"* within the Inkscape preferences *Edit -> Preferences... -> System*.

git repository manual way

`setup.py build` creates the `*.inx` files in the `inkex/` directory.

They then have to be copied in either the global or the per user extension directory of Inkscape. These are `/usr/share/inkscape/extensions/` and `~/.config/inkscape/extensions/` on a unix operating system. On non unix operating the target directories may differ. You can look up the directories *"User extensions"* and *"Inkscape extensions"* within the Inkscape preferences *Edit -> Preferences... -> System*.

As an alternative you can create a symlink to the `inkex/` directory within the desired inkscape extension directory.

2.4 Platform specific instructions

2.4.1 macOS

It is recommended to use Homebrew to install the dependencies for Boxes.py. See brew.sh on how to install Homebrew.

General

1. Install Python 3 and other dependencies:

   ```
   brew install python3 git
   ```

   Optional:

   ```
   brew install pstoedit
   ```

2. Install cairio:

   ```
   brew install pkg-config
   ```

3. Install required Python modules:

   ```
   pip3 install Markdown lxml affine
   ```

4. Download Boxes.py via Git:

   ```
   git clone https://github.com/florianfesti/boxes.git
   ```

5. Run Boxes.py:

   Local web server on port 8000:

   ```
   ./scripts/boxesserver
   ```

   Command line variant (CLI):

   ```
   ./scripts/boxes
   ```

System-wide with Inkscape extension

To install Boxes.py system-wide with the Inkscape extension, following steps are required:

1. Install Inkscape with Homebrew Cask (requires XQuartz):

   ```
   brew cask install inkscape
   ```

2. From the root directory of the repository, run:

   ```
   ./setup.py install
   ```

3. Now `boxes` and `boxesserver` can be executed like other commands and the Inkscape extension should be available.

Troubleshooting

When using the Inkscape extension something like the following error might occur:

```
Traceback (most recent call last):
  File "/Users/martin/.config/inkscape/extensions/boxes", line 107, in <module>
    main()
  File "/Users/martin/.config/inkscape/extensions/boxes", line 47, in main
    run_generator(name, sys.argv[2:])
  File "/Users/martin/.config/inkscape/extensions/boxes", line 73, in run_generator
    box.close()
  File "/usr/local/lib/python3.7/site-packages/boxes-0.1-py3.7.egg/boxes/__init__.py",
↪ line 594, in close
    svgutil.svgMerge(self.output, self.inkscapefile, out)
  File "/usr/local/lib/python3.7/site-packages/boxes-0.1-py3.7.egg/boxes/svgutil.py",␣
↪line 144, in svgMerge
    from lxml import etree as et
ImportError: dlopen(/Applications/Inkscape.app/Contents/Resources/lib/python2.7/site-
↪packages/lxml/etree.so, 2): Symbol not found: _PyBaseString_Type
  Referenced from: /Applications/Inkscape.app/Contents/Resources/lib/python2.7/site-
↪packages/lxml/etree.so
  Expected in: flat namespace
```

This is because Inkscape on macOS ships its own version of Python 2.7 where `lxml` and other dependencies are missing.

A workaround is to edit the file at `/Applications/Inkscape.app/Contents/Resources/bin/inkscape`. At line 79 there should be following code:

```
export PYTHONPATH="$TOP/lib/python$PYTHON_VERS/site-packages/"
```

which needs to be changed to

```
#export PYTHONPATH="$TOP/lib/python$PYTHON_VERS/site-packages/"
```

This forces Inkscape to use the Python version installed by Homebrew which has all the necessary dependencies installed.

Note: This might break other extensions. In this case simply change the line back and restart Inkscape.

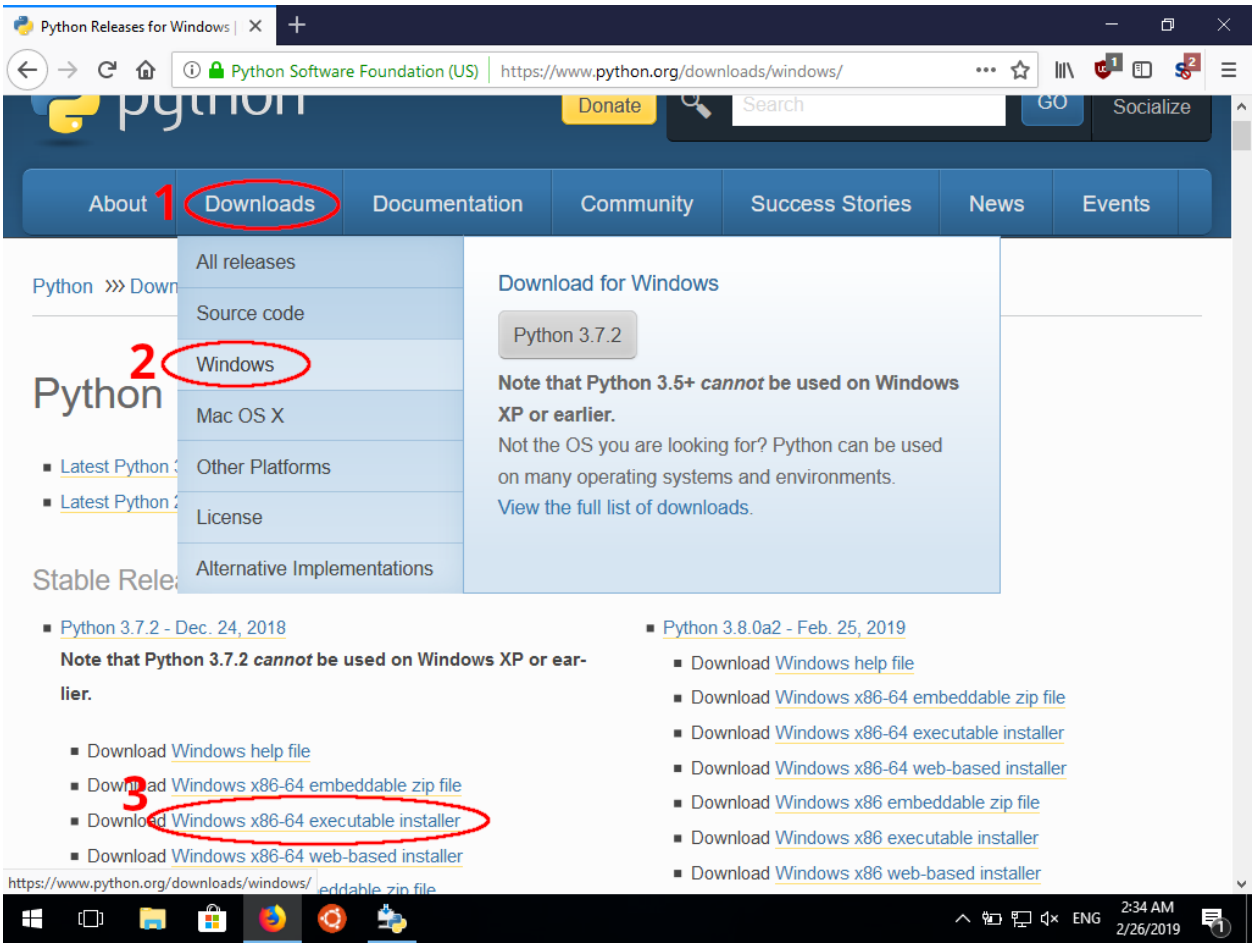
2.4.2 Windows

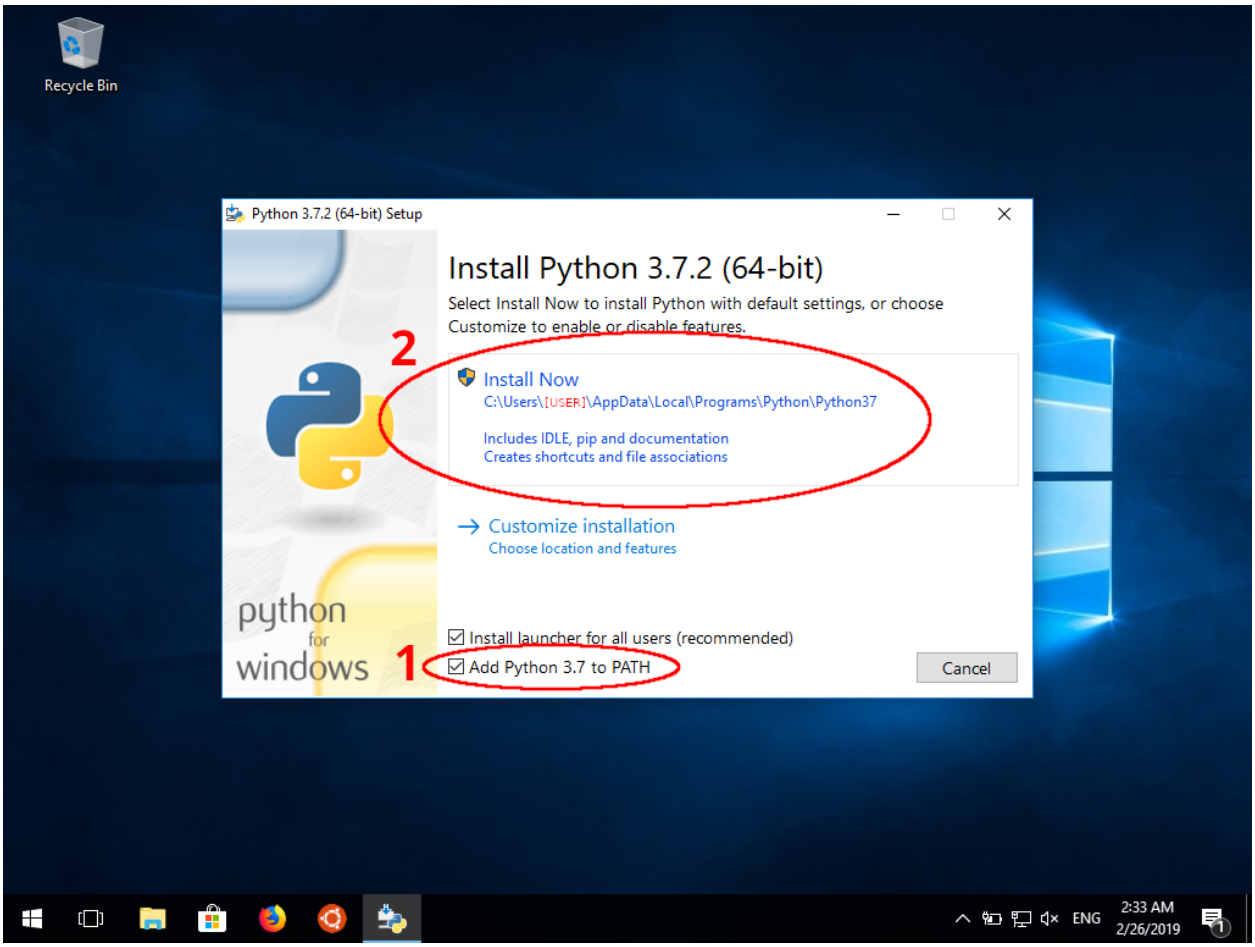
Getting the Inkscape plugins to run will likely need manual installation (see above). Note that Inkscape may come with its own Python. If you run into trouble or have better installation instructions please open a ticket on GitHub.

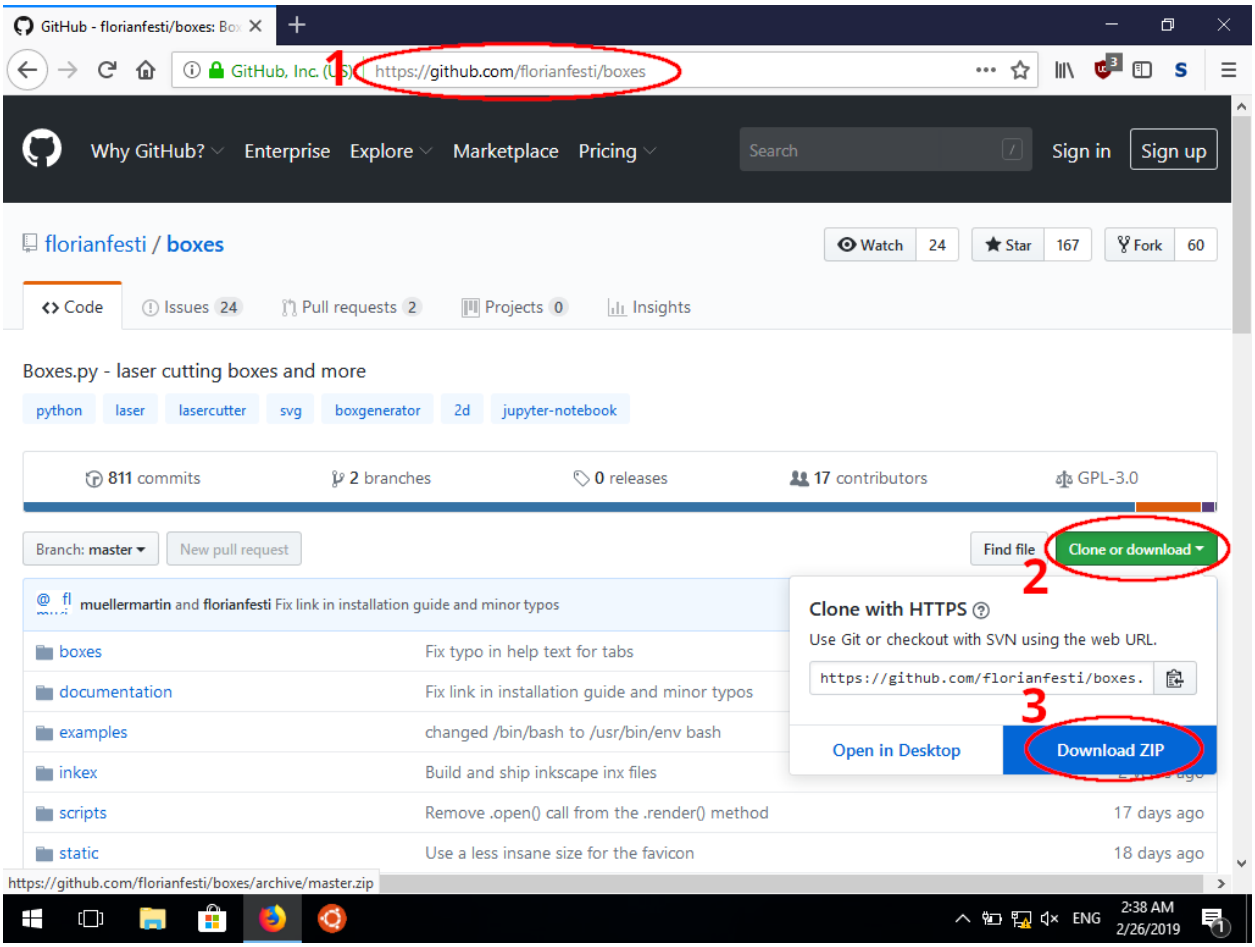
Native

Following steps are known to work under Windows 10 (64-bit):

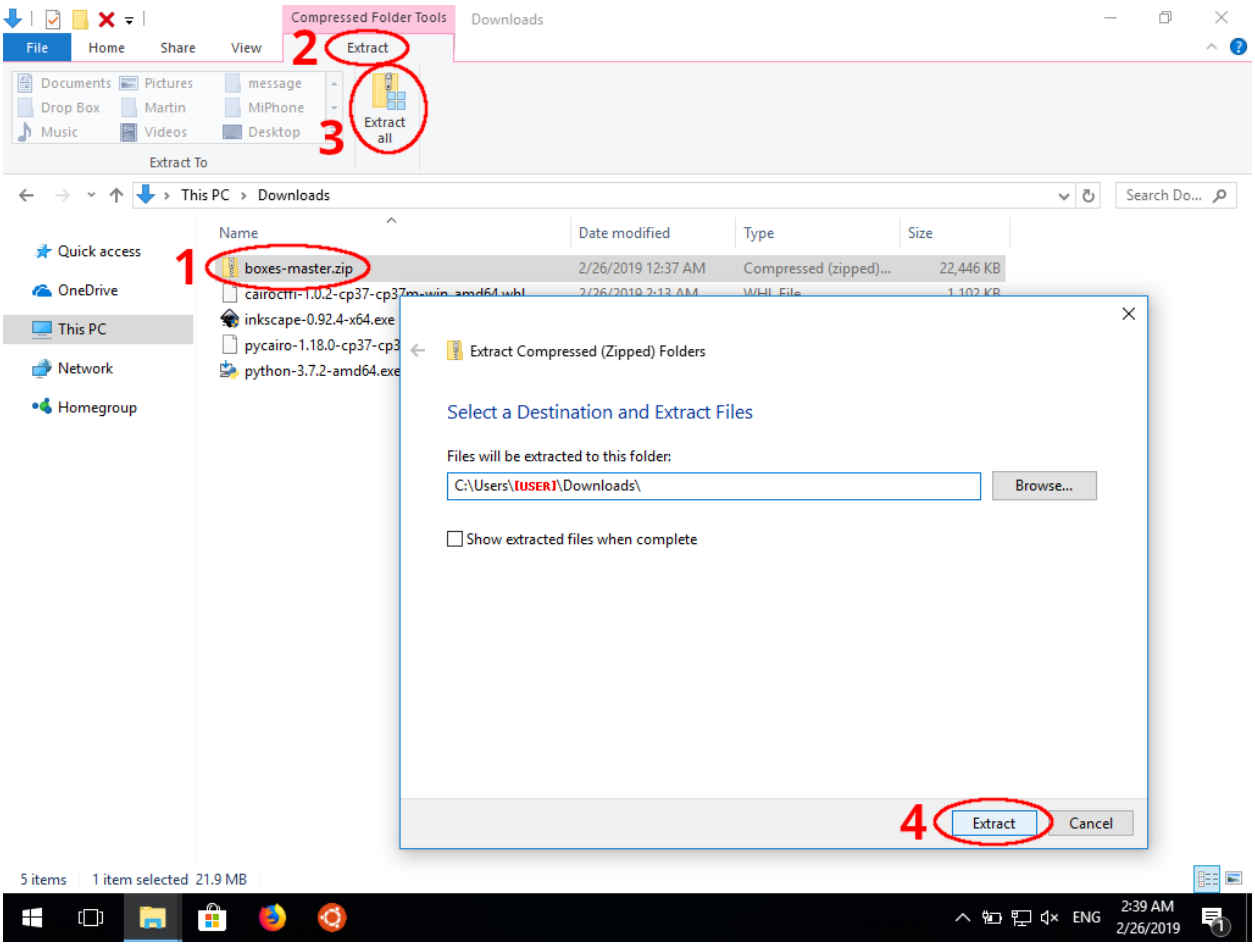
1. Go to https://www.python.org/downloads/windows/ and download the "Windows x86-64 executable installer" for Python 3.7
2. Install Python 3.7 and make sure to check "Add Python 3.7 to PATH" while doing so
3. Run the command `pip install Markdown lxml affine` (Note: If the command pip is not found, you probably forgot to add the Python installation to the PATH environment variable in step 2)


Python Releases for Windows
Python Software Foundation (US)
https://www.python.org/downloads/windows/
Donate
Search
GO
Socialize
About
1
Downloads
Documentation
Community
Success Stories
News
Events
All releases
Source code
Windows
Mac OS X
Other Platforms
License
Alternative Implementations
Download for Windows
Python 3.7.2
Note that Python 3.5+ cannot be used on Windows XP or earlier.
Not the OS you are looking for? Python can be used on many operating systems and environments.
View the full list of downloads.
2
Python
Python 3.7.2 - Dec. 24, 2018
Note that Python 3.7.2 cannot be used on Windows XP or earlier.
Download Windows help file
Download Windows x86-64 embeddable zip file
3
Download Windows x86-64 executable installer
Download Windows x86-64 web-based installer
Python 3.8.0a2 - Feb. 25, 2019
Download Windows help file
Download Windows x86-64 embeddable zip file
Download Windows x86-64 executable installer
Download Windows x86-64 web-based installer
Download Windows x86 embeddable zip file
Download Windows x86 executable installer
Download Windows x86 web-based installer
https://www.python.org/downloads/windows/
ENG
2:34 AM
2/26/2019


Recycle Bin
Python 3.7.2 (64-bit) Setup
Install Python 3.7.2 (64-bit)
Select Install Now to install Python with default settings, or choose
Customize to enable or disable features.
2
Install Now
C:\Users\[USER]\AppData\Local\Programs\Python\Python37
Includes IDLE, pip and documentation
Creates shortcuts and file associations
Customize installation
Choose location and features
python
for
windows
1
Install launcher for all users (recommended)
Add Python 3.7 to PATH
Cancel
ENG
2:33 AM
2/26/2019


GitHub - florianfesti/boxes: Bo
1
GitHub, Inc. (US)
https://github.com/florianfesti/boxes
Why GitHub?
Enterprise
Explore
Marketplace
Pricing
Search
Sign in
Sign up
florianfesti / boxes
Watch 24
Star 167
Fork 60
Code
Issues 24
Pull requests 2
Projects 0
Insights
Boxes.py - laser cutting boxes and more
python
laser
lasercutter
svg
boxgenerator
2d
jupyter-notebook
811 commits
2 branches
0 releases
17 contributors
GPL-3.0
Branch: master
New pull request
Find file
Clone or download
2
muellermartin and florianfesti Fix link in installation guide and minor typos
boxes
Fix typo in help text for tabs
documentation
Fix link in installation guide and minor typos
examples
changed /bin/bash to /usr/bin/env bash
inkex
Build and ship inkscape inx files
scripts
Remove .open() call from the .render() method
17 days ago
static
Use a less insane size for the favicon
18 days ago
Clone with HTTPS
Use Git or checkout with SVN using the web URL.
https://github.com/florianfesti/boxes.
3
Open in Desktop
Download ZIP
https://github.com/florianfesti/boxes/archive/master.zip
ENG
2:38 AM
2/26/2019

4. Download Boxes.py as ZIP archive from GitHub
5. Extract the ZIP archive (e.g. via the built-in Windows feature or other tools like 7-Zip)

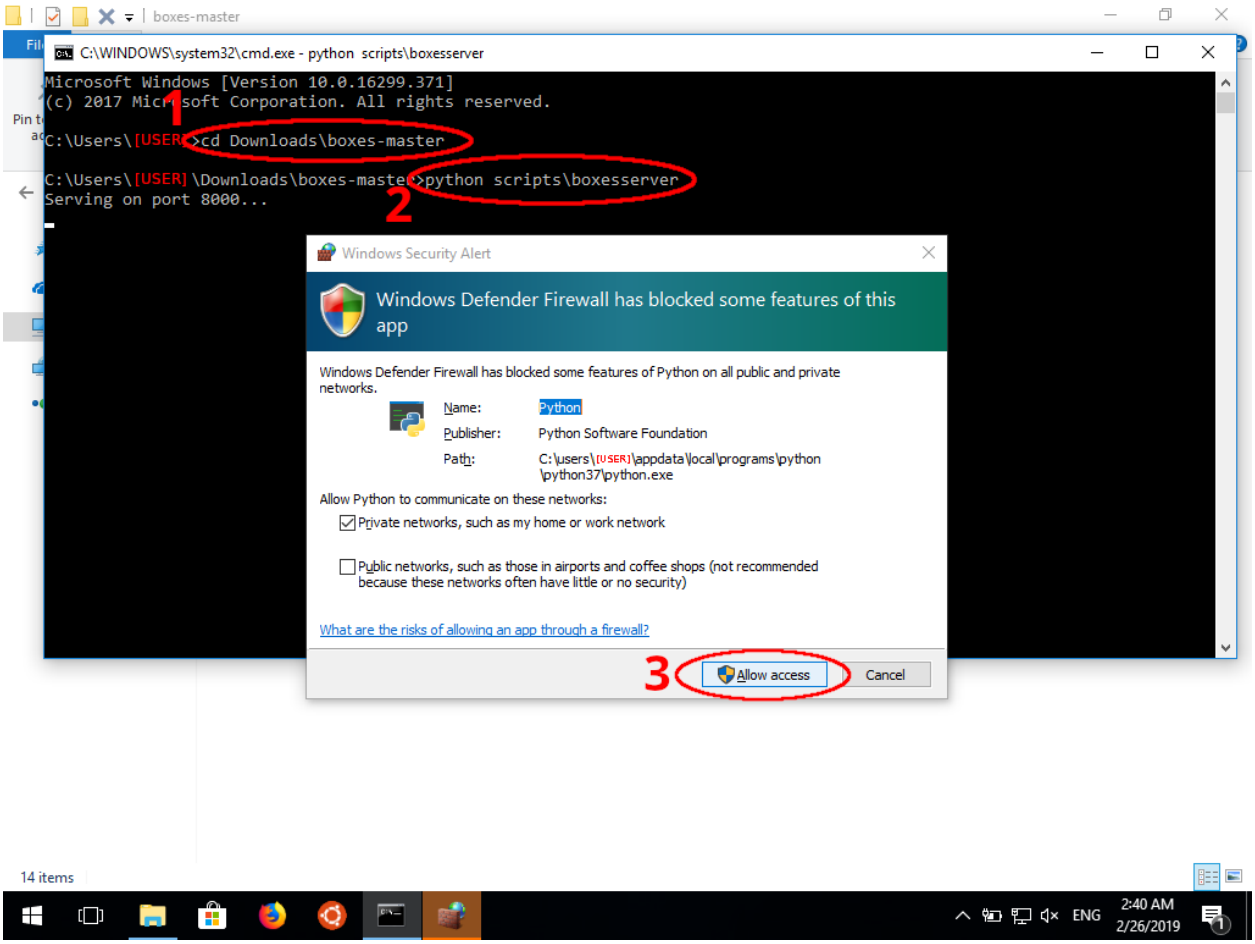


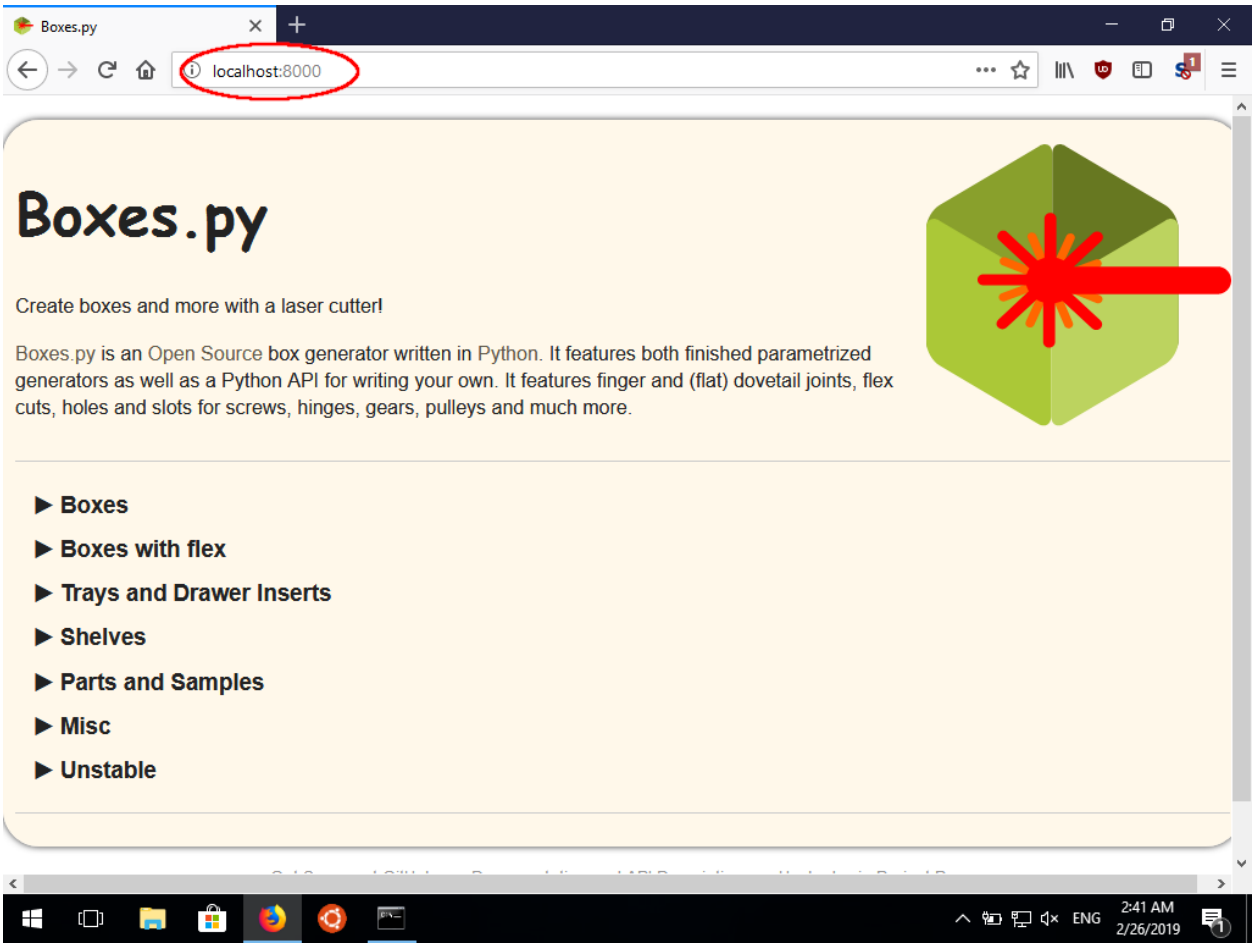
6. **Change into the folder for Boxes.py,** e.g. with the command `cd \Users\[USERNAME]\Downloads\boxes-master`
7. **Run the development server with the command** `python scripts\boxesserver` Note: You likely will be notified by your firewall that it blocked network access. If you want to use boxesserver you need to allow connections.
8. Open the address http://localhost:8000/ in your browser and have fun :)

Additionally the command line version of Boxes.py can be used with the command `python scripts\boxes`.

Windows Subsystem for Linux

Another way of installing Boxes.py on Windows is to use the Windows Subsystem for Linux (WSL). This requires newer versions of Windows 10. Once it is installed (e.g. via the Ubuntu App from the Microsoft Store), the installation is identical to the installation on Linux systems.


boxes-master
C:\WINDOWS\system32\cmd.exe - python scripts\boxesserver
Microsoft Windows [Version 10.0.16299.371]
(c) 2017 Microsoft Corporation. All rights reserved.
C:\Users\[USER]>cd Downloads\boxes-master
C:\Users\[USER]\Downloads\boxes-master>python scripts\boxesserver
Serving on port 8000...
1
2
Windows Security Alert
Windows Defender Firewall has blocked some features of this app
Windows Defender Firewall has blocked some features of Python on all public and private networks.
Name: Python
Publisher: Python Software Foundation
Path: C:\users\[USER]\appdata\local\programs\python\python37\python.exe
Allow Python to communicate on these networks:
Private networks, such as my home or work network
Public networks, such as those in airports and coffee shops (not recommended because these networks often have little or no security)
What are the risks of allowing an app through a firewall?
3
Allow access
Cancel
14 items
ENG
2:40 AM
2/26/2019


Boxes.py
localhost:8000
Boxes.py
Create boxes and more with a laser cutter!
Boxes.py is an Open Source box generator written in Python. It features both finished parametrized generators as well as a Python API for writing your own. It features finger and (flat) dovetail joints, flex cuts, holes and slots for screws, hinges, gears, pulleys and much more.
▶ Boxes
▶ Boxes with flex
▶ Trays and Drawer Inserts
▶ Shelves
▶ Parts and Samples
▶ Misc
▶ Unstable
ENG
2:41 AM
2/26/2019

CHAPTER 3

Using Boxes.py

Boxes.py is made of a library that is not visible to the user and multiple generators – each having its own set of parameters and creating a drawing for it own type of object. These generators are divided up into different groups to make it easier to find them:

- Boxes
- Boxes with flex
- Trays and Drawer Inserts
- Shelves
- Parts and Samples
- Misc
- Unstable

The parameters for each generators also come in groups.

3.1 Units of meassurement

In general all measurements are in Millimeters (mm). There is no option to change the units of measurement and there is no plan to add such a option.

A second way to define lengths is as multiple of the material thickness which is one of the standard parameters described below. This allows features to retain their proportions even if some parts depend on the material thickness.

The description texts should state the unit of each argument - please open a ticket if the units are missing somewhere.

3.2 Default arguments

In the web interface this is the bottom group right before the `Render` button. These are basically all technical settings that have little to do with the object being rendered but more with the material used and the way the drawing and the

material is processed.

The settings are

3.2.1 thickness

The thickness of the material used. This value is used at many places to define the sizes of features like finger joints, hinges, ... It is very important to get the value right - especially if there are fingers that need to fit into some holes. Be aware that many materials may differ from their nominal value. You should **always measure the thickness** for every sheet unless you have a very reliable supply that is known to stick very closely to specifications. For (ply) wood even a 100th of a millimeter makes a notable difference in how stiff the fit is. Harder more brittle materials may be even more picky.

3.2.2 burn

The burn correction aka kerf is the distance the laser has to keep from the edge of the parts. If the laser would cut right on the edge it would cut away the outside perimeter of the part. So the burn value is basically the radius of the laser - or half the width of the laser cut.

The value of the burn parameter depends on your laser cutter, the material cut and the thickness of the material. In addition it depends on whether you want the parts to be over or under sized. Materials that are spongy like wood can be cut oversized (larger burn value) so they can be press fitted with some force and may be assembled without glue. Brittle materials (like Acrylic) need to be cut undersized to leave a gap for the glue.

Note: The way the burn param works is a bit counter intuitive. Bigger burn values make a tighter fit. Smaller values make a looser fit.

Small changes in the burn param can make a notable difference. Typical steps for adjustment are 0.01 or even 0.005mm to choose between different amounts of force needed to press plywood together.

To find the right burn value cut out a rectangle and then meassure how much smaller it is than its nominal size. The burn value should be around half of the difference. To test the fit for several values at once you can use the **BurnTest** generator in the "Parts and Samples" section.

3.2.3 format

Boxes.py is able to create multiple formats. For most of them it requires `ps2edit`. Without `ps2edit` only `SVG` and `postscript` (ps) is supported. Otherwise you can also select

- ai
- dxf
- gcode
- pdf
- plt

Other formats supported by `ps2edit` can be added easily. Please open a ticket on GitHub if you need one.

3.2.4 tabs

Tabs are small bridges between the parts and surrounding material that keep the part from falling out. In theory their width should be affected by the burn parameter. But it is more practical to have both independent so you can tune them separately. Most parts and generators support this features but there may be some that don't.

For plywood values of 0.2 to 0.3mm still allow getting the parts out by hand (Depending on you laser cutter and the exact material). With little more you will need a knife to cut them loose.

3.2.5 debug

Most regular users won't need this option.

It adds some construction lines that are helpful for developing new generators. Only few pieces actually support the parameter. The most notable being finger holes that show the border of the piece they belong to. This helps checking whether the finger holes are placed correctly.

3.2.6 reference

Converting vector graphics is error prone. Many formats have very weird ideas how their internal units translates to real world dimensions. If reference is set to non zero Boxes.py renders a rectangle of the given length. It can be used to check if the drawing is still at the right scale or may give clues on how to scale it back to the right proportions.

3.3 Edge Type parameters

All but the simplest edge types have a number of settings controlling how exactly they should look. Generators are encouraged to offer these settings to the user. In the web interface they are folded up. In the command line interfacce they are grouped together. Users should be aware that not all settings are practical to change. For now Boxes.py does not allow hiding some settings.

CHAPTER 4

Contributing to Boxes.py

You are thinking about contributing to Boxes.py? That's great! Boxes.py is designed to be re-used and extended.

This document gives you some guidelines how your contribution is most likely to impact the development and your changes are most likely to be merged into the upstream repository.

Most of them should be just general best practises and not be surprising. Don't worry if you find them too complicated. It is OK leave the final touch to someone else.

4.1 Writing code for Boxes.py

You will often be compelled to just do a quick thing that will solve your immediate needs. That's fine. But nevertheless it is often worth doing things the right way and be able to submit your changes upstream. For one to give something back to the community. But also for purely selfish reasons like getting the code maintained. Also Boxes.py is designed to make doing things properly the easy way.

Here are some guidelines that make this easier. Depending on what you are up to they may apply to a varying degree. It's ok to submit patches that are not quite ready yet. But please state in the pull request message what you think the status is and whether you want help or are going to finish it on your own.

- Please fork the repository at GitHub before getting started
- Start with creating separate branches for each of your new generators or features
  - You can merge them into your master branch to have them all in one place
  - Please continue your work in the branches and repeatedly merge them to master
- Before submitting a pull request intended to go upstream have clean patches that are self contained and error free
  - Re-order and squash patches with *git rebase -i*
  - The patches should containing meaningful changes and not (necessarily) reflect how the code was created
  - Rebase your branch to the current master branch

  - Be prepared that your code may get reworked before being merged upstream
- Submit a pull request in GitHub based on your feature branch
  - Describe the status of the patch set and your intentions with it in the pull request message

If you want to discuss your idea open a ticket describing it and ask questions there. This is encouraged even if you think you know what you want to do. There are many short cuts in Boxes.py and pointing you in the right direction may save you a lot of work.

If you want feed back on you code feel free to open a PR. State that this is work in progress in the PR message. It's OK if it does not follow the guide lines (yet).

4.1.1 Writing new Generators

Writing new generators is the most straight forward thing to do with Boxes.py. Here are some guidelines that make it easier to get them added:

- Start with a copy of another generator or *boxes/generators/_template.py*
- Commit changes to the library in separate patches
- Use parameters with sane defaults instead of hard coding dimensions
- Simple generators can end up as one single commit
- For more complicated generators there can be multiple patches - each adding another feature

4.2 Improving the Documentation

Boxes.py comes with Sphinx based documentation that is in large parts generated from the doc strings in the code. Nevertheless documentation has a tendency to get outdated. If you encounter outdated pieces of documentation feel free to submit a pull request or open a ticket pointing out what should be changed or even suggesting a better text.

To check your changes docs need to be build with *make html* in *documentation/src*. This places the compiled documentation in *documentation/build/html*. You need to have *sphinx* installed for this to work.

The online documentation gets build and updated automatically by the Travis CI as soon as the changes makes it into the GitHub master branch.

4.3 Improving the User Interface

Coming up with good names and good descriptions is hard. Often writing a new generator is much easier than coming up with a good name for it and its arguments. If you think something deserves a better name or description and you can come up with one please don't hesitate to open a ticket. It is this small things that make something like Boxes.py easy or hard to use.

There is also an - often empty - space for a longer text for each generator that could house assembling instructions, instructions for use or just more detailed descriptions. If you are interested in writing some please open a ticket. Your text does not have to be perfect. We can work on it together.

4.4 Reporting bugs

If you encounter issues with Boxes.py, please open a ticket at GitHub. Please provide all information necessary to reproduce the bug. Often this can be the URL of the broken result. If the issue is easy to spot it may be sufficient to just give a brief description. Otherwise it can be helpful to attach the resulting SVG, a screen shot or the error message. Add a “bug” tag to draw additional attention.

4.5 Suggesting new generators or features

If you have an idea for a new generator or feature please open a ticket. Give some short rational how or where you would use such a thing. Try to give a precise description how it should look like and which features and details are important. The less is left open the easier it is to implement. You can add an “enhancement” tag.

CHAPTER 5

Using the Boxes.py API

If there is no generator fitting your needs you can either adjust an existing one (may be by copying it to another name first) or writing a new one from scratch.

5.1 Architecture

Boxes.py it structured into several distinct tiers.

5.1.1 User Interfaces

User interfaces allow users to render the different generators. They handle the parameters of Generators and convert them to a readable form. The user interfaces are located in *scripts/*. Currently there is

- scripts/boxes – the command line interface
- scripts/boxesserver – the web interface
- scripts/boxes2inx – generates Inkscape extensions
- scripts/boxes_example.ipynb – Jupyter notebook

5.1.2 Generators

A (box) generator is an sub class of boxes.Boxes. It generates one drawing. The sub classes over load .__init__() to set their parameters and implement .render() that does the actual drawing.

Generators are found in `boxes/generators/`. They are included into the web UI and the CLI tool by the name of their class. So whenever you copy either an existing generator or the sceleton in `boxes/generators/_template.py` you need to change the name of the main class first.

5.1.3 Parts

Parts are a single call that draws something according to a set of parameters. There is a number of standard parts. Their typical params are explained in the API docs.

Only real requirement for a part it supporting the move parameter for placement.

Part Callbacks

Most parts support callbacks - either one in the middle for round parts or one for each edge. They allow placing holes or other features on the part.

5.1.4 Navigation and Turtle Graphics

Many drawing commands in Boxes.py are Turtle Graphics commands. They start at the current position and in the current direction and move the coordinate system with them. This way the absolute coordinates are never used and placement and movement is always relative to the current position.

There are a few functions to move the origin to a convenient position or to return to a previously saved position.

5.1.5 Edges

Edges are turtle graphic commands. But they have been elevated to proper Classes to handle outsets. They can be passed as parameters to parts. There is a set of standard edges found in `.edges`. They are associated with a single char which can be used instead of the Edge object itself at most places. This allows passing the edge description of a part as a string.

5.1.6 Turtle graphics

There are a few turtle graphics commands that do the actual drawing. Corners with an positive angle (going counter clockwise) close the part while negative angles (going clockwise) create protrusions. This is inversed for holes which need to be drawn clockwise.

Getting this directions right is important to make the burn correction (aka kerf) work properly.

5.1.7 Simple drawing commands

These also are simple drawing commands. Some of them get `x`, `y` and `angle` parameters to draw somewhere specific. Some just draw right at the current coordinate origin. Often these commands create holes or hole patterns.

5.1.8 Back end

Boxes.py used to use cairo as graphics library. It now uses its own - pure Python - back end. It is not fully encapsulated within the drawing methods of the Boxes class. Although this is the long term goal. Boxes.ctx is the context all drawing is made on.

5.2 Generators

Generators are sub classes of

class boxes.**Boxes**
Main class – Generator should sub class this

Most code is directly in this class. Sub class are supposed to over write the `.__init__()` and `.render()` method.

The Boxes class keeps a canvas object (self.ctx) that all drawing is made on. In addition it keeps a couple of global settings used for various drawing operations. See the `.__init__()` method for the details.

For implementing a new generator forking an existing one or using the `boxes/generators/_template.py` is probably easier than starting from scratch.

Many methods and attributes are for use of the sub classes. These methods are the interface for the user interfaces to interact with the generators:

Boxes.**__init__**()
Initialize self. See help(type(self)) for accurate signature.

Boxes.**parseArgs**(*args=None*)
Parse command line parameters

Parameters **args** – (Default value = None) parameters, None for using sys.argv

Boxes.**render**()
Implement this method in your sub class.

You will typically need to call .parseArgs() before calling this one

Boxes.**open**()
Prepare for rendering

Create canvas and edge and other objects Call this before .render()

Boxes.**close**()
Finish rendering

Flush canvas to disk and convert output to requested format if needed. Call after .render()

5.2.1 Handling Generators

To handle the generators there is code in the `boxes.generators` package.

class boxes.generators.**UIGroup**(*name*, *title=None*, *description=''*)

add(*box*)

boxes.generators.**getAllBoxGenerators**()

boxes.generators.**getAllGeneratorModules**()

This adds generators to the user interfaces automatically. For this to work it is important that the class names are unique. So whenever you start a new generator please change the class name right away.

5.3 Generator Arguments

Boxes.py uses the `argparse` standard library for handling the arguments for the generators. It is used directly for the `boxes` command line tool. But it also handles – with some additional code – the web interface and the Inkscape extensions. To make this work one has to limit the kind of parameters used. Boxes.py supports the following types:

- `int`
- `float`
- `str`
- `boxes.boolarg` – an alternative to `bool` that works with the web interface
- `boxes.argparseSections` – multiple lengths e.g. for dividing up a box in one direction

and

class boxes.**ArgparseEdgeType**(*edges=None*)
argparse type to select from a set of edge types

For the standard types there is code to create HTML and Inkscape extensions. The other types can have `.html()` and `.inx()` methods.

The argument parser need to be built in the `.__init__()` method after calling the method of the super class. Have a look at

BOX.**__init__**()
Initialize self. See help(type(self)) for accurate signature.

As many arguments are used over and over there is a function that can add the most common ones:

Boxes.**buildArgParser**(**l*, ***kw*)
Add commonly used arguments

Parameters

- ***l** – parameter names
- ****kw** – parameters with new default values

Supported parameters are

- floats: x, y, h, hi
- argparseSections: sx, sy, sh
- ArgparseEdgeType: bottom_edge, top_edge
- boolarg: outside
- str (selection): nema_mount

Check the source for details about the single arguments.

Other arguments can be added with the normal argparser API - namely

ArgumentParser.**add_argument**(*dest*, ..., *name=value*, ...)
add_argument(option_string, option_string, . . . , name=value, . . .)

of the `Boxes.argparser` attribute.

5.3.1 Edge style arguments

Edges that work together share a Settings class (and object). These classes can create `argparse` groups:

classmethod `Settings`.**parserArguments**(*parser*, *prefix=None*, ***defaults*)

See

`BOX`.**__init__**()
Initialize self. See help(type(self)) for accurate signature.

for a list of possible edge settings. These regular settings are used in the standard edge instances used everywhere. For special edge instances you can call them with a `prefix` parameter. But you then need to deal with the results on your own.

5.3.2 Default Arguments

The *Default arguments* get added automatically by the super class's constructor.

5.3.3 Accessing the Arguments

For convenience content of the arguments are written to attributes of the Boxes instance before `.render()` is called. This is done by `Boxes.parseArgs`. But most people won't need to care as this is handled by the frame work. Be careful to **not overwrite important methods or attributes by using conflicting argument names**.

5.4 Navigation

The back end can both move the origin and the current point from which the next line is going to start. Boxes.py hides this by using Turtle Graphics commands that also move the origin to the end of the last line. Other drawing commands restore the current position after they are finished.

Moving the origin like this allows ignoring the absolute coordinates and do all movement and drawing to be relative to the current position. The current positions does not only consist of a point on the drawing canvas but also a direction.

To move the origin to a different location there are these to methods:

`Boxes`.**moveTo**(*x*, *y=0.0*, *degrees=0*)
Move coordinate system to given point

Parameters

- **x** –
- **y** – (Default value = 0.0)
- **degrees** – (Default value = 0)

`Boxes`.**moveArc**(*angle*, *r=0.0*)

Parameters

- **angle** –
- **r** – (Default value = 0.0)

Often it is necessary to return to a position e.g. after placing a row of parts. This can be done with the following context manager:

Boxes.**saved_context**()
Generator: for saving and restoring contexts.

It can be used with the following code pattern:

```
with self.saved_context:
    self.rectangularWall(x, h, move="right")
    self.rectangularWall(y, h, move="right")
    self.rectangularWall(y, h, move="right")
    self.rectangularWall(x, h, move="right")
self.rectangularWall(x, h, move="up only")

# continue above the row
```

Parts of the code still directly use the back end primitives **Boxes.ctx.save()** and **Boxes.ctx.restore()**. But this has several disadvantages and is discouraged. For one it requires matchiung calls. It also does not reset the starting point of the next line. This is "healed" by a follow up **.moveTo()**. Use **.moveTo(0, 0)** if in doubt.

5.5 Parts

There are a few parameter shared by many of those parts:

5.5.1 The callback parameter

The callback parameter can take on of the following forms:

- A function (or bound method) that expects one parameter: the number of the side the callback is currently called for.
- A dict with some of the numbers of the sides as keys and functions without parameters as values.
- A list of functions without parameters. The list may contain None as place holder and be shorter than the number of sides.

The callback functions are called with the side of the part at the positive x and y axis. If the edge uses up space this space is below the x axis. You do not have to restore the coordinate settings in the callback.

Instead of functions it can be handy to use a lambda expression calling the one building block function you need (e.g. fingerHolesAt).

For your own parts you can use this helper function:

Boxes.**cc**(*callback*, *number*, *x=0.0*, *y=None*)
Call callback from edge of a part

Parameters

- **callback** – callback (callable or list of callables)
- **number** – number of the callback
- **x** – (Default value = 0.0) x position to be call on
- **y** – (Default value = None) y position to be called on (default does burn correction)

For finding the right piece to the *callback* parameter this function is used:

Boxes.**getEntry**(*param*, *idx*)
Get entry from list or items itself

Parameters

- **param** – list or item
- **idx** – index in list

5.5.2 The move parameter

For placing the parts the `move` parameter can be used. It is string with space separated words - at most one of each of those options:

- left / right
- up / down
- only

If "only" is given the part is not drawn but only the move is done. This can be useful to go in one direction after having placed multiple parts in the other and have returned with `.ctx.restore()`.

For implementing parts the following helper function can be used to implement a `move` parameter:

`Boxes.`**`move`**(*x*, *y*, *where*, *before=False*)

Intended to be used by parts where can be combinations of "up" or "down", "left" or "right", "only", "mirror" and "rotated" when "only" is included the move is only done when before is True "mirror" will flip the part along the y axis "rotated" draws the parts rotated 90 counter clockwise The function returns whether actual drawing of the part should be omited.

Parameters

- **x** – width of part
- **y** – height of part
- **where** – which direction to move
- **before** – (Default value = False) called before or after part being drawn

It needs to be called before and after drawing the actual part with the proper `before` parameter set.

5.5.3 The edges parameter

The `edges` parameter needs to be an iterable of Edge instances to be used as edges of the part. Instead of instances it is possible to pass a single character that is looked up in the `.edges` dict. This allows to pass a string with the desired characters per edge. By default the following character are supported:

- e : straight edge
- E : as above but extended outside by one thickness
- f, F : finger joints
- h : edge with holes for finger joints
- d, D : dove tail joints

Generators can register their own Edges by putting them into the `.edges` dictionary.

Same applies to the parameters of `.surroundingWall` although they denominate single edge (types) only.

5.5.4 PartsMatrix

To place many of the same part partMatrix can used:

Boxes.**partsMatrix**(*n*, *width*, *move*, *part*, **l*, ***kw*)
place many of the same part

Parameters

- **n** – number of parts
- **width** – number of parts in a row (0 for same as n)
- **move** – (Default value = None)
- **part** – callable that draws a part and knows move param
- ***l** – params for part
- ****kw** – keyword params for part

It creates one big block of parts. The move param treat this block like on big part.

5.6 Existing Parts

A couple of commands can create whole parts like walls. Typically the sizes given are the inner dimension not including additional space needed for burn compensation or joints.

Currently there are the following parts:

Boxes.**rectangularWall**(*x*, *y*, *edges='eeee'*, *ignore_widths=[]*, *holesMargin=None*, *holesSettings=None*, *bedBolts=None*, *bedBoltSettings=None*, *callback=None*, *move=None*)
Rectangular wall for all kind of box like objects

Parameters

- **x** – width
- **y** – height
- **edges** – (Default value = "eeee") bottom, right, top, left
- **ignore_widths** – list of edge_widths added to adjacent edge
- **holesMargin** – (Default value = None)
- **holesSettings** – (Default value = None)
- **bedBolts** – (Default value = None)
- **bedBoltSettings** – (Default value = None)
- **callback** – (Default value = None)
- **move** – (Default value = None)

Boxes.**flangedWall**(*x*, *y*, *edges='FFFF'*, *flanges=None*, *r=0.0*, *callback=None*, *move=None*)
Rectangular wall with flanges extending the regular size

This is similar to the rectangularWall but it may extend to either side replacing the F edge with fingerHoles. Use with E and F for edges only.

Parameters

- **x** – width
- **y** – height
- **edges** – (Default value = "FFFF") bottom, right, top, left
- **flanges** – (Default value = None) list of width of the flanges
- **r** – radius of the corners of the flange
- **callback** – (Default value = None)
- **move** – (Default value = None)

Boxes.**rectangularTriangle**(*x*, *y*, *edges='eee'*, *r=0.0*, *num=1*, *bedBolts=None*, *bedBoltSettings=None*, *callback=None*, *move=None*)

Rectangular triangular wall

Parameters

- **x** – width
- **y** – height
- **edges** – (Default value = "eee") bottom, right[, diagonal]
- **r** – radius towards the hypothenuse
- **num** – (Default value = 1) number of triangles
- **bedBolts** – (Default value = None)
- **bedBoltSettings** – (Default value = None)
- **callback** – (Default value = None)
- **move** – (Default value = None)

Boxes.**regularPolygonWall**(*corners=3*, *r=None*, *h=None*, *side=None*, *edges='e'*, *hole=None*, *callback=None*, *move=None*)

Create regular polygone as a wall

Parameters

- **corners** – number of corners of the polygone
- **radius** – distance center to one of the corners
- **h** – distance center to one of the sides (height of sector)
- **side** – length of one side
- **edges** – (Default value = "e", may be string/list of length corners)
- **hole** – diameter of central hole (Default value = 0)
- **callback** – (Default value = None, middle=0, then sides=1..)
- **move** – (Default value = None)

Boxes.**roundedPlate**(*x*, *y*, *r*, *edge='f'*, *callback=None*, *holesMargin=None*, *holesSettings=None*, *bedBolts=None*, *bedBoltSettings=None*, *wallpieces=1*, *extend_corners=True*, *move=None*)

Plate with rounded corner fitting to .surroundingWall()

For the callbacks the sides are counted depending on wallpieces

Parameters

- **x** – width

- **y** – hight
- **r** – radius of the corners
- **callback** – (Default value = None)
- **holesMargin** – (Default value = None) set to get hex holes
- **holesSettings** – (Default value = None)
- **bedBolts** – (Default value = None)
- **bedBoltSettings** – (Default value = None)
- **wallpieces** – (Default value = 1) # of separate surrounding walls
- **extend_corners** – (Default value = True) have corners outset with teh edges
- **move** – (Default value = None)

Boxes.**surroundingWall**(*x*, *y*, *r*, *h*, *bottom='e'*, *top='e'*, *left='D'*, *right='d'*, *pieces=1*, *extend_corners=True*, *callback=None*, *move=None*)

Wall(s) with flex fiting around a roundedPlate()

For the callbacks the sides are counted depending on pieces

Parameters

- **x** – width of matching roundedPlate
- **y** – height of matching roundedPlate
- **r** – corner radius of matching roundedPlate
- **h** – inner height of the wall (without edges)
- **bottom** – (Default value = 'e') Edge type
- **top** – (Default value = 'e') Edge type
- **left** – (Default value = 'D') left edge(s)
- **right** – (Default value = 'd') right edge(s)
- **pieces** – (Default value = 1) number of separate pieces
- **callback** – (Default value = None)
- **move** – (Default value = None)

5.6.1 Parts Class

More parts are available in a separete class. An instance is available as **Boxes.parts**

Parts.**disc**(*diameter*, *hole=0*, *callback=None*, *move=''*)

Simple disc

Parameters

- **diameter** – diameter of the disc
- **hole** – (Default value = 0)
- **callback** – (Default value = None) called in the center
- **move** – (Defaultvalue = None)

Parts.**waivyKnob**(*diameter*, *n=20*, *angle=45*, *hole=0*, *callback=None*, *move=''*)
Disc with a waivy edge to be easier to be gripped

Parameters

- **diameter** – diameter of the knob
- **n** – (Default value = 20) number of waves
- **angle** – (Default value = 45) maximum angle of the wave
- **hole** – (Default value = 0)
- **callback** – (Default value = None) called in the center
- **move** – (Defaultvalue = None)

Parts.**concaveKnob**(*diameter*, *n=3*, *rounded=0.2*, *angle=70*, *hole=0*, *callback=None*, *move=''*)
Knob with dents to be easier to be gripped

Parameters

- **diameter** – diameter of the knob
- **n** – (Default value = 3) number of dents
- **rounded** – (Default value = 0.2) proportion of circumferen remaining
- **angle** – (Default value = 70) angle the dents meet the circumference
- **hole** – (Default value = 0)
- **callback** – (Default value = None) called in the center
- **move** – (Defaultvalue = None)

Parts.**ringSegment**(*r_outside*, *r_inside*, *angle*, *n=1*, *move=None*)
Ring Segment

Parameters

- **r_outside** – outer radius
- **r_inside** – inner radius
- **angle** – anlge the segment is spanning
- **n** – (Default value = 1) number of segments
- **move** – (Defaultvalue = None)

5.7 Edges

Edges are what makes Boxes.py work. They draw a – more or less – straight border to the current piece. They are part of the turtle graphics part of Boxes.py. This means they start at the current position and current direction and move the current position to the end of the edge.

Edge instances have a Settings object associated with them that keeps the details about how the edge should look like. Edges that are supposed to work together share the same Settings object to ensure they fit together - assuming they have the same length. Most edges are symetrical to unsure they fit together even when drawn from different directions. Although there are a few exception - mainly edges that provide special features like hinges.

As edges started out as methods of the main Boxes class they still are callables. It turned out that the edges need to provide a bit more information to allow the surrounding code to handle them properly. When drawing an Edge there is a virtual straight line that is the border the shape of the part (e.g. an rectangle). But the actual Edge has often to

be drawn elsewhere. Best example if probably the `F` Edge that matches the normal finger joints. It has to start one material thickness outside of the virual border of the part so the cutouts for the opposing fingers just touch the border. The Edge classes have a number of methods to deal with these kind of offsets.

A set of instances are kept the `.edges` attribute of the `Boxes` class. It is a dict with strings of length one as keys:

- d : DoveTailJoint
- D : DoveTailJointCounterPart
- e : Edge
- E : OutSetEdge
- f : FingerJointEdge
- F : FingerJointEdgeCounterPart
- g : GrippingEdge
- h : FingerHoleEdge
- ijk : Hinge (start, end, both sides)
- IJK : HingePin (start, end, both sides)
- s : StackableEdge
- S : StackableEdgeTop

5.7.1 Edge base class

class boxes.edges.**BaseEdge**(*boxes*, *settings*)
Abstract base class for all Edges

endAngle()
Not yet supported

margin()
Space needed right of the starting point

spacing()
Space the edge needs outside of the inner space of the part

startAngle()
Not yet supported

startwidth()
Amount of space the beginning of the edge is set below the inner space of the part

BaseEdge.**__call__**(*length*, ***kw*)
Call self as a function.

5.7.2 Settings Class

class boxes.edges.**Settings**(*thickness*, *relative=True*, ***kw*)
Generic Settings class

Used by different other classes to store messurements and details. Supports absolute values and settings that grow with the thickness of the material used.

Overload the absolute_params and relative_params class attributes with the suported keys and default values. The values are available via attribute access.

checkValues()
Check if all values are in the right range. Raise ValueError if needed

edgeObjects(*boxes*, *chars=''*, *add=True*)
Generate Edge objects using this kind of settings

Parameters

- **boxes** – Boxes object
- **chars** – sequence of chars to be used by Edge objects
- **add** – add the resulting Edge objects to the Boxes object's edges

setValues(*thickness*, *relative=True*, ***kw*)
Set values

Parameters

- **thickness** – thickness of the material used
- **relative** – (Default value = True) Do scale by thickness
- ****kw** – parameters to set

5.7.3 Straight Edges

class boxes.edges.**Edge**(*boxes*, *settings*)
Straight edge

class boxes.edges.**OutSetEdge**(*boxes*, *settings*)
Straight edge out set by one thickness

5.7.4 Grip

class boxes.edges.**GripSettings**(*thickness*, *relative=True*, ***kw*)
Settings for GrippingEdge Values:

- absolute_params
- style : "wave : "wave" or "bumps"
- outset : True : extend outward the straight edge
- relative (in multiples of thickness)
- depth : 0.3 : depth of the grooves

class boxes.edges.**GrippingEdge**(*boxes*, *settings*)

5.7.5 Stackable Edges

class boxes.edges.**StackableEdge**(*boxes*, *settings*, *fingerjointsettings*)
Edge for having stackable Boxes. The Edge creates feet on the bottom and has matching recesses on the top corners.

class boxes.edges.**StackableEdgeTop**(*boxes*, *settings*, *fingerjointsettings*)

Stackable Edge Settings

class boxes.edges.**StackableSettings**(*thickness*, *relative=True*, ***kw*)
Settings for Stackable Edges

Values:

- absolute_params
  - angle : 60 : inside angle of the feet
- relative (in multiples of thickness)
  - height : 2.0 : height of the feet
  - width : 4.0 : width of the feet
  - holedistance : 1.0 : distance from finger holes to bottom edge

checkValues()
Check if all values are in the right range. Raise ValueError if needed

edgeObjects(*boxes*, *chars='sS'*, *add=True*, *fingersettings=None*)
Generate Edge objects using this kind of settings

Parameters

- **boxes** – Boxes object
- **chars** – sequence of chars to be used by Edge objects
- **add** – add the resulting Edge objects to the Boxes object's edges

5.7.6 Finger joints

Finger joints are a simple way of joining two sheets (e.g. of plywood). They work best at an 90° angle. There are two different sides matching each other. As a third alternative there are holes that the fingers of one sheet can plug into. This allows stable T connections especially useful for inner walls.

class boxes.edges.**FingerJointEdge**(*boxes*, *settings*)
Finger joint edge

class boxes.edges.**FingerJointEdgeCounterPart**(*boxes*, *settings*)
Finger joint edge - other side

class boxes.edges.**FingerHoleEdge**(*boxes*, *fingerHoles=None*, ***kw*)
Edge with holes for a parallel finger joint

class boxes.edges.**CrossingFingerHoleEdge**(*boxes*, *height*, *fingerHoles=None*, ***kw*)
Edge with holes for finger joints 90° above

In addition there is

class boxes.edges.**FingerHoles**(*boxes*, *settings*)
Hole matching a finger joint edge

which is no Edge but fits `FingerJointEdge`.

An instance of is accessible as **Boxes.fingerHolesAt**.

Finger Joint Settings

class boxes.edges.**FingerJointSettings**(*thickness*, *relative=True*, ***kw*)
Settings for Finger Joints

Values:

- absolute * style : "rectangular" : style of the fingers * surroundingspaces : 2 : maximum space at the start and end in multiple of normal spaces * angle: 90 : Angle of the walls meeting
- relative (in multiples of thickness)
  - space : 2.0 : space between fingers
  - finger : 2.0 : width of the fingers
  - width : 1.0 : width of finger holes
  - edge_width : 1.0 : space below holes of FingerHoleEdge
  - play : 0.0 : extra space to allow finger move in and out

checkValues()
Check if all values are in the right range. Raise ValueError if needed

edgeObjects(*boxes*, *chars='fFh'*, *add=True*)
Generate Edge objects using this kind of settings

Parameters

- **boxes** – Boxes object
- **chars** – sequence of chars to be used by Edge objects
- **add** – add the resulting Edge objects to the Boxes object's edges

Bed Bolts

class boxes.edges.**BoltPolicy**
Abstract class

Distributes (bed) bolts on a number of segments (fingers of a finger joint)

class boxes.edges.**Bolts**(*bolts=1*)
Distribute a fixed number of bolts evenly

5.7.7 Dove Tail Joints

Dovetails joints can only be used to join two pieces flatly. This limits their use to closing some round form created with flex areas or for joining several parts to a bigger one. For this use case they are much stronger than simple finger joints and can also bare pulling forces.

class boxes.edges.**DoveTailJoint**(*boxes*, *settings*)
Edge with dove tail joints

class boxes.edges.**DoveTailJointCounterPart**(*boxes*, *settings*)
Edge for other side of dove joints

Dove Tail Settings

class boxes.edges.**DoveTailSettings**(*thickness*, *relative=True*, ***kw*)
Settings for Dove Tail Joints

Values:

- absolute
  - angle : 50 : how much should fingers widen (-80 to 80)
- relative (in multiples of thickness)
  - size : 3 : from one middle of a dove tail to another
  - depth : 1.5 : how far the dove tails stick out of/into the edge
  - radius : 0.2 : radius used on all four corners

edgeObjects(*boxes*, *chars='dD'*, *add=True*)
Generate Edge objects using this kind of settings

Parameters

- **boxes** – Boxes object
- **chars** – sequence of chars to be used by Edge objects
- **add** – add the resulting Edge objects to the Boxes object's edges

5.7.8 Flex

class boxes.edges.**FlexEdge**(*boxes*, *settings*)
Edge with flex cuts - use straight edge for the opposing side

Flex Settings

class boxes.edges.**FlexSettings**(*thickness*, *relative=True*, ***kw*)
Settings for Flex

Values:

- absolute
- stretch : 1.05 : Hint of how much the flex part should be shortend
- relative (in multiples of thickness)
- distance : 0.5 : width of the pattern perpendicular to the cuts
- connection : 1.0 : width of the gaps in the cuts
- width" : 5.0 : width of the pattern in direction of the cuts

5.7.9 Slots

class boxes.edges.**Slot**(*boxes*, *depth*)
Edge with an slot to slid another pice through

class boxes.edges.**SlottedEdge**(*boxes*, *sections*, *edge='e'*, *slots=0*)
Edge with multiple slots

5.7.10 CompoundEdge

class boxes.edges.**CompoundEdge**(*boxes*, *types*, *lengths*)
Edge composed of multiple different Edges

5.7.11 Hinges

Hinge Settings

class boxes.edges.**HingeSettings**(*thickness*, *relative=True*, ***kw*)
Settings for Hinges and HingePins Values:

- absolute_params
- style : "outset" : "outset" or "flush"
- outset : False : have lid overlap at the sides (similar to OutSetEdge)
- pinwidth : 1.0 : set to lower value to get disks surrounding the pins
- grip_percentage" : 0 : percentage of the lid that should get grips
- relative (in multiples of thickness)
- hingestrength : 1 : thickness of the arc holding the pin in place
- axle : 2 : diameter of the pin hole
- grip_length : 0 : fixed length of the grips on he lids

Hinge

class boxes.edges.**Hinge**(*boxes*, *settings=None*, *layout=1*)

HingePin

class boxes.edges.**HingePin**(*boxes*, *settings=None*, *layout=1*)

5.8 Drawing commands

5.8.1 Turtle Graphics commands

These commands all move the coordinate system with them.

Boxes.**edge**(*length*, *tabs=0*)
Simple line :param length: length in mm

Boxes.**corner**(*degrees*, *radius=0*, *tabs=0*)
Draw a corner

This is what does the burn corrections

Parameters

- **degrees** – angle
- **radius** – (Default value = 0)

Boxes.**curveTo**(*x1*, *y1*, *x2*, *y2*, *x3*, *y3*)
control point 1, control point 2, end point

Parameters

- **x1** –
- **y1** –
- **x2** –
- **y2** –
- **x3** –
- **y3** –

Boxes.**polyline**(**args*)
Draw multiple connected lines

Parameters ***args** – Alternating length in mm and angle in degrees.

lengths may be a tuple (length, #tabs) angles may be tuple (angle, radius)

Special Functions

Boxes.**bedBoltHole**(*length*, *bedBoltSettings=None*, *tabs=0*)
Draw an edge with slot for a bed bolt

Parameters

- **length** – length of the edge in mm
- **bedBoltSettings** – (Default value = None) Dimmensions of the slot

Latch and Grip

These should probably be Edge classes. But right now they are still functions.

Boxes.**grip**(*length*, *depth*)
Corrugated edge useful as an gipping area

Parameters

- **length** – length
- **depth** – depth of the grooves

Boxes.**latch**(*length*, *positive=True*, *reverse=False*)
Latch to fix a flex box door to the box

Parameters

- **length** – length in mm

- **positive** – (Default value = True) False: Door side; True: Box side
- **reverse** – (Default value = False) True when running away from the latch

`Boxes.`**handle**(*x*, *h*, *hl*, *r=30*)
Creates an Edge with a handle

Parameters

- **x** – width in mm
- **h** – height in mm
- **hl** – height if th grip hole
- **r** – (Default value = 30) radius of the corners

Tab support

Tabs are small interuptions in the border of a part to keep it in place. They are enabled with the **tabs** parameter. All **Edges** automatically create about two tabs. So parts like `boxes.Boxes.rectangularWall()` will have 8 tabs holding them in place. Because of this developers often don't need to be concerned about tabs. But some part may be completely drawn by low level Turtle Graphics commands. For those both `boxes.Boxes.edge()` and `boxes.Boxes.corner()` do support a **tabs** parameter. In addition the length of the line segments in `boxes.Boxes.polyline()` can be given as a tuple **(length, tabs)**.

5.8.2 Draw Commands

These commands do not change the coordinate system but get the coordinates passed as parameters. All of them are either som sort of hole or text. These artifacts are placed somewhere independently of some continuous outline of the part their on.

`Boxes.`**hole**(*x*, *y*, *r=0.0*, *d=0.0*, *tabs=0*)
Draw a round hole

Parameters

- **x** – position
- **y** – postion
- **r** – radius

`Boxes.`**rectangularHole**(*x*, *y*, *dx*, *dy*, *r=0*)
Draw an rectangulat hole

Parameters

- **x** – position
- **y** – position
- **dx** – width
- **dy** – height
- **r** – (Default value = 0) radius of the corners

`Boxes.`**dHole**(*x*, *y*, *r=None*, *d=None*, *w=None*, *rel_w=0.75*, *angle=0*)

`Boxes.`**flatHole**(*x*, *y*, *r=None*, *d=None*, *w=None*, *rel_w=0.75*, *angle=0*)

Boxes.**text**(*text, x=0, y=0, angle=0, align='', fontsize=10, color=[0.0, 0.0, 0.0]*)
Draw text

Parameters

- **text** – text to render
- **x** – (Default value = 0)
- **y** – (Default value = 0)
- **angle** – (Default value = 0)
- **align** – (Default value = "") string with combinations of (top|middle|bottom) and (left|center|right) separated by a space

Boxes.**NEMA**(*size*, *x=0*, *y=0*, *angle=0*, *screwholes=None*)
Draw holes for mounting a NEMA stepper motor

Parameters

- **size** – Nominal size in tenths of inches
- **x** – (Default value = 0)
- **y** – (Default value = 0)
- **angle** – (Default value = 0)

Boxes.**TX**(*size*, *x=0*, *y=0*, *angle=0*)
Draw a star pattern

Parameters

- **size** – 1 to 100
- **x** – (Default value = 0)
- **y** – (Default value = 0)
- **angle** – (Default value = 0)

Boxes.**flex2D**(*x*, *y*, *width=1*)
Fill a rectangle with a pattern allowing bending in both axis

Parameters

- **x** – width
- **y** – height
- **width** – width between the lines of the pattern in multiples of thickness

class NutHole

An instance is available as **boxes.Boxes.nutHole()**

An instance of

class boxes.edges.**FingerHoles**(*boxes*, *settings*)
Hole matching a finger joint edge

is accessible as **Boxes.fingerHolesAt**.

Hexagonal Hole patterns

Hexagonal hole patterns are one way to have some ventilation for housings made with Boxes.py. Right now both `.rectangularWall()` and `.roundedPlate()` do supports this pattern directly by passing the parameters to the calls. For other use cases these more low level methods can be used.

For now this is the only supported pattern for ventilation slots. More may be added in the future.

There is a global Boxes.hexHolesSettings object that is used if no settings are passed. It currently is just a tuple of (r, dist, style) defaulting to (5, 3, 'circle') but might be replace by a Settings instance in the future.

`Boxes.`**`hexHolesRectangle`**(*x*, *y*, *settings=None*, *skip=None*)
Fills a rectangle with holes in a hex pattern.

Settings have: r : radius of holes b : space between holes style : what types of holes (not yet implemented)

Parameters

- **`x`** – width
- **`y`** – height
- **`settings`** – (Default value = None)
- **`skip`** – (Default value = None) function to check if hole should be present gets x, y, r, b, posx, posy

`Boxes.`**`hexHolesCircle`**(*d*, *settings=None*)
Fill circle with holes in a hex pattern

Parameters

- **`d`** – diameter of the circle
- **`settings`** – (Default value = None)

`Boxes.`**`hexHolesPlate`**(*x*, *y*, *rc*, *settings=None*)
Fill a plate with holes in a hex pattern

Parameters

- **`x`** – width
- **`y`** – height
- **`rc`** – radius of the corners
- **`settings`** – (Default value = None)

`Boxes.`**`hexHolesHex`**(*h*, *settings=None*, *grow=None*)
Fill a hexagon with holes in a hex pattern

Parameters

- **`h`** – height
- **`settings`** – (Default value = None)
- **`grow`** – (Default value = None)

5.9 Burn correction

The burn correction – aka kerf – is integrated into the low level commands of Boxes.py. So for the most part developer do not need to care about it. Nevertheless they need to understand how it works to catch the places the do need to care.

Burn correction is done by increasing the radius of all outer corners. This moves all the straight lines outward by the same amount. This has the added benefit of not needing to change the length of the straight lines – making them independent of the adjacent angles. An issue arises when it comes to inner corners. If they do have a radius reducing it by the burn value does the right thing. But for small radii and sharp corners (radius zero) this results in a negative values. It turns out flipping over the arc for negative radii allows keeping the lengths of the straight lines unchanged. So this is what Boxes.py does:

This results in the straight lines touching the piece. This leads to overcuts. They are not as nice as proper dog bones as might be used by a dedicated CAM software. But as Boxes.py is meant to be used for laser cutting this is deemed acceptable:

5.9.1 Programmer's perspective

For this to work it is important that outside is drawn in a counter clock wise direction while holes are drawn in a clock wise direction.

`boxes.Boxes.corner()` adjusts the radius automatically according to **.burn**. This propagates to higher level functions. Parts shipped with Boxes.py do take the burn out-set into account and execute callbacks at the correct position.

In case developers move to a feature inside of a part or executing callbacks while implementing a part they need to be aware of the burn correction. `boxes.Boxes.cc()` does correct for the out-set if called without an **y** parameter. But if a value is given one has to add **self.burn** to compensate. Note that the **x** value typically does not have to be corrected as the callbacks are executed from right underneath the part.

A similar approach is necessary when moving to a feature drawn inside the part without the use of callbacks. Here you typically have to correct for the out-set at the outside of the part and again for in-set of the hole one is about to cut. This can be done in **x** or **y** direction depending on whether the cut ist started vertical or horizontally.

5.10 Examples

Decide whether you want to start from scratch or want to rework an existing generator.

You should go over the arguments first. Get at least the most basic arguments done. For things you are still unsure you can just use a attribute set in the .__init__() method and turn it into a proper argument later on.

Depending on what you want to do you can work on the different levels of the API. You can either use what is there and combine it into something new or you can implements new things in the appropriate level.

Here are some examples:

5.10.1 Housing for some electronics

You can use the ElectronicsBox or the ClosedBox as a basis. Write some callbacks to place holes in the walls to allow accessing the ports of the electronics boards. Place some holes to screw spacers into the bottom to mount the PBC on.

5.10.2 NemaMount

This is a good non box example to look at.

class boxes.generators.nemamount.**NemaMount**
Mounting braket for a Nema motor

Note that although it produces a cube like object it uses separate variables (`x`, `y`, `h`) for the different axis. Probably because it started as a copy of another generator like `ClosedBox`.

5.10.3 DisplayShelf

class boxes.generators.displayshelf.**DisplayShelf**
Shelf with forward slanted floors

The DisplayShelf is completely made out of rectangularWalls(). It uses a callback to place all the fingerHolesAt() right places on the sides. While the use of the Boxes.py API is pretty straight forward the calculations needed are a bit more tricky. You can use the `debug` default param to check if you got things right when attempting something like this yourself.

Note that the front walls and the shelfs form a 90° angle so they work with the default FingerJoints.

5.10.4 BinTray

class boxes.generators.bintray.**BinTray**
A Type tray variant to be used up right with sloped walls in front

The BinTray is based on the TypeTray generator:

class boxes.generators.typetray.**TypeTray**
Type tray - allows only continuous walls

TypeTray is an already pretty complicated generator.

BinTray replaces the now vertical front (former top) edges with a special purpose one that does add the triangles:

class boxes.generators.bintray.**BinFrontEdge**(*boxes*, *settings*)

The `hi` (height of inner walls) argument was removed although the variable is still used internally - out of laziness.

To complete the bin the front walls are added. Follow up patches then switched the slots between the vertical and horizontal walls to have better support for the now bottoms of the bins. Another patch adds angled finger joints for connecting the front walls with the bottoms of the bins.

The TrafficLight generator uses a similar technique implementing its own Edge class. But it uses its own code to generate all the wall needed.

5.10.5 Stachel

class boxes.generators.stachel.**Stachel**
Bass Recorder Endpin

Stachel allows mounting a monopod to a bass recorder. It is basically just one part repeated with different parameters. It can't really make use of much of the Boxes.py library. It implements this one part including the `move` parameter and draws everything using the `.polyline()` method. This is pretty painful as lots of angles and distances need to be calculated by hand.

For symmetric sections it passes the parameters to `.polyline` twice – first in normal order and then reversed to get the mirrored section.

This generator is beyond what Boxes.py is designed for. If you need something similar you may want to use another tool like OpenScad or a traditional CAD program.

CHAPTER 6

All Box Generators

Generators are organized in several Groups

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Index

Symbols

__call__() (*boxes.edges.BaseEdge method*), 34
__init__() (*boxes.Boxes method*), 25
__init__() (*boxes.generators._template.BOX method*), 26, 27

A

add() (*boxes.generators.UIGroup method*), 25
add_argument() (*argparse.ArgumentParser method*), 26
ArgparseEdgeType (*class in boxes*), 26

B

BaseEdge (*class in boxes.edges*), 34
bedBoltHole() (*boxes.Boxes method*), 40
BinFrontEdge (*class in boxes.generators.bintray*), 45
BinTray (*class in boxes.generators.bintray*), 45
BoltPolicy (*class in boxes.edges*), 37
Bolts (*class in boxes.edges*), 37
Boxes (*class in boxes*), 25
boxes.generators (*module*), 25
buildArgParser() (*boxes.Boxes method*), 26

C

cc() (*boxes.Boxes method*), 28
checkValues() (*boxes.edges.FingerJointSettings method*), 37
checkValues() (*boxes.edges.Settings method*), 34
checkValues() (*boxes.edges.StackableSettings method*), 36
close() (*boxes.Boxes method*), 25
CompoundEdge (*class in boxes.edges*), 39
concaveKnob() (*boxes.parts.Parts method*), 33
corner() (*boxes.Boxes method*), 39
CrossingFingerHoleEdge (*class in boxes.edges*), 36
curveTo() (*boxes.Boxes method*), 40

D

dHole() (*boxes.Boxes method*), 41
disc() (*boxes.parts.Parts method*), 32
DisplayShelf (*class in boxes.generators.displayshelf*), 45
DoveTailJoint (*class in boxes.edges*), 37
DoveTailJointCounterPart (*class in boxes.edges*), 37
DoveTailSettings (*class in boxes.edges*), 38

E

Edge (*class in boxes.edges*), 35
edge() (*boxes.Boxes method*), 39
edgeObjects() (*boxes.edges.DoveTailSettings method*), 38
edgeObjects() (*boxes.edges.FingerJointSettings method*), 37
edgeObjects() (*boxes.edges.Settings method*), 35
edgeObjects() (*boxes.edges.StackableSettings method*), 36
endAngle() (*boxes.edges.BaseEdge method*), 34

F

FingerHoleEdge (*class in boxes.edges*), 36
FingerHoles (*class in boxes.edges*), 36
FingerJointEdge (*class in boxes.edges*), 36
FingerJointEdgeCounterPart (*class in boxes.edges*), 36
FingerJointSettings (*class in boxes.edges*), 37
flangedWall() (*boxes.Boxes method*), 30
flatHole() (*boxes.Boxes method*), 41
flex2D() (*boxes.Boxes method*), 42
FlexEdge (*class in boxes.edges*), 38
FlexSettings (*class in boxes.edges*), 38

G

getAllBoxGenerators() (*in module boxes.generators*), 25
getAllGeneratorModules() (*in module boxes.generators*), 25
getEntry() (*boxes.Boxes method*), 28

grip() (*boxes.Boxes method*), 40
GrippingEdge (*class in boxes.edges*), 35
GripSettings (*class in boxes.edges*), 35

H

handle() (*boxes.Boxes method*), 41
hexHolesCircle() (*boxes.Boxes method*), 43
hexHolesHex() (*boxes.Boxes method*), 43
hexHolesPlate() (*boxes.Boxes method*), 43
hexHolesRectangle() (*boxes.Boxes method*), 43
Hinge (*class in boxes.edges*), 39
HingePin (*class in boxes.edges*), 39
HingeSettings (*class in boxes.edges*), 39
hole() (*boxes.Boxes method*), 41

L

latch() (*boxes.Boxes method*), 40

M

margin() (*boxes.edges.BaseEdge method*), 34
move() (*boxes.Boxes method*), 29
moveArc() (*boxes.Boxes method*), 27
moveTo() (*boxes.Boxes method*), 27

N

NEMA() (*boxes.Boxes method*), 42
NemaMount (*class in boxes.generators.nemamount*), 44
NutHole (*built-in class*), 42

O

open() (*boxes.Boxes method*), 25
OutSetEdge (*class in boxes.edges*), 35

P

parseArgs() (*boxes.Boxes method*), 25
parserArguments() (*boxes.edges.Settings class method*), 27
partsMatrix() (*boxes.Boxes method*), 30
polyline() (*boxes.Boxes method*), 40

R

rectangularHole() (*boxes.Boxes method*), 41
rectangularTriangle() (*boxes.Boxes method*), 31
rectangularWall() (*boxes.Boxes method*), 30
regularPolygonWall() (*boxes.Boxes method*), 31
render() (*boxes.Boxes method*), 25
ringSegment() (*boxes.parts.Parts method*), 33
roundedPlate() (*boxes.Boxes method*), 31

S

saved_context() (*boxes.Boxes method*), 27
Settings (*class in boxes.edges*), 34
setValues() (*boxes.edges.Settings method*), 35
Slot (*class in boxes.edges*), 38
SlottedEdge (*class in boxes.edges*), 38
spacing() (*boxes.edges.BaseEdge method*), 34
Stachel (*class in boxes.generators.stachel*), 45
StackableEdge (*class in boxes.edges*), 35
StackableEdgeTop (*class in boxes.edges*), 35
StackableSettings (*class in boxes.edges*), 36
startAngle() (*boxes.edges.BaseEdge method*), 34
startwidth() (*boxes.edges.BaseEdge method*), 34
surroundingWall() (*boxes.Boxes method*), 32

T

text() (*boxes.Boxes method*), 41
TX() (*boxes.Boxes method*), 42
TypeTray (*class in boxes.generators.typetray*), 45

U

UIGroup (*class in boxes.generators*), 25

W

waivyKnob() (*boxes.parts.Parts method*), 32