
pymunk Documentation

Release 5.4.0

Victor Blomqvist

Oct 25, 2018

Contents

1	Installation	3
2	Example	5
3	Documentation	7
4	The Pymunk Vision	9
5	Contact & Support	11
6	Dependencies / Requirements	13
7	Python 2 & Python 3	15
8	Chipmunk Compilation	17
9	Contents	19
9.1	News	19
9.1.1	Pymunk 5.4.0	19
9.1.2	Introductory video tutorials	19
9.1.3	Pymunk 5.3.2	20
9.1.4	Pymunk 5.3.1	20
9.1.5	Pymunk 5.3.0	20
9.1.6	New page theme	21
9.1.7	Pymunk on Android	21
9.1.8	Pymunk 5.2.0	21
9.1.9	Pymunk 5.1.0	21
9.1.10	Pymunk 5.0.0	22
9.1.11	Move from ctypes to CFFI?	22
9.1.12	Travis-ci & tox	22
9.1.13	Move to Github	23
9.1.14	pymunk 4.0.0	23
9.1.15	pymunk 3.0.0	24
9.1.16	pymunk 2.1.0	25
9.1.17	pymunk 2.0.0	25
9.1.18	Older news	26
9.2	Installation	26
9.2.1	Install Pymunk	26

9.2.2	Examples & Documentation	26
9.2.3	Advanced install	26
9.2.4	Compile Chipmunk	27
9.2.5	CFFI Installation	27
9.3	Overview	27
9.3.1	Basics	27
9.3.2	Model your physics objects	28
9.3.2.1	Object shape	28
9.3.2.2	Mass, weight and units	28
9.3.2.3	Looks before realism	28
9.3.3	Game loop / moving time forward	29
9.3.4	Unstable simulation?	29
9.3.5	Copy and Load/Save Pymunk objects	29
9.3.6	Additional info	30
9.4	API Reference	30
9.4.1	pymunk Package	30
9.4.1.1	pymunk.autogeometry Module	30
9.4.1.2	pymunk.constraint Module	33
9.4.1.3	pymunk.vec2d Module	43
9.4.1.4	pymunk.matplotlib_util Module	46
9.4.1.5	pymunk.pygame_util Module	48
9.4.1.6	pymunk.pyglet_util Module	50
9.4.1.7	pymunkoptions Module	52
9.5	Examples	84
9.5.1	Jupyter Notebooks	84
9.5.1.1	matplotlib_util_demo.ipynb	84
9.5.1.2	newtons_cradle.ipynb	85
9.5.2	Standalone Python	85
9.5.2.1	arrows.py	86
9.5.2.2	balls_and_lines.py	87
9.5.2.3	basic_test.py	88
9.5.2.4	bouncing_balls.py	88
9.5.2.5	box2d_pyramid.py	89
9.5.2.6	box2d_vertical_stack.py	90
9.5.2.7	breakout.py	91
9.5.2.8	contact_and_no_flip.py	92
9.5.2.9	contact_with_friction.py	93
9.5.2.10	copy_and_pickle.py	94
9.5.2.11	damped_rotary_spring_pointer.py	95
9.5.2.12	deformable.py	96
9.5.2.13	flipper.py	97
9.5.2.14	index_video.py	98
9.5.2.15	kivy_pymunk_demo	99
9.5.2.16	newtons_cradle.py	99
9.5.2.17	no_debug.py	100
9.5.2.18	platformer.py	100
9.5.2.19	playground.py	101
9.5.2.20	point_query.py	102
9.5.2.21	py2exe_setup__basic_test.py	103
9.5.2.22	py2exe_setup__breakout.py	103
9.5.2.23	pygame_util_demo.py	104
9.5.2.24	pyglet_util_demo.py	104
9.5.2.25	run.py	105
9.5.2.26	shapes_for_draw_demos.py	105

9.5.2.27	slide_and_pinjoint.py	105
9.5.2.28	spiderweb.py	106
9.5.2.29	using_sprites.py	107
9.5.2.30	using_sprites_pyglet.py	108
9.6	Showcase	109
9.6.1	Games	111
9.6.2	Non-Games	113
9.6.3	Papers / Science	114
9.7	Tutorials	115
9.7.1	Slide and Pin Joint Demo Step by Step	115
9.7.1.1	Before we start	116
9.7.1.2	An empty simulation	117
9.7.1.3	Falling balls	118
9.7.1.4	A static L	120
9.7.1.5	Joints (1)	122
9.7.1.6	Joints (2)	122
9.7.1.7	Ending	123
9.7.2	External Tutorials	125
9.7.2.1	Pymunk physics in Pyglet	125
9.8	Benchmarks	125
9.8.1	Micro benchmarks	125
9.8.1.1	Results:	126
9.8.2	Compared to Other Physics Libraries	127
9.8.2.1	Cymunk	127
9.9	Advanced	128
9.9.1	Why CFFI?	128
9.9.2	Why ctypes? (OBSOLETE)	128
9.9.3	Code Layout	129
9.9.4	Tests	130
9.9.5	Working with non-wrapped parts of Chipmunk	130
9.9.6	Weak References and <code>__del__</code> Methods	130
9.10	License	130
10	Indices and tables	133
	Python Module Index	135

Pymunk is a easy-to-use pythonic 2d physics library that can be used whenever you need 2d rigid body physics from Python. Perfect when you need 2d physics in your game, demo or other application! It is built on top of the very capable 2d physics library [Chipmunk](#).

The first version was released in 2007 and Pymunk is still actively developed and maintained today.

Pymunk has been used with success in many projects, big and small. For example: 3 Pyweek game competition winners, more than a dozen published scientific papers and even in a self-driving car simulation! See the Showcases section on the Pymunk webpage for some examples.

2007 - 2018, Victor Blomqvist - vb@viblo.se, MIT License

This release is based on the latest Pymunk release (5.4.0), using Chipmunk 7.0 rev d7603e3927 (source included)

CHAPTER 1

Installation

In the normal case pymunk can be installed with pip:

```
> pip install pymunk
```

It has one (or two) dependencies. CFFI and if not on Windows you also need a working gcc compiler.

CHAPTER 2

Example

Quick code example:

```
import pymunk                # Import pymunk..

space = pymunk.Space()       # Create a Space which contain the simulation
space.gravity = 0,-1000      # Set its gravity

body = pymunk.Body(1,1666)   # Create a Body with mass and moment
body.position = 50,100       # Set the position of the body

poly = pymunk.Poly.create_box(body) # Create a box shape and attach to body
space.add(body, poly)        # Add both body and shape to the simulation

while True:                  # Infinite loop simulation
    space.step(0.02)         # Step the simulation one step forward
```

For more detailed and advanced examples, take a look at the included demos (in examples/).

Examples are not included if you install with *pip install pymunk*. Instead you need to download the source archive (pymunk-x.y.z.zip). Download available from <https://pypi.org/project/pymunk/#files>

CHAPTER 3

Documentation

The source distribution of Pymunk ships with a number of demos in the examples directory, and it also contains the full documentation including API reference.

You can also find the full documentation including examples and API reference on the Pymunk homepage, <http://www.pymunk.org>

The Pymunk Vision

“Make 2d physics easy to include in your game”

It is (or is striving to be):

- **Easy to use** - It should be easy to use, no complicated stuff should be needed to add physics to your game/program.
- **“Pythonic”** - It should not be visible that a c-library (chipmunk) is in the bottom, it should feel like a Python library (no strange naming, OO, no memory handling and more)
- **Simple to build & install** - You shouldn't need to have a zillion of libraries installed to make it install, or do a lot of command line tricks.
- **Multi-platform** - Should work on both Windows, *nix and OSX.
- **Non-intrusive** - It should not put restrictions on how you structure your program and not force you to use a special game loop, it should be possible to use with other libraries like Pygame and Pyglet.

CHAPTER 5

Contact & Support

Homepage <http://www.pymunk.org/>

Stackoverflow You can ask questions/browse old ones at Stackoverflow, just look for the Pymunk tag. <http://stackoverflow.com/questions/tagged/pymunk>

Forum Currently Pymunk has no separate forum, but uses the general Chipmunk forum at <http://chipmunk-physics.net/forum/index.php> Many issues are the same, like how to create a rag doll or why a fast moving object pass through a wall. If you have a Pymunk specific question feel free to mark your post with [pymunk] to make it stand out a bit.

E-Mail You can email me directly at vb@viblo.se

Issue Tracker Please use the issue tracker at github to report any issues you find: <https://github.com/viblo/pymunk/issues>

Regardless of the method you use I will try to answer your questions as soon as I see them. (And if you ask on SO or the forum other people might help as well!)

Dependencies / Requirements

Basically Pymunk have been made to be as easy to install and distribute as possible, usually *pip install* will take care of everything for you.

- Python (Runs on CPython 2.7 and 3.X. Pypy and Pypy3)
- Chipmunk (Source included, and on Windows and OSX its already compiled)
- CFFI (will be installed automatically by Pip)
- Setuptools (should be included with Pip)
- GCC and friends (optional, you need it to compile Chipmunk)
- Pygame (optional, you need it to run the Pygame based demos)
- Pyglet (optional, you need it to run the Pyglet based demos)
- Matplotlib & Jupyter Notebook (optional, you need it to run the Matplotlib based demos)
- Sphinx (optional, you need it to build documentation)

CHAPTER 7

Python 2 & Python 3

Pymunk has been tested and runs fine on both Python 2 and Python 3. It has been tested on recent versions of CPython (2 and 3) and Pypy. For an exact list of tested versions see the Travis and Appveyor test configs.

Chipmunk Compilation

This section is only required in case you don't install pymunk the normal way (*pip install* or *setup.py install*). Otherwise it's handled automatically by the install command.

Pymunk is built on top of the C library Chipmunk. It uses CFFI to interface with the Chipmunk library file. Because of this Chipmunk has to be compiled before it can be used with Pymunk. Compilation has to be done with GCC or another compiler that uses the same flags.

The source distribution does not include a pre-compiled Chipmunk library file, instead you need to build it yourself.

There are basically two options, either building it automatically as part of installation using for example Pip:

```
> pip install pymunk-source-dist.zip
```

Or if you have the source unpacked / you got Pymunk by cloning its git repo, you can explicitly tell Pymunk to compile it in place:

```
> python setup.py build_ext --inplace
```

Note that chipmunk is actually not built as a python extension, but distutils / setuptools doesn't currently handle pure native libraries that need to be built in a good way if built with `build_clib`.

The compiled file goes into the /pymunk folder (same as `space.py`, `body.py` and others).

9.1 News

9.1.1 Pymunk 5.4.0

Victor - 2018-10-24

Fix support for MacOS 10.14

Main fix is to allow Pymunk to be installed on latest version of MacOS. This release also contain a bunch of minor fixes and as usual an improvement of the docs, tests and examples.

Changes:

- On newer versions of MacOS only compile in 64bit mode (32bit is deprecated)
- Improved docs, examples and tests
- Fix in `moment_for_*` when passed `Vec2d` instead of tuple
- Fix case when adding or removing more than one obj to space during step.
- Allow threaded solver on Windows.
- Use `msys mingw` to compile `chipmunk` on Windows (prev solution was deprecated).

Enjoy!

9.1.2 Introductory video tutorials

Victor - 2018-02-25

Youtube user Attila has created a series of videos covering the basics of Pymunk. Take a look here for a gentle introduction into Pymunk:

9.1.3 Pymunk 5.3.2

Victor - 2017-09-16

Fixes ContactPointSet updating in Arbiter

This release contains a fix for the ContactPointSet on Arbiters. With this fix its possible to update the contacts during a collision callback, for example to update the normal like in the breakout game example.

Changes:

- Fix Arbiter.contact_point_set

9.1.4 Pymunk 5.3.1

Victor - 2017-07-15

Fix for Pycparser 2.18

This release contains a fix for the recently released Pycparser 2.18 which is used by Pymunk indirectly from its use of CFFI.

Changes:

- Fix broken callbacks when using Pycparser 2.18.

9.1.5 Pymunk 5.3.0

Victor - 2017-06-11

Pickle and copy support!

New in this release is pickle (save and load) and copy support. This has been on my mind for a long time, and when I got a feature request for it on Github by Rick-C-137 I had the final push to make it happen. See [examples/copy_and_pickle.py](#) for an example.

The feature itself is very easy to use, pickle works just as expected, and copy is a simple method call. However, be aware that support for pickle of Spaces with callback functions depends on the pickle protocol version. The oldest pickle protocol have limited capability to pickle functions, so to get maximum functionality use the latest pickle protocol possible.

Changes:

- Pickle support. Most objects can be pickled and un-pickled.
- Copy support and method. Most objects now have a copy() function. Also the standard library copy.deepcopy() function works as expected.
- Fixed bugs in BB.merge and other BB functions.
- Improved documentation and tests.
- New Kivy example (as mentioned in earlier news entry).

I hope you will like it!

9.1.6 New page theme

Victor - 2017-06-07

An mobile friendlier experience!

A couple of days ago I noticed that the Pymunk web page get a significant amount of traffic from mobile, and at the same time the Sphinx theme it uses is not built for mobile browsing. So as a result I decided to change theme to something that can scale down to mobile size better. I hope the new page gives a better experience for everyone!

9.1.7 Pymunk on Android

Victor - 2017-06-04

Pymunk runs on Android!

With the latest version (5.2.0) Pymunk can now be compiled and run on Android phones. Available as an example: [examples/kivy_pymunk_demo](#) is a Kivy example that can be built and run on Android.

Below is a screen cap from my phone (an Xperia X Compact) running the Kivy example. The example itself is an interactive variant of the logo animation used on the front page of Pymunk.org

9.1.8 Pymunk 5.2.0

Victor - 2017-03-25

Customized compile for ARM / Android

The main reason for this release is the ARM / Android cross compilation support thanks to the possibility to override the ccompiler and linker. After this release is out its possible to create a python-for-android build recipe for Pymunk without patching the Pymunk code. It should also be easier to build for other environments.

Changes

- Allow customization of the compilation of chipmunk by allowing overriding the compiler and linker with the CC, CFLAGS, LD and LDFLAGS environment variables. (usually you dont need this, but in some cases its useful)
- Fix sometimes broken Poly draw with pyglet_util.
- Add feature to let you set the mass of shapes and let Pymunk automatically calculate the body mass and moment.
- Dont use separate library naming for 32 and 64 bit builds. (Should not have any visible effect)

9.1.9 Pymunk 5.1.0

Victor - 2016-10-17

A speedier Pymunk has been released!

This release is made as follow up on the [Benchmarks](#) done on Pymunk 5.0 and 4.0. Pymunk 5.0 is already very fast on Pypy, but had some regressions in CPython. Turns out one big part in the change is how Vec2ds are handled in the two versions. Pymunk 5.1 contains optimized code to help reduce a big portion of this difference.

Changes

- Big performance increase compared to Pymunk 5.0 thanks to improved Vec2d handling.
- Documentation improvements.

- Small change in the return type of `Shape.point_query`. Now it correctly return a tuple of (distance, info) as is written in the docs.
- Split `Poly.create_box` into two methods, `Poly.create_box` and `Poly.create_box_bb` to make it more clear what is happening.

I hope you will enjoy this new release!

9.1.10 Pymunk 5.0.0

Victor - 2016-07-17

A new version of Pymunk!

This is a BIG release of Pymunk! Just in time before Pymunk turns 10 next year!

- Support for 64 bit Python on Windows
- Updated to use Chipmunk 7 which includes lots of great improvements
- Updated to use CFFI for wrapping, giving improved development and packaging (wheels, yay!)
- New util module with draw help for matplotlib (with example Jupyter notebooks)
- Support for automatically generate geometry. Can be used for such things as deformable terrain (example included).
- Deprecated obsolete submodule `pymunk.util`.
- Lots of smaller improvements

New in this release is also testing on Travis and Appveyor to ensure good code quality.

I hope you will enjoy this new release!

9.1.11 Move from ctypes to CFFI?

Victor - 2016-05-19

Should pymunk move to CFFI?

To make development of pymunk easier Im planning to move from using ctypes to CFFI for the low level Chipmunk wrapping. The idea is that CFFI is a active project which should mean it will be easier to get help, for example around the 64bit python problems on windows.

Please take a look at Issue 99 on github which tracks this switch. <https://github.com/viblo/pymunk/issues/99>

9.1.12 Travis-ci & tox

Victor - 2014-11-13

pymunk is now using travis-ci for continuous integration

In an effort to make testing and building of pymunk easier travis has been configured to build pymunk. At the same time support for tox was added to streamline local testing.

9.1.13 Move to Github

Victor - 2013-10-04

pymunk has moved its source and issue list to Github!

From the start pymunk has been hosted at Google Code, in the beginning using it for everything, source control, issue tracker, documentation and so on. During that time Github has become more and more popular and overall a better hosting platform.

At the same time distributed version control systems have risen in popularity over traditional ones like Subversion.

Adding to this Google Code will stop hosting binaries in January 2014.

Because of this I have been thinking a while about moving pymunk away from svn and google code. I had an issue open on google code in which all feedback proposed git and github, and that has been my own thought as well. And so, today the move has been completed!

To get the latest source you will need a git client and then do:

```
> git clone https://github.com/viblo/pymunk.git
```

If you prefer a graphical client (I do) I find SourceTree very good.

Issues have been migrated to <https://github.com/viblo/pymunk/issues>

Binaries will be available from PyPI just like before, but the binary hosting at Google Code will not get any updates.

The google code page will from now on only have a redirect to pymunk.org and github.

9.1.14 pymunk 4.0.0

Victor - 2013-08-25

A new release of pymunk is here!

This release is definitely a milestone, pymunk is now over 5 years old! (first version was released in February 2008, for the pyweek competition)

In this release a number of improvements have been made to pymunk. It includes debug drawing for pyglet (debug draw for pygame was introduced in pymunk 3), an updated Chipmunk version with the resulting API adjustments, more and better examples and overall polish as usual.

With the new Chipmunk version (6.2 beta), collision detection might behave a little bit differently as it uses a different algorithm compared to earlier versions. The new algorithm means that segments to segment collisions will be detected now. If you have some segments that you don't want to collide then you can use the sensor property, or a custom collision callback function.

To see the new pymunk.pyglet_util module in action check out the pyglet_util_demo.py example. It has an interface similar to the pygame_util, with a couple of changes because of differences between pyglet and pygame.

Some API additions and changes have been made. Its now legal to add and remove objects such as bodies and shapes during the simulation step (for example in a callback). The actual removal will be scheduled to occur as soon as the simulation step is complete. Other changes are the possibility to change body of a shape, to get the BB of a shape, and create a shape with empty body. On a body you can now retrieve the shapes and constraints attached to it.

This release has been tested and runs on CPython 2.5, 2.6, 2.7, 3.3 and Pypy 2.1. At least one run of the unit tests have been made on the following platforms: 32 bit CPython on Windows, 32 and 64 bit CPython on Linux, and 64 bit CPython on OSX. Pypy 2.1 on one of the above platforms.

Changes

- New draw module to help with pyglet prototyping
- Updated Chipmunk version, with new collision detected code.
- Added, improved and fixed broken examples
- Possible to switch bodies on shapes
- Made it legal to add and remove bodies during a simulation step
- Added shapes and constraints properties to Body
- Possible to get BB of a Shape, and they now allow empty body in constructor
- Added radius property to Poly shapes
- Renamed Poly.get_points to get_vertices
- Renamed the Segment.a and Segment.b properties to unsafe_set
- Added example of using pyinstaller
- Fixed a number of bugs reported
- Improved docs in various places
- General polish and cleanup

I hope you will enjoy this new release!

9.1.15 pymunk 3.0.0

Victor - 2012-09-02

I'm happy to announce pymunk 3!

This release is a definite improvement over the 2.x release line of pymunk. It features a much improved documentation, an updated Chipmunk version with accompanying API adjustments, more and cooler examples. Also, to help to do quick prototyping pymunk now includes a new module `pymunk.pygame_util` that can draw most physics objects on a pygame surface. Check out the new `pygame_util_demo.py` example to get an understanding of how it works.

Another new feature is improved support to run in non-debug mode. Its now possible to pass a compile flag to `setup.py` to build Chipmunk in release mode and there's a new module, `pymunkoptions` that can be used to turn pymunk debug prints off.

This release has been tested and runs on CPython 2.6, 2.7, 3.2. At least one run of the unit tests have been made on the following platforms: 32 bit Python on Windows, 32 and 64 bit Python on Linux, and 32 and 64 bit Python on OSX.

This release has also been tested on Pypy 1.9, with all tests passed!

Changes

- Several new and interesting examples added
- New draw module to help with pygame prototyping
- New `pymunkoptions` module to allow disable of debug
- Tested on OSX, includes a compiled dylib file
- Much extended and reworked documentation and homepage
- Update of Chipmunk
- Smaller API changes
- General polish and cleanup

- Shining new domain: www.pymunk.org

I hope you will like it!

9.1.16 pymunk 2.1.0

Victor - 2011-12-03

A bugfix release of pymunk is here!

The most visible change in this release is that now the source release contains all of it including examples and chipmunk source. :) Other fixes are a new velocity limit property of the body, and some removed methods (Reasoning behind removing them and still on same version: You would get an exception calling them anyway. The removal should not affect code that works). Note, all users should create static bodies by setting the input parameters to None, not using infinity. inf will be removed in an upcoming release.

Changes

- Marked pymunk.inf as deprecated
- Added velocity limit property to the body
- Fixed bug on 64bit python
- Recompiled chipmunk.dll with python 2.5
- Updated chipmunk source.
- New method in Vec2d to get int tuple
- Removed slew and resize hash methods
- Removed pymunk.init calls from examples/tests
- Updated examples/tests to create static bodies the good way

Have fun with it!

9.1.17 pymunk 2.0.0

Victor - 2011-09-04

Today I'm happy to announce the new pymunk 2 release!

New goodies in this release comes mainly from the updated chipmunk library. Its now possible for bodies to sleep, there is a new data structure holding the objects and other smaller improvements. The updated version number comes mainly from the new sleep methods.

Another new item in the release is some simplification, you now don't need to initialize pymunk on your own, thats done automatically on import. Another cool feature is that pymunk passes all its unit tests on the latest pypy source which I think is a great thing! Have not had time to do any performance tests, but pypy claims improvements of the ctypes library over cpython.

Note, this release is not completely backwards compatible with pymunk 1.0, some minor adjustments will be necessary (one of the reasons the major version number were increased).

Changes from the last release:

- Removed init pymunk method, its done automatically on import
- Support for sleeping bodies.
- Updated to latest version of Chipmunk

- More API docs, more unit tests.
- Only dependent on msvcrt.dll on windows now.
- Removed dependency on setuptools
- Minor updates on other API, added some missing properties and methods.

Enjoy!

9.1.18 Older news

Older news items have been archived.

9.2 Installation

Tip: You will find the latest released version at pypi: <https://pypi.python.org/pypi/pymunk>

9.2.1 Install Pymunk

Pymunk can be installed with pip install:

```
> pip install pymunk
```

On non-Windows OS such as OSX and Linux you need to have a GCC-compatible compiler installed.

On OSX you can install one with:

```
> xcode-select --install
```

On Linux you can install one with the package manager, for example on Ubuntu with:

```
> sudo apt-get install build-essential
```

9.2.2 Examples & Documentation

Because of their size are the examples and documentation available in the source distribution of Pymunk, but not the wheels. The source distribution is available from PyPI at <https://pypi.org/project/pymunk/#files> (Named pymunk-x.y.z.zip)

9.2.3 Advanced install

Another option is to use the standard setup.py way, in case you have downloaded the source distribution:

```
> python setup.py install
```

Note that this require a GCC compiler, which can be a bit tricky on Windows. If you are on Mac OS X or Linux you will probably need to run as a privileged user; for example using sudo:


```
> sudo python setup.py install
```

Once installed you should be able to import pymunk just as any other installed library. pymunk should also work just fine with virtualenv in case you want it installed in a contained environment.

9.2.4 Compile Chipmunk

If a compiled binary library of Chipmunk that works on your platform is not included in the release you will need to compile Chipmunk yourself. Another reason to compile chipmunk is if you want to run it in release mode to get rid of the debug prints it generates. If you just use pip install the compilation will happen automatically given that a compiler is available. You can also specifically compile Chipmunk as described below.

To compile Chipmunk:

```
> python setup.py build_ext
```

If you got the source and just want to use it directly you probably want to compile Chipmunk in-place, that way the output is put directly into the correct place in the source folder:

```
> python setup.py build_ext --inplace
```

On Windows you will need to use a gcc-compatible compiler. The pre-built version distributed with pymunk were compiled with the mingwpy GCC compiler at <https://mingwpy.github.io/>

See also:

Module *pymunkoptions* Options module that control runtime options of Pymunk such as debug settings. Use *pymunkoptions* together with release mode compilation to remove all debugs prints.

9.2.5 CFFI Installation

Sometimes you need to manually install the (non-python) dependencies of CFFI. Usually you will notice this as a installation failure when pip tries to install CFFI since CFFI is a dependency of Pymunk. This is not really part of Pymunk, but a brief description is available for your convenience.

You need to install two extra dependencies for CFFI to install properly. This can be handled by the package manager. The dependencies are *python-dev* and *libffi-dev*. Note that they might have slightly different names depending on the distribution, this is for Debian/Ubuntu. Just install them the normal way, for example like this if you use apt and Pip should be able to install CFFI properly:

```
> sudo apt-get install python-dev libffi-dev
```

9.3 Overview

9.3.1 Basics

There are 4 basic classes you will use in Pymunk.

Rigid Bodies (*pymunk.Body*) A rigid body holds the physical properties of an object. (mass, position, rotation, velocity, etc.) It does not have a shape by itself. If you've done physics with particles before, rigid bodies differ mostly in that they are able to rotate. Rigid bodies generally tend to have a 1:1 correlation to sprites in a game. You should structure your game so that you use the position and rotation of the rigid body for drawing your sprite.

Collision Shapes ([`pymunk.Circle`](#), [`pymunk.Segment`](#) and [`pymunk.Poly`](#)) By attaching shapes to bodies, you can define the a body's shape. You can attach many shapes to a single body to define a complex shape, or none if it doesn't require a shape.

Constraints/Joints ([`pymunk.constraint.PinJoint`](#), [`pymunk.constraint.SimpleMotor`](#) and many others) You can attach joints between two bodies to constrain their behavior.

Spaces ([`pymunk.Space`](#)) Spaces are the basic simulation unit in Pymunk. You add bodies, shapes and joints to a space, and then update the space as a whole. They control how all the rigid bodies, shapes, and constraints interact together.

The actual simulation is done by the Space. After adding the objects that should be simulated to the Space time is moved forward in small steps with the [`pymunk.Space.step\(\)`](#) function.

9.3.2 Model your physics objects

9.3.2.1 Object shape

What you see on the screen doesn't necessarily have to be exactly the same shape as the actual physics object. Usually the shape used for collision detection (and other physics simulation) is much simplified version of what is drawn on the screen. Even high end AAA games separate the collision shape from what is drawn on screen.

There are a number of reasons why its good to separate the collision shape and what is drawn.

- Using simpler collision shapes are faster. So if you have a very complicated object, for example a pine tree, maybe it can make sense to simplify its collision shape to a triangle for performance.
- Using a simpler collision shape make the simulation better. Lets say you have a floor made of stone with a small crack in the middle. If you drag a box over this floor it will get stuck on the crack. But if you simplify the floor to just a plane you avoid having to worry about stuff getting stuck in the crack.
- Making the collision shape smaller (or bigger) than the actual object makes gameplay better. Lets say you have a player controlled ship in a shoot-em-up type game. Many times it will feel more fun to play if you make the collision shape a little bit smaller compared to what it should be based on how it looks.

You can see an example of this in the [`using_sprites.py`](#) example included in Pymunk. There the physics shape is a triangle, but what is drawn is 3 boxes in a pyramid with a snake on top. Another example is in the [`platformer.py`](#) example, where the player is drawn as a girl in red and gray. However the physics shape is just a couple of circle shapes on top of each other.

9.3.2.2 Mass, weight and units

Sometimes users of Pymunk can be confused as to what unit everything is defined in. For example, is the mass of a Body in grams or kilograms? Pymunk is unit-less and does not care which unit you use. If you pass in seconds to a function expecting time, then your time unit is seconds. If you pass in pixels to functions that expect a distance, then your unit of distance is pixels.

Then derived units are just a combination of the above. So in the case with seconds and pixels the unit of velocity would be pixels / second.

(This is in contrast to some other physics engines which can have fixed units that you should use)

9.3.2.3 Looks before realism

How heavy is a bird in angry birds? It doest matter, its a cartoon!

Together with the units another key insight when setting up your simulation is to remember that it is a simulation, and in many cases the look and feel is much more important than actual realism. So for example, if you want to model a flipper game, the real power of the flipper and launchers doesn't matter at all, what is important is that the game feels "right" and is fun to use for your users.

Sometimes it make sense to start out with realistic units, to give you a feel for how big mass should be in comparison to gravity for example.

There are exceptions to this of course, when you actually want realism over the looks. In the end it is up to you as a user of Pymunk to decide.

9.3.3 Game loop / moving time forward

The most important part in your game loop is to keep the `dt` argument to the `pymunk.Space.step()` function constant. A constant time step makes the simulation much more stable and reliable.

There are several ways to do this, some more complicated than others. Which one is best for a particular program depends on the requirements.

Some good articles:

- <http://gameprogrammingpatterns.com/game-loop.html>
- <http://gafferongames.com/game-physics/fix-your-timestep/>
- <http://www.koonsolo.com/news/dewitters-gameloop/>

9.3.4 Unstable simulation?

Sometimes the simulation might not behave as expected. In extreme cases it can "blow up" and parts move anywhere without logic.

There a number of things to try if this happens:

- Make all the bodies of similar mass. It is easier for the physics engine to handle bodies with similar weight.
- Dont let two objects with infinite mass touch each other.
- Make the center of gravity in the middle of shapes instead of at the edge.
- Very thin shapes can behave strange, try to make them a little wider.
- Have a fixed time step (see the other sections of this guide).
- Call the `Space.step` function several times with smaller `dt` instead of only one time but with a bigger `dt`. (See the docs of `Space.step`)
- If you use a Motor joint, make sure to set its max force. Otherwise its power will be near infinite.

(Most of these suggestions are the same for most physics engines, not just Pymunk.)

9.3.5 Copy and Load/Save Pymunk objects

Most Pymunk objects can be copied and/or saved with pickle from the standard library. Since the implementation is generic it will also work to use other serializer libraries such as `jsonpickle` (in contrast to pickle the `jsonpickle` serializes to/from json) as long as they make use of the pickle infrastructure.

See the `copy_and_pickle.py` example for an example on how to save, load and copy Pymunk objects.

9.3.6 Additional info

As a complement to the Pymunk docs it can be good to read the [Chipmunk docs](#). Its made for Chipmunk, but Pymunk is build on top of Chipmunk and share most of the concepts, with the main difference being that Pymunk is used from Python while Chipmunk is a C-library.

9.4 API Reference

9.4.1 pymunk Package

Submodules

9.4.1.1 pymunk.autogeometry Module

This module contain functions for automatic generation of geometry, for example from an image.

Example:

```
>>> import pymunk
>>> from pymunk.autogeometry import march_soft
>>> img = [
...     "  xx  ",
...     "  xx  ",
...     "  xx  ",
...     "  xx  ",
...     "  xx  ",
...     " xxxxx",
...     " xxxxx",
... ]
>>> segments = []

>>> def segment_func(v0, v1):
...     segments.append((tuple(v0), tuple(v1)))
>>> def sample_func(point):
...     x = int(point.x)
...     y = int(point.y)
...     return 1 if img[y][x] == "x" else 0

>>> march_soft(pymunk.BB(0,0,6,6), 7, 7, .5, segment_func, sample_func)
>>> print(len(segments))
13
```

The information in segments can now be used to create geometry, for example as a Pymunk Poly or Segment:

```
>>> s = pymunk.Space()
>>> for (a,b) in segments:
...     segment = pymunk.Segment(s.static_body, a, b, 5)
...     s.add(segment)
```

`pymunk.autogeometry.is_closed(polyline)`

Returns true if the first vertex is equal to the last.

Parameters `polyline` (`[(float, float)]`) – Polyline to simplify.

Return type `bool`

`pymunk.autogeometry.simplify_curves` (*polyline, tolerance*)

Returns a copy of a polyline simplified by using the Douglas-Peucker algorithm.

This works very well on smooth or gently curved shapes, but not well on straight edged or angular shapes.

Parameters

- **polyline** (*[(float, float)]*) – Polyline to simplify.
- **tolerance** (*float*) – A higher value means more error is tolerated.

Return type *[(float, float)]*

`pymunk.autogeometry.simplify_vertexes` (*polyline, tolerance*)

Returns a copy of a polyline simplified by discarding “flat” vertexes.

This works well on straight edged or angular shapes, not as well on smooth shapes.

Parameters

- **polyline** (*[(float, float)]*) – Polyline to simplify.
- **tolerance** (*float*) – A higher value means more error is tolerated.

Return type *[(float, float)]*

`pymunk.autogeometry.to_convex_hull` (*polyline, tolerance*)

Get the convex hull of a polyline as a looped polyline.

Parameters

- **polyline** (*[(float, float)]*) – Polyline to simplify.
- **tolerance** (*float*) – A higher value means more error is tolerated.

Return type *[(float, float)]*

`pymunk.autogeometry.convex_decomposition` (*polyline, tolerance*)

Get an approximate convex decomposition from a polyline.

Returns a list of convex hulls that match the original shape to within tolerance.

Note: If the input is a self intersecting polygon, the output might end up overly simplified.

Parameters

- **polyline** (*[(float, float)]*) – Polyline to simplify.
- **tolerance** (*float*) – A higher value means more error is tolerated.

Return type *[(float, float)]*

class `pymunk.autogeometry.PolylineSet`

Bases: `_abcoll.Sequence`

A set of Polylines.

Mainly intended to be used for its `collect_segment()` function when generating geometry with the `march_soft()` and `march_hard()` functions.

__init__ ()

`x.__init__(...)` initializes x; see `help(type(x))` for signature

collect_segment (*v0*, *v1*)

Add a line segment to a polyline set.

A segment will either start a new polyline, join two others, or add to or loop an existing polyline. This is mostly intended to be used as a callback directly from `march_soft()` or `march_hard()`.

Parameters

- **v0** (*(float, float)*) – Start of segment
- **v1** (*(float, float)*) – End of segment

count (*value*) → integer – return number of occurrences of value

index (*value*) → integer – return first index of value.

Raises `ValueError` if the value is not present.

`pymunk.autogeometry.march_soft` (*bb*, *x_samples*, *y_samples*, *threshold*, *segment_func*, *sample_func*)

Trace an *anti-aliased* contour of an image along a particular threshold.

The given number of samples will be taken and spread across the bounding box area using the sampling function and context.

Parameters

- **bb** (*BB*) – Bounding box of the area to sample within
- **x_samples** (*int*) – Number of samples in x
- **y_samples** (*int*) – Number of samples in y
- **threshold** (*float*) – A higher value means more error is tolerated
- **segment_func** (*func* (*v0* : *Vec2d*, *v1* : *Vec2d*)) – The segment function will be called for each segment detected that lies along the density contour for threshold.
- **sample_func** (*func* (*point* : *Vec2d*) → *float*) – The sample function will be called for *x_samples* * *y_samples* spread across the bounding box area, and should return a float.

`pymunk.autogeometry.march_hard` (*bb*, *x_samples*, *y_samples*, *threshold*, *segment_func*, *sample_func*)

Trace an *aliased* curve of an image along a particular threshold.

The given number of samples will be taken and spread across the bounding box area using the sampling function and context.

Parameters

- **bb** (*BB*) – Bounding box of the area to sample within
- **x_samples** (*int*) – Number of samples in x
- **y_samples** (*int*) – Number of samples in y
- **threshold** (*float*) – A higher value means more error is tolerated
- **segment_func** (*func* (*v0* : *Vec2d*, *v1* : *Vec2d*)) – The segment function will be called for each segment detected that lies along the density contour for threshold.
- **sample_func** (*func* (*point* : *Vec2d*) → *float*) – The sample function will be called for *x_samples* * *y_samples* spread across the bounding box area, and should return a float.

9.4.1.2 pymunk.constraint Module

A constraint is something that describes how two bodies interact with each other. (how they constrain each other). Constraints can be simple joints that allow bodies to pivot around each other like the bones in your body, or they can be more abstract like the gear joint or motors.

This submodule contain all the constraints that are supported by Pymunk.

All the constraints support copy and pickle from the standard library. Custom properties set on a constraint will also be copied/pickled.

Chipmunk has a good overview of the different constraint on youtube which works fine to showcase them in Pymunk as well. <http://www.youtube.com/watch?v=ZgJJZTS0aMM>

Example:

```
>>> import pymunk
>>> s = pymunk.Space()
>>> a,b = pymunk.Body(10,10), pymunk.Body(10,10)
>>> c = pymunk.PivotJoint(a, b, (0,0))
>>> s.add(c)
```

class pymunk.constraint.Constraint (constraint=None)
Bases: pymunk._pickle.PickleMixin, object

Base class of all constraints.

You usually don't want to create instances of this class directly, but instead use one of the specific constraints such as the PinJoint.

__init__ (constraint=None)
x.__init__(...) initializes x; see help(type(x)) for signature

a
The first of the two bodies constrained

activate_bodies ()
Activate the bodies this constraint is attached to

b
The second of the two bodies constrained

collide_bodies
Constraints can be used for filtering collisions too.

When two bodies collide, Pymunk ignores the collisions if this property is set to False on any constraint that connects the two bodies. Defaults to True. This can be used to create a chain that self collides, but adjacent links in the chain do not collide.

copy ()
Create a deep copy of this constraint.

error_bias
The percentage of joint error that remains unfixed after a second.

This works exactly the same as the collision bias property of a space, but applies to fixing error (stretching) of joints instead of overlapping collisions.

Defaults to pow(1.0 - 0.1, 60.0) meaning that it will correct 10% of the error every 1/60th of a second.

impulse
The most recent impulse that constraint applied.

To convert this to a force, divide by the timestep passed to `space.step()`. You can use this to implement breakable joints to check if the force they attempted to apply exceeded a certain threshold.

max_bias

The maximum speed at which the constraint can apply error correction.

Defaults to infinity

max_force

The maximum force that the constraint can use to act on the two bodies.

Defaults to infinity

class `pymunk.constraint.PinJoint` (*a*, *b*, *anchor_a*=(0, 0), *anchor_b*=(0, 0))

Bases: `pymunk.constraint.Constraint`

Keeps the anchor points at a set distance from one another.

__init__ (*a*, *b*, *anchor_a*=(0, 0), *anchor_b*=(0, 0))

a and *b* are the two bodies to connect, and *anchor_a* and *anchor_b* are the anchor points on those bodies.

The distance between the two anchor points is measured when the joint is created. If you want to set a specific distance, use the setter function to override it.

a

The first of the two bodies constrained

activate_bodies ()

Activate the bodies this constraint is attached to

anchor_a

anchor_b

b

The second of the two bodies constrained

collide_bodies

Constraints can be used for filtering collisions too.

When two bodies collide, Pymunk ignores the collisions if this property is set to False on any constraint that connects the two bodies. Defaults to True. This can be used to create a chain that self collides, but adjacent links in the chain do not collide.

copy ()

Create a deep copy of this constraint.

distance

error_bias

The percentage of joint error that remains unfixed after a second.

This works exactly the same as the collision bias property of a space, but applies to fixing error (stretching) of joints instead of overlapping collisions.

Defaults to `pow(1.0 - 0.1, 60.0)` meaning that it will correct 10% of the error every 1/60th of a second.

impulse

The most recent impulse that constraint applied.

To convert this to a force, divide by the timestep passed to `space.step()`. You can use this to implement breakable joints to check if the force they attempted to apply exceeded a certain threshold.

max_bias

The maximum speed at which the constraint can apply error correction.

Defaults to infinity

max_force

The maximum force that the constraint can use to act on the two bodies.

Defaults to infinity

class `pymunk.constraint.SlideJoint` (*a, b, anchor_a, anchor_b, min, max*)

Bases: `pymunk.constraint.Constraint`

Like pin joints, but have a minimum and maximum distance. A chain could be modeled using this joint. It keeps the anchor points from getting to far apart, but will allow them to get closer together.

__init__ (*a, b, anchor_a, anchor_b, min, max*)

a and *b* are the two bodies to connect, *anchor_a* and *anchor_b* are the anchor points on those bodies, and *min* and *max* define the allowed distances of the anchor points.

a

The first of the two bodies constrained

activate_bodies ()

Activate the bodies this constraint is attached to

anchor_a

anchor_b

b

The second of the two bodies constrained

collide_bodies

Constraints can be used for filtering collisions too.

When two bodies collide, Pymunk ignores the collisions if this property is set to False on any constraint that connects the two bodies. Defaults to True. This can be used to create a chain that self collides, but adjacent links in the chain do not collide.

copy ()

Create a deep copy of this constraint.

error_bias

The percentage of joint error that remains unfixed after a second.

This works exactly the same as the collision bias property of a space, but applies to fixing error (stretching) of joints instead of overlapping collisions.

Defaults to `pow(1.0 - 0.1, 60.0)` meaning that it will correct 10% of the error every 1/60th of a second.

impulse

The most recent impulse that constraint applied.

To convert this to a force, divide by the timestep passed to `space.step()`. You can use this to implement breakable joints to check if the force they attempted to apply exceeded a certain threshold.

max

max_bias

The maximum speed at which the constraint can apply error correction.

Defaults to infinity

max_force

The maximum force that the constraint can use to act on the two bodies.

Defaults to infinity

min

class pymunk.constraint.PivotJoint(a, b, *args)

Bases: [pymunk.constraint.Constraint](#)

Simply allow two objects to pivot about a single point.

__init__(a, b, *args)

a and b are the two bodies to connect, and pivot is the point in world coordinates of the pivot.

Because the pivot location is given in world coordinates, you must have the bodies moved into the correct positions already. Alternatively you can specify the joint based on a pair of anchor points, but make sure you have the bodies in the right place as the joint will fix itself as soon as you start simulating the space.

That is, either create the joint with PivotJoint(a, b, pivot) or PivotJoint(a, b, anchor_a, anchor_b).

Parameters

- **a** ([Body](#)) – The first of the two bodies
- **b** ([Body](#)) – The second of the two bodies
- **args** ((float, float) or (float, float) (float, float)) – Either one pivot point, or two anchor points

a

The first of the two bodies constrained

activate_bodies()

Activate the bodies this constraint is attached to

anchor_a

anchor_b

b

The second of the two bodies constrained

collide_bodies

Constraints can be used for filtering collisions too.

When two bodies collide, Pymunk ignores the collisions if this property is set to False on any constraint that connects the two bodies. Defaults to True. This can be used to create a chain that self collides, but adjacent links in the chain do not collide.

copy()

Create a deep copy of this constraint.

error_bias

The percentage of joint error that remains unfixed after a second.

This works exactly the same as the collision bias property of a space, but applies to fixing error (stretching) of joints instead of overlapping collisions.

Defaults to pow(1.0 - 0.1, 60.0) meaning that it will correct 10% of the error every 1/60th of a second.

impulse

The most recent impulse that constraint applied.

To convert this to a force, divide by the timestep passed to space.step(). You can use this to implement breakable joints to check if the force they attempted to apply exceeded a certain threshold.

max_bias

The maximum speed at which the constraint can apply error correction.

Defaults to infinity

max_force

The maximum force that the constraint can use to act on the two bodies.

Defaults to infinity

class `pymunk.constraint.GrooveJoint` (*a, b, groove_a, groove_b, anchor_b*)

Bases: `pymunk.constraint.Constraint`

Similar to a pivot joint, but one of the anchors is on a linear slide instead of being fixed.

__init__ (*a, b, groove_a, groove_b, anchor_b*)

The groove goes from *groove_a* to *groove_b* on body *a*, and the pivot is attached to *anchor_b* on body *b*.

All coordinates are body local.

a

The first of the two bodies constrained

activate_bodies ()

Activate the bodies this constraint is attached to

anchor_b**b**

The second of the two bodies constrained

collide_bodies

Constraints can be used for filtering collisions too.

When two bodies collide, Pymunk ignores the collisions if this property is set to False on any constraint that connects the two bodies. Defaults to True. This can be used to create a chain that self collides, but adjacent links in the chain do not collide.

copy ()

Create a deep copy of this constraint.

error_bias

The percentage of joint error that remains unfixed after a second.

This works exactly the same as the collision bias property of a space, but applies to fixing error (stretching) of joints instead of overlapping collisions.

Defaults to `pow(1.0 - 0.1, 60.0)` meaning that it will correct 10% of the error every 1/60th of a second.

groove_a**groove_b****impulse**

The most recent impulse that constraint applied.

To convert this to a force, divide by the timestep passed to `space.step()`. You can use this to implement breakable joints to check if the force they attempted to apply exceeded a certain threshold.

max_bias

The maximum speed at which the constraint can apply error correction.

Defaults to infinity

max_force

The maximum force that the constraint can use to act on the two bodies.

Defaults to infinity

class pymunk.constraint.DampedSpring(*a, b, anchor_a, anchor_b, rest_length, stiffness, damping*)

Bases: `pymunk.constraint.Constraint`

A damped spring

__init__(*a, b, anchor_a, anchor_b, rest_length, stiffness, damping*)
Defined much like a slide joint.

Parameters

- **a** (`Body`) – Body a
- **b** (`Body`) – Body b
- **anchor_a** (`((float,float))`) – Anchor point a, relative to body a
- **anchor_b** (`((float,float))`) – Anchor point b, relative to body b
- **rest_length** (`float`) – The distance the spring wants to be.
- **stiffness** (`float`) – The spring constant (Young’s modulus).
- **damping** (`float`) – How soft to make the damping of the spring.

a

The first of the two bodies constrained

activate_bodies()

Activate the bodies this constraint is attached to

anchor_a

anchor_b

b

The second of the two bodies constrained

collide_bodies

Constraints can be used for filtering collisions too.

When two bodies collide, Pymunk ignores the collisions if this property is set to False on any constraint that connects the two bodies. Defaults to True. This can be used to create a chain that self collides, but adjacent links in the chain do not collide.

copy()

Create a deep copy of this constraint.

damping

How soft to make the damping of the spring.

error_bias

The percentage of joint error that remains unfixed after a second.

This works exactly the same as the collision bias property of a space, but applies to fixing error (stretching) of joints instead of overlapping collisions.

Defaults to `pow(1.0 - 0.1, 60.0)` meaning that it will correct 10% of the error every 1/60th of a second.

impulse

The most recent impulse that constraint applied.

To convert this to a force, divide by the timestep passed to `space.step()`. You can use this to implement breakable joints to check if the force they attempted to apply exceeded a certain threshold.

max_bias

The maximum speed at which the constraint can apply error correction.

Defaults to infinity

max_force

The maximum force that the constraint can use to act on the two bodies.

Defaults to infinity

rest_length

The distance the spring wants to be.

stiffness

The spring constant (Young's modulus).

class `pymunk.constraint.DampedRotarySpring` (*a, b, rest_angle, stiffness, damping*)

Bases: `pymunk.constraint.Constraint`

Like a damped spring, but works in an angular fashion

__init__ (*a, b, rest_angle, stiffness, damping*)

Like a damped spring, but works in an angular fashion.

Parameters

- **a** (`Body`) – Body a
- **b** (`Body`) – Body b
- **rest_angle** (`float`) – The relative angle in radians that the bodies want to have
- **stiffness** (`float`) – The spring constant (Young's modulus).
- **damping** (`float`) – How soft to make the damping of the spring.

a

The first of the two bodies constrained

activate_bodies ()

Activate the bodies this constraint is attached to

b

The second of the two bodies constrained

collide_bodies

Constraints can be used for filtering collisions too.

When two bodies collide, Pymunk ignores the collisions if this property is set to False on any constraint that connects the two bodies. Defaults to True. This can be used to create a chain that self collides, but adjacent links in the chain do not collide.

copy ()

Create a deep copy of this constraint.

damping

How soft to make the damping of the spring.

error_bias

The percentage of joint error that remains unfixed after a second.

This works exactly the same as the collision bias property of a space, but applies to fixing error (stretching) of joints instead of overlapping collisions.

Defaults to `pow(1.0 - 0.1, 60.0)` meaning that it will correct 10% of the error every 1/60th of a second.

impulse

The most recent impulse that constraint applied.

To convert this to a force, divide by the timestep passed to `space.step()`. You can use this to implement breakable joints to check if the force they attempted to apply exceeded a certain threshold.

max_bias

The maximum speed at which the constraint can apply error correction.

Defaults to infinity

max_force

The maximum force that the constraint can use to act on the two bodies.

Defaults to infinity

rest_angle

The relative angle in radians that the bodies want to have

stiffness

The spring constant (Young's modulus).

class `pymunk.constraint.RotaryLimitJoint` (*a, b, min, max*)

Bases: `pymunk.constraint.Constraint`

Constrains the relative rotations of two bodies.

__init__ (*a, b, min, max*)

Constrains the relative rotations of two bodies.

min and *max* are the angular limits in radians. It is implemented so that it's possible to for the range to be greater than a full revolution.

a

The first of the two bodies constrained

activate_bodies ()

Activate the bodies this constraint is attached to

b

The second of the two bodies constrained

collide_bodies

Constraints can be used for filtering collisions too.

When two bodies collide, Pymunk ignores the collisions if this property is set to False on any constraint that connects the two bodies. Defaults to True. This can be used to create a chain that self collides, but adjacent links in the chain do not collide.

copy ()

Create a deep copy of this constraint.

error_bias

The percentage of joint error that remains unfixed after a second.

This works exactly the same as the collision bias property of a space, but applies to fixing error (stretching) of joints instead of overlapping collisions.

Defaults to `pow(1.0 - 0.1, 60.0)` meaning that it will correct 10% of the error every 1/60th of a second.

impulse

The most recent impulse that constraint applied.

To convert this to a force, divide by the timestep passed to `space.step()`. You can use this to implement breakable joints to check if the force they attempted to apply exceeded a certain threshold.

max**max_bias**

The maximum speed at which the constraint can apply error correction.

Defaults to infinity

max_force

The maximum force that the constraint can use to act on the two bodies.

Defaults to infinity

min

class pymunk.constraint.**RatchetJoint** (*a, b, phase, ratchet*)

Bases: [pymunk.constraint.Constraint](#)

Works like a socket wrench.

__init__ (*a, b, phase, ratchet*)

Works like a socket wrench.

ratchet is the distance between “clicks”, phase is the initial offset to use when deciding where the ratchet angles are.

a

The first of the two bodies constrained

activate_bodies ()

Activate the bodies this constraint is attached to

angle**b**

The second of the two bodies constrained

collide_bodies

Constraints can be used for filtering collisions too.

When two bodies collide, Pymunk ignores the collisions if this property is set to False on any constraint that connects the two bodies. Defaults to True. This can be used to create a chain that self collides, but adjacent links in the chain do not collide.

copy ()

Create a deep copy of this constraint.

error_bias

The percentage of joint error that remains unfixed after a second.

This works exactly the same as the collision bias property of a space, but applies to fixing error (stretching) of joints instead of overlapping collisions.

Defaults to pow(1.0 - 0.1, 60.0) meaning that it will correct 10% of the error every 1/60th of a second.

impulse

The most recent impulse that constraint applied.

To convert this to a force, divide by the timestep passed to space.step(). You can use this to implement breakable joints to check if the force they attempted to apply exceeded a certain threshold.

max_bias

The maximum speed at which the constraint can apply error correction.

Defaults to infinity

max_force

The maximum force that the constraint can use to act on the two bodies.

Defaults to infinity

phase
ratchet

class pymunk.constraint.GearJoint (*a, b, phase, ratio*)

Bases: `pymunk.constraint.Constraint`

Keeps the angular velocity ratio of a pair of bodies constant.

__init__ (*a, b, phase, ratio*)

Keeps the angular velocity ratio of a pair of bodies constant.

ratio is always measured in absolute terms. It is currently not possible to set the ratio in relation to a third body's angular velocity. phase is the initial angular offset of the two bodies.

a

The first of the two bodies constrained

activate_bodies ()

Activate the bodies this constraint is attached to

b

The second of the two bodies constrained

collide_bodies

Constraints can be used for filtering collisions too.

When two bodies collide, Pymunk ignores the collisions if this property is set to False on any constraint that connects the two bodies. Defaults to True. This can be used to create a chain that self collides, but adjacent links in the chain do not collide.

copy ()

Create a deep copy of this constraint.

error_bias

The percentage of joint error that remains unfixed after a second.

This works exactly the same as the collision bias property of a space, but applies to fixing error (stretching) of joints instead of overlapping collisions.

Defaults to $\text{pow}(1.0 - 0.1, 60.0)$ meaning that it will correct 10% of the error every 1/60th of a second.

impulse

The most recent impulse that constraint applied.

To convert this to a force, divide by the timestep passed to `space.step()`. You can use this to implement breakable joints to check if the force they attempted to apply exceeded a certain threshold.

max_bias

The maximum speed at which the constraint can apply error correction.

Defaults to infinity

max_force

The maximum force that the constraint can use to act on the two bodies.

Defaults to infinity

phase
ratio

class `pymunk.constraint.SimpleMotor(a, b, rate)`

Bases: `pymunk.constraint.Constraint`

Keeps the relative angular velocity of a pair of bodies constant.

__init__(a, b, rate)

Keeps the relative angular velocity of a pair of bodies constant.

rate is the desired relative angular velocity. You will usually want to set an force (torque) maximum for motors as otherwise they will be able to apply a nearly infinite torque to keep the bodies moving.

a

The first of the two bodies constrained

activate_bodies()

Activate the bodies this constraint is attached to

b

The second of the two bodies constrained

collide_bodies

Constraints can be used for filtering collisions too.

When two bodies collide, Pymunk ignores the collisions if this property is set to False on any constraint that connects the two bodies. Defaults to True. This can be used to create a chain that self collides, but adjacent links in the chain do not collide.

copy()

Create a deep copy of this constraint.

error_bias

The percentage of joint error that remains unfixed after a second.

This works exactly the same as the collision bias property of a space, but applies to fixing error (stretching) of joints instead of overlapping collisions.

Defaults to `pow(1.0 - 0.1, 60.0)` meaning that it will correct 10% of the error every 1/60th of a second.

impulse

The most recent impulse that constraint applied.

To convert this to a force, divide by the timestep passed to `space.step()`. You can use this to implement breakable joints to check if the force they attempted to apply exceeded a certain threshold.

max_bias

The maximum speed at which the constraint can apply error correction.

Defaults to infinity

max_force

The maximum force that the constraint can use to act on the two bodies.

Defaults to infinity

rate

The desired relative angular velocity

9.4.1.3 `pymunk.vec2d` Module

This module contain the `Vec2d` class that is used in all of pymunk when a vector is needed.

The `Vec2d` class is used almost everywhere in `pymunk` for 2d coordinates and vectors, for example to define gravity vector in a space. However, `pymunk` is smart enough to convert tuples or tuple like objects to `Vec2ds` so you usually do not need to explicitly do conversions if you happen to have a tuple:

```
>>> import pymunk
>>> space = pymunk.Space()
>>> space.gravity
Vec2d(0.0, 0.0)
>>> space.gravity = 3,5
>>> space.gravity
Vec2d(3.0, 5.0)
>>> space.gravity += 2,6
>>> space.gravity
Vec2d(5.0, 11.0)
```

More examples:

```
>>> from pymunk.vec2d import Vec2d
>>> Vec2d(7.3, 4.2)
Vec2d(7.3, 4.2)
>>> Vec2d((7.3, 4.2))
Vec2d(7.3, 4.2)
>>> Vec2d(7.3, 4.2) + Vec2d((1,2))
Vec2d(8.3, 6.2)
```

class `pymunk.vec2d.Vec2d` (*x_or_pair=None, y=None*)

Bases: `object`

2d vector class, supports vector and scalar operators, and also provides some high level functions.

__init__ (*x_or_pair=None, y=None*)

`x.__init__(...)` initializes `x`; see `help(type(x))` for signature

angle

Gets or sets the angle (in radians) of a vector

angle_degrees

Gets or sets the angle (in degrees) of a vector

convert_to_basis (*x_vector, y_vector*)

cpvrotate (*other*)

Uses complex multiplication to rotate this vector by the other.

cpvunrotate (*other*)

The inverse of `cpvrotate`

cross (*other*)

The cross product between the vector and other vector `v1.cross(v2) -> v1.x*v2.y - v1.y*v2.x`

Returns The cross product

dot (*other*)

The dot product between the vector and other vector `v1.dot(v2) -> v1.x*v2.x + v1.y*v2.y`

Returns The dot product

get_angle ()

get_angle_between (*other*)

Get the angle between the vector and the other in radians

Returns The angle

get_angle_degrees ()

get_angle_degrees_between (*other*)

Get the angle between the vector and the other in degrees

Returns The angle (in degrees)

get_dist_sqrd (*other*)

The squared distance between the vector and other vector It is more efficient to use this method than to call `get_distance()` first and then do a `sqrt()` on the result.

Returns The squared distance

get_distance (*other*)

The distance between the vector and other vector

Returns The distance

get_length ()

Get the length of the vector.

Returns The length

get_length_sqrd ()

Get the squared length of the vector. It is more efficient to use this method instead of first call `get_length()` or access `.length` and then do a `sqrt()`.

Returns The squared length

int_tuple

Return the x and y values of this vector as ints

interpolate_to (*other, range*)

length

Gets or sets the magnitude of the vector

normalize_return_length ()

Normalize the vector and return its length before the normalization

Returns The length before the normalization

normalized ()

Get a normalized copy of the vector Note: This function will return 0 if the length of the vector is 0.

Returns A normalized vector

static ones ()

A vector where both x and y is 1

perpendicular ()

perpendicular_normal ()

projection (*other*)

rotate (*angle_radians*)

Rotate the vector by `angle_radians` radians.

rotate_degrees (*angle_degrees*)

Rotate the vector by `angle_degrees` degrees.

rotated (*angle_radians*)

Create and return a new vector by rotating this vector by *angle_radians* radians.

Returns Rotated vector

rotated_degrees (*angle_degrees*)

Create and return a new vector by rotating this vector by *angle_degrees* degrees.

Returns Rotade vector

static unit ()

A unit vector pointing up

x

y

static zero ()

A vector of zero length

9.4.1.4 pymunk.matplotlib_util Module

This submodule contains helper functions to help with quick prototyping using pymunk together with pyglet.

Intended to help with debugging and prototyping, not for actual production use in a full application. The methods contained in this module is opinionated about your coordinate system and not very optimized (they use batched drawing, but there is probably room for optimizations still).

class pymunk.matplotlib_util.DrawOptions (*ax*)

Bases: pymunk.space_debug_draw_options.SpaceDebugDrawOptions

DRAW_COLLISION_POINTS

alias of CP_SPACE_DEBUG_DRAW_COLLISION_POINTS

DRAW_CONSTRAINTS

alias of CP_SPACE_DEBUG_DRAW_CONSTRAINTS

DRAW_SHAPES

alias of CP_SPACE_DEBUG_DRAW_SHAPES

__init__ (*ax*)

DrawOptions for space.debug_draw() to draw a space on a *ax* object.

Typical usage:

```
>>> import matplotlib as mpl
>>> import pymunk
>>> import pymunk.matplotlib_util
>>> my_space = pymunk.Space()
>>> fig, ax = mpl.subplot()
>>> options = pymunk.matplotlib_util.DrawOptions(ax)
>>> my_space.debug_draw(options)
```

You can control the color of a Shape by setting *shape.color* to the color you want it drawn in.

```
>>> my_shape.color = (1, 0, 0, 1) # will draw my_shape in red
```

See `matplotlib_util.demo.py` for a full example

Param

ax: `matplotlib.Axes` A matplotlib Axes object.

collision_point_color
color_for_shape (*shape*)
constraint_color
draw_circle (*pos, angle, radius, outline_color, fill_color*)
draw_dot (*size, pos, color*)
draw_fat_segment (*a, b, radius, outline_color, fill_color*)
draw_polygon (*verts, radius, outline_color, fill_color*)
draw_segment (*a, b, color*)

flags

Bit flags which of shapes, joints and collisions should be drawn.

By default all 3 flags are set, meaning shapes, joints and collisions will be drawn.

Example using the basic text only DebugDraw implementation (normally you would use the desired backend instead, such as *pygame_util.DrawOptions* or *pyglet_util.DrawOptions*):

```
>>> import pymunk
>>> s = pymunk.Space()
>>> b = pymunk.Body()
>>> c = pymunk.Circle(b, 10)
>>> c.mass = 3
>>> s.add(b, c)
>>> s.add(pymunk.Circle(s.static_body, 3))
>>> s.step(0.01)
>>> options = pymunk.SpaceDebugDrawOptions()
```

```
>>> # Only draw the shapes, nothing else:
>>> options.flags = pymunk.SpaceDebugDrawOptions.DRAW_SHAPES
>>> s.debug_draw(options)
('draw_circle', (Vec2d(0.0, 0.0), 0.0, 10.0, SpaceDebugColor(r=44.0, g=62.0,
↪b=80.0, a=255.0), SpaceDebugColor(r=52.0, g=152.0, b=219.0, a=255.0)))
('draw_circle', (Vec2d(0.0, 0.0), 0.0, 3.0, SpaceDebugColor(r=44.0, g=62.0,
↪b=80.0, a=255.0), SpaceDebugColor(r=149.0, g=165.0, b=166.0, a=255.0)))
```

```
>>> # Draw the shapes and collision points:
>>> options.flags = pymunk.SpaceDebugDrawOptions.DRAW_SHAPES
>>> options.flags |= pymunk.SpaceDebugDrawOptions.DRAW_COLLISION_POINTS
>>> s.debug_draw(options)
('draw_circle', (Vec2d(0.0, 0.0), 0.0, 10.0, SpaceDebugColor(r=44.0, g=62.0,
↪b=80.0, a=255.0), SpaceDebugColor(r=52.0, g=152.0, b=219.0, a=255.0)))
('draw_circle', (Vec2d(0.0, 0.0), 0.0, 3.0, SpaceDebugColor(r=44.0, g=62.0,
↪b=80.0, a=255.0), SpaceDebugColor(r=149.0, g=165.0, b=166.0, a=255.0)))
('draw_segment', (Vec2d(1.0, 0.0), Vec2d(-8.0, 0.0), SpaceDebugColor(r=231.0,
↪g=76.0, b=60.0, a=255.0)))
```

shape_dynamic_color = SpaceDebugColor(r=52, g=152, b=219, a=255)
shape_kinematic_color = SpaceDebugColor(r=39, g=174, b=96, a=255)
shape_outline_color
shape_sleeping_color = SpaceDebugColor(r=114, g=148, b=168, a=255)
shape_static_color = SpaceDebugColor(r=149, g=165, b=166, a=255)

9.4.1.5 pymunk.pygame_util Module

This submodule contains helper functions to help with quick prototyping using pymunk together with pygame.

Intended to help with debugging and prototyping, not for actual production use in a full application. The methods contained in this module is opinionated about your coordinate system and not in any way optimized.

class pymunk.pygame_util.**DrawOptions** (*surface*)
Bases: pymunk.space_debug_draw_options.SpaceDebugDrawOptions

DRAW_COLLISION_POINTS

alias of CP_SPACE_DEBUG_DRAW_COLLISION_POINTS

DRAW_CONSTRAINTS

alias of CP_SPACE_DEBUG_DRAW_CONSTRAINTS

DRAW_SHAPES

alias of CP_SPACE_DEBUG_DRAW_SHAPES

__init__ (*surface*)

Draw a pymunk.Space on a pygame.Surface object.

Typical usage:

```
>>> import pymunk
>>> import pymunk.pygame_util
>>> surface = pygame.Surface((10,10))
>>> s = pymunk.Space()
>>> options = pymunk.pygame_util.DrawOptions(surface)
>>> s.debug_draw(options)
```

Since pygame uses a coordinate system where y points down (compared to most other cases where a positive y points upwards), we might want to make adjustments for that with the *positive_y_is_up* variable.

By default drawing is done with positive y pointing up, but that will make conversion from pygame coordinate to pymunk coordinate nessecary. If you do a lot of those (for example, lots of mouse input) it might be more convenient to set it to False:

```
>>> positive_y_is_up = False
>>> # Draw everything the pygame way, (0,0) in the top left corner
>>> positive_y_is_up = True
>>> # Draw everything the pymunk way, (0,0) in the bottom left corner
```

You can control the color of a shape by setting shape.color to the color you want it drawn in.

```
>>> c = pymunk.Circle(None, 10)
>>> c.color = pygame.color.THECOLORS["pink"]
```

See pygame_util.demo.py for a full example

Parameters

surface [pygame.Surface] Surface that the objects will be drawn on

collision_point_color

color_for_shape (*shape*)

constraint_color

draw_circle (*pos, angle, radius, outline_color, fill_color*)

draw_dot (*size, pos, color*)

draw_fat_segment (*a, b, radius, outline_color, fill_color*)

draw_polygon (*verts, radius, outline_color, fill_color*)

draw_segment (*a, b, color*)

flags

Bit flags which of shapes, joints and collisions should be drawn.

By default all 3 flags are set, meaning shapes, joints and collisions will be drawn.

Example using the basic text only DebugDraw implementation (normally you would use the desired backend instead, such as *pygame_util.DrawOptions* or *pyglet_util.DrawOptions*):

```
>>> import pymunk
>>> s = pymunk.Space()
>>> b = pymunk.Body()
>>> c = pymunk.Circle(b, 10)
>>> c.mass = 3
>>> s.add(b, c)
>>> s.add(pymunk.Circle(s.static_body, 3))
>>> s.step(0.01)
>>> options = pymunk.SpaceDebugDrawOptions()
```

```
>>> # Only draw the shapes, nothing else:
>>> options.flags = pymunk.SpaceDebugDrawOptions.DRAW_SHAPES
>>> s.debug_draw(options)
('draw_circle', (Vec2d(0.0, 0.0), 0.0, 10.0, SpaceDebugColor(r=44.0, g=62.0,
↳ b=80.0, a=255.0), SpaceDebugColor(r=52.0, g=152.0, b=219.0, a=255.0)))
('draw_circle', (Vec2d(0.0, 0.0), 0.0, 3.0, SpaceDebugColor(r=44.0, g=62.0,
↳ b=80.0, a=255.0), SpaceDebugColor(r=149.0, g=165.0, b=166.0, a=255.0)))
```

```
>>> # Draw the shapes and collision points:
>>> options.flags = pymunk.SpaceDebugDrawOptions.DRAW_SHAPES
>>> options.flags |= pymunk.SpaceDebugDrawOptions.DRAW_COLLISION_POINTS
>>> s.debug_draw(options)
('draw_circle', (Vec2d(0.0, 0.0), 0.0, 10.0, SpaceDebugColor(r=44.0, g=62.0,
↳ b=80.0, a=255.0), SpaceDebugColor(r=52.0, g=152.0, b=219.0, a=255.0)))
('draw_circle', (Vec2d(0.0, 0.0), 0.0, 3.0, SpaceDebugColor(r=44.0, g=62.0,
↳ b=80.0, a=255.0), SpaceDebugColor(r=149.0, g=165.0, b=166.0, a=255.0)))
('draw_segment', (Vec2d(1.0, 0.0), Vec2d(-8.0, 0.0), SpaceDebugColor(r=231.0,
↳ g=76.0, b=60.0, a=255.0)))
```

shape_dynamic_color = SpaceDebugColor(r=52, g=152, b=219, a=255)

shape_kinematic_color = SpaceDebugColor(r=39, g=174, b=96, a=255)

shape_outline_color

shape_sleeping_color = SpaceDebugColor(r=114, g=148, b=168, a=255)

shape_static_color = SpaceDebugColor(r=149, g=165, b=166, a=255)

pymunk.pygame_util.get_mouse_pos (*surface*)

Get position of the mouse pointer in pymunk coordinates.

pymunk.pygame_util.to_pygame (*p, surface*)

Convenience method to convert pymunk coordinates to pygame surface local coordinates.

Note that in case `positive_y_is_up` is `False`, this function won't actually do anything except converting the point to integers.

`pymunk.pygame_util.from_pygame(p, surface)`

Convenience method to convert pygame surface local coordinates to pymunk coordinates

`pymunk.pygame_util.positive_y_is_up = True`

Make increasing values of y point upwards.

When `True`:



When `False`:



9.4.1.6 `pymunk.pyglet_util` Module

This submodule contains helper functions to help with quick prototyping using pymunk together with pyglet.

Intended to help with debugging and prototyping, not for actual production use in a full application. The methods contained in this module are opinionated about your coordinate system and not very optimized (they use batched drawing, but there is probably room for optimizations still).

class `pymunk.pyglet_util.DrawOptions(**kwargs)`

Bases: `pymunk.space_debug_draw_options.SpaceDebugDrawOptions`

DRAW_COLLISION_POINTS

alias of `CP_SPACE_DEBUG_DRAW_COLLISION_POINTS`

DRAW_CONSTRAINTS

alias of `CP_SPACE_DEBUG_DRAW_CONSTRAINTS`

DRAW_SHAPES

alias of `CP_SPACE_DEBUG_DRAW_SHAPES`

__init__ (`**kwargs`)

Draw a `pymunk.Space`.

Typical usage:

```
>>> import pymunk
>>> import pymunk.pygame_util
>>> s = pymunk.Space()
```

(continues on next page)

(continued from previous page)

```
>>> options = pymunk.pyglet_util.DrawOptions()
>>> s.debug_draw(options)
```

You can control the color of a Shape by setting `shape.color` to the color you want it drawn in.

```
>>> c = pymunk.Circle(None, 10)
>>> c.color = (255, 0, 0, 255) # will draw my_shape in red
```

You can optionally pass in a batch to use for drawing. Just remember that you need to call draw yourself.

```
>>> my_batch = pyglet.graphics.Batch()
>>> s = pymunk.Space()
>>> options = pymunk.pyglet_util.DrawOptions(batch=my_batch)
>>> s.debug_draw(options)
>>> my_batch.draw()
```

See `pyglet_util.demo.py` for a full example

Param

kwargs [You can optionally pass in a `pyglet.graphics.Batch`] If a batch is given all drawing will use this batch to draw on. If no batch is given a new batch will be used for the drawing. Remember that if you pass in your own batch you need to call draw on it yourself.

collision_point_color

color_for_shape (*shape*)

constraint_color

draw_circle (*pos, angle, radius, outline_color, fill_color*)

draw_dot (*size, pos, color*)

draw_fat_segment (*a, b, radius, outline_color, fill_color*)

draw_polygon (*verts, radius, outline_color, fill_color*)

draw_segment (*a, b, color*)

flags

Bit flags which of shapes, joints and collisions should be drawn.

By default all 3 flags are set, meaning shapes, joints and collisions will be drawn.

Example using the basic text only DebugDraw implementation (normally you would use the desired backend instead, such as `pygame_util.DrawOptions` or `pyglet_util.DrawOptions`):

```
>>> import pymunk
>>> s = pymunk.Space()
>>> b = pymunk.Body()
>>> c = pymunk.Circle(b, 10)
>>> c.mass = 3
>>> s.add(b, c)
>>> s.add(pymunk.Circle(s.static_body, 3))
>>> s.step(0.01)
>>> options = pymunk.SpaceDebugDrawOptions()
```

```
>>> # Only draw the shapes, nothing else:
>>> options.flags = pymunk.SpaceDebugDrawOptions.DRAW_SHAPES
```

(continues on next page)

(continued from previous page)

```
>>> s.debug_draw(options)
('draw_circle', (Vec2d(0.0, 0.0), 0.0, 10.0, SpaceDebugColor(r=44.0, g=62.0,
↳b=80.0, a=255.0), SpaceDebugColor(r=52.0, g=152.0, b=219.0, a=255.0)))
('draw_circle', (Vec2d(0.0, 0.0), 0.0, 3.0, SpaceDebugColor(r=44.0, g=62.0,
↳b=80.0, a=255.0), SpaceDebugColor(r=149.0, g=165.0, b=166.0, a=255.0)))
```

```
>>> # Draw the shapes and collision points:
>>> options.flags = pymunk.SpaceDebugDrawOptions.DRAW_SHAPES
>>> options.flags |= pymunk.SpaceDebugDrawOptions.DRAW_COLLISION_POINTS
>>> s.debug_draw(options)
('draw_circle', (Vec2d(0.0, 0.0), 0.0, 10.0, SpaceDebugColor(r=44.0, g=62.0,
↳b=80.0, a=255.0), SpaceDebugColor(r=52.0, g=152.0, b=219.0, a=255.0)))
('draw_circle', (Vec2d(0.0, 0.0), 0.0, 3.0, SpaceDebugColor(r=44.0, g=62.0,
↳b=80.0, a=255.0), SpaceDebugColor(r=149.0, g=165.0, b=166.0, a=255.0)))
('draw_segment', (Vec2d(1.0, 0.0), Vec2d(-8.0, 0.0), SpaceDebugColor(r=231.0,
↳g=76.0, b=60.0, a=255.0)))
```

```
shape_dynamic_color = SpaceDebugColor(r=52, g=152, b=219, a=255)
shape_kinematic_color = SpaceDebugColor(r=39, g=174, b=96, a=255)
shape_outline_color
shape_sleeping_color = SpaceDebugColor(r=114, g=148, b=168, a=255)
shape_static_color = SpaceDebugColor(r=149, g=165, b=166, a=255)
```

9.4.1.7 pymunkoptions Module

Use this module to set runtime options of pymunk.

Currently there is one option that can be changed, debug. By setting debug to false debug print outs will be limited. In order to remove all debug prints you will also need to compile chipmunk in release mode. See [Compile Chipmunk](#) for details on how to compile chipmunk.

```
pymunkoptions.options = {'debug': True}
```

Global dict of pymunk options. To change make sure you import pymunk before any sub-packages and then set the option you want. For example:

```
import pymunkoptions
pymunkoptions.options["debug"] = False
import pymunk

#..continue to use pymunk as you normally do
```

pymunk

Pymunk is a easy-to-use pythonic 2d physics library that can be used whenever you need 2d rigid body physics from Python.

Homepage: <http://www.pymunk.org>

This is the main containing module of Pymunk. It contains among other things the very central Space, Body and Shape classes.

When you import this module it will automatically load the chipmunk library file. As long as you haven't turned off the debug mode a print will show exactly which Chipmunk library file it loaded. For example:

```
>>> import pymunk
```

```
Loading chipmunk for Windows (32bit) [C:\code\pymunk\chipmunk.dll]
```

`pymunk.inf = inf`

Infinity that can be passed as mass or inertia to a [Body](#).

Useful when you for example want a body that cannot rotate, just set its moment to inf. Just remember that if two objects with both infinite masses collides the world might explode. Similar effects can happen with infinite moment.

Note: In previous versions of Pymunk you used inf to create static bodies. This has changed. See [Body](#) for details.

`pymunk.version = '5.4.0'`

The release version of this pymunk installation. Valid only if pymunk was installed from a source or binary distribution (i.e. not in a checked-out copy from git).

`pymunk.chipmunk_version = '<Mock object>Rd7603e392782079b691d7948405af2dd66648a7a'`

The Chipmunk version compatible with this pymunk version. Other (newer) Chipmunk versions might also work if the new version does not contain any breaking API changes.

This property does not show a valid value in the compiled documentation, only when you actually import pymunk and do `pymunk.chipmunk_version`

The string is in the following format: `<cpVersionString>R<github commit of chipmunk>` where `cpVersionString` is a version string set by Chipmunk and the git commit hash corresponds to the git hash of the chipmunk source from `github.com/slembcke/Chipmunk2D` included with Pymunk. If the Chipmunk version is a release then the second part will be empty

Note: This is also the version of the Chipmunk source files included in the `chipmunk_src` folder (normally included in the Pymunk source distribution).

class `pymunk.Space` (*threaded=False*)

Bases: `pymunk._pickle.PickleMixin`, `object`

Spaces are the basic unit of simulation. You add rigid bodies, shapes and joints to it and then step them all forward together through time.

A Space can be copied and pickled. Note that any post step callbacks are not copied. Also note that some internal collision cache data is not copied, which can make the simulation a bit unstable the first few steps of the fresh copy.

Custom properties set on the space will also be copied/pickled.

Any collision handlers will also be copied/pickled. Note that depending on the pickle protocol used there are some restrictions on what functions can be copied/pickled.

Example:

```
>>> import pymunk, pickle
>>> space = pymunk.Space()
>>> space2 = space.copy()
>>> space3 = pickle.loads(pickle.dumps(space))
```

`__init__` (*threaded=False*)

Create a new instance of the Space.

If you set `threaded=True` the step function will run in threaded mode which might give a speedup. Note that even when you set `threaded=True` you still have to set `Space.threads=2` to actually use more than one thread.

Also note that threaded mode is not available on Windows, and setting `threaded=True` has no effect on that platform.

add (*objs)

Add one or many shapes, bodies or joints to the space

Unlike Chipmunk and earlier versions of pymunk its now allowed to add objects even from a callback during the simulation step. However, the add will not be performed until the end of the step.

add_collision_handler (collision_type_a, collision_type_b)

Return the `CollisionHandler` for collisions between objects of type `collision_type_a` and `collision_type_b`.

Fill the desired collision callback functions, for details see the `CollisionHandler` object.

Whenever shapes with collision types (`Shape.collision_type`) a and b collide, this handler will be used to process the collision events. When a new collision handler is created, the callbacks will all be set to builtin callbacks that perform the default behavior (call the wildcard handlers, and accept all collisions).

Parameters

- **collision_type_a** (*int*) – Collision type a
- **collision_type_b** (*int*) – Collision type b

Return type `CollisionHandler`

add_default_collision_handler ()

Return a reference to the default collision handler or that is used to process all collisions that don't have a more specific handler.

The default behavior for each of the callbacks is to call the wildcard handlers, ANDing their return values together if applicable.

add_post_step_callback (callback_function, key, *args, **kwargs)

Add a function to be called last in the next simulation step.

Post step callbacks are registered as a function and an object used as a key. You can only register one post step callback per object.

This function was more useful with earlier versions of pymunk where you weren't allowed to use the add and remove methods on the space during a simulation step. But this function is still available for other uses and to keep backwards compatibility.

Note: If you remove a shape from the callback it will trigger the collision handler for the 'separate' event if it the shape was touching when removed.

Note: Post step callbacks are not included in pickle / copy of the space.

Parameters

- **callback_function** (*func(space : Space, key, *args, **kwargs)*) – The callback function

- **key** (*Any*) – This object is used as a key, you can only have one callback for a single object. It is passed on to the callback function.
- **args** – Optional parameters passed to the callback
- **kwargs** – Optional keyword parameters passed on to the callback

Returns True if key was not previously added, False otherwise

add_wildcard_collision_handler (*collision_type_a*)

Add a wildcard collision handler for given collision type.

This handler will be used any time an object with this type collides with another object, regardless of its type. A good example is a projectile that should be destroyed the first time it hits anything. There may be a specific collision handler and two wildcard handlers. It's up to the specific handler to decide if and when to call the wildcard handlers and what to do with their return values.

When a new wildcard handler is created, the callbacks will all be set to builtin callbacks that perform the default behavior. (accept all collisions in *begin()* and *pre_solve()*, or do nothing for *post_solve()* and *separate()*).

Parameters *collision_type_a* (*int*) – Collision type

Return type *CollisionHandler*

bb_query (*bb*, *shape_filter*)

Query space to find all shapes near bb.

The filter is applied to the query and follows the same rules as the collision detection.

Note: Sensor shapes are included in the result

Parameters

- **bb** (*BB*) – Bounding box
- **shape_filter** (*ShapeFilter*) – Shape filter

Return type [*Shape*]

bodies

A list of the bodies added to this space

collision_bias

Determines how fast overlapping shapes are pushed apart.

Pymunk allows fast moving objects to overlap, then fixes the overlap over time. Overlapping objects are unavoidable even if swept collisions are supported, and this is an efficient and stable way to deal with overlapping objects. The bias value controls what percentage of overlap remains unfixed after a second and defaults to ~0.2%. Valid values are in the range from 0 to 1, but using 0 is not recommended for stability reasons. The default value is calculated as $\text{cpfpow}(1.0f - 0.1f, 60.0f)$ meaning that pymunk attempts to correct 10% of error ever 1/60th of a second.

..Note:: Very very few games will need to change this value.

collision_persistence

The number of frames the space keeps collision solutions around for.

Helps prevent jittering contacts from getting worse. This defaults to 3.

..Note:: Very very few games will need to change this value.

collision_slop

Amount of overlap between shapes that is allowed.

To improve stability, set this as high as you can without noticeable overlapping. It defaults to 0.1.

constraints

A list of the constraints added to this space

copy()

Create a deep copy of this space.

current_time_step

Retrieves the current (if you are in a callback from `Space.step()`) or most recent (outside of a `Space.step()` call) timestep.

damping

Amount of simple damping to apply to the space.

A value of 0.9 means that each body will lose 10% of its velocity per second. Defaults to 1. Like gravity, it can be overridden on a per body basis.

debug_draw (*options*)

Debug draw the current state of the space using the supplied drawing options.

If you use a graphics backend that is already supported, such as `pygame` and `pyglet`, you can use the predefined options in their `x_util` modules, for example `pygame_util.DrawOptions`.

Its also possible to write your own graphics backend, see `SpaceDebugDrawOptions`.

If you require any advanced or optimized drawing its probably best to not use this funtion for the drawing since its meant for debugging and quick scripting.

gravity

Global gravity applied to the space.

Defaults to (0,0). Can be overridden on a per body basis by writing custom integration functions.

idle_speed_threshold

Speed threshold for a body to be considered idle.

The default value of 0 means the space estimates a good threshold based on gravity.

iterations

Iterations allow you to control the accuracy of the solver.

Defaults to 10.

Pymunk uses an iterative solver to figure out the forces between objects in the space. What this means is that it builds a big list of all of the collisions, joints, and other constraints between the bodies and makes several passes over the list considering each one individually. The number of passes it makes is the iteration count, and each iteration makes the solution more accurate. If you use too many iterations, the physics should look nice and solid, but may use up too much CPU time. If you use too few iterations, the simulation may seem mushy or bouncy when the objects should be solid. Setting the number of iterations lets you balance between CPU usage and the accuracy of the physics. Pymunk's default of 10 iterations is sufficient for most simple games.

point_query (*point, max_distance, shape_filter*)

Query space at point for shapes within the given distance range.

The filter is applied to the query and follows the same rules as the collision detection. If a `maxDistance` of 0.0 is used, the point must lie inside a shape. Negative `max_distance` is also allowed meaning that the point must be a under a certain depth within a shape to be considered a match.

See [ShapeFilter](#) for details about how the `shape_filter` parameter can be used.

Note: Sensor shapes are included in the result (In `Space.point_query_nearest()` they are not)

Parameters

- **point** (*Vec2d* or (float,float)) – Where to check for collision in the Space
- **max_distance** (*float*) – Match only within this distance
- **shape_filter** (*ShapeFilter*) – Only pick shapes matching the filter

Return type [*PointQueryInfo*]

point_query_nearest (*point, max_distance, shape_filter*)

Query space at point the nearest shape within the given distance range.

The filter is applied to the query and follows the same rules as the collision detection. If a `maxDistance` of 0.0 is used, the point must lie inside a shape. Negative `max_distance` is also allowed meaning that the point must be under a certain depth within a shape to be considered a match.

See [ShapeFilter](#) for details about how the `shape_filter` parameter can be used.

Note: Sensor shapes are not included in the result (In `Space.point_query()` they are)

Parameters

- **point** (*Vec2d* or (float,float)) – Where to check for collision in the Space
- **max_distance** (*float*) – Match only within this distance
- **shape_filter** (*ShapeFilter*) – Only pick shapes matching the filter

Return type *PointQueryInfo* or None

reindex_shape (*shape*)

Update the collision detection data for a specific shape in the space.

reindex_shapes_for_body (*body*)

Reindex all the shapes for a certain body.

reindex_static ()

Update the collision detection info for the static shapes in the space. You only need to call this if you move one of the static shapes.

remove (**objs*)

Remove one or many shapes, bodies or constraints from the space

Unlike Chipmunk and earlier versions of Pymunk its now allowed to remove objects even from a callback during the simulation step. However, the removal will not be performed until the end of the step.

Note: When removing objects from the space, make sure you remove any other objects that reference it. For instance, when you remove a body, remove the joints and shapes attached to it.

segment_query (*start, end, radius, shape_filter*)

Query space along the line segment from start to end with the given radius.

The filter is applied to the query and follows the same rules as the collision detection.

See [ShapeFilter](#) for details about how the `shape_filter` parameter can be used.

Note: Sensor shapes are included in the result (In `Space.segment_query_first()` they are not)

Parameters

- **start** – Starting point
- **end** – End point
- **radius** (*float*) – Radius
- **shape_filter** ([ShapeFilter](#)) – Shape filter

Return type [[SegmentQueryInfo](#)]

segment_query_first (*start, end, radius, shape_filter*)

Query space along the line segment from start to end with the given radius.

The filter is applied to the query and follows the same rules as the collision detection.

Note: Sensor shapes are not included in the result (In `Space.segment_query()` they are)

See [ShapeFilter](#) for details about how the `shape_filter` parameter can be used.

Return type [SegmentQueryInfo](#) or None

shape_query (*shape*)

Query a space for any shapes overlapping the given shape

Note: Sensor shapes are included in the result

Parameters **shape** (*Circle, Poly* or *Segment*) – Shape to query with

Return type [[ShapeQueryInfo](#)]

shapes

A list of all the shapes added to this space

(includes both static and non-static)

sleep_time_threshold

Time a group of bodies must remain idle in order to fall asleep.

The default value of *inf* disables the sleeping algorithm.

static_body

A dedicated static body for the space.

You don't have to use it, but because its memory is managed automatically with the space its very convenient.

step (*dt*)

Update the space for the given time step.

Using a fixed time step is highly recommended. Doing so will increase the efficiency of the contact persistence, requiring an order of magnitude fewer iterations to resolve the collisions in the usual case.

It is not the same to call step 10 times with a dt of 0.1 and calling it 100 times with a dt of 0.01 even if the end result is that the simulation moved forward 100 units. Performing multiple calls with a smaller dt creates a more stable and accurate simulation. Therefor it sometimes make sense to have a little for loop around the step call, like in this example:

```
>>> import pymunk
>>> s = pymunk.Space()
>>> steps = 10
>>> for x in range(steps): # move simulation forward 0.1 seconds:
...     s.step(0.1 / steps)
```

Parameters **dt** (*float*) – Time step length

threads

The number of threads to use for running the step function.

Only valid when the Space was created with threaded=True. Currently the max limit is 2, setting a higher value wont have any effect. The default is 1 regardless if the Space was created with threaded=True, to keep determinism in the simulation.

class `pymunk.Body` (*mass=0, moment=0, body_type=<class 'CP_BODY_TYPE_DYNAMIC'>*)

Bases: `pymunk._pickle.PickleMixin`, `object`

A rigid body

- Use forces to modify the rigid bodies if possible. This is likely to be the most stable.
- Modifying a body's velocity shouldn't necessarily be avoided, but applying large changes can cause strange results in the simulation. Experiment freely, but be warned.
- Don't modify a body's position every step unless you really know what you are doing. Otherwise you're likely to get the position/velocity badly out of sync.

A Body can be copied and pickled. Sleeping bodies that are copied will be awake in the fresh copy. When a Body is copied any spaces, shapes or constraints attached to the body will not be copied.

DYNAMIC

alias of `CP_BODY_TYPE_DYNAMIC`

KINEMATIC

alias of `CP_BODY_TYPE_KINEMATIC`

STATIC

alias of `CP_BODY_TYPE_STATIC`

__init__ (*mass=0, moment=0, body_type=<class 'CP_BODY_TYPE_DYNAMIC'>*)

Create a new Body

Mass and moment are ignored when body_type is KINEMATIC or STATIC.

Guessing the mass for a body is usually fine, but guessing a moment of inertia can lead to a very poor simulation so it's recommended to use Chipmunk's moment calculations to estimate the moment for you.

There are two ways to set up a dynamic body. The easiest option is to create a body with a mass and moment of 0, and set the mass or density of each collision shape added to the body. Chipmunk will automatically calculate the mass, moment of inertia, and center of gravity for you. This is probably preferred in most cases. Note that these will only be correctly calculated after the body and shape are added to a space.

The other option is to set the mass of the body when it's created, and leave the mass of the shapes added to it as 0.0. This approach is more flexible, but is not as easy to use. Don't set the mass of both the body and the shapes. If you do so, it will recalculate and overwrite your custom mass value when the shapes are added to the body.

Examples of the different ways to set up the mass and moment:

```
>>> import pymunk
>>> radius = 2
>>> mass = 3
>>> density = 3
>>> def print_mass_moment(b):
...     print("mass={:.0f} moment={:.0f}".format(b.mass, b.moment))
```

```
>>> # Using Shape.density
>>> s = pymunk.Space()
>>> b = pymunk.Body()
>>> c = pymunk.Circle(b, radius)
>>> c.density = density
>>> print_mass_moment(b)
mass=0 moment=0
>>> s.add(b, c)
>>> print_mass_moment(b)
mass=38 moment=75
```

```
>>> # Using Shape.mass
>>> b = pymunk.Body()
>>> c = pymunk.Circle(b, radius)
>>> c.mass = mass
>>> print_mass_moment(b)
mass=0 moment=0
>>> s.add(b, c)
>>> print_mass_moment(b)
mass=3 moment=6
```

```
>>> # Using Body constructor
>>> moment = pymunk.moment_for_circle(mass, 0, radius)
>>> b = pymunk.Body()
>>> c = pymunk.Circle(b, radius)
>>> c.mass = mass
>>> print_mass_moment(b)
mass=0 moment=0
>>> s.add(b, c)
>>> print_mass_moment(b)
mass=3 moment=6
```

It becomes even more useful to use the mass or density properties of the shape when you attach multiple shapes to one body, like in this example with density:

```
>>> # Using multiple Shape.density
>>> b = pymunk.Body()
>>> c1 = pymunk.Circle(b, radius, offset=(10,0))
>>> c1.density = density
>>> c2 = pymunk.Circle(b, radius, offset=(0,10))
>>> c2.density = density
>>> s.add(b, c1, c2)
>>> print_mass_moment(b)
```

(continues on next page)

(continued from previous page)

```
mass=75 moment=3921
```

activate()

Reset the idle timer on a body.

If it was sleeping, wake it and any other bodies it was touching.

angle

Rotation of the body in radians.

When changing the rotation you may also want to call `Space.reindex_shapes_for_body()` to update the collision detection information for the attached shapes if plan to make any queries against the space. A body rotates around its center of gravity, not its position.

Note: If you get small/no changes to the angle when for example a ball is “rolling” down a slope it might be because the Circle shape attached to the body or the slope shape does not have any friction set.

angular_velocity

The angular velocity of the body in radians per second.

apply_force_at_local_point (*force, point*)

Add the local force force to body as if applied from the body local point.

apply_force_at_world_point (*force, point*)

Add the force force to body as if applied from the world point.

People are sometimes confused by the difference between a force and an impulse. An impulse is a very large force applied over a very short period of time. Some examples are a ball hitting a wall or cannon firing. Chipmunk treats impulses as if they occur instantaneously by adding directly to the velocity of an object. Both impulses and forces are affected the mass of an object. Doubling the mass of the object will halve the effect.

apply_impulse_at_local_point (*impulse, point=(0, 0)*)

Add the local impulse impulse to body as if applied from the body local point.

apply_impulse_at_world_point (*impulse, point=(0, 0)*)

Add the impulse impulse to body as if applied from the world point.

body_type

The type of a body (`Body.DYNAMIC`, `Body.KINEMATIC` or `Body.STATIC`).

When changing an body to a dynamic body, the mass and moment of inertia are recalculated from the shapes added to the body. Custom calculated moments of inertia are not preserved when changing types. This function cannot be called directly in a collision callback.

center_of_gravity

Location of the center of gravity in body local coordinates.

The default value is (0, 0), meaning the center of gravity is the same as the position of the body.

constraints

Get the constraints this body is attached to.

The body only keeps a weak reference to the constraints and a live body wont prevent GC of the attached constraints

copy()

Create a deep copy of this body.

each_arbiter (*func*, *args, **kwargs)

Run func on each of the arbiters on this body.

```
func(arbiter, *args, **kwargs) -> None
```

Callback Parameters

arbiter [*Arbiter*] The Arbiter

args Optional parameters passed to the callback function.

kwargs Optional keyword parameters passed on to the callback function.

Warning: Do not hold on to the Arbiter after the callback!

force

Force applied to the center of gravity of the body.

This value is reset for every time step.

is_sleeping

Returns true if the body is sleeping.

kinetic_energy

Get the kinetic energy of a body.

local_to_world (*v*)

Convert body local coordinates to world space coordinates

Many things are defined in coordinates local to a body meaning that the (0,0) is at the center of gravity of the body and the axis rotate along with the body.

Parameters **v** – Vector in body local coordinates

mass

Mass of the body.

moment

Moment of inertia (MoI or sometimes just moment) of the body.

The moment is like the rotational mass of a body.

position

Position of the body.

When changing the position you may also want to call `Space.reindex_shapes_for_body()` to update the collision detection information for the attached shapes if plan to make any queries against the space.

position_func

The position callback function.

The position callback function is called each time step and can be used to update the body's position.

```
func(body, dt) -> None
```

rotation_vector

The rotation vector for the body.

shapes

Get the shapes attached to this body.

The body only keeps a weak reference to the shapes and a live body wont prevent GC of the attached shapes

sleep()

Forces a body to fall asleep immediately even if it's in midair.

Cannot be called from a callback.

sleep_with_group(*body*)

Force a body to fall asleep immediately along with other bodies in a group.

When objects in Pymunk sleep, they sleep as a group of all objects that are touching or jointed together. When an object is woken up, all of the objects in its group are woken up. `Body.sleep_with_group()` allows you group sleeping objects together. It acts identically to `Body.sleep()` if you pass None as group by starting a new group. If you pass a sleeping body for group, body will be awoken when group is awoken. You can use this to initialize levels and start stacks of objects in a pre-sleeping state.

space

Get the *Space* that the body has been added to (or None).

torque

The torque applied to the body.

This value is reset for every time step.

static update_position(*body*, *dt*)

Default rigid body position integration function.

Updates the position of the body using Euler integration. Unlike the velocity function, it's unlikely you'll want to override this function. If you do, make sure you understand it's source code (in Chipmunk) as it's an important part of the collision/joint correction process.

static update_velocity(*body*, *gravity*, *damping*, *dt*)

Default rigid body velocity integration function.

Updates the velocity of the body using Euler integration.

velocity

Linear velocity of the center of gravity of the body.

velocity_at_local_point(*point*)

Get the absolute velocity of the rigid body at the given body local point

velocity_at_world_point(*point*)

Get the absolute velocity of the rigid body at the given world point

It's often useful to know the absolute velocity of a point on the surface of a body since the angular velocity affects everything except the center of gravity.

velocity_func

The velocity callback function.

The velocity callback function is called each time step, and can be used to set a body's velocity.

```
func(body : Body, gravity, damping, dt)
```

world_to_local(*v*)

Convert world space coordinates to body local coordinates

Parameters *v* – Vector in world space coordinates

class pymunk.Shape(*shape=None*)

Bases: `pymunk._pickle.PickleMixin`, `object`

Base class for all the shapes.

You usually don't want to create instances of this class directly but use one of the specialized shapes instead (*Circle*, *Poly* or *Segment*).

All the shapes can be copied and pickled. If you copy/pickle a shape the body (if any) will also be copied.

`__init__` (*shape=None*)
x.`__init__`(...) initializes x; see `help(type(x))` for signature

`area`
The calculated area of this shape.

`bb`
The bounding box *BB* of the shape.

Only guaranteed to be valid after *Shape.cache_bb()* or *Space.step()* is called. Moving a body that a shape is connected to does not update its bounding box. For shapes used for queries that aren't attached to bodies, you can also use *Shape.update()*.

`body`
The body this shape is attached to. Can be set to None to indicate that this shape doesn't belong to a body.

`cache_bb()`
Update and returns the bounding box of this shape

`center_of_gravity`
The calculated center of gravity of this shape.

`collision_type`
User defined collision type for the shape.

See *Space.add_collision_handler()* function for more information on when to use this property.

`copy()`
Create a deep copy of this shape.

`density`
The density of this shape.

This is useful when you let Pymunk calculate the total mass and inertia of a body from the shapes attached to it. (Instead of setting the body mass and inertia directly)

`elasticity`
Elasticity of the shape.

A value of 0.0 gives no bounce, while a value of 1.0 will give a 'perfect' bounce. However due to inaccuracies in the simulation using 1.0 or greater is not recommended.

`filter`
Set the collision *ShapeFilter* for this shape.

`friction`
Friction coefficient.

Pymunk uses the Coulomb friction model, a value of 0.0 is frictionless.

A value over 1.0 is perfectly fine.

Some real world example values from Wikipedia (Remember that it is what looks good that is important, not the exact value).

Material	Other	Friction
Aluminium	Steel	0.61
Copper	Steel	0.53
Brass	Steel	0.51
Cast iron	Copper	1.05
Cast iron	Zinc	0.85
Concrete (wet)	Rubber	0.30
Concrete (dry)	Rubber	1.0
Concrete	Wood	0.62
Copper	Glass	0.68
Glass	Glass	0.94
Metal	Wood	0.5
Polyethene	Steel	0.2
Steel	Steel	0.80
Steel	Teflon	0.04
Teflon (PTFE)	Teflon	0.04
Wood	Wood	0.4

mass

The mass of this shape.

This is useful when you let Pymunk calculate the total mass and inertia of a body from the shapes attached to it. (Instead of setting the body mass and inertia directly)

moment

The calculated moment of this shape.

point_query (*p*)

Check if the given point lies within the shape.

A negative distance means the point is within the shape.

Returns Tuple of (distance, info)

Return type (float, *PointQueryInfo*)

segment_query (*start*, *end*, *radius=0*)

Check if the line segment from start to end intersects the shape.

Return type *SegmentQueryInfo*

sensor

A boolean value if this shape is a sensor or not.

Sensors only call collision callbacks, and never generate real collisions.

shapes_collide (*b*)

Get contact information about this shape and shape b.

Return type *ContactPointSet*

space

Get the *Space* that shape has been added to (or None).

surface_velocity

The surface velocity of the object.

Useful for creating conveyor belts or players that move around. This value is only used when calculating friction, not resolving the collision.

update (*transform*)

Update, cache and return the bounding box of a shape with an explicit transformation.

Useful if you have a shape without a body and want to use it for querying.

class pymunk.Circle (*body, radius, offset=(0, 0)*)

Bases: pymunk.shapes.Shape

A circle shape defined by a radius

This is the fastest and simplest collision shape

__init__ (*body, radius, offset=(0, 0)*)

body is the body attach the circle to, offset is the offset from the body's center of gravity in body local coordinates.

It is legal to send in None as body argument to indicate that this shape is not attached to a body. However, you must attach it to a body before adding the shape to a space or used for a space shape query.

area

The calculated area of this shape.

bb

The bounding box *BB* of the shape.

Only guaranteed to be valid after *Shape.cache_bb()* or *Space.step()* is called. Moving a body that a shape is connected to does not update it's bounding box. For shapes used for queries that aren't attached to bodies, you can also use *Shape.update()*.

body

The body this shape is attached to. Can be set to None to indicate that this shape doesnt belong to a body.

cache_bb ()

Update and returns the bounding box of this shape

center_of_gravity

The calculated center of gravity of this shape.

collision_type

User defined collision type for the shape.

See *Space.add_collision_handler()* function for more information on when to use this property.

copy ()

Create a deep copy of this shape.

density

The density of this shape.

This is useful when you let Pymunk calculate the total mass and inertia of a body from the shapes attached to it. (Instead of setting the body mass and inertia directly)

elasticity

Elasticity of the shape.

A value of 0.0 gives no bounce, while a value of 1.0 will give a 'perfect' bounce. However due to inaccuracies in the simulation using 1.0 or greater is not recommended.

filter

Set the collision *ShapeFilter* for this shape.

friction

Friction coefficient.

Pymunk uses the Coulomb friction model, a value of 0.0 is frictionless.

A value over 1.0 is perfectly fine.

Some real world example values from Wikipedia (Remember that it is what looks good that is important, not the exact value).

Material	Other	Friction
Aluminium	Steel	0.61
Copper	Steel	0.53
Brass	Steel	0.51
Cast iron	Copper	1.05
Cast iron	Zinc	0.85
Concrete (wet)	Rubber	0.30
Concrete (dry)	Rubber	1.0
Concrete	Wood	0.62
Copper	Glass	0.68
Glass	Glass	0.94
Metal	Wood	0.5
Polyethene	Steel	0.2
Steel	Steel	0.80
Steel	Teflon	0.04
Teflon (PTFE)	Teflon	0.04
Wood	Wood	0.4

mass

The mass of this shape.

This is useful when you let Pymunk calculate the total mass and inertia of a body from the shapes attached to it. (Instead of setting the body mass and inertia directly)

moment

The calculated moment of this shape.

offset

Offset. (body space coordinates)

point_query (*p*)

Check if the given point lies within the shape.

A negative distance means the point is within the shape.

Returns Tuple of (distance, info)

Return type (float, *PointQueryInfo*)

radius

The Radius of the circle

segment_query (*start*, *end*, *radius=0*)

Check if the line segment from start to end intersects the shape.

Return type *SegmentQueryInfo*

sensor

A boolean value if this shape is a sensor or not.

Sensors only call collision callbacks, and never generate real collisions.

shapes_collide (*b*)

Get contact information about this shape and shape *b*.

Return type *ContactPointSet*

space

Get the *Space* that shape has been added to (or *None*).

surface_velocity

The surface velocity of the object.

Useful for creating conveyor belts or players that move around. This value is only used when calculating friction, not resolving the collision.

unsafe_set_offset (*o*)

Unsafe set the offset of the circle.

Note: This change is only picked up as a change to the position of the shape's surface, but not it's velocity. Changing it will not result in realistic physical behavior. Only use if you know what you are doing!

unsafe_set_radius (*r*)

Unsafe set the radius of the circle.

Note: This change is only picked up as a change to the position of the shape's surface, but not it's velocity. Changing it will not result in realistic physical behavior. Only use if you know what you are doing!

update (*transform*)

Update, cache and return the bounding box of a shape with an explicit transformation.

Useful if you have a shape without a body and want to use it for querying.

class pymunk.Poly (*body, vertices, transform=None, radius=0*)

Bases: pymunk.shapes.Shape

A convex polygon shape

Slowest, but most flexible collision shape.

__init__ (*body, vertices, transform=None, radius=0*)

Create a polygon.

A convex hull will be calculated from the vertexes automatically.

Adding a small radius will bevel the corners and can significantly reduce problems where the poly gets stuck on seams in your geometry.

It is legal to send in *None* as body argument to indicate that this shape is not attached to a body. However, you must attach it to a body before adding the shape to a space or used for a space shape query.

Parameters

- **body** (*Body*) – The body to attach the poly to
- **vertices** (*[(float, float)]*) – Define a convex hull of the polygon with a counterclockwise winding.
- **transform** (*Transform*) – Transform will be applied to every vertex.
- **radius** (*float*) – Set the radius of the poly shape

area

The calculated area of this shape.

bb

The bounding box *BB* of the shape.

Only guaranteed to be valid after *Shape.cache_bb()* or *Space.step()* is called. Moving a body that a shape is connected to does not update it's bounding box. For shapes used for queries that aren't attached to bodies, you can also use *Shape.update()*.

body

The body this shape is attached to. Can be set to None to indicate that this shape doesn't belong to a body.

cache_bb()

Update and returns the bounding box of this shape

center_of_gravity

The calculated center of gravity of this shape.

collision_type

User defined collision type for the shape.

See *Space.add_collision_handler()* function for more information on when to use this property.

copy()

Create a deep copy of this shape.

static create_box (*body*, *size*=(10, 10), *radius*=0)

Convenience function to create a box given a width and height.

The boxes will always be centered at the center of gravity of the body you are attaching them to. If you want to create an off-center box, you will need to use the normal constructor *Poly(...)*.

Adding a small radius will bevel the corners and can significantly reduce problems where the box gets stuck on seams in your geometry.

Parameters

- **body** (*Body*) – The body to attach the poly to
- **size** ((*float*, *float*)) – Size of the box as (width, height)
- **radius** (*float*) – Radius of poly

Return type *Poly***static create_box_bb** (*body*, *bb*, *radius*=0)

Convenience function to create a box shape from a *BB*.

The boxes will always be centered at the center of gravity of the body you are attaching them to. If you want to create an off-center box, you will need to use the normal constructor *Poly(..)*.

Adding a small radius will bevel the corners and can significantly reduce problems where the box gets stuck on seams in your geometry.

Parameters

- **body** (*Body*) – The body to attach the poly to
- **bb** (*BB*) – Size of the box
- **radius** (*float*) – Radius of poly

Return type *Poly*

density

The density of this shape.

This is useful when you let Pymunk calculate the total mass and inertia of a body from the shapes attached to it. (Instead of setting the body mass and inertia directly)

elasticity

Elasticity of the shape.

A value of 0.0 gives no bounce, while a value of 1.0 will give a ‘perfect’ bounce. However due to inaccuracies in the simulation using 1.0 or greater is not recommended.

filter

Set the collision *ShapeFilter* for this shape.

friction

Friction coefficient.

Pymunk uses the Coulomb friction model, a value of 0.0 is frictionless.

A value over 1.0 is perfectly fine.

Some real world example values from Wikipedia (Remember that it is what looks good that is important, not the exact value).

Material	Other	Friction
Aluminium	Steel	0.61
Copper	Steel	0.53
Brass	Steel	0.51
Cast iron	Copper	1.05
Cast iron	Zinc	0.85
Concrete (wet)	Rubber	0.30
Concrete (dry)	Rubber	1.0
Concrete	Wood	0.62
Copper	Glass	0.68
Glass	Glass	0.94
Metal	Wood	0.5
Polyethene	Steel	0.2
Steel	Steel	0.80
Steel	Teflon	0.04
Teflon (PTFE)	Teflon	0.04
Wood	Wood	0.4

get_vertices()

Get the vertices in local coordinates for the polygon

If you need the vertices in world coordinates then the vertices can be transformed by adding the body position and each vertex rotated by the body rotation in the following way:

```
>>> import pymunk
>>> b = pymunk.Body()
>>> b.position = 1,2
>>> b.angle = 0.5
>>> shape = pymunk.Poly(b, [(0,0), (10,0), (10,10)])
>>> for v in shape.get_vertices():
...     x,y = v.rotated(shape.body.angle) + shape.body.position
...     (int(x), int(y))
```

(continues on next page)

(continued from previous page)

```
(1, 2)
(9, 6)
(4, 15)
```

Returns The vertices in local coords

Return type [Vec2d]

mass

The mass of this shape.

This is useful when you let Pymunk calculate the total mass and inertia of a body from the shapes attached to it. (Instead of setting the body mass and inertia directly)

moment

The calculated moment of this shape.

point_query (*p*)

Check if the given point lies within the shape.

A negative distance means the point is within the shape.

Returns Tuple of (distance, info)

Return type (float, *PointQueryInfo*)

radius

The radius of the poly shape. Extends the poly in all directions with the given radius

segment_query (*start*, *end*, *radius=0*)

Check if the line segment from start to end intersects the shape.

Return type *SegmentQueryInfo*

sensor

A boolean value if this shape is a sensor or not.

Sensors only call collision callbacks, and never generate real collisions.

shapes_collide (*b*)

Get contact information about this shape and shape b.

Return type *ContactPointSet*

space

Get the *Space* that shape has been added to (or None).

surface_velocity

The surface velocity of the object.

Useful for creating conveyor belts or players that move around. This value is only used when calculating friction, not resolving the collision.

unsafe_set_radius (*radius*)

Unsafe set the radius of the poly.

Note: This change is only picked up as a change to the position of the shape's surface, but not it's velocity. Changing it will not result in realistic physical behavior. Only use if you know what you are doing!

unsafe_set_vertices (*vertices*, *transform=None*)

Unsafe set the vertices of the poly.

Note: This change is only picked up as a change to the position of the shape's surface, but not it's velocity. Changing it will not result in realistic physical behavior. Only use if you know what you are doing!

update (*transform*)

Update, cache and return the bounding box of a shape with an explicit transformation.

Useful if you have a shape without a body and want to use it for querying.

class pymunk.**Segment** (*body*, *a*, *b*, *radius*)

Bases: pymunk.shapes.Shape

A line segment shape between two points

Meant mainly as a static shape. Can be beveled in order to give them a thickness.

__init__ (*body*, *a*, *b*, *radius*)

Create a Segment

It is legal to send in None as body argument to indicate that this shape is not attached to a body. However, you must attach it to a body before adding the shape to a space or used for a space shape query.

Parameters

- **body** (*Body*) – The body to attach the segment to
- **a** – The first endpoint of the segment
- **b** – The second endpoint of the segment
- **radius** (*float*) – The thickness of the segment

a

The first of the two endpoints for this segment

area

The calculated area of this shape.

b

The second of the two endpoints for this segment

bb

The bounding box *BB* of the shape.

Only guaranteed to be valid after *Shape.cache_bb()* or *Space.step()* is called. Moving a body that a shape is connected to does not update it's bounding box. For shapes used for queries that aren't attached to bodies, you can also use *Shape.update()*.

body

The body this shape is attached to. Can be set to None to indicate that this shape doesnt belong to a body.

cache_bb ()

Update and returns the bounding box of this shape

center_of_gravity

The calculated center of gravity of this shape.

collision_type

User defined collision type for the shape.

See `Space.add_collision_handler()` function for more information on when to use this property.

copy()

Create a deep copy of this shape.

density

The density of this shape.

This is useful when you let Pymunk calculate the total mass and inertia of a body from the shapes attached to it. (Instead of setting the body mass and inertia directly)

elasticity

Elasticity of the shape.

A value of 0.0 gives no bounce, while a value of 1.0 will give a ‘perfect’ bounce. However due to inaccuracies in the simulation using 1.0 or greater is not recommended.

filter

Set the collision *ShapeFilter* for this shape.

friction

Friction coefficient.

Pymunk uses the Coulomb friction model, a value of 0.0 is frictionless.

A value over 1.0 is perfectly fine.

Some real world example values from Wikipedia (Remember that it is what looks good that is important, not the exact value).

Material	Other	Friction
Aluminium	Steel	0.61
Copper	Steel	0.53
Brass	Steel	0.51
Cast iron	Copper	1.05
Cast iron	Zinc	0.85
Concrete (wet)	Rubber	0.30
Concrete (dry)	Rubber	1.0
Concrete	Wood	0.62
Copper	Glass	0.68
Glass	Glass	0.94
Metal	Wood	0.5
Polyethene	Steel	0.2
Steel	Steel	0.80
Steel	Teflon	0.04
Teflon (PTFE)	Teflon	0.04
Wood	Wood	0.4

mass

The mass of this shape.

This is useful when you let Pymunk calculate the total mass and inertia of a body from the shapes attached to it. (Instead of setting the body mass and inertia directly)

moment

The calculated moment of this shape.

normal

The normal

point_query (*p*)

Check if the given point lies within the shape.

A negative distance means the point is within the shape.

Returns Tuple of (distance, info)

Return type (float, *PointQueryInfo*)

radius

The radius/thickness of the segment

segment_query (*start*, *end*, *radius=0*)

Check if the line segment from start to end intersects the shape.

Return type *SegmentQueryInfo*

sensor

A boolean value if this shape is a sensor or not.

Sensors only call collision callbacks, and never generate real collisions.

set_neighbors (*prev*, *next*)

When you have a number of segment shapes that are all joined together, things can still collide with the “cracks” between the segments. By setting the neighbor segment endpoints you can tell Chipmunk to avoid colliding with the inner parts of the crack.

shapes_collide (*b*)

Get contact information about this shape and shape b.

Return type *ContactPointSet*

space

Get the *Space* that shape has been added to (or None).

surface_velocity

The surface velocity of the object.

Useful for creating conveyor belts or players that move around. This value is only used when calculating friction, not resolving the collision.

unsafe_set_endpoints (*a*, *b*)

Set the two endpoints for this segment

Note: This change is only picked up as a change to the position of the shape’s surface, but not it’s velocity. Changing it will not result in realistic physical behavior. Only use if you know what you are doing!

unsafe_set_radius (*r*)

Set the radius of the segment

Note: This change is only picked up as a change to the position of the shape’s surface, but not it’s velocity. Changing it will not result in realistic physical behavior. Only use if you know what you are doing!

update (*transform*)

Update, cache and return the bounding box of a shape with an explicit transformation.

Useful if you have a shape without a body and want to use it for querying.

`pymunk.moment_for_circle` (*mass, inner_radius, outer_radius, offset=(0, 0)*)

Calculate the moment of inertia for a hollow circle

inner_radius and *outer_radius* are the inner and outer diameters. (A solid circle has an inner diameter of 0)

`pymunk.moment_for_poly` (*mass, vertices, offset=(0, 0), radius=0*)

Calculate the moment of inertia for a solid polygon shape.

Assumes the polygon center of gravity is at its centroid. The offset is added to each vertex.

`pymunk.moment_for_segment` (*mass, a, b, radius*)

Calculate the moment of inertia for a line segment

The endpoints *a* and *b* are relative to the body

`pymunk.moment_for_box` (*mass, size*)

Calculate the moment of inertia for a solid box centered on the body.

size should be a tuple of (width, height)

class `pymunk.SegmentQueryInfo`

Bases: `pymunk.query_info.SegmentQueryInfo`

Segment queries return more information than just a simple yes or no, they also return where a shape was hit and it's surface normal at the hit point. This object hold that information.

To test if the query hit something, check if `SegmentQueryInfo.shape == None` or not.

Segment queries are like ray casting, but because not all spatial indexes allow processing infinitely long ray queries it is limited to segments. In practice this is still very fast and you don't need to worry too much about the performance as long as you aren't using extremely long segments for your queries.

The properties are as follows

shape Shape that was hit, or None if no collision occurred

point The point of impact.

normal The normal of the surface hit.

alpha The normalized distance along the query segment in the range [0, 1]

__init__

x.__init__(...) initializes *x*; see `help(type(x))` for signature

alpha

Alias for field number 3

count (*value*) → integer – return number of occurrences of *value*

index (*value*[, *start*[, *stop*]]) → integer – return first index of *value*.

Raises `ValueError` if the *value* is not present.

normal

Alias for field number 2

point

Alias for field number 1

shape

Alias for field number 0

class `pymunk.ContactPoint` (*point_a, point_b, distance*)

Bases: `object`

Contains information about a contact point.

point_a and point_b are the contact position on the surface of each shape.

distance is the penetration distance of the two shapes. Overlapping means it will be negative. This value is calculated as `dot(point2 - point1, normal)` and is ignored when you set the `Arbiter.contact_point_set`.

`__init__(point_a, point_b, distance)`
x.`__init__()` initializes x; see `help(type(x))` for signature

distance

point_a

point_b

class `pymunk.ContactPointSet` (*normal, points*)

Bases: `object`

Contact point sets make getting contact information simpler.

normal is the normal of the collision

points is the array of contact points. Can be at most 2 points.

`__init__(normal, points)`
x.`__init__()` initializes x; see `help(type(x))` for signature

normal

points

class `pymunk.Arbiter` (*_arbiter, space*)

Bases: `object`

The Arbiter object encapsulates a pair of colliding shapes and all of the data about their collision.

They are created when a collision starts, and persist until those shapes are no longer colliding.

Warning: Because arbiters are handled by the space you should never hold onto a reference to an arbiter as you don't know when it will be destroyed! Use them within the callback where they are given to you and then forget about them or copy out the information you need from them.

`__init__(_arbiter, space)`
Initialize an Arbiter object from the Chipmunk equivalent struct and the Space.

Note: You should never need to create an instance of this class directly.

contact_point_set

Contact point sets make getting contact information from the Arbiter simpler.

Return *ContactPointSet*

friction

The calculated friction for this collision pair.

Setting the value in a `pre_solve()` callback will override the value calculated by the space. The default calculation multiplies the friction of the two shapes together.

is_first_contact

Returns true if this is the first step the two shapes started touching.

This can be useful for sound effects for instance. If its the first frame for a certain collision, check the energy of the collision in a `post_step()` callback and use that to determine the volume of a sound effect to play.

is_removal

Returns True during a `separate()` callback if the callback was invoked due to an object removal.

restitution

The calculated restitution (elasticity) for this collision pair.

Setting the value in a `pre_solve()` callback will override the value calculated by the space. The default calculation multiplies the elasticity of the two shapes together.

shapes

Get the shapes in the order that they were defined in the collision handler associated with this arbiter

surface_velocity

The calculated surface velocity for this collision pair.

Setting the value in a `pre_solve()` callback will override the value calculated by the space. the default calculation subtracts the surface velocity of the second shape from the first and then projects that onto the tangent of the collision. This is so that only friction is affected by default calculation. Using a custom calculation, you can make something that responds like a pinball bumper, or where the surface velocity is dependent on the location of the contact point.

total_impulse

Returns the impulse that was applied this step to resolve the collision.

This property should only be called from a `post_solve` or `each_arbiter` callback.

total_ke

The amount of energy lost in a collision including static, but not dynamic friction.

This property should only be called from a `post_solve` or `each_arbiter` callback.

class `pymunk.CollisionHandler` (`_handler`, `space`, `*args`, `**kwargs`)

Bases: `object`

A collision handler is a set of 4 function callbacks for the different collision events that Pymunk recognizes.

Collision callbacks are closely associated with Arbiter objects. You should familiarize yourself with those as well.

Note #1: Shapes tagged as sensors (`Shape.sensor == true`) never generate collisions that get processed, so collisions between sensors shapes and other shapes will never call the `post_solve()` callback. They still generate `begin()`, and `separate()` callbacks, and the `pre_solve()` callback is also called every frame even though there is no collision response. Note #2: `pre_solve()` callbacks are called before the sleeping algorithm runs. If an object falls asleep, its `post_solve()` callback won't be called until it's re-awoken.

__init__ (`_handler`, `space`, `*args`, `**kwargs`)

Initialize a CollisionHandler object from the Chipmunk equivalent struct and the Space.

Note: You should never need to create an instance of this class directly.

begin

Two shapes just started touching for the first time this step.

```
func(arbiter, space, data) -> bool
```

Return true from the callback to process the collision normally or false to cause pymunk to ignore the collision entirely. If you return false, the *pre_solve* and *post_solve* callbacks will never be run, but you will still receive a separate event when the shapes stop overlapping.

data

Data property that get passed on into the callbacks.

data is a dictionary and you can not replace it, only fill it with data.

Usefull if the callback needs some extra data to perform its function.

post_solve

Two shapes are touching and their collision response has been processed.

```
func(arbiter, space, data)
```

You can retrieve the collision impulse or kinetic energy at this time if you want to use it to calculate sound volumes or damage amounts. See Arbiter for more info.

pre_solve

Two shapes are touching during this step.

```
func(arbiter, space, data) -> bool
```

Return false from the callback to make pymunk ignore the collision this step or true to process it normally. Additionally, you may override collision values using `Arbiter.friction`, `Arbiter.elasticity` or `Arbiter.surfaceVelocity` to provide custom friction, elasticity, or surface velocity values. See Arbiter for more info.

separate

Two shapes have just stopped touching for the first time this step.

```
func(arbiter, space, data)
```

To ensure that `begin()/separate()` are always called in balanced pairs, it will also be called when removing a shape while its in contact with something or when de-allocating the space.

```
class pymunk.BB(*args)
```

Bases: `pymunk._pickle.PickleMixin`, `object`

Simple bounding box.

Stored as left, bottom, right, top values.

```
__init__(*args)
```

Create a new instance of a bounding box.

Can be created with zero size with `bb = BB()` or with four args defining left, bottom, right and top: `bb = BB(left, bottom, right, top)`

```
area()
```

Return the area

```
bottom
```

```
center()
```

Return the center

```
clamp_vect(v)
```

Returns a copy of the vector `v` clamped to the bounding box

```
contains(other)
```

Returns true if `bb` completely contains the other `bb`

contains_vect (*v*)
Returns true if this bb contains the vector *v*

copy ()
Create a deep copy of this BB.

expand (*v*)
Return the minimal bounding box that contains both this bounding box and the vector *v*

intersects (*other*)
Returns true if the bounding boxes intersect

intersects_segment (*a*, *b*)
Returns true if the segment defined by endpoints *a* and *b* intersect this bb.

left

merge (*other*)
Return the minimal bounding box that contains both this bb and the other bb

merged_area (*other*)
Merges this and other then returns the area of the merged bounding box.

static newForCircle (*p*, *r*)
Convenience constructor for making a BB fitting a circle at position *p* with radius *r*.

right

segment_query (*a*, *b*)
Returns the fraction along the segment query the BB is hit.

Returns infinity if it doesn't hit

top

class pymunk.**ShapeFilter**

Bases: pymunk.shape_filter.ShapeFilter

Pymunk has two primary means of ignoring collisions: groups and category masks.

Groups are used to ignore collisions between parts on a complex object. A ragdoll is a good example. When joining an arm onto the torso, you'll want them to allow them to overlap. Groups allow you to do exactly that. Shapes that have the same group don't generate collisions. So by placing all of the shapes in a ragdoll in the same group, you'll prevent it from colliding against other parts of itself. Category masks allow you to mark which categories an object belongs to and which categories it collides with.

For example, a game has four collision categories: player (0), enemy (1), player bullet (2), and enemy bullet (3). Neither players nor enemies should not collide with their own bullets, and bullets should not collide with other bullets. However, players collide with enemy bullets, and enemies collide with player bullets.

Object	Object Category	Category Mask
Player	0b00001 (1)	0b11000 (4, 5)
Enemy	0b00010 (2)	0b01110 (2, 3, 4)
Player Bullet	0b00100 (3)	0b10001 (1, 5)
Enemy Bullet	0b01000 (4)	0b10010 (2, 5)
Walls	0b10000 (5)	0b01111 (1, 2, 3, 4)

Note that in the table the categories and masks are written as binary values to clearly show the logic. To save space only 5 digits are used. The default type of categories and mask in ShapeFilter is an unsigned int, with a resolution of 32 bits. That means that you have 32 bits to use, in binary notation that is

0b00000000000000000000000000000000 to 0b11111111111111111111111111111111 which can be written in hex as 0x00000000 to 0xFFFFFFFF’.

Everything in this example collides with walls. Additionally, the enemies collide with each other.

By default, objects exist in every category and collide with every category.

Objects can fall into multiple categories. For instance, you might have a category for a red team, and have a red player bullet. In the above example, each object only has one category.

The default type of categories and mask in ShapeFilter is unsigned int which has a resolution of 32 bits on most systems.

There is one last way of filtering collisions using collision handlers. See the section on callbacks for more information. Collision handlers can be more flexible, but can be slower. Fast collision filtering rejects collisions before running the expensive collision detection code, so using groups or category masks is preferred.

Example of how category and mask can be used to filter out player from enemy object:

```
>>> import pymunk
>>> s = pymunk.Space()
>>> player_b = pymunk.Body(1,1)
>>> player_c = pymunk.Circle(player_b, 10)
>>> s.add(player_b, player_c)
>>> player_c.filter = pymunk.ShapeFilter(categories=0b1)
>>> hit = s.point_query_nearest((0,0), 0, pymunk.ShapeFilter())
>>> hit != None
True
>>> filter = pymunk.ShapeFilter(mask=pymunk.ShapeFilter.ALL_MASKS ^ 0b1)
>>> hit = s.point_query_nearest((0,0), 0, filter)
>>> hit == None
True
>>> enemy_b = pymunk.Body(1,1)
>>> enemy_c = pymunk.Circle(enemy_b, 10)
>>> s.add(enemy_b, enemy_c)
>>> hit = s.point_query_nearest((0,0), 0, filter)
>>> hit != None
True
```

ALL_CATEGORIES = 4294967295

ALL_MASKS = 4294967295

__init__

x.__init__(...) initializes x; see help(type(x)) for signature

categories

Alias for field number 1

count (value) → integer – return number of occurrences of value

group

Alias for field number 0

index (value[, start[, stop]]) → integer – return first index of value.

Raises ValueError if the value is not present.

mask

Alias for field number 2

class pymunk.Transform

Bases: pymunk.transform.Transform

Type used for 2x3 affine transforms.

See wikipedia for details: http://en.wikipedia.org/wiki/Affine_transformation

The properties map to the matrix in this way:

a	c	tx
b	d	ty

An instance can be created in this way::

```
>>> Transform(1,2,3,4,5,6)
Transform(a=1, b=2, c=3, d=4, tx=5, ty=6)
```

Or using the default identity in this way::

```
>>> Transform.identity()
Transform(a=1, b=0, c=0, d=1, tx=0, ty=0)
```

Or overriding only some of the values (on a identity matrix):

```
>>> Transform(b=3,ty=5)
Transform(a=1, b=3, c=0, d=1, tx=0, ty=5)
```

__init__

x.__init__(...) initializes x; see help(type(x)) for signature

a

Alias for field number 0

b

Alias for field number 1

c

Alias for field number 2

count (value) → integer – return number of occurrences of value

d

Alias for field number 3

static identity()

The identity transform

index (value[, start[, stop]]) → integer – return first index of value.
Raises ValueError if the value is not present.

tx

Alias for field number 4

ty

Alias for field number 5

class pymunk.PointQueryInfo

Bases: pymunk.query_info.PointQueryInfo

PointQueryInfo holds the result of a point query made on a Shape or Space.

The properties are as follows

shape The nearest shape, None if no shape was within range.

point The closest point on the shape's surface. (in world space coordinates)

distance The distance to the point. The distance is negative if the point is inside the shape.

gradient The gradient of the signed distance function.

The value should be similar to `PointQueryInfo.point/PointQueryInfo.distance`, but accurate even for very small values of `info.distance`.

__init__

`x.__init__(...)` initializes x; see `help(type(x))` for signature

count (*value*) → integer – return number of occurrences of value

distance

Alias for field number 2

gradient

Alias for field number 3

index (*value*[, *start*[, *stop*]]) → integer – return first index of value.

Raises `ValueError` if the value is not present.

point

Alias for field number 1

shape

Alias for field number 0

class `pymunk.ShapeQueryInfo`

Bases: `pymunk.query_info.ShapeQueryInfo`

Shape queries return more information than just a simple yes or no, they also return where a shape was hit. This object hold that information.

__init__

`x.__init__(...)` initializes x; see `help(type(x))` for signature

contact_point_set

Alias for field number 1

count (*value*) → integer – return number of occurrences of value

index (*value*[, *start*[, *stop*]]) → integer – return first index of value.

Raises `ValueError` if the value is not present.

shape

Alias for field number 0

class `pymunk.SpaceDebugDrawOptions`

Bases: `object`

`SpaceDebugDrawOptions` configures debug drawing.

If appropriate its usually easy to use the supplied draw implementations directly: `pymunk.pygame_util`, `pymunk.pyglet_util` and `pymunk.matplotlib_util`.

DRAW_COLLISION_POINTS

alias of `CP_SPACE_DEBUG_DRAW_COLLISION_POINTS`

DRAW_CONSTRAINTS

alias of `CP_SPACE_DEBUG_DRAW_CONSTRAINTS`

DRAW_SHAPES

alias of CP_SPACE_DEBUG_DRAW_SHAPES

__init__()

x.__init__(...) initializes x; see help(type(x)) for signature

collision_point_color**color_for_shape** (*shape*)**constraint_color****draw_circle** (**args*)**draw_dot** (**args*)**draw_fat_segment** (**args*)**draw_polygon** (**args*)**draw_segment** (**args*)**flags**

Bit flags which of shapes, joints and collisions should be drawn.

By default all 3 flags are set, meaning shapes, joints and collisions will be drawn.

Example using the basic text only DebugDraw implementation (normally you would use the desired backend instead, such as *pygame_util.DrawOptions* or *pyglet_util.DrawOptions*):

```
>>> import pymunk
>>> s = pymunk.Space()
>>> b = pymunk.Body()
>>> c = pymunk.Circle(b, 10)
>>> c.mass = 3
>>> s.add(b, c)
>>> s.add(pymunk.Circle(s.static_body, 3))
>>> s.step(0.01)
>>> options = pymunk.SpaceDebugDrawOptions()
```

```
>>> # Only draw the shapes, nothing else:
>>> options.flags = pymunk.SpaceDebugDrawOptions.DRAW_SHAPES
>>> s.debug_draw(options)
('draw_circle', (Vec2d(0.0, 0.0), 0.0, 10.0, SpaceDebugColor(r=44.0, g=62.0,
↳b=80.0, a=255.0), SpaceDebugColor(r=52.0, g=152.0, b=219.0, a=255.0)))
('draw_circle', (Vec2d(0.0, 0.0), 0.0, 3.0, SpaceDebugColor(r=44.0, g=62.0,
↳b=80.0, a=255.0), SpaceDebugColor(r=149.0, g=165.0, b=166.0, a=255.0)))
```

```
>>> # Draw the shapes and collision points:
>>> options.flags = pymunk.SpaceDebugDrawOptions.DRAW_SHAPES
>>> options.flags |= pymunk.SpaceDebugDrawOptions.DRAW_COLLISION_POINTS
>>> s.debug_draw(options)
('draw_circle', (Vec2d(0.0, 0.0), 0.0, 10.0, SpaceDebugColor(r=44.0, g=62.0,
↳b=80.0, a=255.0), SpaceDebugColor(r=52.0, g=152.0, b=219.0, a=255.0)))
('draw_circle', (Vec2d(0.0, 0.0), 0.0, 3.0, SpaceDebugColor(r=44.0, g=62.0,
↳b=80.0, a=255.0), SpaceDebugColor(r=149.0, g=165.0, b=166.0, a=255.0)))
('draw_segment', (Vec2d(1.0, 0.0), Vec2d(-8.0, 0.0), SpaceDebugColor(r=231.0,
↳g=76.0, b=60.0, a=255.0)))
```

```
shape_dynamic_color = SpaceDebugColor(r=52, g=152, b=219, a=255)
```

```
shape_kinematic_color = SpaceDebugColor(r=39, g=174, b=96, a=255)
```

```

shape_outline_color
shape_sleeping_color = SpaceDebugColor(r=114, g=148, b=168, a=255)
shape_static_color = SpaceDebugColor(r=149, g=165, b=166, a=255)

```

9.5 Examples

Here you will find a list of the included examples. Each example have a short description and a screenshot (if applicable).

To look at the source code of an example open it on github by following the link. The examples are also included in the source distribution of Pymunk (but not if you install using the wheel file). You can find the source distribution at PyPI, <https://pypi.org/project/pymunk/#files> (file named pymunk-x.y.z.zip).

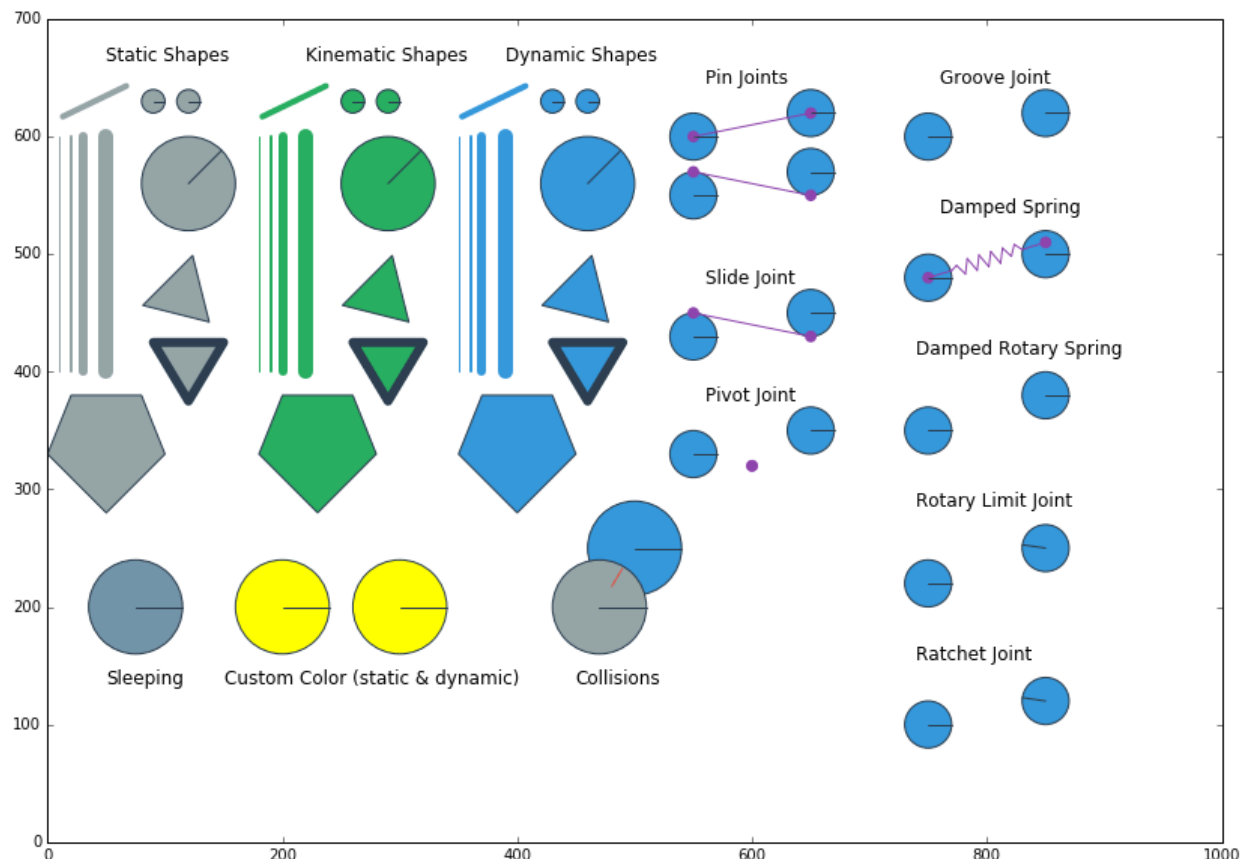
9.5.1 Jupyter Notebooks

There are a couple examples that are provided as Jupyter Notebooks (.ipynb). They are possible to either view online in a browser directly on github, or opened as a Notebook.

9.5.1.1 matplotlib_util_demo.ipynb

Displays the same space as the pygame and pyglet draw demos, but using matplotlib and the notebook.

Source: [examples/matplotlib_util_demo.ipynb](#)



9.5.1.2 newtons_cradle.ipynb

Similar simulation as `newtons_cradle.py`, but this time as a Notebook. Compared to the draw demo this demo will output a animation of the simulated space.

Source: `examples/newtons_cradle.ipynb`

9.5.2 Standalone Python

To run the examples yourself either install pymunk or run it using the convenience `run.py` script.

Given that pymunk is installed where your python will find it:

```
>cd examples
>python breakout.py
```

To run directly without installing anything. From the pymunk source folder:

```
>cd examples
>python run.py breakout.py
```

Each example contains something unique. Not all of the examples use the same style. For example, some use the `pymunk.pygame_util` module to draw stuff, others contain the actual drawing code themselves. However, each example is self contained. Except for external libraries (such as `pygame`) and `pymunk` each example can be run directly to make it easy to read the code and understand what happens even if it means that some code is repeated for each example.

If you have made something that uses pymunk and would like it displayed here or in a showcase section of the site, feel free to contact me!

Example files

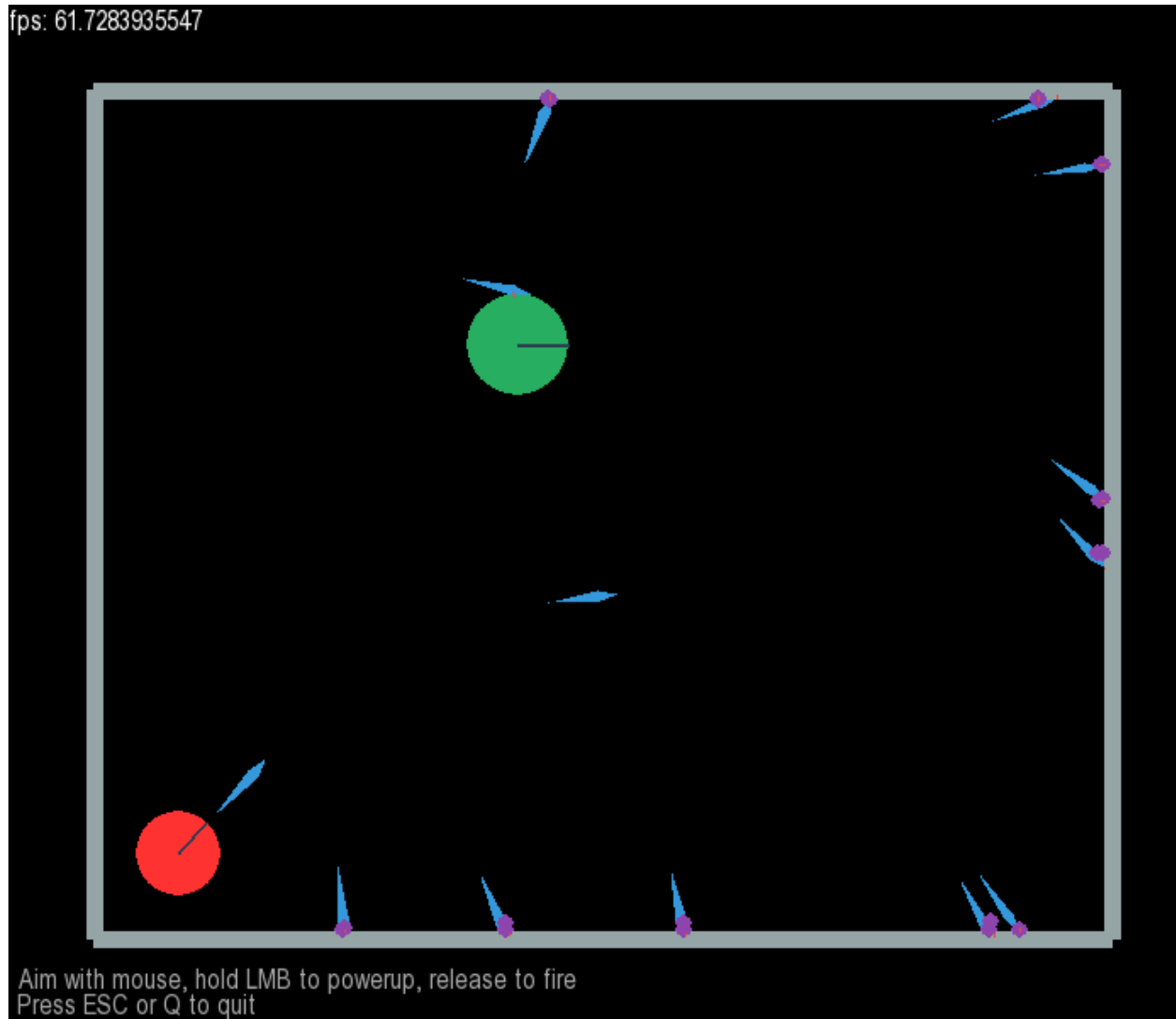
- *arrows.py*
- *balls_and_lines.py*
- *basic_test.py*
- *bouncing_balls.py*
- *box2d_pyramid.py*
- *box2d_vertical_stack.py*
- *breakout.py*
- *contact_and_no_flip.py*
- *contact_with_friction.py*
- *copy_and_pickle.py*
- *damped_rotary_spring_pointer.py*
- *deformable.py*
- *flipper.py*
- *index_video.py*
- *kivy_pymunk_demo*

- *newtons_cradle.py*
- *no_debug.py*
- *platformer.py*
- *playground.py*
- *point_query.py*
- *py2exe_setup__basic_test.py*
- *py2exe_setup__breakout.py*
- *pygame_util_demo.py*
- *pyglet_util_demo.py*
- *run.py*
- *shapes_for_draw_demos.py*
- *slide_and_pinjoint.py*
- *spiderweb.py*
- *using_sprites.py*
- *using_sprites_pyglet.py*

9.5.2.1 arrows.py

Source: [examples/arrows.py](#)

Showcase of flying arrows that can stick to objects in a somewhat realistic looking way.

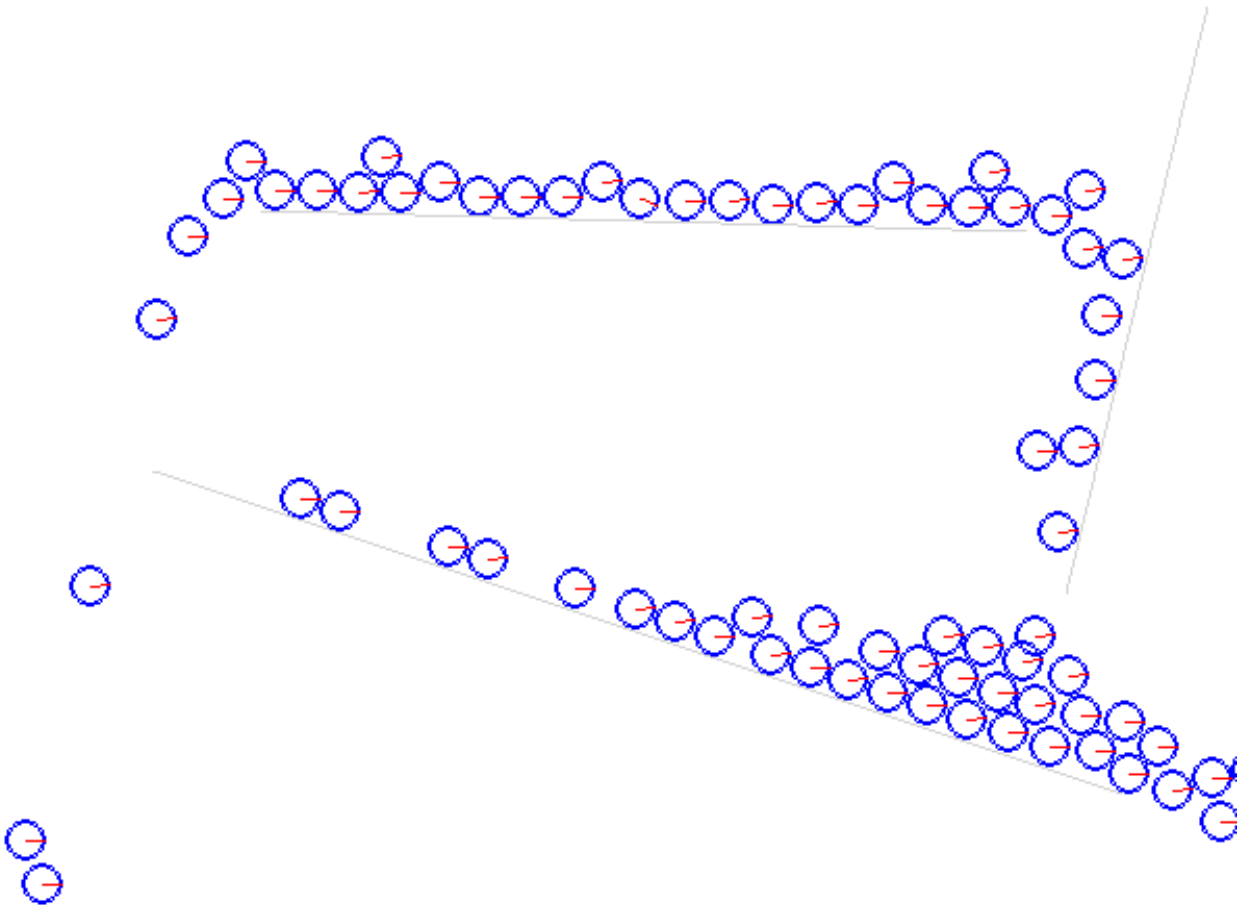


9.5.2.2 balls_and_lines.py

Source: [examples/balls_and_lines.py](#)

This example lets you dynamically create static walls and dynamic balls

LMB: Create ball
LMB + Shift: Create many balls
RMB: Drag to create wall, release to finish
Space: Pause physics simulation



9.5.2.3 basic_test.py

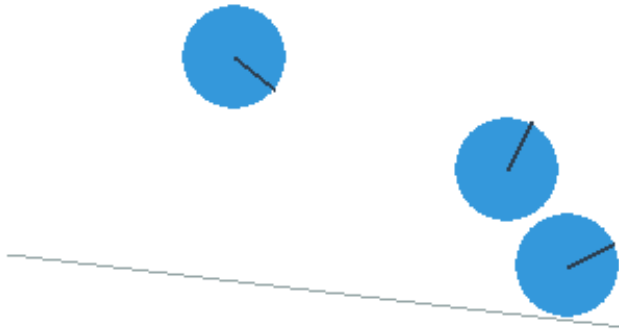
Source: [examples/basic_test.py](#)

Very simple example that does not depend on any third party library such as pygame or pyglet like the other examples.

9.5.2.4 bouncing_balls.py

Source: [examples/bouncing_balls.py](#)

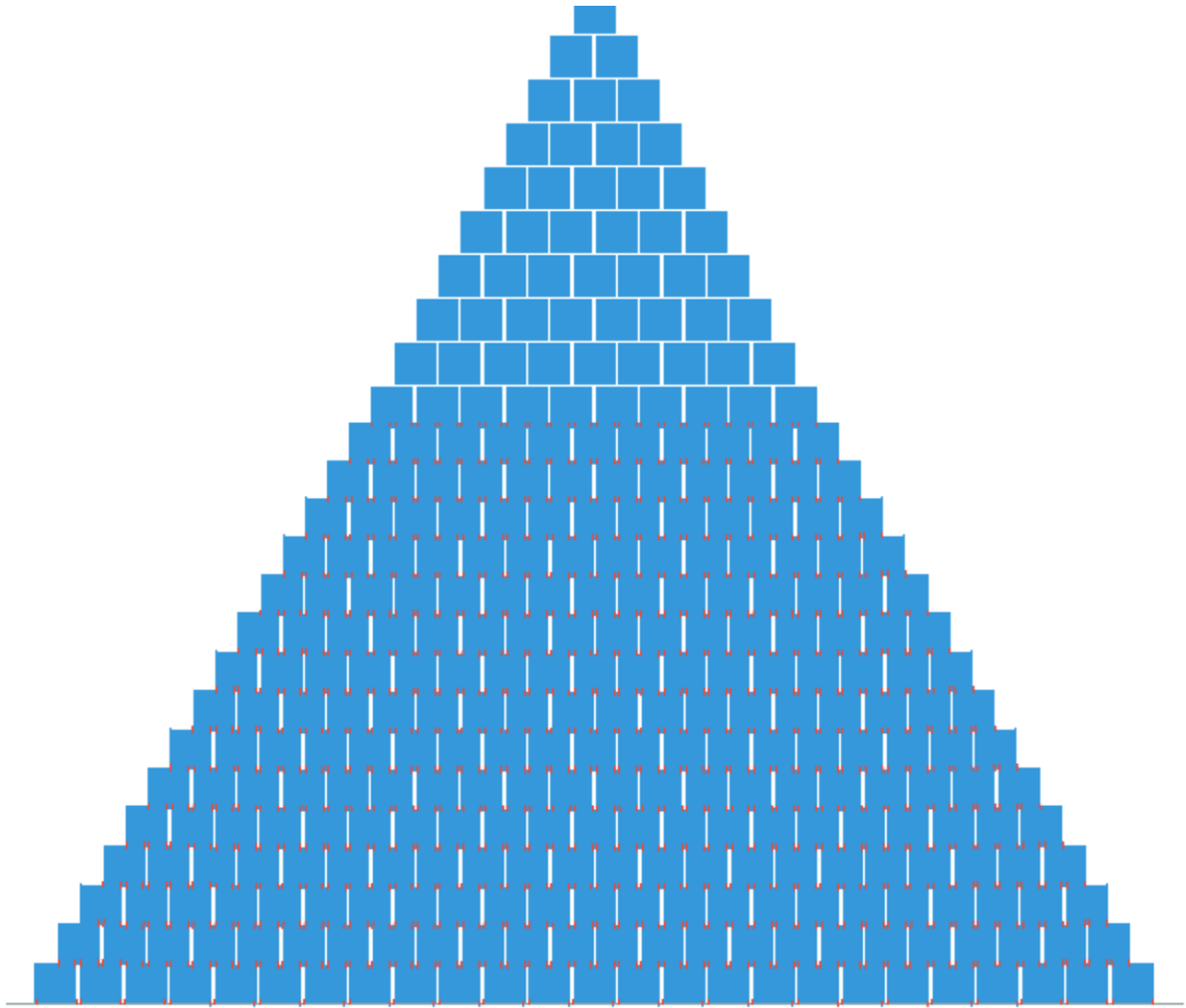
This example spawns (bouncing) balls randomly on a L-shape constructed of two segment shapes. Not interactive.



9.5.2.5 box2d_pyramid.py

Source: [examples/box2d_pyramid.py](#)

Remake of the pyramid demo from the box2d testbed.

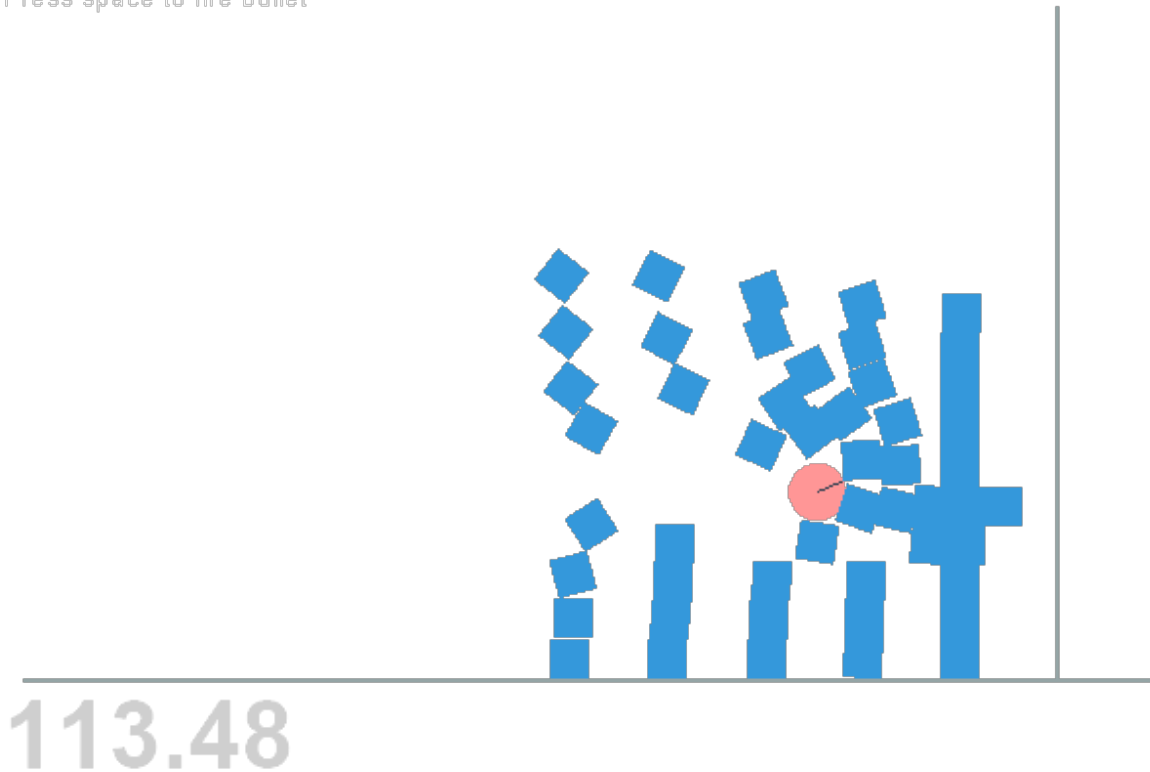


9.5.2.6 box2d_vertical_stack.py

Source: [examples/box2d_vertical_stack.py](#)

Remake of the veritcal stack demo from the box2d testbed.

Press space to fire bullet

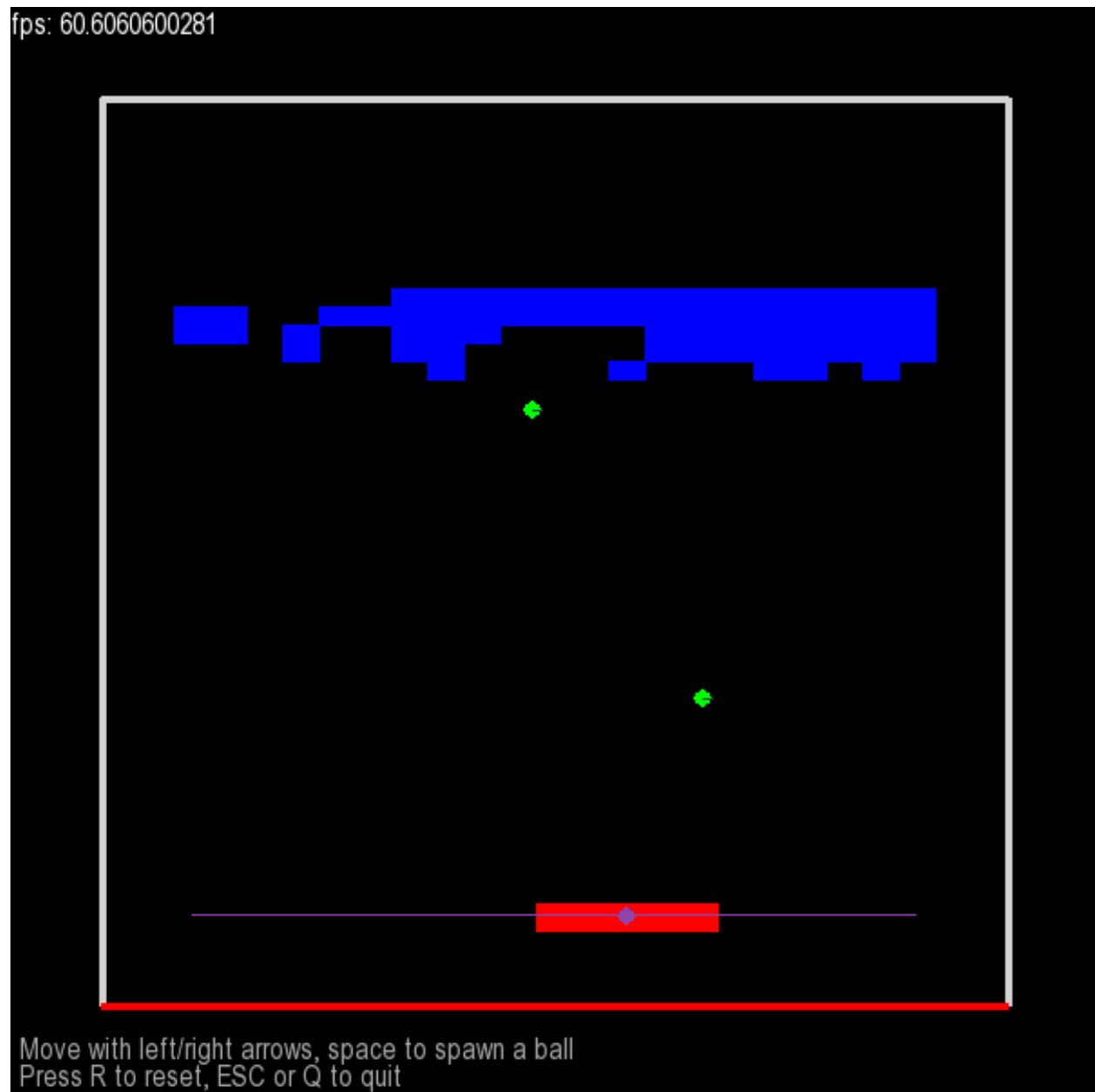


9.5.2.7 breakout.py

Source: [examples/breakout.py](#)

Very simple breakout clone. A circle shape serves as the paddle, then breakable bricks constructed of Poly-shapes.

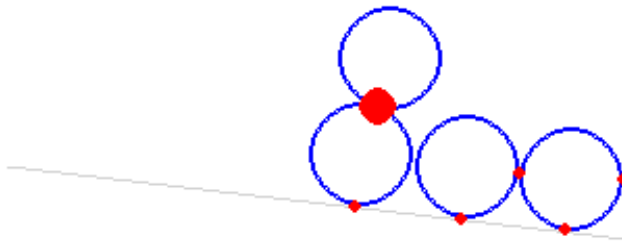
The code showcases several pymunk concepts such as elasticity, impulses, constant object speed, joints, collision handlers and post step callbacks.



9.5.2.8 contact_and_no_flipy.py

Source: [examples/contact_and_no_flipy.py](#)

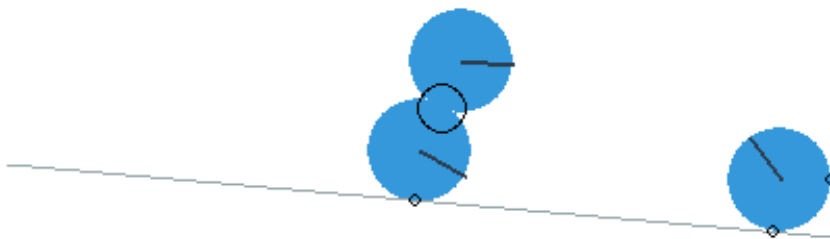
This example spawns (bouncing) balls randomly on a L-shape constructed of two segment shapes. For each collision it draws a red circle with size depending on collision strength. Not interactive.



9.5.2.9 `contact_with_friction.py`

Source: [examples/contact_with_friction.py](#)

This example spawns (bouncing) balls randomly on a L-shape constructed of two segment shapes. Displays collision strength and rotating balls thanks to friction. Not interactive.



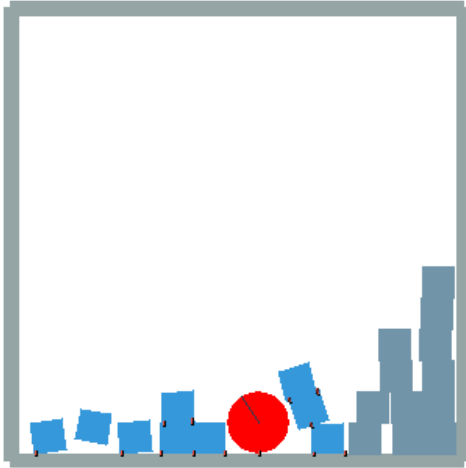
9.5.2.10 `copy_and_pickle.py`

Source: [examples/copy_and_pickle.py](#)

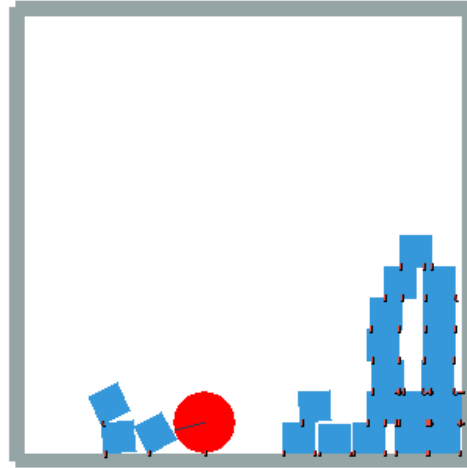
This example shows how you can copy, save and load a space using pickle.

fps: 60.2409629822

space.sleep_time_threshold set to 0.5 seconds



space.sleep_time_threshold set to inf (disabled)



Press SPACE to give an impulse to the ball.
Press S to save the current state to file, press L to load it.
Press R to reset, ESC or Q to quit

9.5.2.11 damped_rotary_spring_pointer.py

Source: [examples/damped_rotary_spring_pointer.py](#)

This example showcase an arrow pointing or aiming towards the cursor.

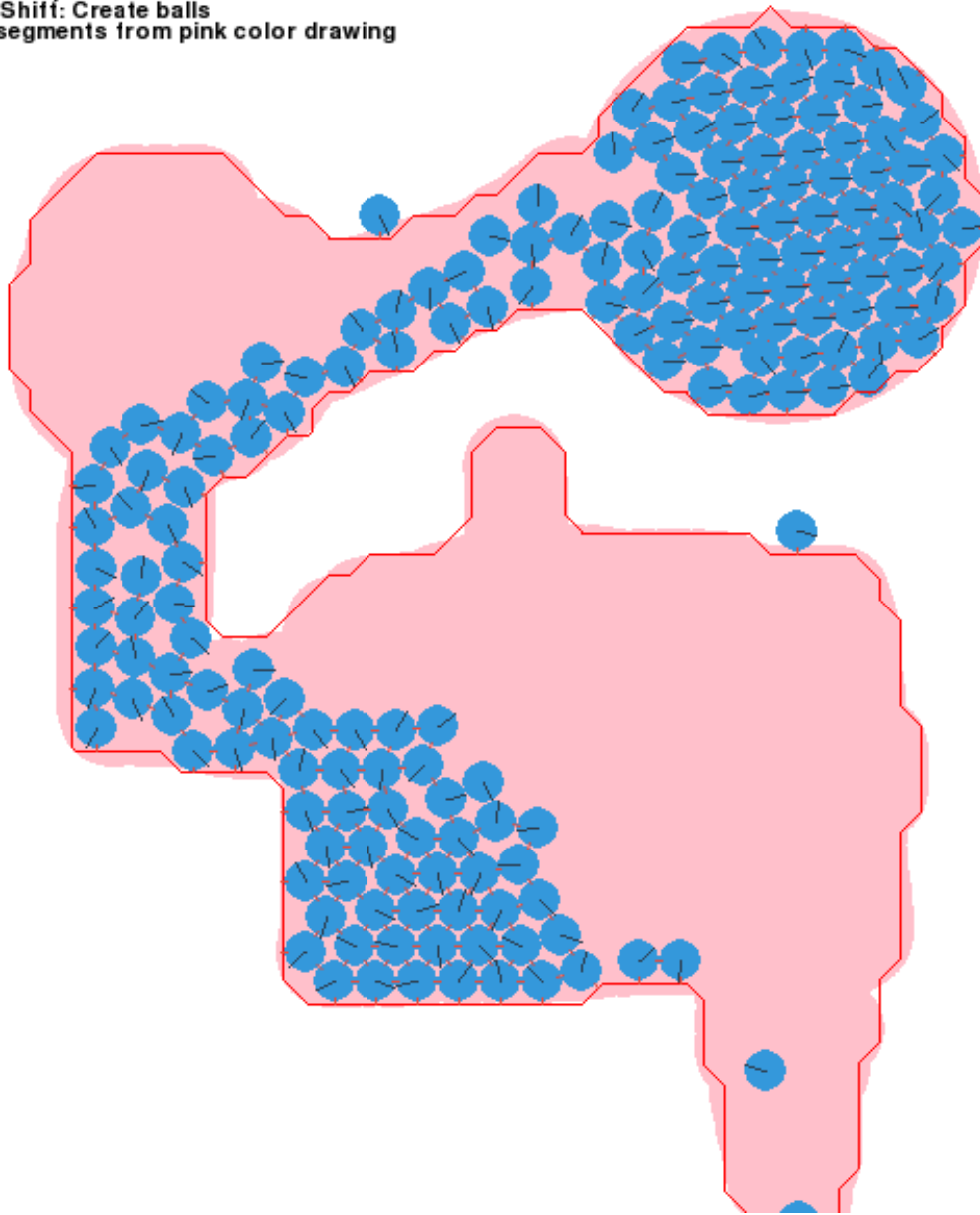


9.5.2.12 deformable.py

Source: [examples/deformable.py](#)

This is an example on how the autogeometry can be used for deformable terrain.

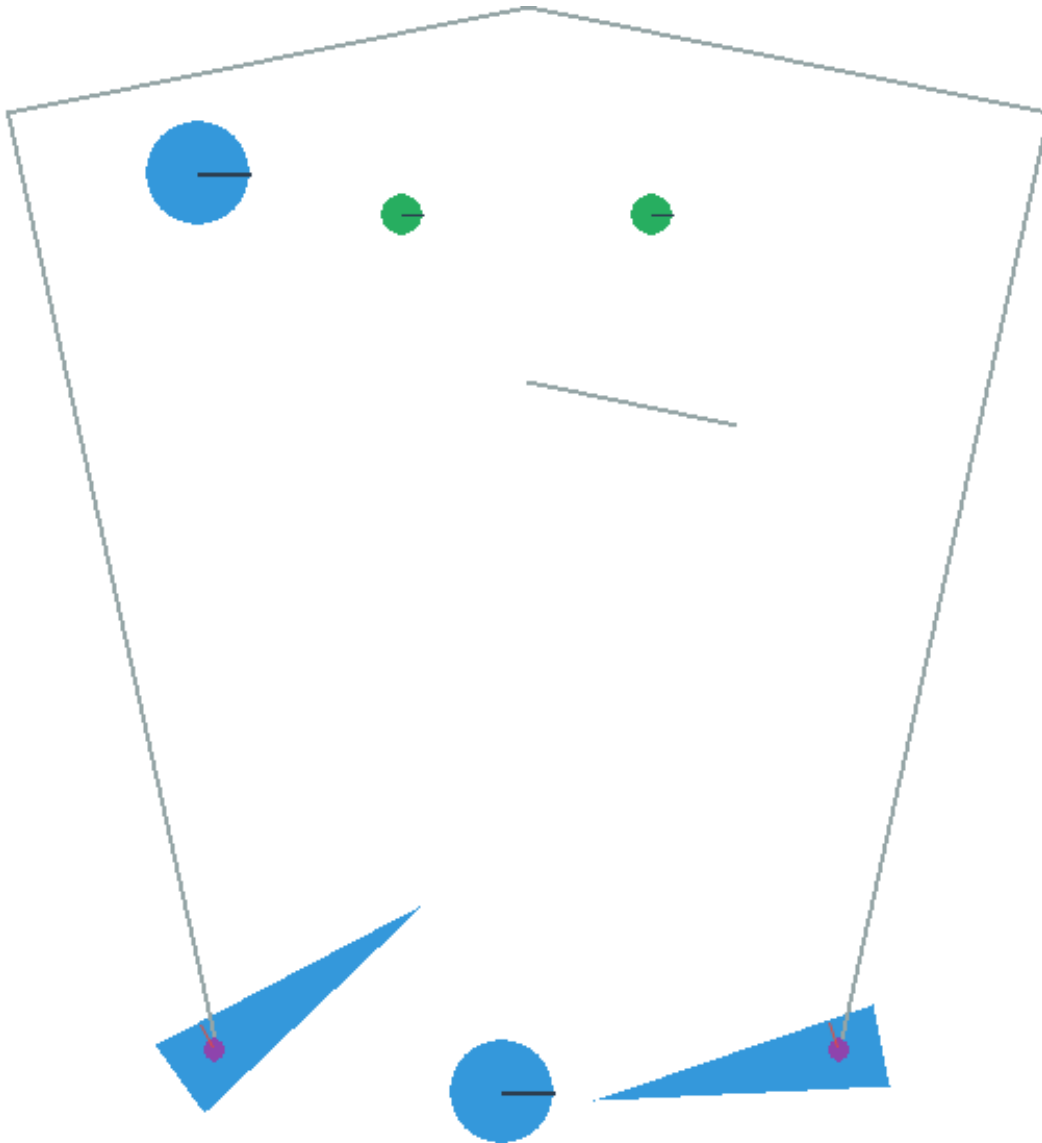
LMB(hold): Draw pink color
LMB(hold) + Shift: Create balls
g: Generate segments from pink color drawing
r: Reset



9.5.2.13 flipper.py

Source: [examples/flipper.py](#)

A very basic flipper game.



9.5.2.14 index_video.py

Source: [examples/index_video.py](#)

Program used to generate the logo animation on the pymunk main page.

This program will showcase several features of Pymunk, such as collisions, debug drawing, automatic generation of shapes from images, motors, joints and sleeping bodies.

Source: [examples/kivy_pymunk_demo](#)

A rudimentary port of the intro video used for the intro animation on [pymunk.org](#). The code is tested on both Windows and Android.

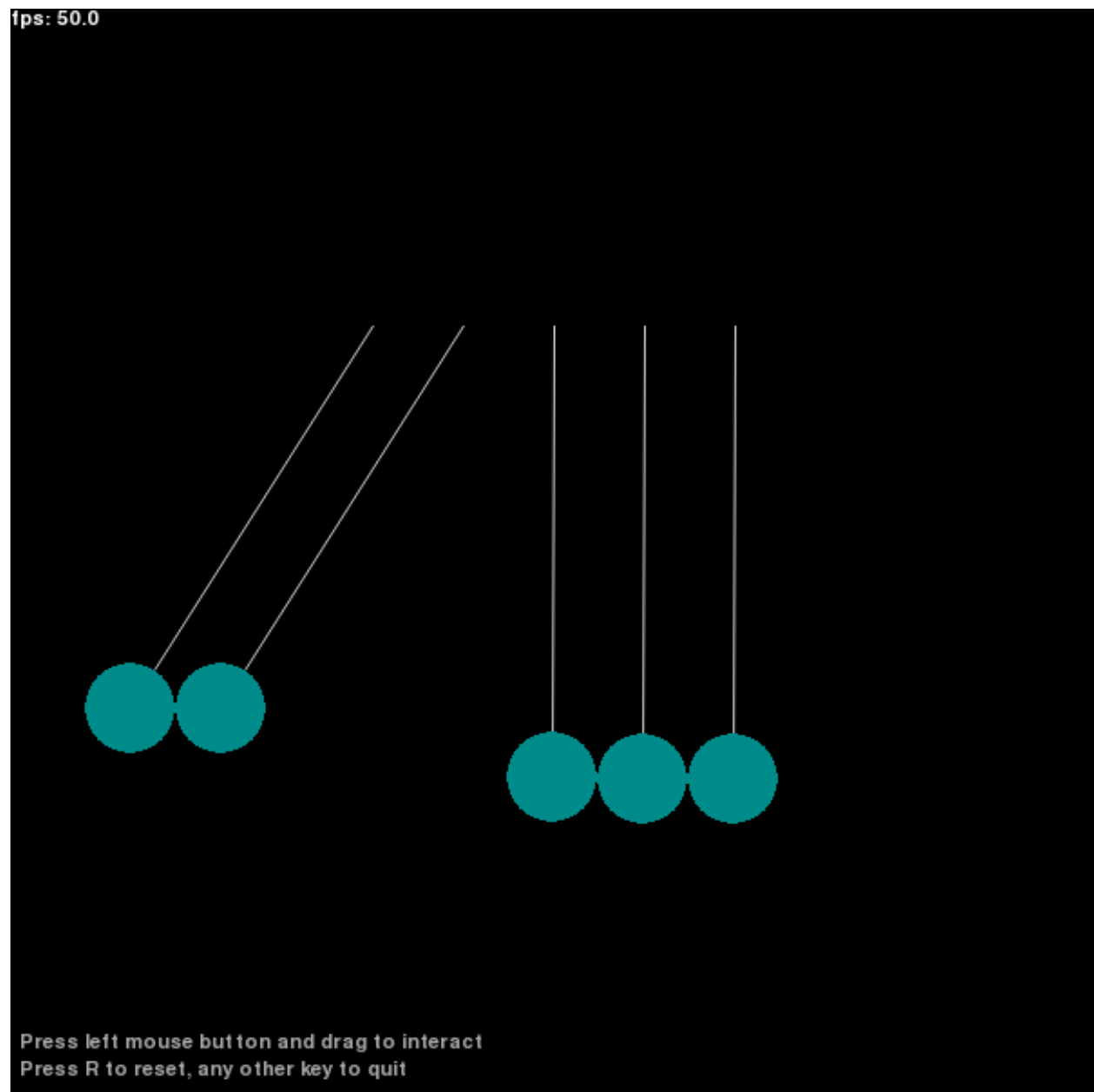
Note that it doesn't display Kivy best practices, the `intro_video` code was just converted to Kivy in the most basic way to show that its possible, its not supposed to show the best way to structure a Kivy application using Pymunk.

A rudimentary port of the intro video used for the intro animation on pymunk.org. The code is tested on both Windows and Android.

Source: [examples/newtons_cradle.py](#)

A screensaver version of Newton's Cradle with an interactive mode.

A screensaver version of Newton's Cradle with an interactive mode.



9.5.2.17 no_debug.py

Source: [examples/no_debug.py](#)

Very simple showcase of how to run pymunk with debug mode off

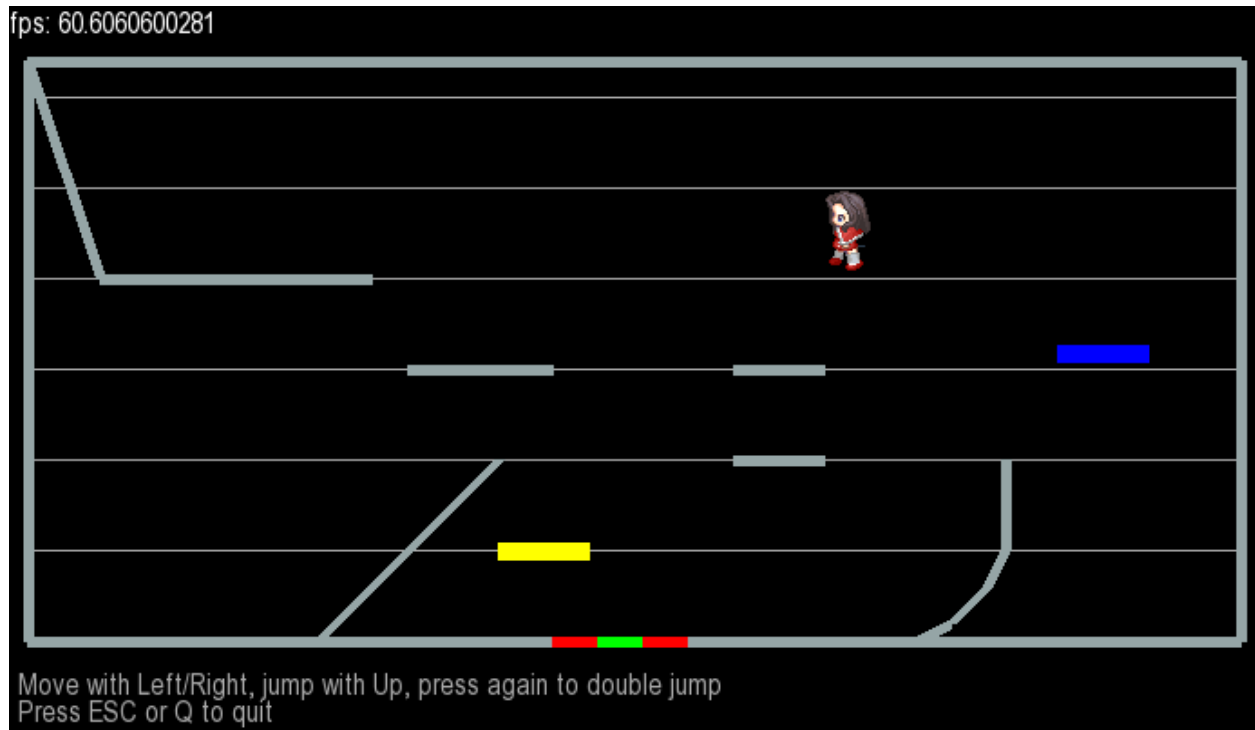
9.5.2.18 platformer.py

Source: [examples/platformer.py](#)

Showcase of a very basic 2d platformer

The red girl sprite is taken from Sithjester's RMXP Resources: <http://untamed.wild-refuge.net/rmxpresources.php?characters>

Note: The code of this example is a bit messy. If you adapt this to your own code you might want to structure it a bit differently.

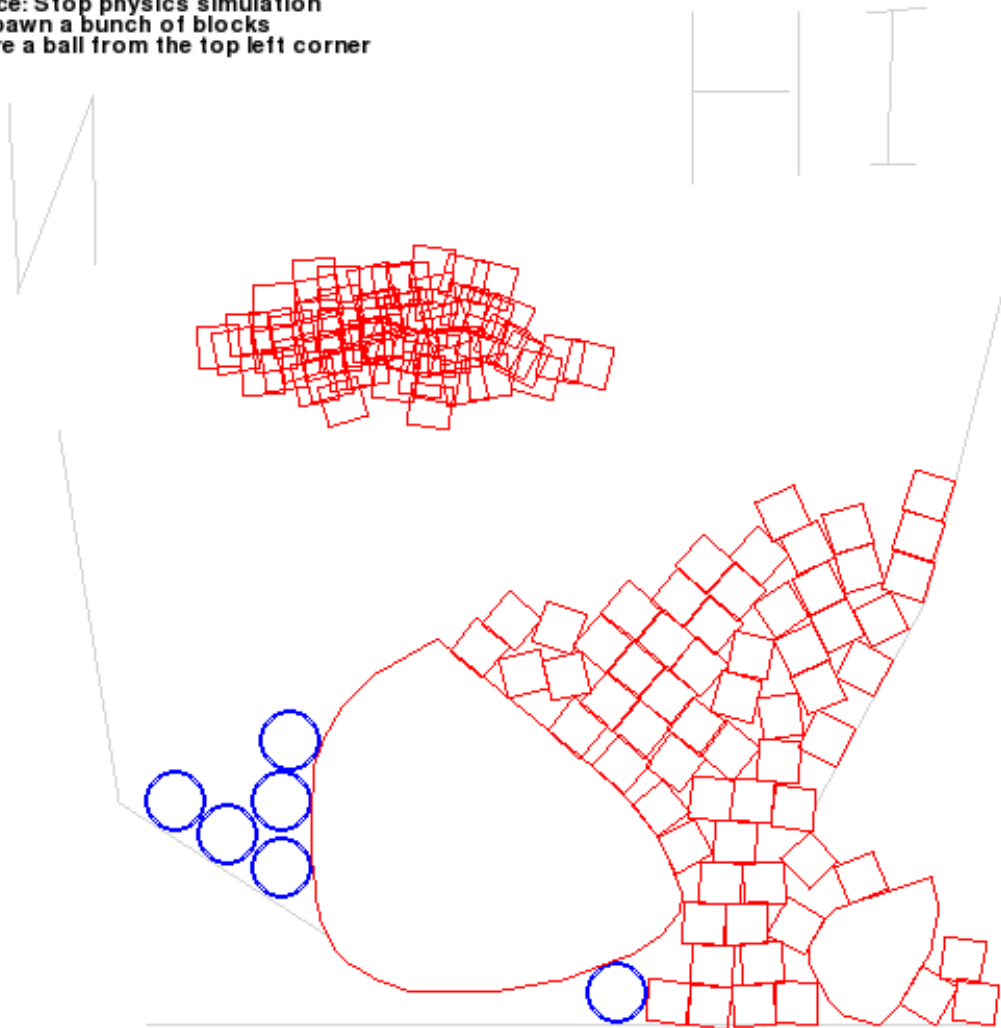


9.5.2.19 playground.py

Source: [examples/playground.py](#)

A basic playground. Most interesting function is draw a shape, basically move the mouse as you want and pymunk will approximate a Poly shape from the drawing.

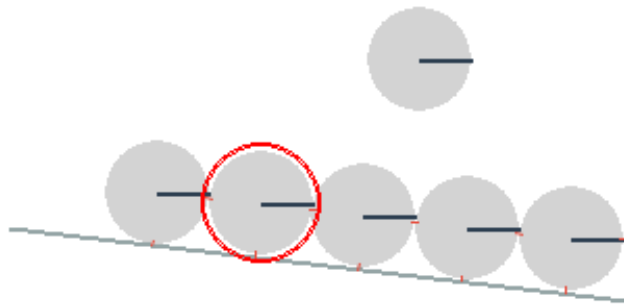
LMB: Create ball
LMB + Shift: Create box
RMB on object: Remove object
RMB(hold) + Shift: Create polygon, release to finish (we be converted to a convex hull of the points)
RMB + Ctrl: Create wall, release to finish
Space: Stop physics simulation
k: Spawn a bunch of blocks
f: Fire a ball from the top left corner



9.5.2.20 point_query.py

Source: [examples/point_query.py](#)

This example showcase point queries by highlighting the shape under the mouse pointer.



9.5.2.21 py2exe_setup__basic_test.py

Source: [examples/py2exe_setup__basic_test.py](#)

Simple example of py2exe to create a exe of the basic_test example.

Tested on py2exe 0.9.2.2 on python 3.4

9.5.2.22 py2exe_setup__breakout.py

Source: [examples/py2exe_setup__breakout.py](#)

Example script to create a exe of the breakout example using py2exe.

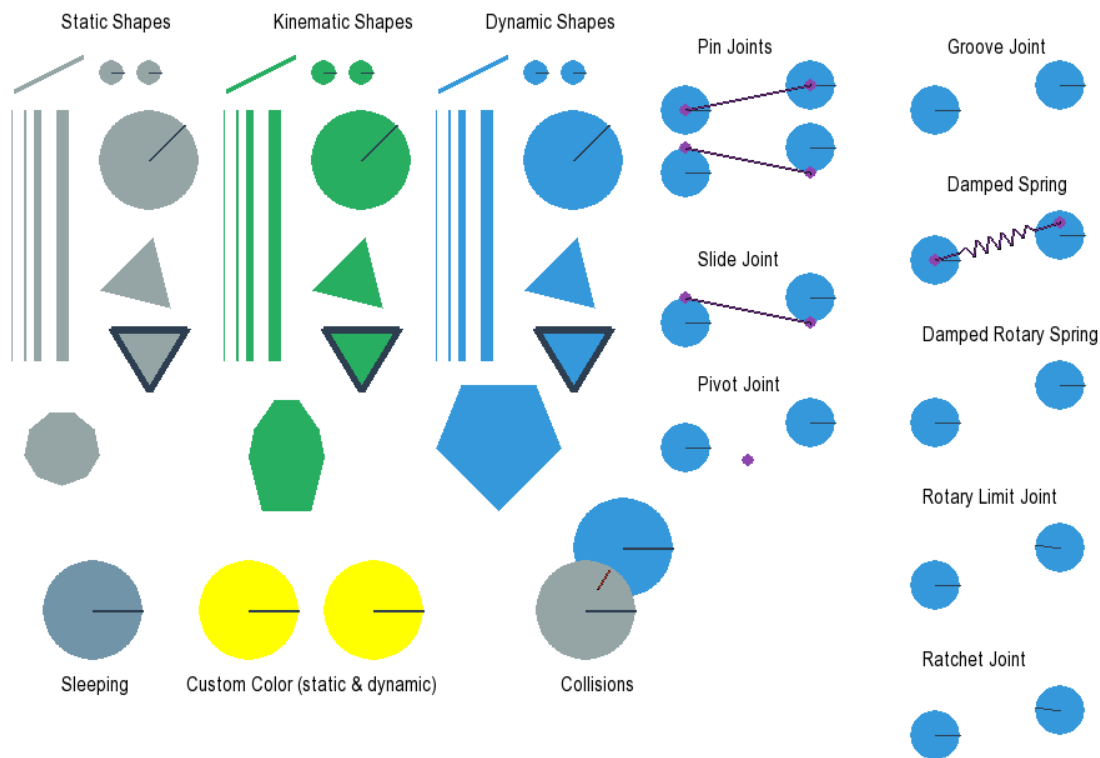
Tested on py2exe 0.9.2.2 on python 3.4

9.5.2.23 pygame_util_demo.py

Source: [examples/pygame_util_demo.py](#)

Showcase what the output of `pymunk.pygame_util` draw methods will look like.

See `pyglet_util_demo.py` for a comparison to `pyglet`.



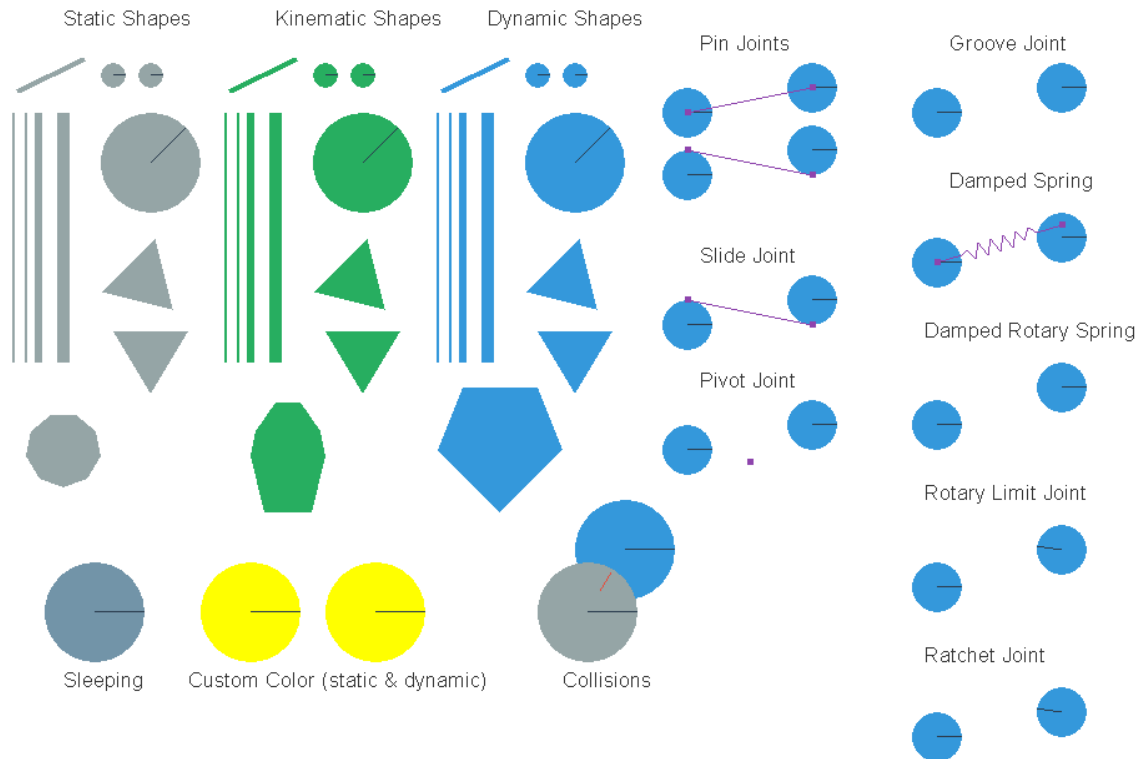
Demo example of `pygame_util.DrawOptions()`

9.5.2.24 pyglet_util_demo.py

Source: [examples/pyglet_util_demo.py](#)

Showcase what the output of `pymunk.pyglet_util` draw methods will look like.

See `pygame_util_demo.py` for a comparison to `pygame`.



Demo example of shapes drawn by `pyglet_util.draw()`

9.5.2.25 run.py

Source: [examples/run.py](#)

Use to run examples using pymunk located one folder level up. Useful if you have the whole pymunk source tree and want to run the examples in a quick and dirty way. (a poor man's virtualenv if you like)

For example, to run the breakout demo:

```
> cd examples
> python run.py breakout.py
```

9.5.2.26 shapes_for_draw_demos.py

Source: [examples/shapes_for_draw_demos.py](#)

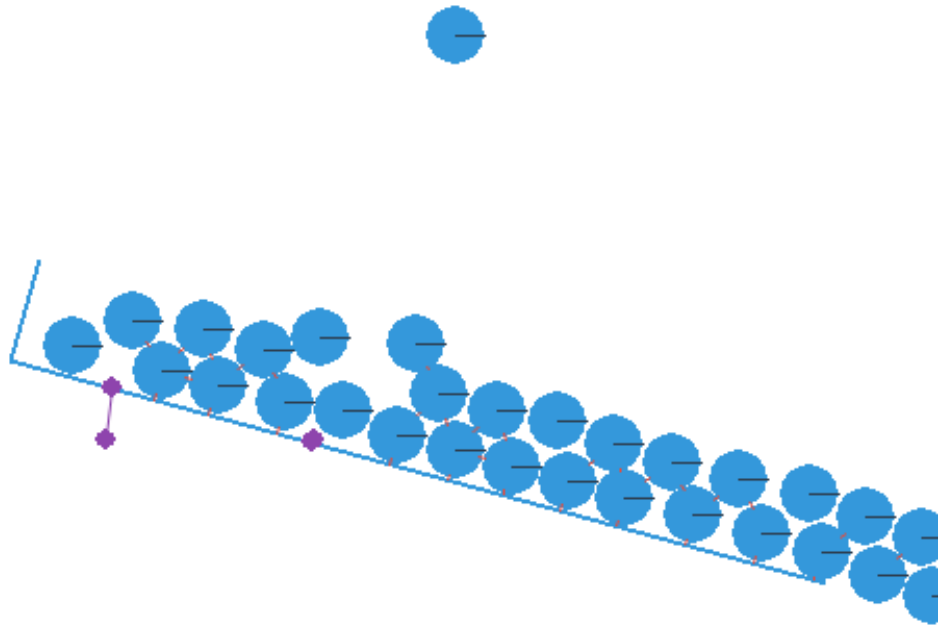
Helper function `fill_space` for the draw demos. Adds a lot of stuff to a space.

9.5.2.27 slide_and_pinjoint.py

Source: [examples/slide_and_pinjoint.py](#)

A L shape attached with a joint and constrained to not tip over.

This example is also used in the Get Started Tutorial.

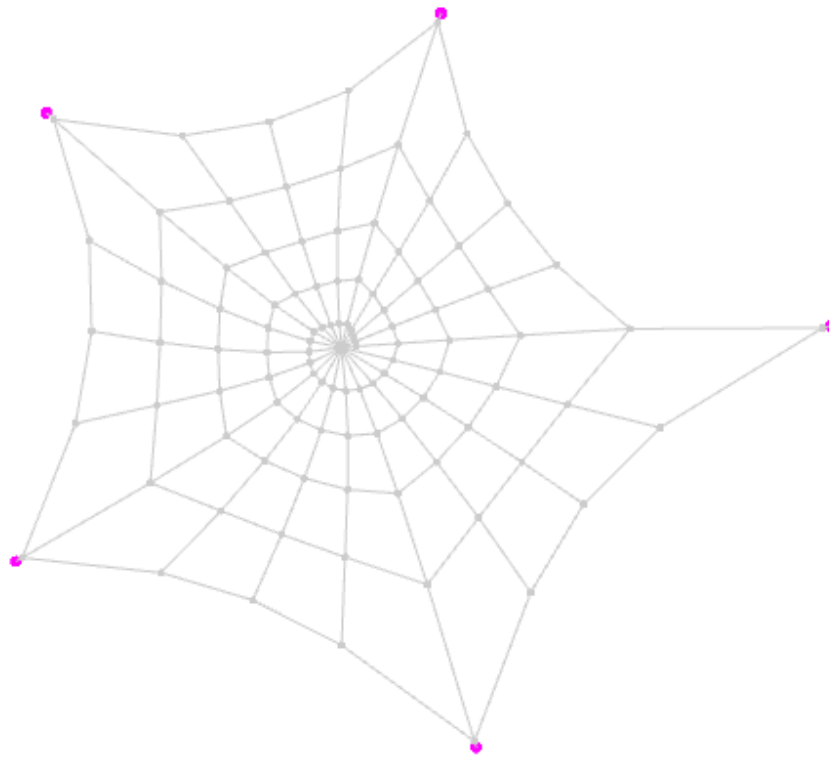


9.5.2.28 spiderweb.py

Source: [examples/spiderweb.py](#)

Showcase of a elastic spiderweb (drawing with pyglet)

It is possible to grab one of the crossings with the mouse

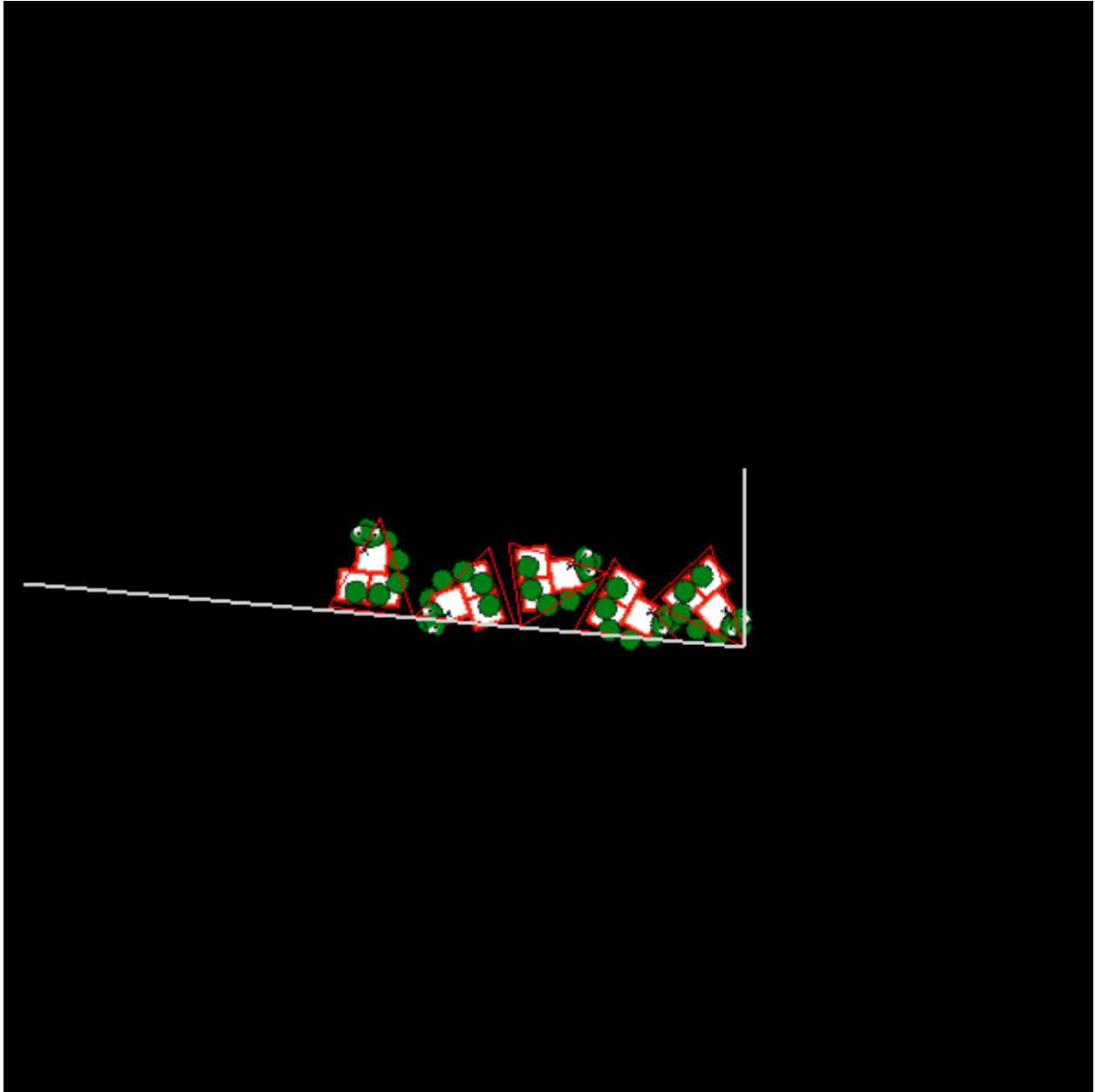


59.42

9.5.2.29 using_sprites.py

Source: [examples/using_sprites.py](#)

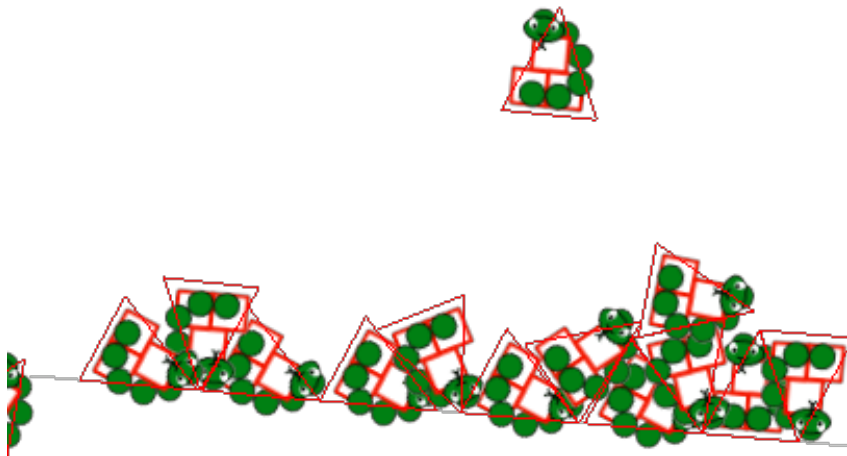
Very basic example of using a sprite image to draw a shape more similar how you would do it in a real game instead of the simple line drawings used by the other examples.



9.5.2.30 using_sprites_pyglet.py

Source: [examples/using_sprites_pyglet.py](#)

This example is a clone of the `using_sprites` example with the difference that it uses `pyglet` instead of `pygame` to showcase sprite drawing.



58.98

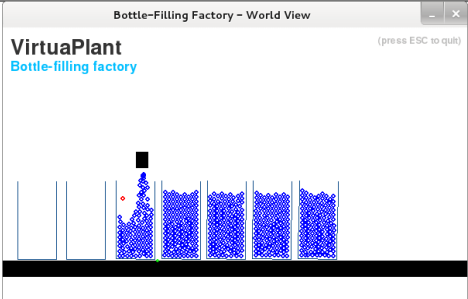
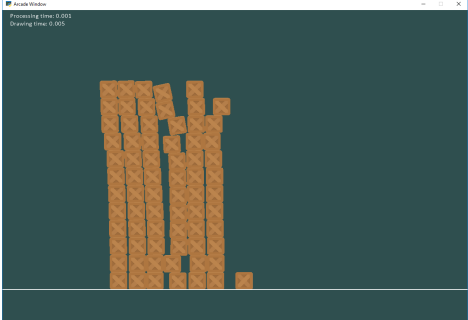

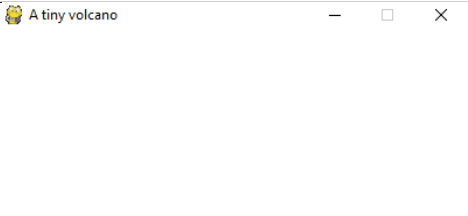
9.6 Showcase

This page shows some uses of Pymunk. If you also have done something using Pymunk please let me know and I can add it here!

9.6.1 Games

	<p>My Sincerest Apologies</p> <p>made by The Larry and Dan show (mauve, larry). Retrieved 2018-10-25</p> <p>Winner of PyWeek 24 (Overall Team Entry)</p> <p>A game of fun, shooting, and “I’m sorry to put you through this”. A fabricator robot on Mars was supposed to make a bunch of robots! But it got lazy and made robots that could make other robots. And it made them smarter than they should have been. Now they’ve all gone off and hidden away behind various tanks and computers. Happily, he knew how to construct <i>you</i>, a simple fighting robot. It’s your job to clean out each area!</p> <p>See Daniel Popes teardown here for additional details</p>
	<p>Beneath the Ice</p> <p>made by Team Chimera (mit-mit, Lucid Design Ar). Retrieved 2016-09-25</p> <p>Winner of PyWeek 22 (Overall Team Entry)</p> <p>Beneath the Ice is a submarine exploration game and puzzle solving adventure! Uncover a mysterious pariah who can’t let you discover his secrets, who can’t let you in! Team Chimera take 3!</p>
	<p>Invisipin</p>

9.6.2 Non-Games

	<p>VirtuaPlant</p> <p>made by Jan Seidl. Retrieved 2018-06-13</p> <p>VirtuaPlant is a Industrial Control Systems simulator which adds a “similar to real-world control logic” to the basic “read/write tags” feature of most PLC simulators. Paired with a game library and 2d physics engine, VirtuaPlant is able to present a GUI simulating the “world view” behind the control system allowing the user to have a vision of the would-be actions behind the control systems.</p>
	<p>The Python Arcade Library</p> <p>made by Paul. Retrieved 2018-03-05</p> <p>Arcade is an easy-to-learn Python library for creating 2D video games. It is not directly tied to Pymunk, but includes a number of examples and helper classes to use Pymunk physics from a Arcade application.</p>
	<p>billiARds A Game of Augmented Reality Pool</p> <p>made by Alex Baikovitz. Retrieved 2017-05-21</p> <p>Alex built billiARds for his 15-112 (Fundamentals of Programming and Computer Science) term project at Carnegie Mellon University. Made in Python3 using OpenCV, Pygame, and Pymunk. Users can simply use a pool cue stick and run the program on any ordinary surface.</p>
	

9.6.3 Papers / Science

Pymunk has been used or referenced in a number of scientific papers

- Caselles-Dupré, Hugo, Louis Annabi, Oksana Hagen, Michael Garcia-Ortiz, and David Filliat. “Flatland: a Lightweight First-Person 2-D Environment for Reinforcement Learning.” arXiv preprint arXiv:1809.00510 (2018).
- Yingzhen, Li, and Stephan Mandt. “Disentangled Sequential Autoencoder.” In International Conference on Machine Learning, pp. 5656-5665. 2018.
- Melnik, Andrew. “Sensorimotor Processing in the Human Brain and in Cognitive Architectures.” (2018).
- Li, Yingzhen, and Stephan Mandt. “A Deep Generative Model for Disentangled Representations of Sequential Data.” arXiv preprint arXiv:1803.02991 (2018).
- Hongsuk Yi, Eunsoo Park and Seungil Kim (, , and .) “Deep Reinforcement Learning for Autonomous Vehicle Driving” (“ .”) 2017 Korea Software Engineering Conference ((2017): 784-786.)
- Fraccaro, Marco, Simon Kamronn, Ulrich Paquet, and Ole Winther. “A Disentangled Recognition and Nonlinear Dynamics Model for Unsupervised Learning.” arXiv preprint arXiv:1710.05741 (2017).
- Kister, Ulrike, Konstantin Klamka, Christian Tominski, and Raimund Dachsel. “GraSp: Combining Spatiallyaware Mobile Devices and a Display Wall for Graph Visualization and Interaction.” In Computer Graphics Forum, vol. 36, no. 3, pp. 503-514. 2017.
- Kim, Neil H., Gloria Lee, Nicholas A. Sherer, K. Michael Martini, Nigel Goldenfeld, and Thomas E. Kuhlman. “Real-time transposable element activity in individual live cells.” *Proceedings of the National Academy of Sciences* 113, no. 26 (2016): 7278-7283.
- Baheti, Ashutosh, and Arobinda Gupta. “Non-linear barrier coverage using mobile wireless sensors.” In Computers and Communications (ISCC), 2017 IEEE Symposium on, pp. 804-809. IEEE, 2017.
- Espeso, David R., Esteban Martínez-García, Victor De Lorenzo, and Ángel Goñi-Moreno. “Physical forces shape group identity of swimming *Pseudomonas putida* cells.” *Frontiers in Microbiology* 7 (2016).
- Goni-Moreno, Angel, and Martyn Amos. “DiSCUS: A Simulation Platform for Conjugation Computing.” In International Conference on Unconventional Computation and Natural Computation, pp. 181-191. Springer International Publishing, 2015.
- Amos, Martyn, et al. “Bacterial computing with engineered populations.” *Phil. Trans. R. Soc. A* 373.2046 (2015): 20140218.
- Crane, Beth, and Stephen Sherratt. “rUNSWift 2D Simulator; Behavioural Simulation Integrated with the rUNSWift Architecture.” *UNSW School of Computer Science and Engineering* (2013).
- Miller, Chreston Allen. “Structural model discovery in temporal event data streams.” Diss. Virginia Polytechnic Institute and State University, 2013.
- Pumar García, César. “Simulación de evolución dirigida de bacteriófagos en poblaciones de bacterias en 2D.” (2013).
- Simoes, Manuel, and Caroline GL Cao. “Leonardo: a first step towards an interactive decision aid for port-placement in robotic surgery.” *Systems, Man, and Cybernetics (SMC), 2013 IEEE International Conference on.* IEEE, 2013.
- Goni-Moreno, Angel, and Martyn Amos. “Discrete modelling of bacterial conjugation dynamics.” *arXiv preprint arXiv:1211.1146* (2012).
- Matthews, Elizabeth A. “ATLAS CHRONICLE: A STORY-DRIVEN SYSTEM TO CREATE STORY-DRIVEN MAPS.” Diss. Clemson University, 2012.

- Matthews, Elizabeth, and Brian Malloy. “Procedural generation of story-driven maps.” *Computer Games (CGAMES), 2011 16th International Conference on*. IEEE, 2011.
- Miller, Chreston, and Francis Quek. “Toward multimodal situated analysis.” *Proceedings of the 13th international conference on multimodal interfaces*. ACM, 2011.
- Verdie, Yannick. “Surface gesture & object tracking on tabletop devices.” Diss. Virginia Polytechnic Institute and State University, 2010.
- Agrawal, Vivek, and Ryan Kerwin. “Dynamic Robot Path Planning Among Crowds in Emergency Situations.”

List last updated 2018-09-09. If something is missing or wrong, please contact me!

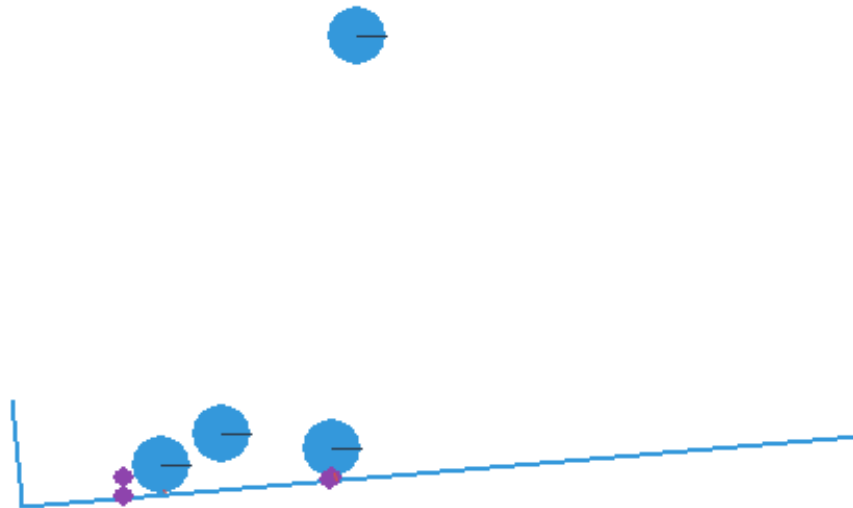
9.7 Tutorials

Pymunk has one tutorial that show a simple simulation from start to end.

After reading it make sure to also check out the [Examples](#) as most of them are easy to follow and showcase many of the things you can do with pymunk.

9.7.1 Slide and Pin Joint Demo Step by Step

This is a step by step tutorial explaining the demo `slide_and_pinjoint.py` included in pymunk. You will find a screenshot of it in the list of [examples](#). It is probably a good idea to have the file near by if I miss something in the tutorial or something is unclear.



9.7.1.1 Before we start

For this tutorial you will need:

- Python (of course)
- Pygame (found at www.pygame.org)
- Pymunk

Pygame is required for this tutorial and some of the included demos, but it is not required to run just pymunk. Pymunk should work just fine with other similar libraries as well, for example you could easily translate this tutorial to use Pyglet instead.

Pymunk is built on top of the 2d physics library Chipmunk. Chipmunk itself is written in C meaning Pymunk need

to call into the c code. The Cffi library helps with this, however if you are on a platform that I haven't been able to compile it on you might have to do it yourself. The good news is that it is very easy to do, in fact if you got Pymunk by Pip install its arelady done!

When you have pymunk installed, try to import it from the python prompt to make sure it works and can be imported:

```
>>> import pymunk
```

If you get an error message it might be because pymunk could not find the chipmunk library that it depends on. If you install pymunk with pip or setup.py install everything should already be correct, but if you got the source and want to use it as a stand alone folder you need to build it inplace:

```
> python setup.py build_ext --inplace
```

More information on installation can be found here: [Installation](#)

If it doesnt work or you have some kind of problem, feel free to write a post in the chipmunk forum, contact me directly or add your problem to the issue tracker: [Contact & Support](#)

9.7.1.2 An empty simulation

Ok, lets start. Chipmunk (and therefore pymunk) has a couple of central concepts, which is explained pretty good in this citation from the Chipmunk docs:

Rigid bodies A rigid body holds the physical properties of an object. (mass, position, rotation, velocity, etc.) It does not have a shape by itself. If you've done physics with particles before, rigid bodies differ mostly in that they are able to rotate.

Collision shapes By attaching shapes to bodies, you can define the a body's shape. You can attach many shapes to a single body to define a complex shape, or none if it doesn't require a shape.

Constraints/joints You can attach joints between two bodies to constrain their behavior.

Spaces Spaces are the basic simulation unit in Chipmunk. You add bodies, shapes and joints to a space, and then update the space as a whole.

The documentation for Chipmunk can be found here: <http://chipmunk-physics.net/release/ChipmunkLatest-Docs/> It is for the c-library but is a good complement to the Pymunk documentation as the concepts are the same, just that Pymunk is more pythonic to use.

The API documentation for Pymunk can be found here: [API Reference](#).

Anyway, we are now ready to write some code:

```
import sys
import pygame
from pygame.locals import *
import pymunk #1

def main():
    pygame.init()
    screen = pygame.display.set_mode((600, 600))
    pygame.display.set_caption("Joints. Just wait and the L will tip over")
    clock = pygame.time.Clock()

    space = pymunk.Space() #2
    space.gravity = (0.0, -900.0)

    while True:
```

(continues on next page)

(continued from previous page)

```
for event in pygame.event.get():
    if event.type == QUIT:
        sys.exit(0)
    elif event.type == KEYDOWN and event.key == K_ESCAPE:
        sys.exit(0)

screen.fill((255,255,255))

space.step(1/50.0) #3

pygame.display.flip()
clock.tick(50)

if __name__ == '__main__':
    sys.exit(main())
```

The code will display a blank window, and will run a physics simulation of an empty space.

1. We need to import pymunk in order to use it...
2. We then create a space and set its gravity to something good. Remember that what is important is what looks good on screen, not what the real world value is. -900 will make a good looking simulation, but feel free to experiment when you have the full code ready.
3. In our game loop we call the step() function on our space. The step function steps the simulation one step forward in time.

Note: It is best to keep the step size constant and not adjust it depending on the framerate. The physic simulation will work much better with a constant step size.

9.7.1.3 Falling balls

The easiest shape to handle (and draw) is the circle. Therefore our next step is to make a ball spawn once in while. In many of the example demos all code is in one big pile in the main() function as they are so small and easy, but I will extract some methods in this tutorial to make it more easy to follow. First, a function to add a ball to a space:

```
def add_ball(space):
    mass = 1
    radius = 14
    moment = pymunk.moment_for_circle(mass, 0, radius) # 1
    body = pymunk.Body(mass, moment) # 2
    x = random.randint(120, 380)
    body.position = x, 550 # 3
    shape = pymunk.Circle(body, radius) # 4
    space.add(body, shape) # 5
    return shape
```

1. All bodies must have their moment of inertia set. If our object is a normal ball we can use the predefined function moment_for_circle to calculate it given its mass and radius. However, you could also select a value by experimenting with what looks good for your simulation.
2. After we have the inertia we can create the body of the ball.
3. And we set its position
4. And in order for it to collide with things, it needs to have one (or many) collision shape(s).

5. Finally we add the body and shape to the space to include it in our simulation.

Now that we can create balls we want to display them. Either we can use the built in `pymunk_util` package to draw the whole space directly, or we can do it manually. The debug drawing functions included with Pymunk are good for putting something together easy and quickly, while a polished game for example most probably will want to make its own drawing code.

If we want to draw manually, our draw function could look something like this:

```
def draw_ball(screen, ball):
    p = int(ball.body.position.x), 600-int(ball.body.position.y)
    pygame.draw.circle(screen, (0,0,255), p, int(ball.radius), 2)
```

And then called in this way (given we collected all the ball shapes in a list called `balls`):

```
for ball in balls:
    draw_ball(screen, ball)
```

However, as we use `pygame` in this example we can instead use the `debug_draw` method already included in Pymunk to simplify a bit. In that case we first have to create a `DrawOptions` object with the options (mainly what surface to draw on):

```
draw_options = pymunk.pygame_util.DrawOptions(screen)
```

And after that when we want to draw all our shapes we would just do it in this way:

```
space.debug_draw(draw_options)
```

Most of the examples included with Pymunk uses this way of drawing.

With the `add_ball` function and the `debug_draw` call and a little code to spawn balls you should see a couple of balls falling. Yay!

```
import sys, random
import pygame
from pygame.locals import *
import pymunk

#def add_ball(space):

def main():
    pygame.init()
    screen = pygame.display.set_mode((600, 600))
    pygame.display.set_caption("Joints. Just wait and the L will tip over")
    clock = pygame.time.Clock()

    space = pymunk.Space()
    space.gravity = (0.0, -900.0)

    balls = []
    draw_options = pymunk.pygame_util.DrawOptions(screen)

    ticks_to_next_ball = 10
    while True:
        for event in pygame.event.get():
            if event.type == QUIT:
                sys.exit(0)
```

(continues on next page)

(continued from previous page)

```

        elif event.type == KEYDOWN and event.key == K_ESCAPE:
            sys.exit(0)

        ticks_to_next_ball -= 1
        if ticks_to_next_ball <= 0:
            ticks_to_next_ball = 25
            ball_shape = add_ball(space)
            balls.append(ball_shape)

        space.step(1/50.0)

        screen.fill((255,255,255))
        space.debug_draw(draw_options)

        pygame.display.flip()
        clock.tick(50)

if __name__ == '__main__':
    main()

```

9.7.1.4 A static L

Falling balls are quite boring. We don't see any physics simulation except basic gravity, and everyone can do gravity without help from a physics library. So lets add something the balls can land on, two static lines forming an L. As with the balls we start with a function to add an L to the space:

```

def add_static_L(space):
    body = pymunk.Body(body_type = pymunk.Body.STATIC) # 1
    body.position = (300, 300)
    l1 = pymunk.Segment(body, (-150, 0), (255, 0), 5) # 2
    l2 = pymunk.Segment(body, (-150, 0), (-150, 50), 5)

    space.add(l1, l2) # 3
    return l1, l2

```

1. We create a “static” body. The important step is to never add it to the space like the dynamic ball bodies. Note how static bodies are created by setting the `body_type` of the body.
2. A line shaped shape is created here.
3. Again, we only add the segments, not the body to the space.

Since we use `Space.debug_draw` to draw the space we dont need to do any special draw code for the Segments, but I still include a possible draw function here just to show what it could look like:

```

def draw_lines(screen, lines):
    for line in lines:
        body = line.body
        pv1 = body.position + line.a.rotated(body.angle) # 1
        pv2 = body.position + line.b.rotated(body.angle)
        p1 = to_pygame(pv1) # 2
        p2 = to_pygame(pv2)
        pygame.draw.lines(screen, THECOLORS["lightgray"], False, [p1,p2])

```

1. In order to get the position with the line rotation we use this calculation. `line.a` is the first endpoint of the line, `line.b` the second. At the moment the lines are static, and not rotated so we don't really have to do this extra

calculation, but we will soon make them move and rotate.

2. This is a little function to convert coordinates from pymunk to pygame world. Now that we have it we can use it in the `draw_ball()` function as well. We want to flip the y coordinate (`-p.y`), and then offset it with the screen height (`+600`). It looks like this:

```
def to_pygame(p):
    """Small hack to convert pymunk to pygame coordinates"""
    return int(p.x), int(-p.y+600)
```

With the full code we should something like the below, and now we should see an inverted L shape in the middle will balls spawning and hitting the shape.

```
import sys, random
import pygame
from pygame.locals import *
import pymunk
import math

#def to_pygame(p):
#def add_ball(space):
#def add_static_l(space):

def main():
    pygame.init()
    screen = pygame.display.set_mode((600, 600))
    pygame.display.set_caption("Joints. Just wait and the L will tip over")
    clock = pygame.time.Clock()

    space = pymunk.Space()
    space.gravity = (0.0, -900.0)

    lines = add_static_L(space)
    balls = []
    draw_options = pymunk.pygame_util.DrawOptions(screen)

    ticks_to_next_ball = 10
    while True:
        for event in pygame.event.get():
            if event.type == QUIT:
                sys.exit(0)
            elif event.type == KEYDOWN and event.key == K_ESCAPE:
                sys.exit(0)

        ticks_to_next_ball -= 1
        if ticks_to_next_ball <= 0:
            ticks_to_next_ball = 25
            ball_shape = add_ball(space)
            balls.append(ball_shape)

        space.step(1/50.0)

        screen.fill((255,255,255))
        space.debug_draw(draw_options)

        pygame.display.flip()
        clock.tick(50)
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    (main())
```

9.7.1.5 Joints (1)

A static L shape is pretty boring. So lets make it a bit more exciting by adding two joints, one that it can rotate around, and one that prevents it from rotating too much. In this part we only add the rotation joint, and in the next we constrain it. As our static L shape won't be static anymore we also rename the function to add_L().

```
def add_L(space):
    rotation_center_body = pymunk.Body(body_type = pymunk.Body.STATIC) # 1
    rotation_center_body.position = (300, 300)

    body = pymunk.Body(10, 10000) # 2
    body.position = (300, 300)
    l1 = pymunk.Segment(body, (-150, 0), (255.0, 0.0), 5.0)
    l2 = pymunk.Segment(body, (-150.0, 0), (-150.0, 50.0), 5.0)

    rotation_center_joint = pymunk.PinJoint(body, rotation_center_body, (0,0), (0,0))
    ↪ # 3

    space.add(l1, l2, body, rotation_center_joint)
    return l1,l2
```

1. This is the rotation center body. Its only purpose is to act as a static point in the joint so the line can rotate around it. As you see we never add any shapes to it.
2. The L shape will now be moving in the world, and therefor it can no longer be a static body. I have precalculated the inertia to 10000. (ok, I just took a number that worked, the important thing is that it looks good on screen!).
3. A pin joint allow two objects to pivot about a single point. In our case one of the objects will be stuck to the world.

9.7.1.6 Joints (2)

In the previous part we added a pin joint, and now its time to constrain the rotating L shape to create a more interesting simulation. In order to do this we modify the add_L() function:

```
def add_L(space):
    rotation_center_body = pymunk.Body(body_type = pymunk.Body.STATIC)
    rotation_center_body.position = (300,300)

    rotation_limit_body = pymunk.Body(body_type = pymunk.Body.STATIC) # 1
    rotation_limit_body.position = (200,300)

    body = pymunk.Body(10, 10000)
    body.position = (300,300)
    l1 = pymunk.Segment(body, (-150, 0), (255.0, 0.0), 5.0)
    l2 = pymunk.Segment(body, (-150.0, 0), (-150.0, 50.0), 5.0)

    rotation_center_joint = pymunk.PinJoint(body, rotation_center_body, (0,0), (0,0))
    joint_limit = 25
    rotation_limit_joint = pymunk.SlideJoint(body, rotation_limit_body, (-100,0), (0,
    ↪ 0), 0, joint_limit) # 2
```

(continues on next page)

(continued from previous page)

```
space.add(l1, l2, body, rotation_center_joint, rotation_limit_joint)
return l1,l2
```

1. We add a body..
2. Create a slide joint. It behaves like pin joints but have a minimum and maximum distance. The two bodies can slide between the min and max, and in our case one of the bodies is static meaning only the body attached with the shapes will move.

9.7.1.7 Ending

You might notice that we never delete balls. This will make the simulation require more and more memory and use more and more cpu, and this is of course not what we want. So in the final step we add some code to remove balls from the simulation when they are below the screen.

```
balls_to_remove = []
for ball in balls:
    if ball.body.position.y < 0: # 1
        balls_to_remove.append(ball) # 2

for ball in balls_to_remove:
    space.remove(ball, ball.body) # 3
    balls.remove(ball) # 4
```

1. Loop the balls and check if the body.position is less than 0.
2. If that is the case, we add it to our list of balls to remove.
3. To remove an object from the space, we need to remove its shape and its body.
4. And then we remove it from our list of balls.

And now, done! You should have an inverted L shape in the middle of the screen being filled with balls, tipping over releasing them, tipping back and start over. You can check `slide_and_pinjoint.py` included in pymunk, but it doesn't follow this tutorial exactly as I factored out a couple of blocks to functions to make it easier to follow in tutorial form.

If anything is unclear, not working feel free to raise an issue on github. If you have an idea for another tutorial you want to read, or some example code you want to see included in pymunk, please write it somewhere (like in the chipmunk forum)

The full code for this tutorial is:

```
import sys, random
import pygame
from pygame.locals import *
import pymunk
import pymunk.pygame_util

def add_ball(space):
    """Add a ball to the given space at a random position"""
    mass = 1
    radius = 14
    inertia = pymunk.moment_for_circle(mass, 0, radius, (0,0))
    body = pymunk.Body(mass, inertia)
    x = random.randint(120,380)
    body.position = x, 550
```

(continues on next page)

(continued from previous page)

```

    shape = pymunk.Circle(body, radius, (0,0))
    space.add(body, shape)
    return shape

def add_L(space):
    """Add a inverted L shape with two joints"""
    rotation_center_body = pymunk.Body(body_type = pymunk.Body.STATIC)
    rotation_center_body.position = (300,300)

    rotation_limit_body = pymunk.Body(body_type = pymunk.Body.STATIC)
    rotation_limit_body.position = (200,300)

    body = pymunk.Body(10, 10000)
    body.position = (300,300)
    l1 = pymunk.Segment(body, (-150, 0), (255.0, 0.0), 5.0)
    l2 = pymunk.Segment(body, (-150.0, 0), (-150.0, 50.0), 5.0)

    rotation_center_joint = pymunk.PinJoint(body, rotation_center_body, (0,0), (0,0))
    joint_limit = 25
    rotation_limit_joint = pymunk.SlideJoint(body, rotation_limit_body, (-100,0), (0,
→0), 0, joint_limit)

    space.add(l1, l2, body, rotation_center_joint, rotation_limit_joint)
    return l1,l2

def main():
    pygame.init()
    screen = pygame.display.set_mode((600, 600))
    pygame.display.set_caption("Joints. Just wait and the L will tip over")
    clock = pygame.time.Clock()

    space = pymunk.Space()
    space.gravity = (0.0, -900.0)

    lines = add_L(space)
    balls = []
    draw_options = pymunk.pygame_util.DrawOptions(screen)

    ticks_to_next_ball = 10
    while True:
        for event in pygame.event.get():
            if event.type == QUIT:
                sys.exit(0)
            elif event.type == KEYDOWN and event.key == K_ESCAPE:
                sys.exit(0)

        ticks_to_next_ball -= 1
        if ticks_to_next_ball <= 0:
            ticks_to_next_ball = 25
            ball_shape = add_ball(space)
            balls.append(ball_shape)

        screen.fill((255,255,255))

        balls_to_remove = []
        for ball in balls:
            if ball.body.position.y < 150:

```

(continues on next page)

(continued from previous page)

```

        balls_to_remove.append(ball)

    for ball in balls_to_remove:
        space.remove(ball, ball.body)
        balls.remove(ball)

    space.debug_draw(draw_options)

    space.step(1/50.0)

    pygame.display.flip()
    clock.tick(50)

if __name__ == '__main__':
    main()

```

9.7.2 External Tutorials

If you have made a tutorial that is using Pymunk in any way and want it mentioned here please send me a link and I will happily add it. I also accept full tutorials to include directly here if you prefer, as long as they are of reasonable quality and style. Check the source to see how the existing ones are built.

9.7.2.1 Pymunk physics in Pyglet

Created by Attila Toth. Retrieved 2018-02-24

Youtube user Attila Toth has created a series of youtube videos that gives a good introduction of Pymunk. The videos covers among other things the 3 types of Bodies, the different Shapes and how to use sprite with Pyglet together with Pymunk.

9.8 Benchmarks

To get a grip of the actual performance of Pymunk this page contains a number of benchmarks.

The full code of all benchmarks are available under the [benchmarks](#) folder.

9.8.1 Micro benchmarks

In order to measure the overhead created by Pymunk in the most common cases I have created two micro benchmarks. They should show the speed of the actual wrapping code, which can tell how big overhead Pymunk creates, and how big difference different wrapping methods does.

The most common thing a typical program using Pymunk does is to read out the position and angle from a Pymunk object. Usually this is done each frame for every object in the simulation, so this is a important factor in how fast something will be.

Given this our first test is:

```
t += b.position.x + b.position.y + b.angle
```

(see *pymunk-get.py*)

Running it is simple, for example like this for pymunk 4.0:

```
> python -m pip install pymunk==4.0
> python pymunk-get.py
```

The second test we do is based on the second heavy thing we can do, and that is using a callback, for example as a collision handler or a position function:

```
def f(b, dt):
    b.position += (1, 0)

s.step(0.01)
```

(see *pymunk-callback.py*)

9.8.1.1 Results:

Tests run on a HP G1 1040 laptop with a Intel i7-4600U. Laptop runs Windows, and the tests were run inside a VirtualBox VM running 64bit Debian. The CPython tests uses CPython from Conda, while the Pypy tests used a manually downloaded Pypy. CPython 2.7 is using Cffi 1.7, the other tests Cffi 1.8.

Remember that these results doesn't tell you how you game/application will perform, they can more be seen as a help to identify performance issues and know differences between Pythons.

Pymunk-Get:

	CPython 2.7.12	CPython 3.5.2	Pypy 5.4.1
Pymunk 5.1	2.1s	2.2s	0.36s
Pymunk 5.0	4.3s	4.5s	0.37s
Pymunk 4.0	1.0s	0.9s	0.52s

Pymunk-Callback:

	CPython 2.7.12	CPython 3.5.2	Pypy 5.4.1
Pymunk 5.1	5.7s	6.8s	1.1s
Pymunk 5.0	6.5s	7.3s	1.0s
Pymunk 4.0	5.1s	6.5s	4.5s

What we can see from these results is that you should use Pypy if you have the possibility since that is much faster than regular CPython. We can also see that moving from Ctypes to Cffi between Pymunk 4 and 5 had a negative impact in CPython, but positive impact on Pypy, and Pymunk 5 together with Pypy is with a big margin the fastest option.

The speed increase between 5.0 and 5.1 happened because the Vec2d class and how its handled internally in Pymunk was changed to improve performance.

9.8.2 Compared to Other Physics Libraries

9.8.2.1 Cymunk

Cymunk is an alternative wrapper around Chipmunk. In contrast to Pymunk it uses Cython for wrapping (Pymunk uses CFFI) which gives it a different performance profile. However, since both are built around Chipmunk the overall speed will be very similar, only when information passes from/to Chipmunk will there be a difference. This is exactly the kind of overhead that the micro benchmarks are made to measure.

Cymunk is not as feature complete as Pymunk, so in order to compare with Pymunk we have to make some adjustments. A major difference is that it does not implement the *position_func* function, so instead we do an alternative callback test using the collision handler:

```
h = s.add_default_collision_handler()
def f(arb):
    return False
h.pre_solve = f
s.step(0.01)
```

(see *pymunk-collision-callback.py* and *cymunk-collision-callback.py*)

Results

Tests run on a HP G1 1040 laptop with a Intel i7-4600U. Laptop runs Windows, and the tests were run inside a VirtualBox VM running 64bit Debian. The CPython tests uses CPython from Conda, while the Pypy tests used a manually downloaded Pypy. Cffi version 1.10.0 and Cython 0.25.2.

Since Cymunk doesn't have a proper release I used the latest master from its Github repository, hash 24845cc retrieved on 2017-09-16.

Get:

	CPython 3.5.3	Pypy 5.8
Pymunk 5.3	2.14s	0.33s
Cymunk 20170916	0.41s	(10.0s)

Collision-Callback:

	CPython 3.5.3	Pypy 5.8
Pymunk 5.3	3.71s	0.58s
Pymunk 20170916	0.95s	(7.01s)

(Cymunk results on Pypy within parentheses since Cython is well known to be slow on Pypy)

What we can see from these results is that Cymunk on CPython is much faster than Pymunk on CPython, but Pymunk takes the overall victory when we include Pypy.

Something we did not take into account is that you can trade convenience for performance and use Cython in the application code as well to speed things up. I think this is the approach used in KivEnt which is the primary user of Cymunk. However, that requires a much more complicated setup when you develop your application because of the compiler requirements and code changes.

9.9 Advanced

In this section different “Advanced” topics are covered, things you normally don't need to worry about when you use Pymunk but might be of interest if you want a better understanding of Pymunk for example to extend it.

First off, Pymunk is a pythonic wrapper around the C-library Chipmunk.

To wrap Chipmunk Pymunk uses CFFI. On top of the CFFI wrapping is a handmade pythonic layer to make it nice to use from Python code.

9.9.1 Why CFFI?

This is a straight copy from the github issue tracking the CFFI upgrade. <https://github.com/viblo/pymunk/issues/99>

CFFI have a number of advantages but also a downsides.

Advantages (compared to ctypes):

- Its an active project. The developers and users are active, there are new releases being made and its possible to ask and get answers within a day on the CFFI mailing list.
- Its said to be the way forward for pypy, with promise of better performance compares to ctypes.
- A little easier than ctypes to wrap things since you can just copy-paste the c headers.

Disadvantages (compared to ctypes):

- ctypes is part of the CPython standard library, CFFI is not. That means that it will be more difficult to install Pymunk if it uses CFFI, since a copy-paste install is no longer possible in an easy way.

For me I see the 1st advantage as the main point. I have had great difficulties with strange segfaults with 64bit pythons on windows, and also sometimes on 32bit python, and support for 64bit python on both windows and linux is something I really want. Hopefully those problems will be easier to handle with CFFI since it has an active community.

Then comes the 3rd advantage, that its a bit easier to wrap the c code. For ctypes I have a automatic wrapping script that does most of the low level wrapping, but its not supported, very difficult to set up (I only managed inside a VM with linux) and quite annoying. CFFI would be a clear improvement.

For the disadvantage of ctypes I think it will be acceptable, even if not ideal. Many python packages have to be installed in some way (like pygame), and nowadays with pip its very easy to do. So I hope that it will be ok.

See the next section on why ctypes was used initially.

9.9.2 Why ctypes? (OBSOLETE)

The reasons for ctypes instead of [your favorite wrapping solution] can be summarized as

- You only need to write pure python code when wrapping. This is good for several reasons. I can not really code in c. Sure, I can read it and write easy things, but Im not a good c coder. What I do know quite well is python. I imagine that the same is true for most people using pymunk, after all its a python library. :) Hopefully this means that users of pymunk can look at how stuff is actually done very easily, and for example add a missing chipmunk method/property on their own in their own code without much problem, and without being required to compile/build anything.
- ctypes is included in the standard library. Anyone with python has it already, no dependencies on 3rd party libraries, and some guarantee that it will stick around for a long time.
- The only thing required to run pymunk is python and a c compiler (in those cases a prebuilt version of chipmunk is not included). This should maximize the multiplatformness of pymunk, only thing that would even better would be a pure python library (which might be a bad idea for other reasons, mainly speed).

- Not much magic going on. Working with ctypes is quite straight forward. Sure, pymunk uses a generator which is a bit of a pain, but at least its possible to sidestep it if required, which Ive done in some cases. Ive also got a share amount of problems when stuff didnt work as expected, but I imagine it would have been even worse with other solutions. At least its only the c library and python, and not some 3rd party in between.
- Non api-breaking fixes in chipmunk doesnt affect pymunk. If a bugfix, some optimization or whatever is done in chipmunk that doesnt affect the API, then its enough with a recompile of chipmunk with the new code to benefit from the fix. Easy for everyone.
- Ctypes can run on other python implementations than cpython. Right now pypy feels the most promising and it is be able to run ctypes just fine.

As I see it, the main benefit another solution could give would be speed. However, there are a couple of arguments why I dont find this as important as the benefits of ctypes

- You are writing your game in python in the first place, if you really required top performance than maybe rewrite the whole thing in c would be better anyway? Or make a optimized binding to chipmunk.

For example, if you really need excellent performance then one possible optimization would be to write the drawing code in c as well, and have that interact with chipmunk directly. That way it can be made more performant than any generic wrapping solution as it would skip the whole layer.

- The bottleneck in a full game/application is somewhere else than in the physics wrapping in many cases. If your game has AI, logic and so on in python, then the wrapper overhead added by ctypes is not so bad in comparison.
- Pypy. ctypes on pypy has the potential to be very quick. However, right now with pypy-1.9 the speed of pymunk is actually a bit slower on pypy than on cpython. Hopefully this will improve in the future.

Note that pymunk has been around since late 2007 which means not all wrapping options that exist today did exist or was not stable/complete enough for use by pymunk in the beginning. There are more options available today, and using ctypes is not set in stone. If a better alternative comes around then pymunk might switch given the improvements are big enough.

9.9.3 Code Layout

Most of Pymunk should be quite straight forward.

Except for the documented API Pymunk has a couple of interesting parts. Low level bindings to Chipmunk, a custom library load function, a custom documentation generation extension and a customized setup.py file to allow compilation of Chipmunk.

The low level chipmunk bindings are located in the two files `_chipmunk_cffi.py` and `_chipmunk_cffi_abi.py`. In order to locate and load the compiled chipmunk library file pymunk uses a custom `load_library` function in `_libload.py`

docs/src/ext/autoexample.py A Sphinx extension that scans a directory and extracts the toplevel docstring. Used to autogenerate the examples documentation.

pymunk/_chipmunk_cffi.py This file only contains a call to `_chipmunk_cffi_abi.py`, and exists mostly as a wrapper to be able to switch between abi and api mode of Cffi. This is currently not in use in the relased code, but is used during experimentation.

pymunk/_chipmunk_cffi_abi.py This file contains the pure Cffi wrapping definitons. Bascially a giant string created by copy-paster from the relevant header files of Chipmunk.

pymunk/_libload.py This file contains the custom Cffi library load function that is used by the rest of pymunk to load the Chipmunk library file.

setup.py Except for the standard setup stuff this file also contain the custom build commands to build Chipmunk from source, using a `build_ext` extension.

tests/* Collection of (unit) tests. Does not cover all cases, but most core things are there. The tests require a working chipmunk library file.

tools/* Collection of helper scripts that can be used to various development tasks such as generating documentation.

9.9.4 Tests

There are a number of unit tests included in the tests folder. Not exactly all the code is tested, but most of it (at the time of writing its about 85% of the core parts).

There is a helper script in the tools folder to easily run the tests:

```
> cd tools
> python run_tests.py
```

9.9.5 Working with non-wrapped parts of Chipmunk

In case you need to use something that exist in Chipmunk but currently is not included in pymunk the easiest method is to add it manually.

For example, lets assume that the `is_sleeping` property of a body was not wrapped by pymunk. The Chipmunk method to get this property is named `cpBodyIsSleeping`.

First we need to check if its included in the cdef definition in `_chipmunk_cffi.abi.py`. If its not just add it.

```
cpBool cpBodyIsSleeping(const cpBody *body);
```

Then to make it easy to use we want to create a python method that looks nice:

```
def is_sleeping(body) :
    return cp.cpBodyIsSleeping(body._body)
```

Now we are ready with the mapping and ready to use our new method.

9.9.6 Weak References and `__del__` Methods

Internally Pymunk allocates structs from Chipmunk (the c library). For example a `Body` struct is created from inside the constructor method when a `pymunk.Body` is created. Because of this several Pymunk objects uses a `__del__()` method that cleans up the underlying c struct when the object is deleted.

Use of a `__del__()` method prevents the normal CPython GC (garbage collection) from handling cyclic references since it wont know in which order to run the `__del__()` methods. Some Pymunk objects naturally keeps cyclic references to each other to make them easier to use. One such example is the body and shape object. A shape is attached to a body, and a body has a set of all shapes that has been attached. To make it easier for the user of the library these cyclic references have been broken up so that the reference in one direction is weak and dont affect GC. Usually the user do not need to worry about this, but in the cases a reference is weak it is marked in the API documentation of the specific method.

9.10 License

Copyright (c) 2007-2018 Victor Blomqvist

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use,

copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER 10

Indices and tables

- `genindex`
- `modindex`
- `search`

a

`pymunk.autogeometry`, 30

c

`pymunk.constraint`, 33

m

`pymunk.matplotlib_util`, 46

p

`pymunk`, 52

`pymunk.pygame_util`, 48

`pymunk.pyglet_util`, 50

`pymunkoptions`, 52

v

`pymunk.vec2d`, 43

Symbols

__init__ (pymunk.PointQueryInfo attribute), 82
 __init__ (pymunk.SegmentQueryInfo attribute), 75
 __init__ (pymunk.ShapeFilter attribute), 80
 __init__ (pymunk.ShapeQueryInfo attribute), 82
 __init__ (pymunk.Transform attribute), 81
 __init__() (pymunk.Arbitrator method), 76
 __init__() (pymunk.BB method), 78
 __init__() (pymunk.Body method), 59
 __init__() (pymunk.Circle method), 66
 __init__() (pymunk.CollisionHandler method), 77
 __init__() (pymunk.ContactPoint method), 76
 __init__() (pymunk.ContactPointSet method), 76
 __init__() (pymunk.Poly method), 68
 __init__() (pymunk.Segment method), 72
 __init__() (pymunk.Shape method), 64
 __init__() (pymunk.Space method), 53
 __init__() (pymunk.SpaceDebugDrawOptions method), 83
 __init__() (pymunk.autogeometry.PolylineSet method), 31
 __init__() (pymunk.constraint.Constraint method), 33
 __init__() (pymunk.constraint.DampedRotarySpring method), 39
 __init__() (pymunk.constraint.DampedSpring method), 38
 __init__() (pymunk.constraint.GearJoint method), 42
 __init__() (pymunk.constraint.GrooveJoint method), 37
 __init__() (pymunk.constraint.PinJoint method), 34
 __init__() (pymunk.constraint.PivotJoint method), 36
 __init__() (pymunk.constraint.RatchetJoint method), 41
 __init__() (pymunk.constraint.RotaryLimitJoint method), 40
 __init__() (pymunk.constraint.SimpleMotor method), 43
 __init__() (pymunk.constraint.SlideJoint method), 35
 __init__() (pymunk.matplotlib_util.DrawOptions method), 46
 __init__() (pymunk.pygame_util.DrawOptions method), 48

__init__() (pymunk.pyglet_util.DrawOptions method), 50
 __init__() (pymunk.vec2d.Vec2d method), 44

A

a (pymunk.constraint.Constraint attribute), 33
 a (pymunk.constraint.DampedRotarySpring attribute), 39
 a (pymunk.constraint.DampedSpring attribute), 38
 a (pymunk.constraint.GearJoint attribute), 42
 a (pymunk.constraint.GrooveJoint attribute), 37
 a (pymunk.constraint.PinJoint attribute), 34
 a (pymunk.constraint.PivotJoint attribute), 36
 a (pymunk.constraint.RatchetJoint attribute), 41
 a (pymunk.constraint.RotaryLimitJoint attribute), 40
 a (pymunk.constraint.SimpleMotor attribute), 43
 a (pymunk.constraint.SlideJoint attribute), 35
 a (pymunk.Segment attribute), 72
 a (pymunk.Transform attribute), 81
 activate() (pymunk.Body method), 61
 activate_bodies() (pymunk.constraint.Constraint method), 33
 activate_bodies() (pymunk.constraint.DampedRotarySpring method), 39
 activate_bodies() (pymunk.constraint.DampedSpring method), 38
 activate_bodies() (pymunk.constraint.GearJoint method), 42
 activate_bodies() (pymunk.constraint.GrooveJoint method), 37
 activate_bodies() (pymunk.constraint.PinJoint method), 34
 activate_bodies() (pymunk.constraint.PivotJoint method), 36
 activate_bodies() (pymunk.constraint.RatchetJoint method), 41
 activate_bodies() (pymunk.constraint.RotaryLimitJoint method), 40
 activate_bodies() (pymunk.constraint.SimpleMotor method), 43
 activate_bodies() (pymunk.constraint.SlideJoint method), 35

[add\(\)](#) (pymunk.Space method), 54
[add_collision_handler\(\)](#) (pymunk.Space method), 54
[add_default_collision_handler\(\)](#) (pymunk.Space method), 54
[add_post_step_callback\(\)](#) (pymunk.Space method), 54
[add_wildcard_collision_handler\(\)](#) (pymunk.Space method), 55
[ALL_CATEGORIES](#) (pymunk.ShapeFilter attribute), 80
[ALL_MASKS](#) (pymunk.ShapeFilter attribute), 80
[alpha](#) (pymunk.SegmentQueryInfo attribute), 75
[anchor_a](#) (pymunk.constraint.DampedSpring attribute), 38
[anchor_a](#) (pymunk.constraint.PinJoint attribute), 34
[anchor_a](#) (pymunk.constraint.PivotJoint attribute), 36
[anchor_a](#) (pymunk.constraint.SlideJoint attribute), 35
[anchor_b](#) (pymunk.constraint.DampedSpring attribute), 38
[anchor_b](#) (pymunk.constraint.GrooveJoint attribute), 37
[anchor_b](#) (pymunk.constraint.PinJoint attribute), 34
[anchor_b](#) (pymunk.constraint.PivotJoint attribute), 36
[anchor_b](#) (pymunk.constraint.SlideJoint attribute), 35
[angle](#) (pymunk.Body attribute), 61
[angle](#) (pymunk.constraint.RatchetJoint attribute), 41
[angle](#) (pymunk.vec2d.Vec2d attribute), 44
[angle_degrees](#) (pymunk.vec2d.Vec2d attribute), 44
[angular_velocity](#) (pymunk.Body attribute), 61
[apply_force_at_local_point\(\)](#) (pymunk.Body method), 61
[apply_force_at_world_point\(\)](#) (pymunk.Body method), 61
[apply_impulse_at_local_point\(\)](#) (pymunk.Body method), 61
[apply_impulse_at_world_point\(\)](#) (pymunk.Body method), 61
[Arbiter](#) (class in pymunk), 76
[area](#) (pymunk.Circle attribute), 66
[area](#) (pymunk.Poly attribute), 68
[area](#) (pymunk.Segment attribute), 72
[area](#) (pymunk.Shape attribute), 64
[area\(\)](#) (pymunk.BB method), 78

B

[b](#) (pymunk.constraint.Constraint attribute), 33
[b](#) (pymunk.constraint.DampedRotarySpring attribute), 39
[b](#) (pymunk.constraint.DampedSpring attribute), 38
[b](#) (pymunk.constraint.GearJoint attribute), 42
[b](#) (pymunk.constraint.GrooveJoint attribute), 37
[b](#) (pymunk.constraint.PinJoint attribute), 34
[b](#) (pymunk.constraint.PivotJoint attribute), 36
[b](#) (pymunk.constraint.RatchetJoint attribute), 41
[b](#) (pymunk.constraint.RotaryLimitJoint attribute), 40
[b](#) (pymunk.constraint.SimpleMotor attribute), 43
[b](#) (pymunk.constraint.SlideJoint attribute), 35
[b](#) (pymunk.Segment attribute), 72
[b](#) (pymunk.Transform attribute), 81

[BB](#) (class in pymunk), 78
[bb](#) (pymunk.Circle attribute), 66
[bb](#) (pymunk.Poly attribute), 69
[bb](#) (pymunk.Segment attribute), 72
[bb](#) (pymunk.Shape attribute), 64
[bb_query\(\)](#) (pymunk.Space method), 55
[begin](#) (pymunk.CollisionHandler attribute), 77
[bodies](#) (pymunk.Space attribute), 55
[Body](#) (class in pymunk), 59
[body](#) (pymunk.Circle attribute), 66
[body](#) (pymunk.Poly attribute), 69
[body](#) (pymunk.Segment attribute), 72
[body](#) (pymunk.Shape attribute), 64
[body_type](#) (pymunk.Body attribute), 61
[bottom](#) (pymunk.BB attribute), 78

C

[c](#) (pymunk.Transform attribute), 81
[cache_bb\(\)](#) (pymunk.Circle method), 66
[cache_bb\(\)](#) (pymunk.Poly method), 69
[cache_bb\(\)](#) (pymunk.Segment method), 72
[cache_bb\(\)](#) (pymunk.Shape method), 64
[categories](#) (pymunk.ShapeFilter attribute), 80
[center\(\)](#) (pymunk.BB method), 78
[center_of_gravity](#) (pymunk.Body attribute), 61
[center_of_gravity](#) (pymunk.Circle attribute), 66
[center_of_gravity](#) (pymunk.Poly attribute), 69
[center_of_gravity](#) (pymunk.Segment attribute), 72
[center_of_gravity](#) (pymunk.Shape attribute), 64
[chipmunk_version](#) (in module pymunk), 53
[Circle](#) (class in pymunk), 66
[clamp_vect\(\)](#) (pymunk.BB method), 78
[collect_segment\(\)](#) (pymunk.autogeometry.PolylineSet method), 31
[collide_bodies](#) (pymunk.constraint.Constraint attribute), 33
[collide_bodies](#) (pymunk.constraint.DampedRotarySpring attribute), 39
[collide_bodies](#) (pymunk.constraint.DampedSpring attribute), 38
[collide_bodies](#) (pymunk.constraint.GearJoint attribute), 42
[collide_bodies](#) (pymunk.constraint.GrooveJoint attribute), 37
[collide_bodies](#) (pymunk.constraint.PinJoint attribute), 34
[collide_bodies](#) (pymunk.constraint.PivotJoint attribute), 36
[collide_bodies](#) (pymunk.constraint.RatchetJoint attribute), 41
[collide_bodies](#) (pymunk.constraint.RotaryLimitJoint attribute), 40
[collide_bodies](#) (pymunk.constraint.SimpleMotor attribute), 43

- `collide_bodies` (pymunk.constraint.SlideJoint attribute), 35
 - `collision_bias` (pymunk.Space attribute), 55
 - `collision_persistence` (pymunk.Space attribute), 55
 - `collision_point_color` (pymunk.matplotlib_util.DrawOptions attribute), 46
 - `collision_point_color` (pymunk.pygame_util.DrawOptions attribute), 48
 - `collision_point_color` (pymunk.pyglet_util.DrawOptions attribute), 51
 - `collision_point_color` (pymunk.SpaceDebugDrawOptions attribute), 83
 - `collision_slop` (pymunk.Space attribute), 55
 - `collision_type` (pymunk.Circle attribute), 66
 - `collision_type` (pymunk.Poly attribute), 69
 - `collision_type` (pymunk.Segment attribute), 72
 - `collision_type` (pymunk.Shape attribute), 64
 - `CollisionHandler` (class in pymunk), 77
 - `color_for_shape()` (pymunk.matplotlib_util.DrawOptions method), 47
 - `color_for_shape()` (pymunk.pygame_util.DrawOptions method), 48
 - `color_for_shape()` (pymunk.pyglet_util.DrawOptions method), 51
 - `color_for_shape()` (pymunk.SpaceDebugDrawOptions method), 83
 - `Constraint` (class in pymunk.constraint), 33
 - `constraint_color` (pymunk.matplotlib_util.DrawOptions attribute), 47
 - `constraint_color` (pymunk.pygame_util.DrawOptions attribute), 48
 - `constraint_color` (pymunk.pyglet_util.DrawOptions attribute), 51
 - `constraint_color` (pymunk.SpaceDebugDrawOptions attribute), 83
 - `constraints` (pymunk.Body attribute), 61
 - `constraints` (pymunk.Space attribute), 56
 - `contact_point_set` (pymunk.Arbitrator attribute), 76
 - `contact_point_set` (pymunk.ShapeQueryInfo attribute), 82
 - `ContactPoint` (class in pymunk), 75
 - `ContactPointSet` (class in pymunk), 76
 - `contains()` (pymunk.BB method), 78
 - `contains_vect()` (pymunk.BB method), 78
 - `convert_to_basis()` (pymunk.vec2d.Vec2d method), 44
 - `convex_decomposition()` (in module pymunk.autogeometry), 31
 - `copy()` (pymunk.BB method), 79
 - `copy()` (pymunk.Body method), 61
 - `copy()` (pymunk.Circle method), 66
 - `copy()` (pymunk.constraint.Constraint method), 33
 - `copy()` (pymunk.constraint.DampedRotarySpring method), 39
 - `copy()` (pymunk.constraint.DampedSpring method), 38
 - `copy()` (pymunk.constraint.GearJoint method), 42
 - `copy()` (pymunk.constraint.GrooveJoint method), 37
 - `copy()` (pymunk.constraint.PinJoint method), 34
 - `copy()` (pymunk.constraint.PivotJoint method), 36
 - `copy()` (pymunk.constraint.RatchetJoint method), 41
 - `copy()` (pymunk.constraint.RotaryLimitJoint method), 40
 - `copy()` (pymunk.constraint.SimpleMotor method), 43
 - `copy()` (pymunk.constraint.SlideJoint method), 35
 - `copy()` (pymunk.Poly method), 69
 - `copy()` (pymunk.Segment method), 73
 - `copy()` (pymunk.Shape method), 64
 - `copy()` (pymunk.Space method), 56
 - `count()` (pymunk.autogeometry.PolylineSet method), 32
 - `count()` (pymunk.PointQueryInfo method), 82
 - `count()` (pymunk.SegmentQueryInfo method), 75
 - `count()` (pymunk.ShapeFilter method), 80
 - `count()` (pymunk.ShapeQueryInfo method), 82
 - `count()` (pymunk.Transform method), 81
 - `cpvrotate()` (pymunk.vec2d.Vec2d method), 44
 - `cpvunrotate()` (pymunk.vec2d.Vec2d method), 44
 - `create_box()` (pymunk.Poly static method), 69
 - `create_box_bb()` (pymunk.Poly static method), 69
 - `cross()` (pymunk.vec2d.Vec2d method), 44
 - `current_time_step` (pymunk.Space attribute), 56
- ## D
- `d` (pymunk.Transform attribute), 81
 - `DampedRotarySpring` (class in pymunk.constraint), 39
 - `DampedSpring` (class in pymunk.constraint), 37
 - `damping` (pymunk.constraint.DampedRotarySpring attribute), 39
 - `damping` (pymunk.constraint.DampedSpring attribute), 38
 - `damping` (pymunk.Space attribute), 56
 - `data` (pymunk.CollisionHandler attribute), 78
 - `debug_draw()` (pymunk.Space method), 56
 - `density` (pymunk.Circle attribute), 66
 - `density` (pymunk.Poly attribute), 69
 - `density` (pymunk.Segment attribute), 73
 - `density` (pymunk.Shape attribute), 64
 - `distance` (pymunk.constraint.PinJoint attribute), 34
 - `distance` (pymunk.ContactPoint attribute), 76
 - `distance` (pymunk.PointQueryInfo attribute), 82
 - `dot()` (pymunk.vec2d.Vec2d method), 44
 - `draw_circle()` (pymunk.matplotlib_util.DrawOptions method), 47
 - `draw_circle()` (pymunk.pygame_util.DrawOptions method), 48
 - `draw_circle()` (pymunk.pyglet_util.DrawOptions method), 51

[draw_circle\(\)](#) (pymunk.SpaceDebugDrawOptions method), 83
[DRAW_COLLISION_POINTS](#) (pymunk.matplotlib_util.DrawOptions attribute), 46
[DRAW_COLLISION_POINTS](#) (pymunk.pygame_util.DrawOptions attribute), 48
[DRAW_COLLISION_POINTS](#) (pymunk.pyglet_util.DrawOptions attribute), 50
[DRAW_COLLISION_POINTS](#) (pymunk.SpaceDebugDrawOptions attribute), 82
[DRAW_CONSTRAINTS](#) (pymunk.matplotlib_util.DrawOptions attribute), 46
[DRAW_CONSTRAINTS](#) (pymunk.pygame_util.DrawOptions attribute), 48
[DRAW_CONSTRAINTS](#) (pymunk.pyglet_util.DrawOptions attribute), 50
[DRAW_CONSTRAINTS](#) (pymunk.SpaceDebugDrawOptions attribute), 82
[draw_dot\(\)](#) (pymunk.matplotlib_util.DrawOptions method), 47
[draw_dot\(\)](#) (pymunk.pygame_util.DrawOptions method), 48
[draw_dot\(\)](#) (pymunk.pyglet_util.DrawOptions method), 51
[draw_dot\(\)](#) (pymunk.SpaceDebugDrawOptions method), 83
[draw_fat_segment\(\)](#) (pymunk.matplotlib_util.DrawOptions method), 47
[draw_fat_segment\(\)](#) (pymunk.pygame_util.DrawOptions method), 49
[draw_fat_segment\(\)](#) (pymunk.pyglet_util.DrawOptions method), 51
[draw_fat_segment\(\)](#) (pymunk.SpaceDebugDrawOptions method), 83
[draw_polygon\(\)](#) (pymunk.matplotlib_util.DrawOptions method), 47
[draw_polygon\(\)](#) (pymunk.pygame_util.DrawOptions method), 49
[draw_polygon\(\)](#) (pymunk.pyglet_util.DrawOptions method), 51
[draw_polygon\(\)](#) (pymunk.SpaceDebugDrawOptions method), 83
[draw_segment\(\)](#) (pymunk.matplotlib_util.DrawOptions method), 47
[draw_segment\(\)](#) (pymunk.pygame_util.DrawOptions method), 49
[draw_segment\(\)](#) (pymunk.pyglet_util.DrawOptions method), 51
[draw_segment\(\)](#) (pymunk.SpaceDebugDrawOptions method), 83
[DRAW_SHAPES](#) (pymunk.matplotlib_util.DrawOptions attribute), 46
[DRAW_SHAPES](#) (pymunk.pygame_util.DrawOptions attribute), 48
[DRAW_SHAPES](#) (pymunk.pyglet_util.DrawOptions attribute), 50
[DRAW_SHAPES](#) (pymunk.SpaceDebugDrawOptions attribute), 82
[DrawOptions](#) (class in pymunk.matplotlib_util), 46
[DrawOptions](#) (class in pymunk.pygame_util), 48
[DrawOptions](#) (class in pymunk.pyglet_util), 50
[DYNAMIC](#) (pymunk.Body attribute), 59

E

[each_arbiter\(\)](#) (pymunk.Body method), 61
[elasticity](#) (pymunk.Circle attribute), 66
[elasticity](#) (pymunk.Poly attribute), 70
[elasticity](#) (pymunk.Segment attribute), 73
[elasticity](#) (pymunk.Shape attribute), 64
[error_bias](#) (pymunk.constraint.Constraint attribute), 33
[error_bias](#) (pymunk.constraint.DampedRotarySpring attribute), 39
[error_bias](#) (pymunk.constraint.DampedSpring attribute), 38
[error_bias](#) (pymunk.constraint.GearJoint attribute), 42
[error_bias](#) (pymunk.constraint.GrooveJoint attribute), 37
[error_bias](#) (pymunk.constraint.PinJoint attribute), 34
[error_bias](#) (pymunk.constraint.PivotJoint attribute), 36
[error_bias](#) (pymunk.constraint.RatchetJoint attribute), 41
[error_bias](#) (pymunk.constraint.RotaryLimitJoint attribute), 40
[error_bias](#) (pymunk.constraint.SimpleMotor attribute), 43
[error_bias](#) (pymunk.constraint.SlideJoint attribute), 35
[expand\(\)](#) (pymunk.BB method), 79

F

[filter](#) (pymunk.Circle attribute), 66
[filter](#) (pymunk.Poly attribute), 70
[filter](#) (pymunk.Segment attribute), 73
[filter](#) (pymunk.Shape attribute), 64
[flags](#) (pymunk.matplotlib_util.DrawOptions attribute), 47
[flags](#) (pymunk.pygame_util.DrawOptions attribute), 49
[flags](#) (pymunk.pyglet_util.DrawOptions attribute), 51
[flags](#) (pymunk.SpaceDebugDrawOptions attribute), 83
[force](#) (pymunk.Body attribute), 62
[friction](#) (pymunk.Arbiter attribute), 76
[friction](#) (pymunk.Circle attribute), 66
[friction](#) (pymunk.Poly attribute), 70
[friction](#) (pymunk.Segment attribute), 73

friction (pymunk.Shape attribute), 64
 from_pygame() (in module pymunk.pygame_util), 50

G

GearJoint (class in pymunk.constraint), 42
 get_angle() (pymunk.vec2d.Vec2d method), 44
 get_angle_between() (pymunk.vec2d.Vec2d method), 44
 get_angle_degrees() (pymunk.vec2d.Vec2d method), 45
 get_angle_degrees_between() (pymunk.vec2d.Vec2d method), 45
 get_dist_sqrd() (pymunk.vec2d.Vec2d method), 45
 get_distance() (pymunk.vec2d.Vec2d method), 45
 get_length() (pymunk.vec2d.Vec2d method), 45
 get_length_sqrd() (pymunk.vec2d.Vec2d method), 45
 get_mouse_pos() (in module pymunk.pygame_util), 49
 get_vertices() (pymunk.Poly method), 70
 gradient (pymunk.PointQueryInfo attribute), 82
 gravity (pymunk.Space attribute), 56
 groove_a (pymunk.constraint.GrooveJoint attribute), 37
 groove_b (pymunk.constraint.GrooveJoint attribute), 37
 GrooveJoint (class in pymunk.constraint), 37
 group (pymunk.ShapeFilter attribute), 80

I

identity() (pymunk.Transform static method), 81
 idle_speed_threshold (pymunk.Space attribute), 56
 impulse (pymunk.constraint.Constraint attribute), 33
 impulse (pymunk.constraint.DampedRotarySpring attribute), 39
 impulse (pymunk.constraint.DampedSpring attribute), 38
 impulse (pymunk.constraint.GearJoint attribute), 42
 impulse (pymunk.constraint.GrooveJoint attribute), 37
 impulse (pymunk.constraint.PinJoint attribute), 34
 impulse (pymunk.constraint.PivotJoint attribute), 36
 impulse (pymunk.constraint.RatchetJoint attribute), 41
 impulse (pymunk.constraint.RotaryLimitJoint attribute), 40
 impulse (pymunk.constraint.SimpleMotor attribute), 43
 impulse (pymunk.constraint.SlideJoint attribute), 35
 index() (pymunk.autogeometry.PolylineSet method), 32
 index() (pymunk.PointQueryInfo method), 82
 index() (pymunk.SegmentQueryInfo method), 75
 index() (pymunk.ShapeFilter method), 80
 index() (pymunk.ShapeQueryInfo method), 82
 index() (pymunk.Transform method), 81
 inf (in module pymunk), 53
 int_tuple (pymunk.vec2d.Vec2d attribute), 45
 interpolate_to() (pymunk.vec2d.Vec2d method), 45
 intersects() (pymunk.BB method), 79
 intersects_segment() (pymunk.BB method), 79
 is_closed() (in module pymunk.autogeometry), 30
 is_first_contact (pymunk.Arbitrator attribute), 76
 is_removal (pymunk.Arbitrator attribute), 77
 is_sleeping (pymunk.Body attribute), 62

iterations (pymunk.Space attribute), 56

K

KINEMATIC (pymunk.Body attribute), 59
 kinetic_energy (pymunk.Body attribute), 62

L

left (pymunk.BB attribute), 79
 length (pymunk.vec2d.Vec2d attribute), 45
 local_to_world() (pymunk.Body method), 62

M

march_hard() (in module pymunk.autogeometry), 32
 march_soft() (in module pymunk.autogeometry), 32
 mask (pymunk.ShapeFilter attribute), 80
 mass (pymunk.Body attribute), 62
 mass (pymunk.Circle attribute), 67
 mass (pymunk.Poly attribute), 71
 mass (pymunk.Segment attribute), 73
 mass (pymunk.Shape attribute), 65
 max (pymunk.constraint.RotaryLimitJoint attribute), 40
 max (pymunk.constraint.SlideJoint attribute), 35
 max_bias (pymunk.constraint.Constraint attribute), 34
 max_bias (pymunk.constraint.DampedRotarySpring attribute), 40
 max_bias (pymunk.constraint.DampedSpring attribute), 38
 max_bias (pymunk.constraint.GearJoint attribute), 42
 max_bias (pymunk.constraint.GrooveJoint attribute), 37
 max_bias (pymunk.constraint.PinJoint attribute), 34
 max_bias (pymunk.constraint.PivotJoint attribute), 36
 max_bias (pymunk.constraint.RatchetJoint attribute), 41
 max_bias (pymunk.constraint.RotaryLimitJoint attribute), 41
 max_bias (pymunk.constraint.SimpleMotor attribute), 43
 max_bias (pymunk.constraint.SlideJoint attribute), 35
 max_force (pymunk.constraint.Constraint attribute), 34
 max_force (pymunk.constraint.DampedRotarySpring attribute), 40
 max_force (pymunk.constraint.DampedSpring attribute), 39
 max_force (pymunk.constraint.GearJoint attribute), 42
 max_force (pymunk.constraint.GrooveJoint attribute), 37
 max_force (pymunk.constraint.PinJoint attribute), 35
 max_force (pymunk.constraint.PivotJoint attribute), 36
 max_force (pymunk.constraint.RatchetJoint attribute), 41
 max_force (pymunk.constraint.RotaryLimitJoint attribute), 41
 max_force (pymunk.constraint.SimpleMotor attribute), 43
 max_force (pymunk.constraint.SlideJoint attribute), 35
 merge() (pymunk.BB method), 79
 merged_area() (pymunk.BB method), 79
 min (pymunk.constraint.RotaryLimitJoint attribute), 41

min (pymunk.constraint.SlideJoint attribute), 35
moment (pymunk.Body attribute), 62
moment (pymunk.Circle attribute), 67
moment (pymunk.Poly attribute), 71
moment (pymunk.Segment attribute), 73
moment (pymunk.Shape attribute), 65
moment_for_box() (in module pymunk), 75
moment_for_circle() (in module pymunk), 74
moment_for_poly() (in module pymunk), 75
moment_for_segment() (in module pymunk), 75

N

newForCircle() (pymunk.BB static method), 79
normal (pymunk.ContactPointSet attribute), 76
normal (pymunk.Segment attribute), 73
normal (pymunk.SegmentQueryInfo attribute), 75
normalize_return_length() (pymunk.vec2d.Vec2d method), 45
normalized() (pymunk.vec2d.Vec2d method), 45

O

offset (pymunk.Circle attribute), 67
ones() (pymunk.vec2d.Vec2d static method), 45
options (in module pymunkoptions), 52

P

perpendicular() (pymunk.vec2d.Vec2d method), 45
perpendicular_normal() (pymunk.vec2d.Vec2d method), 45
phase (pymunk.constraint.GearJoint attribute), 42
phase (pymunk.constraint.RatchetJoint attribute), 42
PinJoint (class in pymunk.constraint), 34
PivotJoint (class in pymunk.constraint), 36
point (pymunk.PointQueryInfo attribute), 82
point (pymunk.SegmentQueryInfo attribute), 75
point_a (pymunk.ContactPoint attribute), 76
point_b (pymunk.ContactPoint attribute), 76
point_query() (pymunk.Circle method), 67
point_query() (pymunk.Poly method), 71
point_query() (pymunk.Segment method), 74
point_query() (pymunk.Shape method), 65
point_query() (pymunk.Space method), 56
point_query_nearest() (pymunk.Space method), 57
PointQueryInfo (class in pymunk), 81
points (pymunk.ContactPointSet attribute), 76
Poly (class in pymunk), 68
PolylineSet (class in pymunk.autogeometry), 31
position (pymunk.Body attribute), 62
position_func (pymunk.Body attribute), 62
positive_y_is_up (in module pymunk.pygame_util), 50
post_solve (pymunk.CollisionHandler attribute), 78
pre_solve (pymunk.CollisionHandler attribute), 78
projection() (pymunk.vec2d.Vec2d method), 45
pymunk (module), 52

pymunk.autogeometry (module), 30
pymunk.constraint (module), 33
pymunk.matplotlib_util (module), 46
pymunk.pygame_util (module), 48
pymunk.pyglet_util (module), 50
pymunk.vec2d (module), 43
pymunkoptions (module), 52

R

radius (pymunk.Circle attribute), 67
radius (pymunk.Poly attribute), 71
radius (pymunk.Segment attribute), 74
ratchet (pymunk.constraint.RatchetJoint attribute), 42
RatchetJoint (class in pymunk.constraint), 41
rate (pymunk.constraint.SimpleMotor attribute), 43
ratio (pymunk.constraint.GearJoint attribute), 42
reindex_shape() (pymunk.Space method), 57
reindex_shapes_for_body() (pymunk.Space method), 57
reindex_static() (pymunk.Space method), 57
remove() (pymunk.Space method), 57
rest_angle (pymunk.constraint.DampedRotarySpring attribute), 40
rest_length (pymunk.constraint.DampedSpring attribute), 39
restitution (pymunk.Arbitrator attribute), 77
right (pymunk.BB attribute), 79
RotaryLimitJoint (class in pymunk.constraint), 40
rotate() (pymunk.vec2d.Vec2d method), 45
rotate_degrees() (pymunk.vec2d.Vec2d method), 45
rotated() (pymunk.vec2d.Vec2d method), 45
rotated_degrees() (pymunk.vec2d.Vec2d method), 46
rotation_vector (pymunk.Body attribute), 62

S

Segment (class in pymunk), 72
segment_query() (pymunk.BB method), 79
segment_query() (pymunk.Circle method), 67
segment_query() (pymunk.Poly method), 71
segment_query() (pymunk.Segment method), 74
segment_query() (pymunk.Shape method), 65
segment_query() (pymunk.Space method), 57
segment_query_first() (pymunk.Space method), 58
SegmentQueryInfo (class in pymunk), 75
sensor (pymunk.Circle attribute), 67
sensor (pymunk.Poly attribute), 71
sensor (pymunk.Segment attribute), 74
sensor (pymunk.Shape attribute), 65
separate (pymunk.CollisionHandler attribute), 78
set_neighbors() (pymunk.Segment method), 74
Shape (class in pymunk), 63
shape (pymunk.PointQueryInfo attribute), 82
shape (pymunk.SegmentQueryInfo attribute), 75
shape (pymunk.ShapeQueryInfo attribute), 82

- shape_dynamic_color (pymunk.matplotlib_util.DrawOptions attribute), 47
 - shape_dynamic_color (pymunk.pygame_util.DrawOptions attribute), 49
 - shape_dynamic_color (pymunk.pyglet_util.DrawOptions attribute), 52
 - shape_dynamic_color (pymunk.SpaceDebugDrawOptions attribute), 83
 - shape_kinematic_color (pymunk.matplotlib_util.DrawOptions attribute), 47
 - shape_kinematic_color (pymunk.pygame_util.DrawOptions attribute), 49
 - shape_kinematic_color (pymunk.pyglet_util.DrawOptions attribute), 52
 - shape_kinematic_color (pymunk.SpaceDebugDrawOptions attribute), 83
 - shape_outline_color (pymunk.matplotlib_util.DrawOptions attribute), 47
 - shape_outline_color (pymunk.pygame_util.DrawOptions attribute), 49
 - shape_outline_color (pymunk.pyglet_util.DrawOptions attribute), 52
 - shape_outline_color (pymunk.SpaceDebugDrawOptions attribute), 83
 - shape_query() (pymunk.Space method), 58
 - shape_sleeping_color (pymunk.matplotlib_util.DrawOptions attribute), 47
 - shape_sleeping_color (pymunk.pygame_util.DrawOptions attribute), 49
 - shape_sleeping_color (pymunk.pyglet_util.DrawOptions attribute), 52
 - shape_sleeping_color (pymunk.SpaceDebugDrawOptions attribute), 84
 - shape_static_color (pymunk.matplotlib_util.DrawOptions attribute), 47
 - shape_static_color (pymunk.pygame_util.DrawOptions attribute), 49
 - shape_static_color (pymunk.pyglet_util.DrawOptions attribute), 52
 - shape_static_color (pymunk.SpaceDebugDrawOptions attribute), 84
 - ShapeFilter (class in pymunk), 79
 - ShapeQueryInfo (class in pymunk), 82
 - shapes (pymunk.Arbitrator attribute), 77
 - shapes (pymunk.Body attribute), 62
 - shapes (pymunk.Space attribute), 58
 - shapes_collide() (pymunk.Circle method), 67
 - shapes_collide() (pymunk.Poly method), 71
 - shapes_collide() (pymunk.Segment method), 74
 - shapes_collide() (pymunk.Shape method), 65
 - SimpleMotor (class in pymunk.constraint), 42
 - simplify_curves() (in module pymunk.autogeometry), 30
 - simplify_vertexes() (in module pymunk.autogeometry), 31
 - sleep() (pymunk.Body method), 63
 - sleep_time_threshold (pymunk.Space attribute), 58
 - sleep_with_group() (pymunk.Body method), 63
 - SlideJoint (class in pymunk.constraint), 35
 - Space (class in pymunk), 53
 - space (pymunk.Body attribute), 63
 - space (pymunk.Circle attribute), 68
 - space (pymunk.Poly attribute), 71
 - space (pymunk.Segment attribute), 74
 - space (pymunk.Shape attribute), 65
 - SpaceDebugDrawOptions (class in pymunk), 82
 - STATIC (pymunk.Body attribute), 59
 - static_body (pymunk.Space attribute), 58
 - step() (pymunk.Space method), 58
 - stiffness (pymunk.constraint.DampedRotarySpring attribute), 40
 - stiffness (pymunk.constraint.DampedSpring attribute), 39
 - surface_velocity (pymunk.Arbitrator attribute), 77
 - surface_velocity (pymunk.Circle attribute), 68
 - surface_velocity (pymunk.Poly attribute), 71
 - surface_velocity (pymunk.Segment attribute), 74
 - surface_velocity (pymunk.Shape attribute), 65
- ## T
- threads (pymunk.Space attribute), 59
 - to_convex_hull() (in module pymunk.autogeometry), 31
 - to_pygame() (in module pymunk.pygame_util), 49
 - top (pymunk.BB attribute), 79
 - torque (pymunk.Body attribute), 63
 - total_impulse (pymunk.Arbitrator attribute), 77
 - total_ke (pymunk.Arbitrator attribute), 77
 - Transform (class in pymunk), 80
 - tx (pymunk.Transform attribute), 81
 - ty (pymunk.Transform attribute), 81
- ## U
- unit() (pymunk.vec2d.Vec2d static method), 46
 - unsafe_set_endpoints() (pymunk.Segment method), 74
 - unsafe_set_offset() (pymunk.Circle method), 68
 - unsafe_set_radius() (pymunk.Circle method), 68
 - unsafe_set_radius() (pymunk.Poly method), 71
 - unsafe_set_radius() (pymunk.Segment method), 74

`unsafe_set_vertices()` (pymunk.Poly method), [71](#)
`update()` (pymunk.Circle method), [68](#)
`update()` (pymunk.Poly method), [72](#)
`update()` (pymunk.Segment method), [74](#)
`update()` (pymunk.Shape method), [65](#)
`update_position()` (pymunk.Body static method), [63](#)
`update_velocity()` (pymunk.Body static method), [63](#)

V

`Vec2d` (class in pymunk.vec2d), [44](#)
`velocity` (pymunk.Body attribute), [63](#)
`velocity_at_local_point()` (pymunk.Body method), [63](#)
`velocity_at_world_point()` (pymunk.Body method), [63](#)
`velocity_func` (pymunk.Body attribute), [63](#)
`version` (in module pymunk), [53](#)

W

`world_to_local()` (pymunk.Body method), [63](#)

X

`x` (pymunk.vec2d.Vec2d attribute), [46](#)

Y

`y` (pymunk.vec2d.Vec2d attribute), [46](#)

Z

`zero()` (pymunk.vec2d.Vec2d static method), [46](#)