9.5.2 Non-Games .......................................................... 138
9.5.3 Papers / Science ...................................................... 139
9.6 Tutorials ............................................................... 142
  9.6.1 Slide and Pin Joint Demo Step by Step ......................... 143
  9.6.2 External Tutorials .................................................. 153
9.7 Benchmarks ............................................................ 154
  9.7.1 Micro benchmarks ................................................. 154
  9.7.2 Compared to Other Physics Libraries .......................... 156
9.8 Advanced .............................................................. 157
  9.8.1 Impulse Solver Algorithm ....................................... 157
  9.8.2 Collision Detection Algorithm .................................. 158
  9.8.3 Why CFFI? ......................................................... 158
  9.8.4 Why ctypes? (OBSOLETE) ....................................... 158
  9.8.5 Code Layout ...................................................... 159
  9.8.6 Tests .............................................................. 160
  9.8.7 Working with non-wrapped parts of Chipmunk ................ 160
  9.8.8 Weak References and free Methods .............................. 160
9.9 Changelog ............................................................. 161
  9.9.1 Pymunk 6.3.0 (2022-11-04) .................................... 161
  9.9.2 Pymunk 6.2.1 (2021-10-31) .................................... 161
  9.9.3 Pymunk 6.2.0 (2021-08-25) .................................... 161
  9.9.4 Pymunk 6.1.0 (2021-08-11) .................................... 162
  9.9.5 Pymunk 6.0.0 (2020-12-07) .................................... 162
  9.9.6 Pymunk 5.7.0 (2020-09-16) .................................... 165
  9.9.7 Pymunk 5.6.0 (2019-11-02) .................................... 165
  9.9.8 Pymunk 5.5.0 (2019-05-03) .................................... 166
  9.9.9 Pymunk 5.4.2 (2019-01-07) .................................... 166
  9.9.10 Pymunk 5.4.1 (2018-12-31) ................................... 167
  9.9.11 Pymunk 5.4.0 (2018-10-24) ................................... 167
  9.9.12 Pymunk 5.3.2 (2017-09-16) ................................... 167
  9.9.13 Pymunk 5.3.1 (2017-07-15) ................................... 168
  9.9.14 Pymunk 5.3.0 (2017-06-11) ................................... 168
  9.9.15 Pymunk 5.2.0 (2017-03-25) ................................... 168
  9.9.16 Pymunk 5.1.0 (2016-10-17) ................................... 169
  9.9.17 Pymunk 5.0.0 (2016-07-17) ................................... 169
  9.9.18 Pymunk 4.0.0 (2013-08-25) ................................... 169
  9.9.19 Pymunk 3.0.0 (2012-09-02) ................................... 170
  9.9.20 Pymunk 2.1.0 (2011-12-03) ................................... 171
  9.9.21 Pymunk 2.0.0 (2011-09-04) ................................... 171
  9.9.22 Pymunk 1.0.0 (2010-07-16) ................................... 172
  9.9.23 Pymunk 0.8.3 (2009-07-26) ................................... 172
  9.9.24 Pymunk 0.8.2 (2009-04-22) ................................... 172
  9.9.25 Pymunk 0.8.1 (2008-11-02) ................................... 172
  9.9.26 Pymunk 0.8 (2008-06-15) ..................................... 172
  9.9.27 Pymunk 0.1 (2007-08-01) ..................................... 172
9.10 License ............................................................... 173

10 Indices and tables ..................................................... 173

Python Module Index ..................................................... 175

Index ................................................................. 177
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 simulation! 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, more than 10 years of active development!

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 - 2022, Victor Blomqvist - vb@viblo.se, MIT License

This release is based on the latest Pymunk release (6.3.0), using Chipmunk 7 rev 0593976ef47fcb3957166bd342f6b2baf4d0e44.
In the normal case pymunk can be installed from PyPI with pip:

> pip install pymunk

It has one direct dependency, CFFI.

Pymunk can also be installed with conda, from the conda-forge channel:

> conda install -c conda-forge pymunk
Quick code example:

```python
import pymunk  # Import pymunk..

space = pymunk.Space()  # Create a Space which contain the simulation
space.gravity = 0, -981  # Set its gravity

body = pymunk.Body()  # Create a Body
body.position = 50, 100  # Set the position of the body

poly = pymunk.Poly.create_box(body)  # Create a box shape and attach to body
poly.mass = 10  # Set the mass on the shape
space.add(body, poly)  # Add both body and shape to the simulation

print_options = pymunk.SpaceDebugDrawOptions()  # For easy printing

while True:  # Infinite loop simulation
    space.step(0.02)  # Step the simulation one step forward
    space.debug_draw(print_options)  # Print the state of the simulation
```

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
The source distribution of Pymunk ships with a number of demos of different simulations 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
CHAPTER
FOUR

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 code should be needed to add physics to your game or 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
FIVE

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

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 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 3.6 and later and Pypy3)
- Chipmunk (Compiled library already included on common platforms)
- CFFI (will be installed automatically by Pip)
- Setuptools (should be included with Pip)
- GCC and friends (optional, you need it to compile Pymunk from source. On windows Visual Studio is required to compile)
- 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 & afigure & sphinx_autodoc_typehints (optional, you need it to build documentation)
Support for Python 2 (and Python 3.0 - 3.5) has been dropped with Pymunk 6.0. If you use these legacy versions of Python, please use Pymunk 5.x.
INSTALL FROM SOURCE / CHIPMUNK COMPILATION

This section is only required in case you do not install pymunk from the prebuild binary wheels (normally if you do not use pip install or you are on a uncommon platform).

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 together with Pymunk as an extension module.

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 inplace:

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

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

9.1.1 Install Pymunk

Pymunk can be installed with pip install:

> pip install pymunk

Pymunk can also be installed with conda install, from the conda-forge channel:

> conda install -c conda-forge pymunk

Once Pymunk is installed you can verify that the installation works by running the tests:

> python -m pymunk.tests -f test

Sometimes on more uncommon platforms you will need to have a GCC-compatible c-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

9.1.2 Examples & Documentation

Because of their size the examples and the documentation are 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)
9.1.3 Troubleshooting

Check that no files are named pymunk.py
Check that conda install works https://stackoverflow.com/questions/39811929/package-installed-by-conda-python-cannot-find-it

9.1.4 Advanced - Android Install

Pymunk can run on Android phones/tablets/computers.

Kivy
Kivy is a open source Python library for rapid development of applications that make use of innovative user interfaces, such as multi-touch apps, and can run on Android (and a number of other platforms such as Linux, Windows, OS X, iOS and Raspberry Pi).

Pymunk should work out of the box when used with Kivy. Note however that the recipe used to build Pymunk specifies a specific version of Pymunk that might not be the latest, see the recipe script here: https://github.com/kivy/python-for-android/blob/master/pythonforandroid/recipes/pymunk/__init__.py

Termux
Termux is an Android terminal emulator and Linux environment app that works directly with no rooting or setup required.

There are no binary wheels of pymunk for Termux/Android, or for its dependency cffi, so you will need to install a couple of packages first, before pymunk can be installed.

1. Install python and other needed dependencies (run inside Termux):

   $ pkg install python python-dev clang libffi-dev

2. Install pymunk with pip:

   $ pip install pymunk

3. Verify that it works:

   $ python -m pymunk.tests test

9.1.5 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 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.
Advanced - Running without installation

If you do not want to install Pymunk, for example because you want to bundle it with your code, its also possible to run it directly inplace. Given that you have the source code the first thing to do is to compile chipmunk with the inplace option, as described in the Compile Chipmunk section.

To actually import pymunk from its folder you need to do a small path hack, since the pymunk root folder (where setup.py and the README are located) is not part of the package. Instead you should add the path to the pymunk package folder (where files such as space.py and body.py are located):

```
mycodefolder/
|-- mycode.py
|-- ...
|-- pymunk/
   |-- README.rst
   |-- setup.py
   |-- pymunk/
   |   |-- space.py
   |   |-- body.py
   |   |-- ...
   |   |-- ...
```

Then inside you code file (mycode.py) import sys and add the pymunk folder to the path:

```
import sys
sys.path.insert(1, 'pymunk')
import pymunk
```

9.1.6 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 Visual Studio matching your Python version.
9.1.7 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
```

9.2 Overview

9.2.1 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 constraints between two bodies to constrain their behavior, for example to keep a fixed distance between two bodies.

**Spaces** (*pymunk.Space*)

Spaces are the basic simulation unit in Pymunk. You add bodies, shapes and constraints 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.

9.2.2 Model your physics objects

**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. Let’s 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. Let’s 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.

### Center of gravity

An important part of creating the shape of an object is to ensure its center of gravity is where it should. In most cases you want it to be in the center of the shape(s), but just like in real life it can create interesting objects if the center of gravity is not in the actual center.

Below is a couple of examples how the center easily ends up in one corner of the shape:

Note however that a Circle is created at the center automatically, and that a box created by the helper `pymunk.Poly.create_box()` will also have its center of gravity in the middle.

The center of gravity can be moved in a couple of different ways:

• `Segment(body, (0,0), (6,6))` can be changed to `Segment(body, (-3,-3), (-3,-3))`.
• The center of gravity can be adjusted directly on the body: `body.center_of_gravity = (3,3)`
• Poly shapes can be transformed with a `pymunk.Transform`. `Poly(body, [...], pymunk.Transform.translation(3,3))`
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)

Looks before realism

How heavy is a bird in angry birds? It does matter, it's 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.

9.2.3 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/

9.2.4 Object tunneling

Sometimes an object can pass through another object even though its not supposed to. Usually this happens because the object is moving so fast, that during a single call to space.step() the object moves from one side to the other.

There are several ways to mitigate this problem. Sometimes it might be a good idea to do more than one of these.
• Make sure the velocity of objects never get too high. One way to do that is to use a custom velocity function with a limit built in on the bodies that have a tendency to move too fast:

```python
def limit_velocity(body, gravity, damping, dt):
    max_velocity = 1000
    pymunk.Body.update_velocity(body, gravity, damping, dt)
    l = body.velocity.length
    if l > max_velocity:
        scale = max_velocity / l
        body.velocity = body.velocity * scale

body_to_limit.velocity_func = limit_velocity
```

Depending on the requirements it might make more sense to clamp the velocity over multiple frames instead. Then the limit function could look like this instead:

```python
def limit_velocity(body, gravity, damping, dt):
    max_velocity = 1000
    pymunk.Body.update_velocity(body, gravity, damping, dt)
    if body.velocity.length > max_velocity:
        body.velocity = body.velocity * 0.99
```

• For objects such as bullets, use a space query such as space.segment_query or space.segment_first.

• Use a smaller value for dt in the call to space.step. A simple way is to call space.step multiple times each frame in your application. This will also help to make the overall simulation more stable.

• Double check that the center of gravity is at a reasonable point for all objects.

## 9.2.5 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 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.

• Double check that the center of gravity is at a reasonable point for all objects.

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

Various tips that can improve performance:

- Run Python with optimizations on (will disable various useful but non-critical asserts). `python -O mycode.py`
- If possible use Pypy instead of CPython. See `Benchmarks` for some examples of the speed difference.
- Tweak the `Space.iterations` property.
- If possible let objects fall asleep with `Space.sleep_time_threshold`.
- Reduce usage of callback methods (like collision callbacks or custom update functions). These are much slower than the default built in code.

Note that many times the actual simulation is quick enough, but reading out the result after each step and manipulating the objects manually can have a significant overhead and performance cost.

9.2.7 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.

Note that the version of Pymunk used must be the same for the code saving as the version used when loading the saved object.

9.2.8 Additional info

As a complement to the Pymunk docs it can be good to read the Chipmunk docs. It's 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.

9.3 API Reference

9.3.1 pymunk Package

Submodules

`pymunk.autogeometry` Module

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

Example:

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

>>> def sample_func(point):
...     x = int(point[0])
...     y = int(point[1])
...     return 1 if img[y][x] == "x" else 0

>>> pl_set = march_soft(pymunk.BB(0,0,6,6), 7, 7, .5, sample_func)
>>> print(len(pl_set))
2

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

```python
>>> s = pymunk.Space()
>>> for poly_line in pl_set:
>>>     for i in range(len(poly_line) - 1):
>>>         a = poly_line[i]
>>>         b = poly_line[i + 1]
>>>         segment = pymunk.Segment(s.static_body, a, b, 1)
>>>         s.add(segment)
```

```python
class pymunk.autogeometry.PolylineSet
Bases: Sequence[List[Vec2d]]

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__`() → None

Initialize a new PolylineSet

`collect_segment`(v0: Tuple[float, float], v1: Tuple[float, float]) → None

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` (`Tuple[float, float]`) -- Start of segment
- `v1` (`Tuple[float, float]`) -- End of segment

`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.

Supporting start and stop arguments is optional, but recommended.


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)]

```python
def pymunk.autogeometry.is_closed(polyline: Union[List[Tuple[float, float]], List[Vec2d]]) -> bool:
    Returns true if the first vertex is equal to the last.
    Parameters
    polyline ([(float, float)]) – Polyline to simplify.
    Return type
    bool```

```python
def pymunk.autogeometry.march_hard(bb: BB, x_samples: int, y_samples: int, threshold: float, sample_func: Callable[[Tuple[float, float]], float]) -> PolylineSet:
    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
    • sample_func (func(point: Tuple[float, float]) -> float) – The sample function will be called for x_samples * y_samples spread across the bounding box area, and should return a float.
    Returns
    PolylineSet with the polylines found.
```

```python
def pymunk.autogeometry.march_soft(bb: BB, x_samples: int, y_samples: int, threshold: float, sample_func: Callable[[Tuple[float, float]], float]) -> PolylineSet:
    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
- **sample_func** *(func(point: Tuple[float, float]) -> float)* – The sample function will be called for x_samples * y_samples spread across the bounding box area, and should return a float.

**Returns**
Polylineset with the polylines found.

```python
pymunk.autogeometry.simplify_curves(polyline: Union[List[Tuple[float, float]], List[Vec2d]], tolerance: float) -> List[Vec2d]
```

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)]

```python
pymunk.autogeometry.simplify_vertexes(polyline: Union[List[Tuple[float, float]], List[Vec2d]], tolerance: float) -> List[Vec2d]
```

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)]

```python
pymunk.autogeometry.to_convex_hull(polyline: Union[List[Tuple[float, float]], List[Vec2d]], tolerance: float) -> List[Vec2d]
```

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.constraints 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=ZgJjZS0aMM](http://www.youtube.com/watch?v=ZgJjZS0aMM)
Example:

```python
>>> import pymunk
>>> import pymunk.constraints

s = pymunk.Space()

a, b = pymunk.Body(10, 10), pymunk.Body(10, 10)

c = pymunk.constraints.PivotJoint(a, b, (0,0))

s.add(c)
```

class pymunk.constraints.Constraint(constraint: CData)

   Bases: PickleMixin, TypingAttrMixing, 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: CData) → None

property a: Body

   The first of the two bodies constrained

activate_bodies() → None

   Activate the bodies this constraint is attached to

property b: Body

   The second of the two bodies constrained

property collide_bodies: bool

   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() → T

   Create a deep copy of this object.

property error_bias: float

   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.

property impulse: float

   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.

property max_bias: float

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

   Defaults to infinity

property max_force: float

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

   Defaults to infinity
property post_solve: Optional[Callable[[Constraint, Space], None]]

The post-solve function is called after the constraint solver runs.

Note that None can be used to reset it to default value.

```python
>>> import pymunk
>>> j = pymunk.PinJoint(pymunk.Body(1,2), pymunk.Body(3,4), (0,0))
>>> def post_solve_func(constraint, space):
...   print("Hello from pre-solve")
>>> j.post_solve = post_solve_func
>>> j.post_solve = None
```

property pre_solve: Optional[Callable[[Constraint, Space], None]]

The pre-solve function is called before the constraint solver runs.

Note that None can be used to reset it to default value.

```python
>>> import pymunk
>>> j = pymunk.PinJoint(pymunk.Body(1,2), pymunk.Body(3,4), (0,0))
>>> def pre_solve_func(constraint, space):
...   print("Hello from pre-solve")
>>> j.pre_solve = pre_solve_func
>>> j.pre_solve = None
```

class pymunk.constraints.DampedRotarySpring(a: Body, b: Body, rest_angle: float, stiffness: float, damping: float)

Bases: Constraint

DampedRotarySpring works like the DampedSpring but in a angular fashion.

__init__(a: Body, b: Body, rest_angle: float, stiffness: float, damping: float) → None

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.

property a: Body

The first of the two bodies constrained

activate_bodies() → None

Activate the bodies this constraint is attached to

property b: Body

The second of the two bodies constrained

property collide_bodies: bool

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() → T
Create a deep copy of this object.

property damping: float
How soft to make the damping of the spring.

property error_bias: float
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.

property impulse: float
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.

property max_bias: float
The maximum speed at which the constraint can apply error correction.

Defaults to infinity

property max_force: float
The maximum force that the constraint can use to act on the two bodies.

Defaults to infinity

property post_solve: Optional[Callable[[Constraint, Space], None]]
The post-solve function is called after the constraint solver runs.

Note that None can be used to reset it to default value.

```python
>>> import pymunk
>>> j = pymunk.PinJoint(pymunk.Body(1,2), pymunk.Body(3,4), (0,0))
>>> def post_solve_func(constraint, space):
...     print("Hello from pre-solve")
>>> j.post_solve = post_solve_func
>>> j.post_solve = None
```

property pre_solve: Optional[Callable[[Constraint, Space], None]]
The pre-solve function is called before the constraint solver runs.

Note that None can be used to reset it to default value.

```python
>>> import pymunk
>>> j = pymunk.PinJoint(pymunk.Body(1,2), pymunk.Body(3,4), (0,0))
>>> def pre_solve_func(constraint, space):
...     print("Hello from pre-solve")
>>> j.pre_solve = pre_solve_func
>>> j.pre_solve = None
```

property rest_angle: float
The relative angle in radians that the bodies want to have

property stiffness: float
The spring constant (Young’s modulus).
class pymunk.constraints.DampedSpring(a: Body, b: Body, anchor_a: Tuple[float, float], anchor_b: Tuple[float, float], rest_length: float, stiffness: float, damping: float)

Bases: Constraint

DampedSpring is a damped spring.
The spring allows you to define the rest length, stiffness and damping.

__init__(a: Body, b: Body, anchor_a: Tuple[float, float], anchor_b: Tuple[float, float], rest_length: float, stiffness: float, damping: float) → None

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.

property a: Body

The first of the two bodies constrained

activate_bodies() → None

Activate the bodies this constraint is attached to

property anchor_a: Vec2d

property anchor_b: Vec2d

property b: Body

The second of the two bodies constrained

property collide_bodies: bool

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() → T

Create a deep copy of this object.

property damping: float

How soft to make the damping of the spring.

property error_bias: float

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.
property impulse: float
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.

property max_bias: float
The maximum speed at which the constraint can apply error correction.
Defaults to infinity

property max_force: float
The maximum force that the constraint can use to act on the two bodies.
Defaults to infinity

property post_solve: Optional[Callable[[Constraint, Space], None]]
The post-solve function is called after the constraint solver runs.
Note that None can be used to reset it to default value.

```python
>>> import pymunk
>>> j = pymunk.PinJoint(pymunk.Body(1,2), pymunk.Body(3,4), (0,0))
>>> def post_solve_func(constraint, space):
...    print("Hello from pre-solve")
>>> j.post_solve = post_solve_func
>>> j.post_solve = None
```

property pre_solve: Optional[Callable[[Constraint, Space], None]]
The pre-solve function is called before the constraint solver runs.
Note that None can be used to reset it to default value.

```python
>>> import pymunk
>>> j = pymunk.PinJoint(pymunk.Body(1,2), pymunk.Body(3,4), (0,0))
>>> def pre_solve_func(constraint, space):
...    print("Hello from pre-solve")
>>> j.pre_solve = pre_solve_func
>>> j.pre_solve = None
```

property rest_length: float
The distance the spring wants to be.

property stiffness: float
The spring constant (Young's modulus).

class pymunk.constraints.GearJoint(a: Body, b: Body, phase: float, ratio: float)
Bases: Constraint
GearJoint keeps the angular velocity ratio of a pair of bodies constant.

__init__(a: Body, b: Body, phase: float, ratio: float)
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.

property a: Body
The first of the two bodies constrained
activate_bodies() → None

Activate the bodies this constraint is attached to.

property b: Body

The second of the two bodies constrained.

property collide_bodies: bool

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() → T

Create a deep copy of this object.

property error_bias: float

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.

property impulse: float

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.

property max_bias: float

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

Defaults to infinity

property max_force: float

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

Defaults to infinity

property phase: float

property post_solve: Optional[Callable[[Constraint, Space], None]]

The post-solve function is called after the constraint solver runs.

Note that None can be used to reset it to default value.

```python
>>> import pymunk
>>> j = pymunk.PinJoint(pymunk.Body(1,2), pymunk.Body(3,4), (0,0))
>>> def post_solve_func(constraint, space):
...     print("Hello from post-solve")
>>> j.post_solve = post_solve_func
>>> j.post_solve = None
```

property pre_solve: Optional[Callable[[Constraint, Space], None]]

The pre-solve function is called before the constraint solver runs.

Note that None can be used to reset it to default value.
import pymunk
j = pymunk.PinJoint(pymunk.Body(1,2), pymunk.Body(3,4), (0,0))
def pre_solve_func(constraint, space):
... print("Hello from pre-solve")
j.pre_solve = pre_solve_func
j.pre_solve = None

property ratio: float

class pymunk.constraints.GrooveJoint(a: Body, b: Body, groove_a: Tuple[float, float], groove_b: Tuple[float, float], anchor_b: Tuple[float, float])

Bases: Constraint

GrooveJoint is similar to a PivotJoint, but with a linear slide.
One of the anchor points is a line segment that the pivot can slide in instead of being fixed.

__init__(a: Body, b: Body, groove_a: Tuple[float, float], groove_b: Tuple[float, float], anchor_b: Tuple[float, float]) → None

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.

property a: Body
The first of the two bodies constrained

activate_bodies() → None
Activate the bodies this constraint is attached to

property anchor_b: Vec2d

property b: Body
The second of the two bodies constrained

property collide_bodies: bool
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() → T
Create a deep copy of this object.

property error_bias: float
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.

property groove_a: Vec2d

property groove_b: Vec2d
**property impulse**: float

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.

**property max_bias**: float

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

Defaults to infinity

**property max_force**: float

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

Defaults to infinity

**property post_solve**: Optional[Callable[[Constraint, Space], None]]

The post-solve function is called after the constraint solver runs.

Note that None can be used to reset it to default value.

```python
>>> import pymunk
>>> j = pymunk.PinJoint(pymunk.Body(1,2), pymunk.Body(3,4), (0,0))
>>> def post_solve_func(constraint, space):
...     print("Hello from pre-solve")
>>> j.post_solve = post_solve_func
>>> j.post_solve = None
```

**property pre_solve**: Optional[Callable[[Constraint, Space], None]]

The pre-solve function is called before the constraint solver runs.

Note that None can be used to reset it to default value.

```python
>>> import pymunk
>>> j = pymunk.PinJoint(pymunk.Body(1,2), pymunk.Body(3,4), (0,0))
>>> def pre_solve_func(constraint, space):
...     print("Hello from pre-solve")
>>> j.pre_solve = pre_solve_func
>>> j.pre_solve = None
```

**class pymunk.constraints.PinJoint**(a: Body, b: Body, anchor_a: Tuple[float, float] = (0, 0), anchor_b: Tuple[float, float] = (0, 0))

Bases: Constraint

PinJoint links shapes with a solid bar or pin.

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

__init__(a: Body, b: Body, anchor_a: Tuple[float, float] = (0, 0), anchor_b: Tuple[float, float] = (0, 0)) → None

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.

**property a**: Body

The first of the two bodies constrained
activate_bodies() → None
Activate the bodies this constraint is attached to

property anchor_a: Vec2d

property anchor_b: Vec2d

property b: Body
The second of the two bodies constrained

property collide_bodies: bool
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() → T
Create a deep copy of this object.

property distance: float

property error_bias: float
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.

property impulse: float
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.

property max_bias: float
The maximum speed at which the constraint can apply error correction.
Defaults to infinity

property max_force: float
The maximum force that the constraint can use to act on the two bodies.
Defaults to infinity

property post_solve: Optional[Callable[[Constraint, Space], None]]
The post-solve function is called after the constraint solver runs.
Note that None can be used to reset it to default value.

>>> import pymunk
>>> j = pymunk.PinJoint(pymunk.Body(1,2), pymunk.Body(3,4), (0,0))
>>> def post_solve_func(constraint, space):
...     print("Hello from pre-solve")
>>> j.post_solve = post_solve_func
>>> j.post_solve = None
property pre_solve: Optional[Callable[[Constraint, Space], None]]

The pre-solve function is called before the constraint solver runs.

Note that None can be used to reset it to default value.

```python
>>> import pymunk
>>> j = pymunk.PinJoint(pymunk.Body(1, 2), pymunk.Body(3, 4), (0, 0))
>>> def pre_solve_func(constraint, space):
...     print("Hello from pre-solve")
>>> j.pre_solve = pre_solve_func
>>> j.pre_solve = None
```

class pymunk.constraints.PivotJoint(a: Body, b: Body, *args: Union[Tuple[float, float], Tuple[Tuple[float, float], Tuple[float, float]]])

Bases: Constraint

PivotJoint allow two objects to pivot about a single point.

Its like a swivel.

```python
__init__(a: Body, b: Body, *args: Union[Tuple[float, float], Tuple[Tuple[float, float], Tuple[float, float]]]) → None
```

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

property a: Body

The first of the two bodies constrained

activate_bodies() → None

Activate the bodies this constraint is attached to

property anchor_a: Vec2d

property anchor_b: Vec2d

property b: Body

The second of the two bodies constrained

property collide_bodies: bool

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() \rightarrow T

Create a deep copy of this object.

**property error_bias**: float

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.

**property impulse**: float

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.

**property max_bias**: float

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

Defaults to infinity

**property max_force**: float

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

Defaults to infinity

**property post_solve**: Optional[Callable[[Constraint, Space], None]]

The post-solve function is called after the constraint solver runs.

Note that None can be used to reset it to default value.

```python
>>> import pymunk
>>> j = pymunk.PinJoint(pymunk.Body(1,2), pymunk.Body(3,4), (0,0))
>>> def post_solve_func(constraint, space):
...     print("Hello from pre-solve")
>>> j.post_solve = post_solve_func
>>> j.post_solve = None
```

**property pre_solve**: Optional[Callable[[Constraint, Space], None]]

The pre-solve function is called before the constraint solver runs.

Note that None can be used to reset it to default value.

```python
>>> import pymunk
>>> j = pymunk.PinJoint(pymunk.Body(1,2), pymunk.Body(3,4), (0,0))
>>> def pre_solve_func(constraint, space):
...     print("Hello from pre-solve")
>>> j.pre_solve = pre_solve_func
>>> j.pre_solve = None
```

class pymunk.constraints.RatchetJoint\((a: Body, b: Body, phase: float, ratchet: float)\)

Bases: Constraint

RatchetJoint is a rotary ratchet, it works like a socket wrench.
__init__(a: Body, b: Body, phase: float, ratchet: float) → None

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.

**property a:** Body

The first of the two bodies constrained

**activate_bodies() → None**

Activate the bodies this constraint is attached to

**property angle:** float

**property b:** Body

The second of the two bodies constrained

**property collide_bodies:** bool

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() → T**

Create a deep copy of this object.

**property error_bias:** float

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.

**property impulse:** float

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.

**property max_bias:** float

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

Defaults to infinity

**property max_force:** float

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

Defaults to infinity

**property phase:** float

**property post_solve:** Optional[Callable[[Constraint, Space], None]]

The post-solve function is called after the constraint solver runs.

Note that None can be used to reset it to default value.
>>> import pymunk
>>> j = pymunk.PinJoint(pymunk.Body(1,2), pymunk.Body(3,4), (0,0))
>>> def post_solve_func(constraint, space):
    ...    print("Hello from pre-solve")
>>> j.post_solve = post_solve_func
>>> j.post_solve = None

property pre_solve: Optional[Callable[[Constraint, Space], None]]

The pre-solve function is called before the constraint solver runs.

Note that None can be used to reset it to default value.

>>> import pymunk
>>> j = pymunk.PinJoint(pymunk.Body(1,2), pymunk.Body(3,4), (0,0))
>>> def pre_solve_func(constraint, space):
    ...    print("Hello from pre-solve")
>>> j.pre_solve = pre_solve_func
>>> j.pre_solve = None

property ratchet: float

class pymunk.constraints.RotaryLimitJoint(a: Body, b: Body, min: float, max: float)
    Bases: Constraint

RotaryLimitJoint constrains the relative rotations of two bodies.

__init__(a: Body, b: Body, min: float, max: float) -> None

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.

property a: Body

The first of the two bodies constrained

activate_bodies() -> None

Activate the bodies this constraint is attached to

property b: Body

The second of the two bodies constrained

property collide_bodies: bool

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() -> T

Create a deep copy of this object.

property error_bias: float

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.
property impulse: float
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.

property max: float

property max_bias: float
The maximum speed at which the constraint can apply error correction.
Defaults to infinity

property max_force: float
The maximum force that the constraint can use to act on the two bodies.
Defaults to infinity

property min: float

property post_solve: Optional[Callable[[Constraint, Space], None]]
The post-solve function is called after the constraint solver runs.
Note that None can be used to reset it to default value.

```python
>>> import pymunk
>>> j = pymunk.PinJoint(pymunk.Body(1,2), pymunk.Body(3,4), (0,0))
>>> def post_solve_func(constraint, space):
...     print("Hello from pre-solve")
>>> j.post_solve = post_solve_func
>>> j.post_solve = None
```

property pre_solve: Optional[Callable[[Constraint, Space], None]]
The pre-solve function is called before the constraint solver runs.
Note that None can be used to reset it to default value.

```python
>>> import pymunk
>>> j = pymunk.PinJoint(pymunk.Body(1,2), pymunk.Body(3,4), (0,0))
>>> def pre_solve_func(constraint, space):
...     print("Hello from pre-solve")
>>> j.pre_solve = pre_solve_func
>>> j.pre_solve = None
```

class pymunk.constraints.SimpleMotor(a: Body, b: Body, rate: float)
Bases: Constraint
SimpleMotor keeps the relative angular velocity constant.

__init__(a: Body, b: Body, rate: float) → None
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.

property a: Body
The first of the two bodies constrained
activate_bodies() → None
Activate the bodies this constraint is attached to

property b:  Body
The second of the two bodies constrained

property collide_bodies:  bool
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() → T
Create a deep copy of this object.

property error_bias:  float
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.

property impulse:  float
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.

property max_bias:  float
The maximum speed at which the constraint can apply error correction.
Defaults to infinity

property max_force:  float
The maximum force that the constraint can use to act on the two bodies.
Defaults to infinity

property post_solve:  Optional[Callable[[Constraint, Space], None]]
The post-solve function is called after the constraint solver runs.
Note that None can be used to reset it to default value.

>>> import pymunk
>>> j = pymunk.PinJoint(pymunk.Body(1,2), pymunk.Body(3,4), (0,0))
>>> def post_solve_func(constraint, space):
...     print("Hello from pre-solve")
>>> j.post_solve = post_solve_func
>>> j.post_solve = None

property pre_solve:  Optional[Callable[[Constraint, Space], None]]
The pre-solve function is called before the constraint solver runs.
Note that None can be used to reset it to default value.
>>> import pymunk
>>> j = pymunk.PinJoint(pymunk.Body(1,2), pymunk.Body(3,4), (0,0))
>>> def pre_solve_func(constraint, space):
...     print("Hello from pre-solve")
>>> j.pre_solve = pre_solve_func
>>> j.pre_solve = None

**property rate**: float

The desired relative angular velocity

class pymunk.constraints.SlideJoint(a: Body, b: Body, anchor_a: Tuple[float, float], anchor_b: Tuple[float, float], min: float, max: float)

Bases: Constraint

SlideJoint is like a PinJoint, 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.

```python
__init__(a: Body, b: Body, anchor_a: Tuple[float, float], anchor_b: Tuple[float, float], min: float, max: float) → None
```

- `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.

**property a**: Body

The first of the two bodies constrained

**activate_bodies()** → None

Activate the bodies this constraint is attached to

**property anchor_a**: Vec2d

**property anchor_b**: Vec2d

**property b**: Body

The second of the two bodies constrained

**property collide_bodies**: bool

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()** → T

Create a deep copy of this object.

**property error_bias**: float

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.
property impulse: float
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.

property max: float

property max_bias: float
The maximum speed at which the constraint can apply error correction.
Defaults to infinity

property max_force: float
The maximum force that the constraint can use to act on the two bodies.
Defaults to infinity

property min: float

property post_solve: Optional[Callable[[Constraint, Space], None]]
The post-solve function is called after the constraint solver runs.
Note that None can be used to reset it to default value.

>>> import pymunk
>>> j = pymunk.PinJoint(pymunk.Body(1,2), pymunk.Body(3,4), (0,0))
>>> def post_solve_func(constraint, space):
...     print("Hello from pre-solve")
>>> j.post_solve = post_solve_func
>>> j.post_solve = None

property pre_solve: Optional[Callable[[Constraint, Space], None]]
The pre-solve function is called before the constraint solver runs.
Note that None can be used to reset it to default value.

>>> import pymunk
>>> j = pymunk.PinJoint(pymunk.Body(1,2), pymunk.Body(3,4), (0,0))
>>> def pre_solve_func(constraint, space):
...     print("Hello from pre-solve")
>>> j.pre_solve = pre_solve_func
>>> j.pre_solve = None

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:

```python
>>> from pymunk.vec2d import Vec2d

>>> Vec2d(7.3, 4.2)
Vec2d(7.3, 4.2)

>>> Vec2d(7.3, 4.2) + Vec2d(1, 2)
Vec2d(8.3, 6.2)
```

```python
class pymunk.vec2d(Vec2d(x: float, y: float)
    Bases: tuple

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

    __abs__() → float
    Return the length of the Vec2d

    >>> abs(Vec2d(3,4))
    5.0

    __add__(other: Tuple[float, float]) → Vec2d
    Add a Vec2d with another Vec2d or Tuple of size 2

    >>> Vec2d(3,4) + Vec2d(1,2)
    Vec2d(4, 6)
    >>> Vec2d(3,4) + (1,2)
    Vec2d(4, 6)

    __floordiv__(other: float) → Vec2d
    Floor division by a float (also known as integer division)

    >>> Vec2d(3,6) // 2.0
    Vec2d(1.0, 3.0)

    __mul__(other: float) → Vec2d
    Multiply with a float

    >>> Vec2d(3,6) * 2.5
    Vec2d(7.5, 15.0)

    __neg__() → Vec2d
    Return the negated version of the Vec2d

    >>> -Vec2d(1,-2)
    Vec2d(-1, 2)
```
Return the unary pos of the Vec2d.

```python
>>> +Vec2d(1, -2)
Vec2d(1, -2)
```

Subtract a Vec2d with another Vec2d or Tuple of size 2

```python
>>> Vec2d(3, 4) - Vec2d(1, 2)
Vec2d(2, 2)

>>> Vec2d(3, 4) - (1, 2)
Vec2d(2, 2)
```

Division by a float

```python
>>> Vec2d(3, 6) / 2.0
Vec2d(1.5, 3.0)
```

The angle (in radians) of the vector

```python
property angle: float
```

The angle (in degrees) of a vector

```python
property angle_degrees: float
```

Uses complex multiplication to rotate this vector by the other.

```python
cpvrotate(other: Tuple[float, float]) → Vec2d
```

The inverse of cpvrotate

```python
cpvunrotate(other: Tuple[float, float]) → Vec2d
```

The cross product between the vector and other vector

```python
cross(other: Tuple[float, float]) → float
```

Returns

The cross product

The dot product between the vector and other vector

```python
dot(other: Tuple[float, float]) → float
```

Returns

The dot product
get_angle_between \( \text{other: Tuple[float, float]} \) \rightarrow \text{float}  
Get the angle between the vector and the other in radians  

Returns  
The angle  

get_angle_degrees_between \( \text{other: Vec2d} \) \rightarrow \text{float}  
Get the angle between the vector and the other in degrees  

Returns  
The angle (in degrees)  

get_dist_sqrd \( \text{other: Tuple[float, float]} \) \rightarrow \text{float}  
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 \( \text{other: Tuple[float, float]} \) \rightarrow \text{float}  
The distance between the vector and other vector  

Returns  
The distance  

get_length_sqrd() \rightarrow \text{float}  
Get the squared length of the vector. If the squared length is enough it is more efficient to use this method instead of first calling get_length() or access .length and then do a \( x**2 \).  

Returns  
The squared length  

index \( \text{value, start=0, stop=9223372036854775807, /} \)  
Return first index of value.  

Raises ValueError if the value is not present.  


definitions  

property int_tuple: Tuple[int, int]  
The x and y values of this vector as a tuple of ints. Uses round() to round to closest int.  

interpolate_to \( \text{other: Tuple[float, float]}, \text{range: float} \) \rightarrow \text{Vec2d}  

property length: float  
Get the length of the vector.  

>>> v = Vec2d(3, 4)  
>>> v.get_length_sqrd() == v.length**2  
True
Returns
The length

normalized() → Vec2d
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

normalized_and_length() → Tuple[Vec2d, float]
Normalize the vector and return its length before the normalization.

Returns
The length before the normalization

static ones() → Vec2d
A vector where both x and y is 1

>>> Vec2d.ones()
Vec2d(1, 1)

perpendicular() → Vec2d

perpendicular_normal() → Vec2d

projection(other: Tuple[float, float]) → Vec2d
Project this vector on top of other vector

rotated(angle_radians: float) → Vec2d
Create and return a new vector by rotating this vector by angle_radians radians.

Returns
Rotated vector

rotated_degrees(angle_degrees: float) → Vec2d
Create and return a new vector by rotating this vector by angle_degrees degrees.

Returns
Rotated vector

scale_to_length(length: float) → Vec2d
Return a copy of this vector scaled to the given length.

>>> '%.2f, %.2f' % Vec2d(10, 20).scale_to_length(20)
'8.94, 17.89'

static unit() → Vec2d
A unit vector pointing up

>>> Vec2d.unit()
Vec2d(0, 1)

property x
Alias for field number 0

property y
Alias for field number 1
**pymunk Documentation, Release 6.3.0**

**static zero() → Vec2d**

A vector of zero length.

```python
>>> Vec2d.zero()
Vec2d(0, 0)
```

### 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: Any)
Bases: SpaceDebugDrawOptions

**DRAW_COLLISION_POINTS**

alias of <class 'CP_SPACE_DEBUG_DRAW_COLLISION_POINTS'>

**DRAW_CONSTRAINTS**

alias of <class 'CP_SPACE_DEBUG_DRAW_CONSTRAINTS'>

**DRAW_SHAPES**

alias of <class 'CP_SPACE_DEBUG_DRAW_SHAPES'>

**__init__(ax: Any) → None**

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

Typical usage:

```python
>>> import matplotlib as mpl
>>> import matplotlib.pyplot as plt
>>> import pymunk
>>> import pymunk.matplotlib_util

>>> space = pymunk.Space()

>>> ax = plt.subplot()

>>> options = pymunk.matplotlib_util.DrawOptions(ax)

>>> space.debug_draw(options)
```

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

```python
>>> shape = pymunk.Circle(space.static_body, 10)
>>> shape.color = (1, 0, 0, 1)  # will draw shape in red
```

See matplotlib_util.demo.py for a full example

**Param**

- **ax: matplotlib.Axes**
  - A matplotlib Axes object.

**property collision_point_color: SpaceDebugColor**

The color of collisions.

Should be a tuple of 4 ints between 0 and 255 (r,g,b,a).

Example:
```python
>>> import pymunk
>>> s = pymunk.Space()
>>> b = pymunk.Body(1, 10)
>>> c1 = pymunk.Circle(b, 10)
>>> c2 = pymunk.Circle(s.static_body, 10)
>>> s.add(b, c1, c2)
>>> s.step(1)
>>> options = pymunk.SpaceDebugDrawOptions()
>>> s.debug_draw(options)
```

```python
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, 10.0, 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(8.0, 0.0), Vec2d(-8.0, 0.0), SpaceDebugColor(r=231.0, g=76.0, b=60.0, a=255.0))
```

```python
>>> options.collision_point_color = (10, 20, 30, 40)
>>> s.debug_draw(options)
```

```python
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, 10.0, 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(8.0, 0.0), Vec2d(-8.0, 0.0), SpaceDebugColor(r=10.0, g=20.0, b=30.0, a=40.0))
```

color_for_shape(shape: Shape) → SpaceDebugColor

**property constraint_color:** SpaceDebugColor

The color of constraints.

Should be a tuple of 4 ints between 0 and 255 (r,g,b,a).

Example:

```python
>>> import pymunk
>>> s = pymunk.Space()
>>> b = pymunk.Body(1, 10)
>>> j = pymunk.PivotJoint(s.static_body, b, (0,0))
>>> s.add(j)
>>> options = pymunk.SpaceDebugDrawOptions()
>>> s.debug_draw(options)
```

```python
draw_dot (5.0, Vec2d(0.0, 0.0), SpaceDebugColor(r=142.0, g=68.0, b=173.0, a=255.0))
draw_dot (5.0, Vec2d(0.0, 0.0), SpaceDebugColor(r=142.0, g=68.0, b=173.0, a=255.0))
```

```python
>>> options.constraint_color = (10, 20, 30, 40)
>>> s.debug_draw(options)
```

```python
draw_dot (5.0, Vec2d(0.0, 0.0), SpaceDebugColor(r=10.0, g=20.0, b=30.0, a=40.0))
draw_dot (5.0, Vec2d(0.0, 0.0), SpaceDebugColor(r=10.0, g=20.0, b=30.0, a=40.0))
```

draw_circle(pos: Vec2d, angle: float, radius: float, outline_color: SpaceDebugColor, fill_color: SpaceDebugColor) → None

draw_dot(size: float, pos: Vec2d, color: SpaceDebugColor) → None

**draw_polygon**(*verts*: Sequence[Vec2d], *radius*: float, *outline_color*: SpaceDebugColor, *fill_color*: SpaceDebugColor) ➞ None

**draw_segment**(*a*: Vec2d, *b*: Vec2d, *color*: SpaceDebugColor) ➞ None

**draw_shape**(*shape*: Shape) ➞ None

**property flags**: int
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 the desired backend instead, such as `pygame_util.DrawOptions` or `pyglet_util.DrawOptions`):

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

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

**property shape_outline_color**: SpaceDebugColor
The outline color of shapes.

Should be a tuple of 4 ints between 0 and 255 (r,g,b,a).

Example:
>>> import pymunk
>>> s = pymunk.Space()
>>> c = pymunk.Circle(s.static_body, 10)
>>> s.add(c)
>>> options = pymunk.SpaceDebugDrawOptions()
>>> 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=149.0, g=165.0, b=166.0, a=255.0))

options.shape_outline_color = (10,20,30,40)

options = pymunk.SpaceDebugDrawOptions()
options.transform = pymunk.Transform.scaling(2)
options.transform = pymunk.Transform.translation(2,3)

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

property transform: Transform
The transform is applied before drawing, e.g for scaling or translation.

Example:

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: Surface)
    Bases: SpaceDebugDrawOptions

Note: Not all tranformations are supported by the debug drawing logic. Uniform scaling and translation are supported, but not rotation, linear stretching or shearing.
**DRAW_COLLISION_POINTS**

alias of `<class 'CP_SPACE_DEBUG_DRAW_COLLISION_POINTS'>`

**DRAW_CONSTRAINTS**

alias of `<class 'CP_SPACE_DEBUG_DRAW_CONSTRAINTS'>`

**DRAW_SHAPES**

alias of `<class 'CP_SPACE_DEBUG_DRAW_SHAPES'>`

```python
__init__(surface: Surface) \rightarrow None
```

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

Typical usage:

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

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

```python
>>> c = pymunk.Circle(None, 10)
>>> c.color = pygame.Color("pink")
```

See pygame_util.demo.py for a full example

Since pygame uses a coordinate system where y points down (in contrast to many other cases), you either have to make the physics simulation with Pymunk also behave in that way, or flip everything when you draw.

The easiest is probably to just make the simulation behave the same way as Pygame does. In that way all coordinates used are in the same orientation and easy to reason about:

```python
>>> space = pymunk.Space()
>>> space.gravity = (0, -1000)
>>> body = pymunk.Body()
>>> body.position = (0, 0) # will be positioned in the top left corner
>>> space.debug_draw(options)
```

To flip the drawing its possible to set the module property `positive_y_is_up` to True. Then the pygame drawing will flip the simulation upside down before drawing:

```python
>>> positive_y_is_up = True
>>> body = pymunk.Body()
>>> body.position = (0, 0)
>>> # Body will be position in bottom left corner
```

**Parameters**

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

- `property collision_point_color: SpaceDebugColor`

  The color of collisions.

  Should be a tuple of 4 ints between 0 and 255 (r,g,b,a).
Example:

```python
>>> import pymunk
>>> s = pymunk.Space()  
>>> b = pymunk.Body(1,10)  
>>> c1 = pymunk.Circle(b, 10)  
>>> c2 = pymunk.Circle(s.static_body, 10)  
>>> s.add(b, c1, c2)  
>>> s.step(1)  
>>> options = pymunk.SpaceDebugDrawOptions()  
>>> s.debug_draw(options)  
```

```python
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))  
```

```python
draw_segment (Vec2d(8.0, 0.0), Vec2d(-8.0, 0.0), SpaceDebugColor(r=231.0, g=76.0, b=60.0, a=255.0))  
```

```python
color_for_shape (shape: Shape) → SpaceDebugColor  
```

```python
property constraint_color: SpaceDebugColor  
```

The color of constraints.

Should be a tuple of 4 ints between 0 and 255 (r,g,b,a).

Example:

```python
>>> import pymunk
>>> s = pymunk.Space()  
>>> b = pymunk.Body(1, 10)  
>>> j = pymunk.PivotJoint(s.static_body, b, (0,0))  
>>> s.add(j)  
>>> options = pymunk.SpaceDebugDrawOptions()  
>>> s.debug_draw(options)  
```

```python
draw_dot (5.0, Vec2d(0.0, 0.0), SpaceDebugColor(r=142.0, g=68.0, b=173.0, a=255.0))  
```

```python
draw_circle (Vec2d(0.0, 0.0), 0.0, 10.0, SpaceDebugColor(r=142.0, g=68.0, b=173.0, a=255.0))  
```

```python
draw_segment (Vec2d(8.0, 0.0), Vec2d(-8.0, 0.0), SpaceDebugColor(r=231.0, g=76.0, b=60.0, a=255.0))  
```

```python
color_for_shape (shape: Shape) → SpaceDebugColor  
```

```python
property constraint_color: SpaceDebugColor  
```

The color of constraints.

Should be a tuple of 4 ints between 0 and 255 (r,g,b,a).

Example:

```python
>>> import pymunk
>>> s = pymunk.Space()  
>>> b = pymunk.Body(1, 10)  
>>> j = pymunk.PivotJoint(s.static_body, b, (0,0))  
>>> s.add(j)  
>>> options = pymunk.SpaceDebugDrawOptions()  
>>> s.debug_draw(options)  
```

```python
draw_dot (5.0, Vec2d(0.0, 0.0), SpaceDebugColor(r=142.0, g=68.0, b=173.0, a=255.0))  
```

```python
draw_circle (Vec2d(0.0, 0.0), 0.0, 10.0, SpaceDebugColor(r=142.0, g=68.0, b=173.0, a=255.0))  
```

```python
draw_segment (Vec2d(8.0, 0.0), Vec2d(-8.0, 0.0), SpaceDebugColor(r=231.0, g=76.0, b=60.0, a=255.0))  
```

```python
color_for_shape (shape: Shape) → SpaceDebugColor  
```

```python
property constraint_color: SpaceDebugColor  
```

The color of constraints.

Should be a tuple of 4 ints between 0 and 255 (r,g,b,a).

Example:

```python
>>> import pymunk
>>> s = pymunk.Space()  
>>> b = pymunk.Body(1, 10)  
>>> j = pymunk.PivotJoint(s.static_body, b, (0,0))  
>>> s.add(j)  
>>> options = pymunk.SpaceDebugDrawOptions()  
>>> s.debug_draw(options)  
```

```python
draw_dot (5.0, Vec2d(0.0, 0.0), SpaceDebugColor(r=142.0, g=68.0, b=173.0, a=255.0))  
```

```python
draw_circle (Vec2d(0.0, 0.0), 0.0, 10.0, SpaceDebugColor(r=142.0, g=68.0, b=173.0, a=255.0))  
```

```python
draw_segment (Vec2d(8.0, 0.0), Vec2d(-8.0, 0.0), SpaceDebugColor(r=231.0, g=76.0, b=60.0, a=255.0))  
```

```python
color_for_shape (shape: Shape) → SpaceDebugColor  
```

```python
property constraint_color: SpaceDebugColor  
```

The color of constraints.

Should be a tuple of 4 ints between 0 and 255 (r,g,b,a).
**draw_fat_segment** *(a: Tuple[float, float], b: Tuple[float, float], radius: float, outline_color: SpaceDebugColor, fill_color: SpaceDebugColor) → None*

**draw_polygon** *(verts: Sequence[Tuple[float, float]], radius: float, outline_color: SpaceDebugColor, fill_color: SpaceDebugColor) → None*

**draw_segment** *(a: Vec2d, b: Vec2d, color: SpaceDebugColor) → None*

**draw_shape** *(shape: Shape) → None*

**property flags: int**

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 the desired backend instead, such as *pygame_util.DrawOptions* or *pyglet_util.DrawOptions*):

```python
>>> 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_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 = SpaceDebugColor(r=52, g=152, b=219, a=255)**

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

**property shape_outline_color: SpaceDebugColor**

The outline color of shapes.

Should be a tuple of 4 ints between 0 and 255 (r,g,b,a).

Example:
>>> import pymunk
>>> s = pymunk.Space()
>>> c = pymunk.Circle(s.static_body, 10)
>>> s.add(c)
>>> options = pymunk.SpaceDebugDrawOptions()
>>> 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=149.0, g=165.0, b=166.0, a=255.0))

>>> options.shape_outline_color = (10, 20, 30, 40)
>>> s.debug_draw(options)

draw_circle (Vec2d(0.0, 0.0), 0.0, 10.0, SpaceDebugColor(r=10.0, g=20.0, b=30.0, a=40.0), SpaceDebugColor(r=149.0, g=165.0, b=166.0, a=255.0))

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

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

property transform:  Transform

The transform is applied before drawing, e.g for scaling or translation.

Example:

>>> import pymunk
>>> s = pymunk.Space()
>>> c = pymunk.Circle(s.static_body, 10)
>>> s.add(c)
>>> options = pymunk.SpaceDebugDrawOptions()
>>> 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=149.0, g=165.0, b=166.0, a=255.0))

>>> options.transform = pymunk.Transform.scaling(2)
>>> s.debug_draw(options)

draw_circle (Vec2d(0.0, 0.0), 0.0, 20.0, 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))

>>> options.transform = pymunk.Transform.translation(2,3)
>>> s.debug_draw(options)

draw_circle (Vec2d(2.0, 3.0), 0.0, 10.0, 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))

Note:  Not all tranformations are supported by the debug drawing logic. Uniform scaling and translation are supported, but not rotation, linear stretching or shearing.

pymunk.pygame_util.from_pygame(p: Tuple[float, float], surface: Surface) \(\rightarrow\) Tuple[int, int]

Convenience method to convert pygame surface local coordinates to pymunk coordinates

pymunk.pygame_util.get_mouse_pos(surface: Surface) \(\rightarrow\) Tuple[int, int]

Get position of the mouse pointer in pymunk coordinates.

pymunk.pygame_util.positive_y_is_up:  bool = False

Make increasing values of y point upwards.

When True:
Pymunk Documentation, Release 6.3.0

```
| |
| . (3, 3)
| |
| . (2, 2)
| |
+------ > x
```

When False:

```
+------ > x
| |
| . (2, 2)
| |
| . (3, 3)
v
```

```
pymunk.pygame_util.to_pygame(p: Tuple[float, float], surface: Surface) → Tuple[int, int]
```

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.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 is 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: Any)
```

Bases: SpaceDebugDrawOptions

**DRAW_COLLISION_POINTS**

alias of <class 'CP_SPACE_DEBUG_DRAW_COLLISION_POINTS'>

**DRAW_CONSTRAINTS**

alias of <class 'CP_SPACE_DEBUG_DRAW_CONSTRAINTS'>

**DRAW_SHAPES**

alias of <class 'CP_SPACE_DEBUG_DRAW_SHAPES'>

```
__init__(**kwargs: Any) → None
```

Draw a pymunk.Space.

Typical usage:

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

9.3. API Reference

59
```python
>>> 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.

```python
>>> 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 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.

**property collision_point_color:**  SpaceDebugColor

The color of collisions.

Should be a tuple of 4 ints between 0 and 255 (r,g,b,a).

Example:

```python
>>> import pymunk
>>> s = pymunk.Space()
>>> b = pymunk.Body(1, 10)
>>> c1 = pymunk.Circle(b, 10)
>>> c2 = pymunk.Circle(s.static_body, 10)
>>> s.add(b, c1, c2)
>>> s.step(1)
>>> options = pymunk.SpaceDebugDrawOptions()
>>> s.debug_draw(options)
```

```python
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, 10.0, 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(8.0, 0.0), Vec2d(-8.0, 0.0), SpaceDebugColor(r=231.0, g=76.0, b=60.0, a=255.0))
```

```python
>>> options.collision_point_color = (10, 20, 30, 40)
>>> s.debug_draw(options)
```

```python
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, 10.0, 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(8.0, 0.0), Vec2d(-8.0, 0.0), SpaceDebugColor(r=10.0, g=20.0, b=30.0, a=40.0))
```

**color_for_shape**(shape: Shape) → SpaceDebugColor

**property constraint_color:**  SpaceDebugColor

The color of constraints.

Should be a tuple of 4 ints between 0 and 255 (r,g,b,a).
Example:

```
>>> import pymunk
>>> s = pymunk.Space()
>>> b = pymunk.Body(1, 10)
>>> j = pymunk.PivotJoint(s.static_body, b, (0,0))
>>> s.add(j)
>>> options = pymunk.SpaceDebugDrawOptions()
>>> s.debug_draw(options)
draw_dot (5.0, Vec2d(0.0, 0.0), SpaceDebugColor(r=142.0, g=68.0, b=173.0, a=255.0))
>>> options.constraint_color = (10,20,30,40)
>>> s.debug_draw(options)
draw_dot (5.0, Vec2d(0.0, 0.0), SpaceDebugColor(r=10.0, g=20.0, b=30.0, a=40.0))
```

draw_circle(pos: Vec2d, angle: float, radius: float, outline_color: SpaceDebugColor, fill_color: SpaceDebugColor) → None

draw_dot(size: float, pos: Vec2d, color: SpaceDebugColor) → None

draw_fat_segment(a: Vec2d, b: Vec2d, radius: float, outline_color: SpaceDebugColor, fill_color: SpaceDebugColor) → None

draw_polygon(verts: Sequence[Vec2d], radius: float, outline_color: SpaceDebugColor, fill_color: SpaceDebugColor) → None

draw_segment(a: Vec2d, b: Vec2d, color: SpaceDebugColor) → None

draw_shape(shape: Shape) → None

property flags: int

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 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()
```

```python
>>> # 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))
shape_dynamic_color: SpaceDebugColor = SpaceDebugColor(r=52, g=152, b=219, a=255)

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

property shape_outline_color: SpaceDebugColor
    The outline color of shapes.
    Should be a tuple of 4 ints between 0 and 255 (r,g,b,a).
    Example:

>>> import pymunk
>>> s = pymunk.Space()
>>> c = pymunk.Circle(s.static_body, 10)
>>> s.add(c)
>>> options = pymunk.SpaceDebugDrawOptions()
>>> 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=149.0, g=165.0, b=166.0, a=255.0))
>>> options.shape_outline_color = (10,20,30,40)
>>> s.debug_draw(options)
    draw_circle (Vec2d(0.0, 0.0), 0.0, 10.0, SpaceDebugColor(r=10.0, g=20.0, b=30.0, a=40.0), SpaceDebugColor(r=149.0, g=165.0, b=166.0, a=255.0))

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

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

property transform: Transform
    The transform is applied before drawing, e.g for scaling or translation.
    Example:

>>> import pymunk
>>> s = pymunk.Space()
>>> c = pymunk.Circle(s.static_body, 10)
>>> s.add(c)
>>> options = pymunk.SpaceDebugDrawOptions()
>>> 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=149.0, g=165.0, b=166.0, a=255.0))

```python
>>> options.transform = pymunk.Transform.scaling(2)
```

```python
>>> s.debug_draw(options)
```

draw_circle (Vec2d(0.0, 0.0), 0.0, 20.0, 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))

```python
>>> options.transform = pymunk.Transform.translation(2,3)
```

```python
>>> s.debug_draw(options)
```

draw_circle (Vec2d(2.0, 3.0), 0.0, 10.0, 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))

---

**Note:** Not all transformations are supported by the debug drawing logic. Uniform scaling and translation are supported, but not rotation, linear stretching or shearing.

---

### pymunk.tests Module

**Pymunk**

Pymunk is an 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](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.

**class** `pymunk.Arbiter`(_arbiter: CData, space: 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.

```python
__init__(_arbiter: CData, space: Space) → None
```

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.

**property** `contact_point_set`: `ContactPointSet`

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

Return `ContactPointSet`
**property friction**: float

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.

**property is_first_contact**: bool

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.

**property is_removal**: bool

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

**property normal**: Vec2d

Returns the normal of the collision.

**property restitution**: float

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.

**property shapes**: Tuple[Shape, Shape]

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

**property surface_velocity**: Vec2d

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.

**property total_impulse**: Vec2d

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.

**property total_ke**: float

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.

```python
class pymunk.BB(left: float = 0, bottom: float = 0, right: float = 0, top: float = 0)

Bases: tuple

Simple axis-aligned 2D bounding box.

Stored as left, bottom, right, top values.

An instance can be created in this way:

```python
>>> BB(left=1, bottom=5, right=20, top=10)
BB(left=1, bottom=5, right=20, top=10)
```

Or partially, for example like this:
area() \rightarrow float
Return the area

**property bottom**
Alias for field number 1

center() \rightarrow Vec2d
Return the center

clamp_vect(v: Tuple[float, float]) \rightarrow Vec2d
Returns a copy of the vector v clamped to the bounding box

contains(other: BB) \rightarrow bool
Returns true if bb completley contains the other bb

contains_vect(v: Tuple[float, float]) \rightarrow bool
Returns true if this bb contains the vector v

count(value, /)
Return number of occurrences of value.

expand(v: Tuple[float, float]) \rightarrow BB
Return the minimal bounding box that contains both this bounding box and the vector v

index(value, start=0, stop=9223372036854775807, /)
Return first index of value.
Raises ValueError if the value is not present.

intersects(other: BB) \rightarrow bool
Returns true if the bounding boxes intersect

intersects_segment(a: Tuple[float, float], b: Tuple[float, float]) \rightarrow bool
Returns true if the segment defined by endpoints a and b intersect this bb.

**property left**
Alias for field number 0

merge(other: BB) \rightarrow BB
Return the minimal bounding box that contains both this bb and the other bb

merged_area(other: BB) \rightarrow float
Merges this and other then returns the area of the merged bounding box.

static newForCircle(p: Tuple[float, float], r: float) \rightarrow BB
Convenience constructor for making a BB fitting a circle at position p with radius r.

**property right**
Alias for field number 2

segment_query(a: Tuple[float, float], b: Tuple[float, float]) \rightarrow float
Returns the fraction along the segment query the BB is hit.
Returns infinity if it doesn't hit
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 `<class ‘CP_BODY_TYPE_DYNAMIC’>`

**KINEMATIC**

alias of `<class ‘CP_BODY_TYPE_KINEMATIC’>`

**STATIC**

alias of `<class ‘CP_BODY_TYPE_STATIC’>`

```python
__init__(mass: float = 0, moment: float = 0, body_type: int = <class ‘CP_BODY_TYPE_DYNAMIC’>) → None
```

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:

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

```python
>>> def print_mass_moment(b):
...     print("mass={:.0f} moment={:.0f}".format(b.mass, b.moment))
```
# Using Shape.density

```python
generate Serialize流传提取

>>> # 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:

```python
>>> # 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)
mass=75 moment=3921
```

activate() → None
Reset the idle timer on a body.
If it was sleeping, wake it and any other bodies it was touching.

property angle: float
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.

**property angular_velocity: float**

The angular velocity of the body in radians per second.

**apply_force_at_local_point**

(force: Tuple[float, float], point: Tuple[float, float] = (0, 0)) → None

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

**apply_force_at_world_point**

(force: Tuple[float, float], point: Tuple[float, float]) → None

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: Tuple[float, float], point: Tuple[float, float] = (0, 0)) → None

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

**apply_impulse_at_world_point**

(impulse: Tuple[float, float], point: Tuple[float, float]) → None

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

**property body_type: int**

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.

**property center_of_gravity: Vec2d**

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.

**property constraints: Set[Constraint]**

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()** → T

Create a deep copy of this object.

**each_arbiter**

(func: Callable[..., None], *args: Any, **kwargs: Any) → None

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!

**property force:** *Vec2d*
Force applied to the center of gravity of the body.

This value is reset for every time step. Note that this is not the total of forces acting on the body (such as from collisions), but the force applied manually from the apply force functions.

**property is_sleeping:** *bool*
Returns true if the body is sleeping.

**property kinetic_energy:** *float*
Get the kinetic energy of a body.

**local_to_world:** (*pymunk.Vec2d*)
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

**property mass:** *float*
Mass of the body.

**property moment:** *float*
Moment of inertia (MoI or sometimes just moment) of the body.

The moment is like the rotational mass of a body.

**property position:** *Vec2d*
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.

**property 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.

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

**property rotation_vector:** *Vec2d*
The rotation vector for the body.

**property shapes:** *Set[Shape]*
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()** → None

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

Cannot be called from a callback.

**sleep_with_group**(body: Body) → None

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.

**property space:** Optional[Space]

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

**property torque:** float

The torque applied to the body.

This value is reset for every time step.

**static update_position**(body: Body, dt: float) → None

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: Body, gravity: Tuple[float, float], damping: float, dt: float) → None

Default rigid body velocity integration function.

Updates the velocity of the body using Euler integration.

**property velocity:** Vec2d

Linear velocity of the center of gravity of the body.

**velocity_at_local_point**(point: Tuple[float, float]) → Vec2d

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

**velocity_at_world_point**(point: Tuple[float, float]) → Vec2d

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.

**property 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)
```

There are many cases when this can be useful. One example is individual gravity for some bodies, and another is to limit the velocity which is useful to prevent tunneling.

Example of a callback that sets gravity to zero for a object.
>>> import pymunk
>>> space = pymunk.Space()
>>> space.gravity = 0, 10
>>> body = pymunk.Body(1,2)
>>> space.add(body)
>>> def zero_gravity(body, gravity, damping, dt):
...     pymunk.Body.update_velocity(body, (0,0), damping, dt)
...
>>> body.velocity_func = zero_gravity
>>> space.step(1)
>>> space.step(1)
>>> print(body.position, body.velocity)
Vec2d(0.0, 0.0) Vec2d(0.0, 0.0)

Example of a callback that limits the velocity:

>>> import pymunk
>>> body = pymunk.Body(1,2)
>>> def limit_velocity(body, gravity, damping, dt):
...     max_velocity = 1000
...     pymunk.Body.update_velocity(body, gravity, damping, dt)
...     l = body.velocity.length
...     if l > max_velocity:
...         scale = max_velocity / l
...         body.velocity = body.velocity * scale
...
>>> body.velocity_func = limit_velocity

world_to_local\((v: Tuple[\text{float}, \text{float}]) \rightarrow \text{Vec2d}\)

Convert world space coordinates to body local coordinates

Parameters
v – Vector in world space coordinates

class pymunk.Circle(body: Optional[Body], radius: float, offset: Tuple[float, float] = (0, 0))

Bases: Shape

A circle shape defined by a radius

This is the fastest and simplest collision shape

__init__(body: Optional[Body], radius: float, offset: Tuple[float, float] = (0, 0)) \rightarrow None

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.

property area: float

The calculated area of this shape.

property bb: 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().
property body: Optional[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() → BB
   Update and returns the bounding box of this shape

property center_of_gravity: Vec2d
   The calculated center of gravity of this shape.

property collision_type: int
   User defined collision type for the shape.
   See Space.add_collision_handler() function for more information on when to use this property.

copy() → T
   Create a deep copy of this object.

property density: float
   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)

property elasticity: float
   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.

property filter: ShapeFilter
   Set the collision ShapeFilter for this shape.

property friction: float
   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).

<table>
<thead>
<tr>
<th>Material</th>
<th>Other</th>
<th>Friction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aluminium</td>
<td>Steel</td>
<td>0.61</td>
</tr>
<tr>
<td>Copper</td>
<td>Steel</td>
<td>0.53</td>
</tr>
<tr>
<td>Brass</td>
<td>Steel</td>
<td>0.51</td>
</tr>
<tr>
<td>Cast iron</td>
<td>Copper</td>
<td>1.05</td>
</tr>
<tr>
<td>Cast iron</td>
<td>Zinc</td>
<td>0.85</td>
</tr>
<tr>
<td>Concrete (wet)</td>
<td>Rubber</td>
<td>0.30</td>
</tr>
<tr>
<td>Concrete (dry)</td>
<td>Rubber</td>
<td>1.0</td>
</tr>
<tr>
<td>Concrete</td>
<td>Wood</td>
<td>0.62</td>
</tr>
<tr>
<td>Copper</td>
<td>Glass</td>
<td>0.68</td>
</tr>
<tr>
<td>Glass</td>
<td>Glass</td>
<td>0.94</td>
</tr>
<tr>
<td>Metal</td>
<td>Wood</td>
<td>0.5</td>
</tr>
<tr>
<td>Polyethene</td>
<td>Steel</td>
<td>0.2</td>
</tr>
<tr>
<td>Steel</td>
<td>Steel</td>
<td>0.80</td>
</tr>
<tr>
<td>Steel</td>
<td>Teflon</td>
<td>0.04</td>
</tr>
<tr>
<td>Teflon (PTFE)</td>
<td>Teflon</td>
<td>0.04</td>
</tr>
<tr>
<td>Wood</td>
<td>Wood</td>
<td>0.4</td>
</tr>
</tbody>
</table>
property mass: float
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)

property moment: float
The calculated moment of this shape.

property offset: Vec2d
Offset. (body space coordinates)

point_query(p: Tuple[float, float]) → PointQueryInfo
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)

property radius: float
The Radius of the circle

segment_query(start: Tuple[float, float], end: Tuple[float, float], radius: float = 0) → SegmentQueryInfo
Check if the line segment from start to end intersects the shape.

Return type
SegmentQueryInfo

property sensor: bool
A boolean value if this shape is a sensor or not.
Sensors only call collision callbacks, and never generate real collisions.

shapes_collide(b: Shape) → ContactPointSet
Get contact information about this shape and shape b.

Return type
ContactPointSet

property space: Optional[Space]
Get the Space that shape has been added to (or None).

property surface_velocity: Vec2d
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: Tuple[float, float]) → None
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: float) → None
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: Transform) → BB
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.CollisionHandler(_handler: Any, space: Space)
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-awaken.

__init__(_handler: Any, space: Space) → None
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.

property begin: Optional[Callable[[Arbiter, Space, Any], bool]]
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 recieve a separate event when the shapes stop overlapping.

property data: Dict[Any, Any]
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.

property post_solve: Optional[Callable[[Arbiter, Space, Any], None]]
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.

property pre_solve: Optional[Callable[[Arbiter, Space, Any], bool]]
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.

**property separate:** Optional[Callable[[Arbiter, Space, Any], None]]

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

```python
def func(arbiter, space, data):
    pass
```

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.

```python
class pymunk.ContactPoint(point_a: Vec2d, point_b: Vec2d, distance: float)
    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: Vec2d, point_b: Vec2d, distance: float) → None
    distance: float
    point_a: Vec2d
    point_b: Vec2d
```

```python
class pymunk.ContactPointSet(normal: Vec2d, points: List[ContactPoint])
    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: Vec2d, points: List[ContactPoint]) → None
    normal: Vec2d
    points: List[ContactPoint]
```

```python
class pymunk.PointQueryInfo(shape: Optional[Shape], point: Vec2d, distance: float, gradient: Vec2d)
    Bases: tuple
    PointQueryInfo holds the result of a point query made on a Shape or Space.
    count(value, /)
        Return number of occurrences of value.
    property distance
        The distance to the point. The distance is negative if the point is inside the shape.
    property 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.
```

9.3. API Reference 75
index(value, start=0, stop=9223372036854775807, /)
Return first index of value.
Raises ValueError if the value is not present.

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

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

class pymunk.Poly(body: Optional[Body], vertices: Sequence[Tuple[float, float]], transform: Optional[Transform] = None, radius: float = 0)
Bases: Shape
A convex polygon shape
Slowest, but most flexible collision shape.
__init__(body: Optional[Body], vertices: Sequence[Tuple[float, float]], transform: Optional[Transform] = None, radius: float = 0) → None
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.

Note: Make sure to put the vertices around (0,0) or the shape might behave strange.

Either directly place the vertices like the below example:

```python
>>> import pymunk
>>> w, h = 10, 20
>>> vs = [(-w/2,-h/2), (w/2,-h/2), (w/2,h/2), (-w/2,h/2)]
>>> poly_good = pymunk.Poly(None, vs)
>>> print(poly_good.center_of_gravity)
Vec2d(0.0, 0.0)
```

Or use a transform to move them:

```python
>>> import pymunk
>>> width, height = 10, 20
>>> vs = [(0, 0), (width, 0), (width, height), (0, height)]
>>> poly_bad = pymunk.Poly(None, vs)
>>> print(poly_bad.center_of_gravity)
Vec2d(5.0, 10.0)
>>> t = pymunk.Transform(tx=-width/2, ty=-height/2)
>>> poly_good = pymunk.Poly(None, vs, transform=t)
>>> print(poly_good.center_of_gravity)
Vec2d(0.0, 0.0)
```
• **body** ([Body](#body)) – The body to attach the poly to

• **vertices** ([([float, float])](#vertices)) – Define a convex hull of the polygon with a counter-clockwise winding.

• **transform** ([Transform](#transform)) – Transform will be applied to every vertex.

• **radius** ([float](#radius)) – Set the radius of the poly shape

**property area**: [float](#property-area)

The calculated area of this shape.

**property bb**: [BB](#property-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()`.

**property body**: [Optional[Body]](#property-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()** → [BB](#cache-bb)

Update and returns the bounding box of this shape.

**property center_of_gravity**: [Vec2d](#property-center-of-gravity)

The calculated center of gravity of this shape.

**property collision_type**: [int](#property-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()** → T

Create a deep copy of this object.

**static create_box**([body: Optional[Body], size: Tuple[float, float] = (10, 10), radius: float = 0]) → Poly

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](#body)) – The body to attach the poly to

• **size** ([float, float]) – Size of the box as (width, height)

• **radius** ([float](#radius)) – Radius of poly

**Return type**

Poly

**static create_box_bb**([body: Optional[Body], bb: BB, radius: float = 0]) → Poly

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*

**property density**: *float*

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)

**property elasticity**: *float*

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.

**property filter**: *ShapeFilter*

Set the collision *ShapeFilter* for this shape.

**property friction**: *float*

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).

<table>
<thead>
<tr>
<th>Material</th>
<th>Other</th>
<th>Friction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aluminium</td>
<td>Steel</td>
<td>0.61</td>
</tr>
<tr>
<td>Copper</td>
<td>Steel</td>
<td>0.53</td>
</tr>
<tr>
<td>Brass</td>
<td>Steel</td>
<td>0.51</td>
</tr>
<tr>
<td>Cast iron</td>
<td>Copper</td>
<td>1.05</td>
</tr>
<tr>
<td>Cast iron</td>
<td>Zinc</td>
<td>0.85</td>
</tr>
<tr>
<td>Concrete (wet)</td>
<td>Rubber</td>
<td>0.30</td>
</tr>
<tr>
<td>Concrete (dry)</td>
<td>Rubber</td>
<td>1.0</td>
</tr>
<tr>
<td>Concrete</td>
<td>Wood</td>
<td>0.62</td>
</tr>
<tr>
<td>Copper</td>
<td>Glass</td>
<td>0.68</td>
</tr>
<tr>
<td>Glass</td>
<td>Glass</td>
<td>0.94</td>
</tr>
<tr>
<td>Metal</td>
<td>Wood</td>
<td>0.5</td>
</tr>
<tr>
<td>Polythene</td>
<td>Steel</td>
<td>0.2</td>
</tr>
<tr>
<td>Steel</td>
<td>Steel</td>
<td>0.80</td>
</tr>
<tr>
<td>Steel</td>
<td>Teflon</td>
<td>0.04</td>
</tr>
<tr>
<td>Teflon (PTFE)</td>
<td>Teflon</td>
<td>0.04</td>
</tr>
<tr>
<td>Wood</td>
<td>Wood</td>
<td>0.4</td>
</tr>
</tbody>
</table>
get_vertices() → List[Vec2d]

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:

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

Returns
The vertices in local coords

Return type
[List[Vec2d]]

property mass: float
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)

property moment: float
The calculated moment of this shape.

point_query(p: Tuple[float, float]) → PointQueryInfo
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)

property radius: float
The radius of the poly shape.

Extends the poly in all directions with the given radius.

segment_query(start: Tuple[float, float], end: Tuple[float, float], radius: float = 0) → SegmentQueryInfo
Check if the line segment from start to end intersects the shape.

Return type
SegmentQueryInfo

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

Sensors only call collision callbacks, and never generate real collisions.
shapes_collide(b: Shape) → ContactPointSet

Get contact information about this shape and shape b.

Return type

ContactPointSet

property space: Optional[Space]

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

property surface_velocity: Vec2d

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: float) → None

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: Sequence[Tuple[float, float]], transform: Optional[Transform] = None) → 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: Transform) → BB

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: Optional[Body], a: Tuple[float, float], b: Tuple[float, float], radius: float)

Bases: 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: Optional[Body], a: Tuple[float, float], b: Tuple[float, float], radius: float) → None

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

property a: Vec2d

The first of the two endpoints for this segment
property area: float
    The calculated area of this shape.

property b: Vec2d
    The second of the two endpoints for this segment

property bb: 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().

property body: Optional[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() → BB
    Update and returns the bounding box of this shape

property center_of_gravity: Vec2d
    The calculated center of gravity of this shape.

property collision_type: int
    User defined collision type for the shape.
    See Space.add_collision_handler() function for more information on when to use this property.

copy() → T
    Create a deep copy of this object.

property density: float
    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)

property elasticity: float
    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 inaccu-
    racies in the simulation using 1.0 or greater is not recommended.

property filter: ShapeFilter
    Set the collision ShapeFilter for this shape.

property friction: float
    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).
### Friction Coefficients

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

**property mass:** float

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)

**property moment:** float

The calculated moment of this shape.

**property normal:** Vec2d

The normal

**point_query** *(p: Tuple[float, float]) → PointQueryInfo*

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)

**property radius:** float

The radius/thickness of the segment

**segment_query** *(start: Tuple[float, float], end: Tuple[float, float], radius: float = 0) → SegmentQueryInfo*

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

- **Return type**
  - SegmentQueryInfo

**property sensor:** bool

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

Sensors only call collision callbacks, and never generate real collisions.
**set_neighbors** *(prev: Tuple[float, float], next: Tuple[float, float]) → None*

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: Shape) → ContactPointSet*

Get contact information about this shape and shape b.

**Return type**

ContactPointSet

**property space:** Optional[Space]

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

**property surface_velocity:** Vec2d

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: Tuple[float, float], b: Tuple[float, float]) → None*

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 its velocity. Changing it will not result in realistic physical behavior. Only use if you know what you are doing!

**unsafe_set_radius** *(r: float) → None*

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 its velocity. Changing it will not result in realistic physical behavior. Only use if you know what you are doing!

**update** *(transform: Transform) → BB*

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.SegmentQueryInfo** *(shape: Optional[Shape], point: Vec2d, normal: Vec2d, alpha: float)*

**Bases:** tuple

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.

**property alpha**

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

**count** *(value, /)*

Return number of occurrences of value.
\textbf{index}(\texttt{value, start=0, stop=922372036854775807, /})

Return first index of value.

Raises ValueError if the value is not present.

\textbf{property normal}

The normal of the surface hit.

\textbf{property point}

The point of impact.

\textbf{property shape}

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

\textbf{class pymunk.Shape}(\texttt{shape: Shape})

Bases: PickleMixin, TypingAttrMixing, 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 (\texttt{Circle}, \texttt{Poly} or \texttt{Segment}).

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

\textbf{__init__}(\texttt{shape: Shape}) \rightarrow None

\textbf{property area}: \texttt{float}

The calculated area of this shape.

\textbf{property bb}: \texttt{BB}

The bounding box \texttt{BB} of the shape.

Only guaranteed to be valid after \texttt{Shape.cache_bb()} or \texttt{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 \texttt{Shape.update()}.

\textbf{property body}: \texttt{Optional[Body]}

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

\textbf{cache_bb}() \rightarrow \texttt{BB}

Update and returns the bounding box of this shape.

\textbf{property center_of_gravity}: \texttt{Vec2d}

The calculated center of gravity of this shape.

\textbf{property collision_type}: \texttt{int}

User defined collision type for the shape.

See \texttt{Space.add_collision_handler()} function for more information on when to use this property.

\textbf{copy}() \rightarrow \texttt{T}

Create a deep copy of this object.

\textbf{property density}: \texttt{float}

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)
property elasticity: float
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.

property filter: ShapeFilter
Set the collision ShapeFilter for this shape.

property friction: float
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).

<table>
<thead>
<tr>
<th>Material</th>
<th>Other</th>
<th>Friction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aluminium</td>
<td>Steel</td>
<td>0.61</td>
</tr>
<tr>
<td>Copper</td>
<td>Steel</td>
<td>0.53</td>
</tr>
<tr>
<td>Brass</td>
<td>Steel</td>
<td>0.51</td>
</tr>
<tr>
<td>Cast iron</td>
<td>Copper</td>
<td>1.05</td>
</tr>
<tr>
<td>Cast iron</td>
<td>Zinc</td>
<td>0.85</td>
</tr>
<tr>
<td>Concrete (wet)</td>
<td>Rubber</td>
<td>0.30</td>
</tr>
<tr>
<td>Concrete (dry)</td>
<td>Rubber</td>
<td>1.0</td>
</tr>
<tr>
<td>Concrete</td>
<td>Wood</td>
<td>0.62</td>
</tr>
<tr>
<td>Copper</td>
<td>Glass</td>
<td>0.68</td>
</tr>
<tr>
<td>Glass</td>
<td>Glass</td>
<td>0.94</td>
</tr>
<tr>
<td>Metal</td>
<td>Wood</td>
<td>0.5</td>
</tr>
<tr>
<td>Polyethylene</td>
<td>Steel</td>
<td>0.2</td>
</tr>
<tr>
<td>Steel</td>
<td>Steel</td>
<td>0.80</td>
</tr>
<tr>
<td>Steel</td>
<td>Teflon</td>
<td>0.04</td>
</tr>
<tr>
<td>Teflon (PTFE)</td>
<td>Teflon</td>
<td>0.04</td>
</tr>
<tr>
<td>Wood</td>
<td>Wood</td>
<td>0.4</td>
</tr>
</tbody>
</table>

property mass: float
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)

property moment: float
The calculated moment of this shape.

point_query(p: Tuple[float, float]) → PointQueryInfo
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: Tuple[float, float], end: Tuple[float, float], radius: float = 0) → SegmentQueryInfo

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

Return type
SegmentQueryInfo

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

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

shapes_collide(b: Shape) → ContactPointSet
Get contact information about this shape and shape b.

Return type
ContactPointSet

property space: Optional[Space]
Get the Space that shape has been added to (or None).

property surface_velocity: Vec2d
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: Transform) → BB
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.ShapeFilter(group: int = 0, categories: int = 4294967295, mask: int = 4294967295)

Bases: tuple

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 jointing 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.

<table>
<thead>
<tr>
<th>Object</th>
<th>Object Category</th>
<th>Category Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>Player</td>
<td>0b00001 (1)</td>
<td>0b1100 (4, 5)</td>
</tr>
<tr>
<td>Enemy</td>
<td>0b00010 (2)</td>
<td>0b01110 (2, 3, 4)</td>
</tr>
<tr>
<td>Player Bullet</td>
<td>0b00100 (3)</td>
<td>0b10001 (1, 5)</td>
</tr>
<tr>
<td>Enemy Bullet</td>
<td>0b01000 (4)</td>
<td>0b10010 (2, 5)</td>
</tr>
<tr>
<td>Walls</td>
<td>0b10000 (5)</td>
<td>0b11111 (1, 2, 3, 4)</td>
</tr>
</tbody>
</table>

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 the 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:

```python
>>> 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
```

**static ALL_CATEGORIES() → int**

**static ALL_MASKS() → int**

**property categories**

A bitmask of user definable categories that this object belongs to.

The category/mask combinations of both objects in a collision must agree for a collision to occur.

**count(value, /)**

Return number of occurrences of value.

**property group**

Two objects with the same non-zero group value do not collide.

This is generally used to group objects in a composite object together to disable self collisions.

**index(value, start=0, stop=9223372036854775807, /)**

Return first index of value.

Raises ValueError if the value is not present.
property mask
A bitmask of user definable category types that this object object collides with.

The category/mask combinations of both objects in a collision must agree for a collision to occur.

class pymunk.ShapeQueryInfo(shape: Optional[Shape], contact_point_set: ContactPointSet)
Bases: tuple
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.

property contact_point_set
Alias for field number 1

count(value, /)
Return number of occurrences of value.

index(value, start=0, stop=9223372036854775807, /)
Return first index of value.
Raises ValueError if the value is not present.

property shape
Shape that was hit, or None if no collision occured

class pymunk.Space(threaded: bool = False)
Bases: 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:

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

__init__(threaded: bool = False) → None
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: Union[Body, Shape, Constraint]) → None
Add one or many shapes, bodies or constraints (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: int, collision_type_b: int) → CollisionHandler

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() → CollisionHandler

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: Callable[..., None], key: Hashable, *args: Any, **kwargs: Any) → bool

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: int) → CollisionHandler

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: BB, shape_filter: ShapeFilter) → List[Shape]

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 – Bounding box
- shape_filter – Shape filter

Return type

List[Shape]

property bodies: List[Body]

A list of the bodies added to this space

property collision_bias: float

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 cpfpow(1.0f - 0.1f, 60.0f) meaning that pymunk attempts to correct 10% of error every 1/60th of a second.

..Note::

Very very few games will need to change this value.

property collision_persistence: float

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.
property collision_slop: float
    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.

property constraints: List[Constraint]
    A list of the constraints added to this space

copy() → T
    Create a deep copy of this object.

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

property damping: float
    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: SpaceDebugDrawOptions) → None
    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 pre-defined options in their x_util modules, for example pygame_util.DrawOptions.
    It's 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 function for the drawing since its meant for debugging and quick scripting.

property gravity: Vec2d
    Global gravity applied to the space.
    Defaults to (0,0). Can be overridden on a per body basis by writing custom integration functions and set it on the body: pymunk.Body.velocity_func().

property idle_speed_threshold: float
    Speed threshold for a body to be considered idle.
    The default value of 0 means the space estimates a good threshold based on gravity.

property iterations: int
    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: Tuple[float, float], max_distance: float, shape_filter: ShapeFilter) → List[PointQueryInfo]

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 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: Tuple[float, float], max_distance: float, shape_filter: ShapeFilter) → Optional[PointQueryInfo]

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: Shape) → None

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

reindex_shapes_for_body(body: Body) → None

Reindex all the shapes for a certain body.

reindex_static() → None

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: Union[Body, Shape, Constraint]) → None

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: Tuple[float, float], end: Tuple[float, float], radius: float, shape_filter: ShapeFilter) → List[SegmentQueryInfo]

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: Tuple[float, float], end: Tuple[float, float], radius: float, shape_filter: ShapeFilter) → Optional[SegmentQueryInfo]

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: Shape) → List[ShapeQueryInfo]

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

9.3. API Reference
Return type

[ShapeQueryInfo]

**property shapes:** List[Shape]

A list of all the shapes added to this space

(includes both static and non-static)

**property sleep_time_threshold:** float

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

The default value of \( \text{inf} \) disables the sleeping algorithm.

**property static_body:** Body

A dedicated static body for the space.

You don't have to use it, but many times it can be convenient to have a static body together with the space.

**step**(dt: float) → None

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. Therefore it sometimes make sense to have a little for loop around the step call, like in this example:

```python
>>> 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** – Time step length

**property threads:** int

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 won't have any effect. The default is 1 regardless if the Space was created with threaded=True, to keep determinism in the simulation. Note that Windows does not support the threaded solver.

**use_spatial_hash**(dim: float, count: int) → None

Switch the space to use a spatial hash instead of the bounding box tree.

Pymunk supports two spatial indexes. The default is an axis-aligned bounding box tree inspired by the one used in the Bullet Physics library, but caching of overlapping leaves was added to give it very good temporal coherence. The tree requires no tuning, and most games will find that they get the best performance using from the tree. The other available spatial index type available is a spatial hash, which can be much faster when you have a very large number (1000s) of objects that are all the same size. For smaller numbers of objects, or objects that vary a lot in size, the spatial hash is usually much slower. It also requires tuning (usually through experimentation) to get the best possible performance.

The spatial hash data is fairly size sensitive. dim is the size of the hash cells. Setting dim to the average collision shape size is likely to give the best performance. Setting dim too small will cause the shape to be inserted into many cells, setting it too low will cause too many objects into the same hash slot.
count is the suggested minimum number of cells in the hash table. If there are too few cells, the spatial hash will return many false positives. Too many cells will be hard on the cache and waste memory. Setting count to ~10x the number of objects in the space is probably a good starting point. Tune from there if necessary.

Parameters
- `dim` – the size of the hash cells
- `count` – the suggested minimum number of cells in the hash table

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 <class ‘CP_SPACE_DEBUG_DRAW_COLLISION_POINTS’>

DRAW_CONSTRAINTS
alias of <class ‘CP_SPACE_DEBUG_DRAW_CONSTRAINTS’>

DRAW_SHAPES
alias of <class ‘CP_SPACE_DEBUG_DRAW_SHAPES’>

__init__() → None

property collision_point_color: SpaceDebugColor
The color of collisions.
Should be a tuple of 4 ints between 0 and 255 (r,g,b,a).

Example:
```
>>> import pymunk
>>> s = pymunk.Space()
>>> b = pymunk.Body(1, 10)
>>> c1 = pymunk.Circle(b, 10)
>>> c2 = pymunk.Circle(s.static_body, 10)
>>> s.add(b, c1, c2)
>>> s.step(1)
>>> options = pymunk.SpaceDebugDrawOptions()
>>> 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, 10.0, 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(8.0, 0.0), Vec2d(-8.0, 0.0), SpaceDebugColor(r=231.0, g=76.0, b=60.0, a=255.0))
>>> options.collision_point_color = (10, 20, 30, 40)
>>> 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, 10.0, 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(8.0, 0.0), Vec2d(-8.0, 0.0), SpaceDebugColor(r=10.0, g=20.0, b=30.0, a=40.0))
```
**color_for_shape** *(shape: Shape) → SpaceDebugColor*

**property constraint_color: SpaceDebugColor**

The color of constraints.

Should be a tuple of 4 ints between 0 and 255 (r,g,b,a).

Example:

```python
>>> import pymunk
>>> s = pymunk.Space()
>>> b = pymunk.Body(1, 10)
>>> j = pymunk.PivotJoint(s.static_body, b, (0, 0))
>>> s.add(j)
>>> options = pymunk.SpaceDebugDrawOptions()
>>> s.debug_draw(options)
```

```text
draw_dot (5.0, Vec2d(0.0, 0.0), SpaceDebugColor(r=142.0, g=68.0, b=173.0, a=255.0))
```

```python
options.constraint_color = (10, 20, 30, 40)
```

```python
s.debug_draw(options)
```

```text
draw_dot (5.0, Vec2d(0.0, 0.0), SpaceDebugColor(r=10.0, g=20.0, b=30.0, a=40.0))
```

**draw_circle** *(pos: Vec2d, angle: float, radius: float, outline_color: SpaceDebugColor, fill_color: SpaceDebugColor) → None*

**draw_dot** *(size: float, pos: Vec2d, color: SpaceDebugColor) → None*

**draw_fat_segment** *(a: Vec2d, b: Vec2d, radius: float, outline_color: SpaceDebugColor, fill_color: SpaceDebugColor) → None*

**draw_polygon** *(verts: Sequence[Vec2d], radius: float, outline_color: SpaceDebugColor, fill_color: SpaceDebugColor) → None*

**draw_segment** *(a: Vec2d, b: Vec2d, color: SpaceDebugColor) → None*

**draw_shape** *(shape: Shape) → None*

**property flags: int**

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 the desired backend instead, such as pygame_util.DrawOptions or pyglet_util.DrawOptions):

```python
>>> 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()
```
```python
>>> # 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 = SpaceDebugColor(r=52, g=152, b=219, a=255)
shape_kinematic_color: SpaceDebugColor = SpaceDebugColor(r=39, g=174, b=96, a=255)

property shape_outline_color: SpaceDebugColor
The outline color of shapes.
Should be a tuple of 4 ints between 0 and 255 (r,g,b,a).

Example:

```python
>>> import pymunk

```python
>>> s = pymunk.Space()

```python
>>> c = pymunk.Circle(s.static_body, 10)

```python
>>> s.add(c)

```python
>>> options = pymunk.SpaceDebugDrawOptions()

```python
>>> 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=149.0, g=165.0, b=166.0, a=255.0))

```python
>>> options.shape_outline_color = (10,20,30,40)

```python
>>> s.debug_draw(options)

draw_circle (Vec2d(0.0, 0.0), 0.0, 10.0, SpaceDebugColor(r=10.0, g=20.0, b=30.0, a=40.0), SpaceDebugColor(r=149.0, g=165.0, b=166.0, a=255.0))

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

property transform: Transform
The transform is applied before drawing, e.g for scaling or translation.

Example:

```python
>>> import pymunk

```python
>>> s = pymunk.Space()

```python
>>> c = pymunk.Circle(s.static_body, 10)

```python
```
>>> s.add(c)
>>> options = pymunk.SpaceDebugDrawOptions()
>>> 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=149.0, g=165.0, b=166.0, a=255.0))
>>> options.transform = pymunk.Transform.scaling(2)
>>> s.debug_draw(options)
draw_circle (Vec2d(0.0, 0.0), 0.0, 20.0, 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))
>>> options.transform = pymunk.Transform.translation(2,3)
>>> s.debug_draw(options)
draw_circle (Vec2d(2.0, 3.0), 0.0, 10.0, 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))

Note: Not all transformations are supported by the debug drawing logic. Uniform scaling and translation are supported, but not rotation, linear stretching or shearing.

class pymunk.Transform(a: float = 1, b: float = 0, c: float = 0, d: float = 1, tx: float = 0, ty: float = 0)
Bases: tuple
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:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>c</td>
<td>tx</td>
</tr>
<tr>
<td>b</td>
<td>d</td>
<td>ty</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

An instance can be created in this way:

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

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

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

Or using one of the static methods like identity or translation (see each method for details).

The Transform supports the matrix multiplication operator (@) with a Transform, Vec2d or tuple as second operand, which produces a transformed Transform or Vec2d as result:

```python
>>> Transform.scaling(2) @ Transform.scaling(3)
Transform(a=6, b=0, c=0, d=6, tx=0, ty=0)
>>> Transform.scaling(2) @ Vec2d(1, 2)
Vec2d(4)
```

__matmul__(other: Tuple[float, float]) → Vec2d
__matmul__(other: Tuple[float, float, float, float, float, float]) → Transform

Multiply this transform with a Transform, Vec2d or Tuple of size 2 or 6.

Examples (Transform @ Transform):

```python
g>>> Transform() @ Transform() == Transform.identity()
True
g>>> Transform.translation(2,3) @ Transform.translation(4,5)
Transform(a=1, b=0, c=0, d=1, tx=6, ty=8)
g>>> Transform.scaling(2) @ Transform.scaling(3)
Transform(a=6, b=0, c=0, d=6, tx=0, ty=0)
g>>> Transform.scaling(2) @ Transform.translation(3,4)
Transform(a=2, b=0, c=0, d=2, tx=6, ty=8)
g>>> Transform.translation(3,4) @ Transform.scaling(2)
Transform(a=2, b=0, c=0, d=2, tx=3, ty=4)
```

Examples (Transform @ Vec2d):

```python
g>>> Transform.identity() @ Vec2d(1, 2)
Vec2d(1, 2)
g>>> Transform.scaling(2) @ Vec2d(1, 2)
Vec2d(2, 4)
g>>> Transform.translation(3,5) @ Vec2d(1, 2)
Vec2d(4, 7)
g>>> Transform.rotation(1) @ Vec2d(1, 2) == Vec2d(1, 2).rotated(1)
True
```

property a

Alias for field number 0

property b

Alias for field number 1

property c

Alias for field number 2

property d

Alias for field number 3

static identity() → Transform

The identity transform

Example:

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

Returns a Transform with this matrix:

```
| 1 0 0 |
| 0 1 0 |
| 0 0 1 |
```
\texttt{index}(\texttt{value}, \texttt{start=0, stop=9223372036854775807, /})

Return first index of value.

Raises \texttt{ValueError} if the value is not present.

\texttt{rotated}(\texttt{t: float} \rightarrow \texttt{Transform})

Rotate this Transform and return the result.

\begin{verbatim}
>>> '%.2f, %.2f, %.2f, %.2f, %.2f' % Transform.rotation(1).rotated(0.5)
'0.07, 1.00, -1.00, 0.07, 0.00'
>>> '%.2f, %.2f, %.2f, %.2f, %.2f' % Transform.rotation(1.5)
'0.07, 1.00, -1.00, 0.07, 0.00'
\end{verbatim}

\texttt{static rotation}(\texttt{t: float} \rightarrow \texttt{Transform})

A rotation transform

Example to rotate by 1 rad:

\begin{verbatim}
>>> '%.2f, %.2f, %.2f, %.2f, %.2f' % Transform.rotation(1)
'0.54, 0.84, -0.84, 0.54, 0.00'
\end{verbatim}

Returns a Transform with this matrix:

\[
\begin{bmatrix}
\cos(t) & -\sin(t) & 0 \\
\sin(t) & \cos(t) & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

\texttt{scaled}(\texttt{s: float} \rightarrow \texttt{Transform})

Scale this Transform and return the result.

Example:

\begin{verbatim}
>>> Transform.translation(3,4).scaled(2)
Transform(a=2, b=0, c=0, d=2, tx=3, ty=4)
\end{verbatim}

\texttt{static scaling}(\texttt{s: float} \rightarrow \texttt{Transform})

A scaling transform

Example to scale 4x:

\begin{verbatim}
>>> Transform.scaling(4)
Transform(a=4, b=0, c=0, d=4, tx=0, ty=0)
\end{verbatim}

Returns a Transform with this matrix:

\[
\begin{bmatrix}
s & 0 & 0 \\
0 & s & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

\texttt{translated}(\texttt{x: float, y: float} \rightarrow \texttt{Transform})

Translate this Transform and return the result.

Example: >>> Transform.scaling(2).translated(3,4) Transform(a=2, b=0, c=0, d=2, tx=6, ty=8)
**static translation** *(x: float, y: float) → Transform*

A translation transform

Example to translate (move) by 3 on x and 5 in y axis:

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

Returns a Transform with this matrix:

```
<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>y</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

**property tx**

Alias for field number 4

**property ty**

Alias for field number 5

class **pymunk.Vec2d** *(x: float, y: float)*

Bases: tuple

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

**property angle**: float

The angle (in radians) of the vector

**property angle_degrees**: float

Gets the angle (in degrees) of a vector

**convert_to_basis** *(x_vector: Tuple[float, float], y_vector: Tuple[float, float]) → Vec2d*

**count**(value, /)

Return number of occurrences of value.

**cpvrotate** *(other: Tuple[float, float]) → Vec2d*

Uses complex multiplication to rotate this vector by the other.

**cpvunrotate** *(other: Tuple[float, float]) → Vec2d*

The inverse of cpvrotate

**cross** *(other: Tuple[float, float]) → float*

The cross product between the vector and other vector

\[ v1 \cdot cross(v2) \rightarrow v1.x \cdot v2.y - v1.y \cdot v2.x \]

Returns

The cross product

**dot** *(other: Tuple[float, float]) → float*

The dot product between the vector and other vector

\[ v1 \cdot dot(v2) \rightarrow v1.x \cdot v2.x + v1.y \cdot v2.y \]

Returns

The dot product
get_angle_between(other: Tuple[float, float]) → float
Get the angle between the vector and the other in radians

Returns
The angle

get_angle_degrees_between(other: Vec2d) → float
Get the angle between the vector and the other in degrees

Results
The angle (in degrees)

get_dist_sqrd(other: Tuple[float, float]) → float
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: Tuple[float, float]) → float
The distance between the vector and other vector

Returns
The distance

get_length_sqrd() → float
Get the squared length of the vector. If the squared length is enough it is more efficient to use this method instead of first calling get_length() or access .length and then do a x**2.

>>> v = Vec2d(3, 4)
>>> v.get_length_sqrd() == v.length**2
True

Returns
The squared length

index(value, start=0, stop=9223372036854775807, /)
Return first index of value.
 Raises ValueError if the value is not present.

property int_tuple: Tuple[int, int]
The x and y values of this vector as a tuple of ints. Uses round() to round to closest int.

>>> Vec2d(0.9, 2.4).int_tuple
(1, 2)

interpolate_to(other: Tuple[float, float], range: float) → Vec2d

property length: float
Get the length of the vector.

>>> Vec2d(10, 0).length
10.0
>>> '%.2f' % Vec2d(10, 20).length
'22.36'
Returns
The length

normalized() → Vec2d
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

normalized_and_length() → Tuple[Vec2d, float]
Normalize the vector and return its length before the normalization

Returns
The length before the normalization

static ones() → Vec2d
A vector where both x and y is 1

>>> Vec2d.ones()
Vec2d(1, 1)

perpendicular() → Vec2d

perpendicular_normal() → Vec2d

projection(other: Tuple[float, float]) → Vec2d
Project this vector on top of other vector

rotated(angle_radians: float) → Vec2d
Create and return a new vector by rotating this vector by angle_radians radians.

Returns
Rotated vector

rotated_degrees(angle_degrees: float) → Vec2d
Create and return a new vector by rotating this vector by angle_degrees degrees.

Returns
Rotated vector

scale_to_length(length: float) → Vec2d
Return a copy of this vector scaled to the given length.

>>> '%.2f, %.2f' % Vec2d(10, 20).scale_to_length(20)
'8.94, 17.89'

static unit() → Vec2d
A unit vector pointing up

>>> Vec2d.unit()
Vec2d(0, 1)

property x
Alias for field number 0

property y
Alias for field number 1
**static zero() → Vec2d**
A vector of zero length.

```python
>>> Vec2d.zero()
Vec2d(0, 0)
```

pymunk.chipmunk_version: str = '<Mock object>-0593976ef47fcb3957166bd342f6b2bafe4d0e44'
The Chipmunk version used with this Pymunk version.

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<githubcommitofchipmunk> 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/viblo/Chipmunk2D included with Pymunk.

pymunk.moment_for_box(mass: float, size: Tuple[float, float]) → float
Calculate the moment of inertia for a solid box centered on the body.
size should be a tuple of (width, height)

pymunk.moment_for_circle(mass: float, inner_radius: float, outer_radius: float, offset: Tuple[float, float] = (0, 0)) → float
Calculate the moment of inertia for a hollow circle
(A solid circle has an inner radius of 0)

pymunk.moment_for_poly(mass: float, vertices: Sequence[Tuple[float, float]], offset: Tuple[float, float] = (0, 0),
radius: float = 0) → float
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: float, a: Tuple[float, float], b: Tuple[float, float], radius: float) → float
Calculate the moment of inertia for a line segment
The endpoints a and b are relative to the body

pymunk.version: str = '6.3.0'
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).

## 9.4 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).
9.4.1 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.

**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

![Diagram of various shapes and joints](image)

**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
9.4.2 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:

```bash
>cd examples
>python 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
- camera.py
- constraints.py
- contact_and_no_flipy.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
- planet.py
- platformer.py
- playground.py
- point_query.py
- py2exe_setup__basic_test.py
arrows.py

Source: examples/arrows.py

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

Source: examples/balls_and_lines.py

This example lets you dynamically create static walls and dynamic balls.
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.
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.
box2d_pyramid.py

Source: examples/box2d_pyramid.py

Remake of the pyramid demo from the box2d testbed.
**box2d_vertical_stack.py**

Source: examples/box2d_vertical_stack.py

Remake of the vertical stack demo from the box2d testbed.

---

**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.
Basic showcase on how the transform property on SpaceDebugDrawOptions can be used as a camera to allow panning. Use arrows to move the camera.

Source: examples/camera.py

**camera.py**

Basic showcase on how the transform property on SpaceDebugDrawOptions can be used as a camera to allow panning. Use arrows to move the camera.
Use Arrows (up, down, left, right) to move the camera, and a and z to zoom in / out.

constraints.py

Source: examples/constraints.py
Pymunk constraints demo. Showcase of all the constraints included in Pymunk.
Adapted from the Chipmunk Joints demo: https://github.com/slembcke/Chipmunk2D/blob/master/demo/Joints.c
GrooveJoint is similar to a PivotJoint, but with a linear slide.

One of the anchor points is a line segment that the pivot can slide in instead of being fixed.

**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.
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.
This example shows how you can copy, save and load a space using pickle.

**copy_and_pickle.py**

Source: examples/copy_and_pickle.py

This example shows how you can copy, save and load a space using pickle.
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

**damped_rotary_spring_pointer.py**

Source: examples/damped_rotary_spring_pointer.py

This example showcase an arrow pointing or aiming towards the cursor.
This is an example on how the autogeometry can be used for deformable terrain.
A very basic flipper game.

flipper.py

Source: examples/flipper.py

A very basic flipper game.
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.
kivy_pymunk_demo

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.
newtons_cradle.py

Source: examples/newtons_cradle.py

A screensaver version of Newton’s Cradle with an interactive mode.

Press left mouse button and drag to interact
Press R to reset, any other key to quit
planet.py

Source: examples/planet.py

Showcase of planets/satellites (small boxes) orbiting around a star.

Uses a custom velocity function to manually calculate the gravity, assuming the star is in the middle and is massive enough that the satellites does not affect it.

This is a port of the Planet demo included in Chipmunk.
**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.

---

**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.
point_query.py

Source: examples/point_query.py

This example showcase point queries by highlighting the shape under the mouse pointer.
**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
**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

**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.

**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 pygame_util.draw()

**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.

**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.
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
tank.py

Source: examples/tank.py

Port of the Chipmunk tank demo. Showcase a topdown tank driving towards the mouse, and hitting obstacles on the way.
Use the mouse to drive the tank, it will follow the cursor.

**usingSprites.py**

Source: examples/usingSprites.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.
This example is a clone of the using_sprites example with the difference that it uses pyglet instead of pygame to showcase sprite drawing.
9.5 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!
### 9.5.1 Games

<table>
<thead>
<tr>
<th>Image</th>
<th>Description</th>
</tr>
</thead>
</table>
| ![Guide The Ball](image1) | **Guide The Ball**  
A game combining Pymunk with Kivy that runs on Android mobiles!  
In Guide The Ball the task is to guide a ball through 19 exciting and challenging levels! |
| ![My Sincerest Apologies](image2) | **My Sincerest Apologies**  
Made by The Larry and Dan show (mauve, larry). Retrieved 2018-10-25  
Winner of PyWeek 24 (Overall Team Entry)  
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 you, a simple fighting robot. It’s your job to clean out each area!  
See Daniel Popes teardown [here](#) for additional details |
| ![Beneath the Ice](image3) | **Beneath the Ice**  
Made by Team Chimera (mit-mit, Lucid Design Ar). Retrieved 2016-09-25  
Winner of PyWeek 22 (Overall Team Entry)  
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! |
| ![Invisipin](image4) | **Invisipin**  
Made by Tee. Retrieved 2016-01-25  
Winner of PyWeek 20 (Overall Individual Entry)  
A Pachinko-like puzzle game. Play some balls and watch their movement carefully (i.e. collect data) to reconstruct the board! |
| ![Angry Birds in Python](image5) | **Angry Birds in Python**  
Made by Estevao Fonseca. Retrieved 2016-10-30  
An Angry Birds game written in python using pygame and pymunk |
| ![SubTerrex](image6) | **SubTerrex**  
Made by Paul Paterson. Retrieved 2016-01-25  
A cave exploration game where you explore caves by descending into them on ropes. |
## 9.5.2 Non-Games

### Bouncing Balls, Beautiful Patterns

*made by Alessandro Giusti. Retrieved 2022-01-25*

Satisfying simulations of bouncing balls obeying physical laws; for a fleeting moment during the simulation, the balls pass through a beautiful regular arrangement.

Each sequence is obtained by joining two simulations, both starting from the time in which the balls are arranged regularly. One simulates forward in time, one backwards.

### manim-physics

*made by pdcxs, Matheart & Iced-Tea3. Retrieved 2021-07-05*

This is a 2D physics simulation plugin that allows you to generate complicated scenes in various branches of Physics such as rigid mechanics, electromagnetism, wave etc.

### Computer Vision and Physics

*made by Amirabbas Asadi. Retrieved 2021-06-25*

**Just playing with OpenCV + a hand tracking model + a physics engine :)**

A very cool blog post (and video) explaining how to combine computer vision and a little physics to create a simple environment for Augmented Reality. In the blog Amirabbas Asadi shows how OpenCV, Mediapipe and Pymunk can be combined to make an app where simulated balls bounce on the hand of the user.
9.5.3 Papers / Science

Pymunk has been used or referenced in a number of scientific papers.

List of papers which has used or mentioned Pymunk:


43. Wong, Eric C. “Example Based Hebbian Learning may be sufficient to support Human Intelligence.” bioRxiv (2019): 758375.


List last updated 2022-10-03. If something is missing or wrong, please contact me!

Cite Pymunk

If you use Pymunk in a published work and want to cite it, below is a citation.cff example. If you prefer you can also visit the Github repository of Pymunk and use the shorthand “cite this repository” feature: https://github.com/viblo/pymunk

Feel free to modify to fit your style. (Make sure to modify the version number if included.):

```cff

cff-version: 1.2.0
message: "If you use this software and want to cite it, please do so as below."
authors:
  - family-names: "Blomqvist"
    given-names: "Victor"
title: "Pymunk"
abstract: "A easy-to-use pythonic rigid body 2d physics library"
version: 6.3.0
date-released: 2022-11-04
url: "https://pymunk.org"
```

9.6 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. Also take a look at the external tutorials. While they are not created by the Pymunk creator they can still give a good start in how to use Pymunk.
9.6.1 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.
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:

```python
>>> import pymunk
```

More information on installation can be found here: [Installation](http://chipmunk-physics.net/release/ChipmunkLatest-Docs/)

If it doesn’t 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](http://chipmunk-physics.net/release/ChipmunkLatest-Docs/)

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 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/](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](http://chipmunk-physics.net/release/ChipmunkLatest-Docs/)

Anyway, we are now ready to write some code:

```python
import sys
import pygame
import pymunk
```

(continues on next page)
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:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit(0)
            elif event.type == pygame.KEYDOWN and event.key == pygame.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 each time called.

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.

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 = 3
    radius = 25
    body = pymunk.Body() # 1

(continues on next page)
x = random.randint(120, 300)
body.position = x, 50  # 2
shape = pymunk.Circle(body, radius)  # 3
shape.mass = mass  # 4
shape.friction = 1
space.add(body, shape)  # 5
return shape

1. We first create the body of the ball.
2. And we set its position
3. And in order for it to collide with things, it needs to have one (or many) collision shape(s).
4. All bodies must have their moment of inertia set. In most cases its easiest to let Pymunk handle calculation from shapes. So we set the mass of each shape, and then when added to space the body will automatically get a proper mass and moment set. Another option is to set the density of each shape, or its also possible to set the values directly on the body (or even adjust them afterwards).
5. To make the balls roll we set friction on the shape. (By default its 0).
6. Finally we add the body and shape to the space to include it in our simulation. Note that the body must always be added to the space before or at the same time as any shapes attached to it.

Now that we can create balls we want to display them. Either we can use the built in pymunk_util package do 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 for example a polished game most probably will want to make its own drawing code.

If we want to draw manually, our draw function could look something like this:

```python
def draw_ball(screen, ball):
    p = int(ball.body.position.x), 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):

```python
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. It first needs to be imported, and next we have to create a DrawOptions object with the options (what surface to draw on in the case of Pygame):

```python
import pymunk.pygame_util
...  
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:

```python
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
random.seed(1)  # make the simulation the same each time, easier to debug
import pygame
import pymunk
import pymunk.pygame_util

#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 == pygame.QUIT:
                sys.exit(0)
            elif event.type == pygame.KEYDOWN and event.key == pygame.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()
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 let’s 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:

```python
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)
    l1.friction = 1 # 3
    l2.friction = 1
    space.add(body, l1, l2) # 4
    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. Many times it’s easier to use the already existing static body in the space (`space.static_body`), but we will make the L shape dynamic in just a little bit.

2. A line shaped shape is created here.

3. Set the friction.

4. Again, we only add the segments, not the body to the space.

Since we use `Space.debug_draw` to draw the space we don’t 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:

```python
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.

```python
def to_pygame(p):
    """Small helper to convert pymunk vec2d to pygame integers""
    return round(p.x), round(p.y)
```

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.

```python
import sys, random
random.seed(1) # make the simulation the same each time, easier to debug
import pygame
```
import pymunk
import pymunk.pygame_util

#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 == pygame.QUIT:
                sys.exit(0)
            elif event.type == pygame.KEYDOWN and event.key == pygame.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()
Joints (1)

A static L shape is pretty boring. So let’s 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().

```python
def add_L(space):
    rotation_center_body = pymunk.Body(body_type=pymunk.Body.STATIC)  # 1
    rotation_center_body.position = (300, 300)

    body = pymunk.Body()
    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)
    l1.friction = 1
    l2.friction = 1
    l1.mass = 8  # 2
    l2.mass = 1
    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 therefore it cannot be a static body. Here we see the benefit of setting the mass on the shapes instead of the body, no need to figure out how big the moment should be, and Pymunk will automatically calculate the center of gravity.

3. A pin joint allows two objects to pivot about a single point. In our case one of the objects will be stuck to the world.

Joints (2)

In the previous part we added a pin joint, and now it’s time to constrain the rotating L shape to create a more interesting simulation. In order to do this we modify the add_L() function:

```python
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()
    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)
    l1.friction = 1
    l2.friction = 1
    l1.mass = 8  # 4
```

(continues on next page)
l2.mass = 1

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

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.

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 bellow 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 will 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:

```python
import sys, random
random.seed(1) # make the simulation the same each time, easier to debug
import pygame
import pymunk
import pymunk.pygame_util
```
```python
def add_ball(space):
    """Add a ball to the given space at a random position""
    mass = 3
    radius = 25
    inertia = pymunk.moment_for_circle(mass, 0, radius, (0,0))
    body = pymunk.Body(mass, inertia)
    x = random.randint(120,300)
    body.position = x, 50
    shape = pymunk.Circle(body, radius, (0,0))
    shape.friction = 1
    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)
    l1.friction = 1
    l2.friction = 1
    l1.mass = 8
    l2.mass = 1

    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 == pygame.QUIT:
            sys.exit(0)
        elif event.type == pygame.KEYDOWN and event.key == pygame.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 > 550:
            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()
Pymunk Basics

Created by Ear of Corn Programming. Retrieved 2021-03-07

Youtube user Ear of Corn Programming has a comprehensive introduction to Pymunk together with Pygame over several videos.

Simulating physics in Python

Created by Clear Code. Retrieved 2020-07-09

Youtube user Clear Code has nice introduction to using Pymunk together with Pygame.

Breakout game in Python, Pyglet and Pymunk

Created by Attila Toth. Retrieved 2019-04-14

Youtube user Attila Toth has created a series of Youtube videos that guide through creating a Breakout like game, similar to the breakout example that is included with Pymunk. If you like his first series (Pymunk physics in Pyglet) then this is a great followup.

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.

9.7 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.

Note that the the benchmarks are not yet updated for Pymunk 6.0, but tests look promising.

9.7.1 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:

\[
t += b\text{.position.x + b\text{.position.y + b\text{.angle}}}
\]

(see pymunk-get.py)

Running it is simple, for example like this for pymunk 4.0:
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:

```python
def f(b, dt):
    b.position += (1, 0)

s.step(0.01)
```

(see `pymunk-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. 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:**

<table>
<thead>
<tr>
<th>Pymunk</th>
<th>CPython 2.7.12</th>
<th>CPython 3.5.2</th>
<th>Pypy 5.4.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>2.1s</td>
<td>2.2s</td>
<td>0.36s</td>
</tr>
<tr>
<td>5.0</td>
<td>4.3s</td>
<td>4.5s</td>
<td>0.37s</td>
</tr>
<tr>
<td>4.0</td>
<td>1.0s</td>
<td>0.9s</td>
<td>0.52s</td>
</tr>
</tbody>
</table>

**Pymunk-Callback:**

<table>
<thead>
<tr>
<th>Pymunk</th>
<th>CPython 2.7.12</th>
<th>CPython 3.5.2</th>
<th>Pypy 5.4.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>5.7s</td>
<td>6.8s</td>
<td>1.1s</td>
</tr>
<tr>
<td>5.0</td>
<td>6.5s</td>
<td>7.3s</td>
<td>1.0s</td>
</tr>
<tr>
<td>4.0</td>
<td>5.1s</td>
<td>6.5s</td>
<td>4.5s</td>
</tr>
</tbody>
</table>

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.

9.7. Benchmarks
9.7.2 Compared to Other Physics Libraries

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:

```python
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:

<table>
<thead>
<tr>
<th></th>
<th>CPython 3.5.3</th>
<th>Pypy 5.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pymunk 5.3</td>
<td>2.14s</td>
<td>0.33s</td>
</tr>
<tr>
<td>Cymunk 20170916</td>
<td>0.41s</td>
<td>(10.0s)</td>
</tr>
</tbody>
</table>

Collision-Callback:

<table>
<thead>
<tr>
<th></th>
<th>CPython 3.5.3</th>
<th>Pypy 5.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pymunk 5.3</td>
<td>3.71s</td>
<td>0.58s</td>
</tr>
<tr>
<td>Pymunk 20170916</td>
<td>0.95s</td>
<td>(7.01s)</td>
</tr>
</tbody>
</table>

(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.

## 9.8 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 library built around the C-library Chipmunk2D, which provides almost all of the base functionality around the physics simulation such as collision detection, impulse solving etc. Basically it runs the simulation, and Pymunk calls it with input, and receives the result.

To wrap Chipmunk Pymunk uses CFFI in API mode. On top of the CFFI wrapping is a handmade pythonic layer to make it nice to use from Python code.

### 9.8.1 Impulse Solver Algorithm

Pymunk in itself only performs a minimum amount of physics calculation, instead those are handled by the underlying C-library Chipmunk 2D. Chipmunk2D (and therefore Pymunk) uses standard Euler to perform the integration.

Scott/slembcke (the creator of Chipmunk2D), wrote this to describe the method used on the Chipmunk2D Forum:

Chipmunk works like this:

1. Integrate the positions of everything and finds colliding pairs.
2. Pre-calculate a number of properties of the contacts and joints. (mass properties, bounce velocities, etc)
3. Integrate the velocities of everything.
4. Run a number of solver iterations, to fix velocity constraints.

Case 1, a box sitting on the ground:

1. The box is at rest, it’s velocity is (very near) zero, it’s position doesn’t change. Generate a contact with the ground.
2. Precalculate the contact properties, if elasticity is set, the “bounce” velocity is calculated now (as 0).
3. Integrate the velocity, gravity makes the object accelerate downwards.
4. Solve the contact. Velocity should converge back to 0. If elasticity was set, it will be handled correctly without resorting to threshold velocities.

#1 is really the most important part. If you were to update the velocity before the position, it would cause the box to move itself into a position where it would intersect the ground. While Chipmunk has to solve these overlaps anyway, avoiding them seems desirable. Another very useful property is that when cpSpaceStep() returns, all of the collision detection data structures are up to date. No need to reindex them twice in a single frame if you want to make queries.
9.8.2 Collision Detection Algorithm

Just as the impulse solver, the collision detection is also handled by the underlying C-library Chipmunk2D.

Chipmunk uses GJK/EPA to find collisions between the tricky cases (e.g. polygons, segment shapes). There is a blog post here with more details.

9.8.3 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.

Disadvatages (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.

9.8.4 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 I've done in some cases. I've also got a share amount of problems when stuff didn't 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 doesn't affect pymunk. If a bugfix, some optimization or whatever is done in chipmunk that doesn't 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 don't 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 then 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 wraper 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.

9.8.5 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 documentation generation extension and a customized setup.py file to allow compilation of Chipmunk.

The low level chipmunk bindings are located in the file pymunk_extension_build.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 released 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.

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.

pymunk/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.

### 9.8.6 Tests

There are a number of unit tests included in the `pymunk.tests` package. Not exactly all the code is tested, but most of it (at the time of writing its about 85% of the core parts).

The tests can be run by calling the module

```bash
> python -m pymunk.tests
```

It's possible to control which tests to run, by specifying a filtering argument. The matching is as broad as possible, so `UnitTest` matches all the unit tests, `test_arbiter` all tests in `test_arbiter.py` and `testRestitution` matches the exact `testRestitution` test case

```bash
> python -m pymunk.tests -f testRestitution
```

To see all options to the tests command use `-h`

```bash
> python -m pymunk.tests -h
```

Since the tests cover even the optional parts, you either have to make sure all the optional dependencies are installed, or filter out those tests.

### 9.8.7 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, let's 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 `pymunk_extension_build.py`. If its not just add it.

```c
cpBool cpBodyIsSleeping(const cpBody *body);
```

Then to make it easy to use we want to create a python method that looks nice:

```python
def is_sleeping(body):
    return cp.cpBodyIsSleeping(body._body)
```

Now we are ready with the mapping and ready to use our new method.

### 9.8.8 Weak References and free 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 its important that the corresponding c side memory is deallocated properly when not needed anymore, usually when the Python side object is garbage collected. Most Pymunk objects use `ffi.gc` with a custom free function to do this. Note that the order of freeing is very important to avoid errors.
9.9 Changelog

9.9.1 Pymunk 6.3.0 (2022-11-04)

Build wheel for CPython 3.11!
This is a minor update with changes to be build pipe to build wheels for CPython 3.11. Some internal parts have been rewritten as well.

Changes:
- Update callbacks implementation to the cffi recommended way
- Improve asserts to catch errors earlier
- Improve type hints
- Build wheels for more targets
- Remove experimental body._id

9.9.2 Pymunk 6.2.1 (2021-10-31)

Build wheel for CPython 3.10!
This is a minor update with changes to the build pipe to build wheels for more cases, notably the recently released CPython 3.10.

Changes:
- Use pyproject.toml
- Require CFFI 1.15 to make sure wheels are build ok on Apple ARM64/M1.
- Doc improvements
- Build wheels for more targets

9.9.3 Pymunk 6.2.0 (2021-08-25)

Improved transforms for debug drawing!
This release contains enhancements to transform usage with debug drawing, and an update to latest git version of Chipmunk. It also contains a new example of how gravity in the center could be implemented.

Changes:
- Updated Chipmunk to latest git version
- Updated debug draw to support rotation, and fixed scaling of constraints
- New example of “planet” gravity (ported from Chipmunk)
- Fixed potential corner case bug in garbage collection logic
9.9.4 Pymunk 6.1.0 (2021-08-11)

Transforms for debug drawing!

The main improvement in this release is that it's now possible to set a Transform on the SpaceDebugDrawOptions object, which is applied before anything is drawn. This works in all the debug draw implementation, e.g. for pygame. In this way it's possible to easily implement features such as camera panning easily for debug draw code. See the new camera.py example for an example of this.

Changes:

- Added transform property to SpaceDebugDrawOptions.
- Extended Transform to allow allow matrix multiplication using @, either with another Transform or with a Vec2d.
- Improved error handling when adding objects to a space.
- Improved docs.

9.9.5 Pymunk 6.0.0 (2020-12-07)

Typehints, dropped Python 2, and Vec2d rework and wrapping upgrade!

This release is a very big update to Pymunk, with a number of breaking API changes. It is likely that most users of Pymunk that upgrade will need to do some changes to work, but in the majority of cases the changes should be minor.

Highlights - Major changes:

- Python 3.6 or newer required. Support for older Pythons including 2.7 has been dropped.
- Type hints added. Type hints have been added for all public interfaces.
- Vec2d (the vector class) has been completely overhauled. It is now an immutable subclass of NamedTuple, with a streamlined API interface. See below for details.

Vec2d changes:

- Vec2d no longer accept objects that have .x and .y properties, but do not support __getitem__ for [0] & [1] in the constructor. If you have such an objects, rewrite Vec2d(myobj) to Vec2d(myobj.x, myobj.y).
- Vec2d is now Immutable.
  - removed __setitem__ (you can not do Vec2d(1,2)[1] = 3 anymore).
  - not possible to set the length property. Vec2d(1,2).length = 10, instead use Vec2d(1,2).scale_to_length(10).
  - removed Vec2d.get_length method (use the length property instead).
  - removed Vec2d.rotate() method. use Vec2d.rotated() instead.
  - removed Vec2d.rotate_degrees() method. use Vec2d.rotated_degrees instead.
  - not possible to set the angle property (Vec2d(1,2).angle = 3.14). Use Vec2d.rotated() instead.
  - removed Vec2d.get_angle method (use the angle property instead).
  - not possible to set the angle_degrees property (Vec2d(1,2).angle_degrees = 180). Use Vec2d.rotated_degrees instead.
  - removed Vec2d.get_angle_degrees method (use the angle_degrees property instead)
  - removed Vec2d.normalize_return_length method (use Vec2d.length and Vec2d.normalized(), or the new Vec2d.normalized_and_length method).
• Removed __iadd__, __isub__, __imul__, __ifloordiv__ and __itruediv__.

• Removed __nonzero__ magic. This never worked in Python 3, and was not included in any tests.

• Removed __pow__ and __rpow__ magic. Its no longer possible to do Vec2d(1,2)**2, instead you need to do the calculation manually.

• Removed __invert__ magic. Its no longer possible to do ~Vec2d(1,2).

• Removed __mod__ and __divmod__ magic. Its no longer possible to do Vec2d(1,2) % 2 or divmod(Vec2d(1,2), 2).

• Removed bit operations right shift, left shift, or, and, xor. (<<, >>, |, &, ^).

• Changed abs(Vec2d(1,2)) to return the expected vector length instead of Vec2d(abs(x), abs(y)).

• Vec2d now only support addition with other Vec2ds or tuples (sequences) of length 2.

• Vec2d now only support subtraction with other Vec2ds or tuples (sequences) of length 2.

• Vec2d now only support multiplicaton with ints and floats.

• Vec2d now only support division by ints and floats. Note that reverse division is not supported, i.e. 1 / Vec2d(1,2).

• Vec2d now only support floor division (//) by ints and floats. Note that reverse floor division is not supported, i.e. 1 // Vec2d(1,2).

• Improved error checking in Vec2d when an operator (magics like __add__) is used with incompatible types.

• Removed option to create a zero Vec2d with empty constructor. Vec2d() should be replaced with Vec2d.zero().

• Made Vec2d a subclass of NamedTuple.
  – Vec2ds has to be constructed with separate x and a y values.
  – Vec2d((1,2)) can be changed to Vec2d(*((1,2))).
  – Vec2d(Vec2d(1,2)) can be changed to Vec2d(*Vec2d(1,2)).
  – Vec2d() can be changed to Vec2d(0,0) or Vec2d.zero().
  – Vec2d(1,2) is no longer equal to [1,2] since they are of different types. (but Vec2d(1,2) == (1,2) is still true)

• Relaxed get_angle_between, convert_to_basis, cpvrotate and cpunvrotate to accept tuples of size 2 as arguments just like most other methods on Vec2d.

General Changes:

• add_collision_handler(a,b) and add_collision_handler(b,a) will return the same handler. Issue #132.

• Bodies used by shapes must be added to the space before (or at the same time) the shape is added. This change will help users of Pymunk uncover bugs, and it should be straightforward to fix old code.

• Python 3.6+ required. If you use a older Python, please continue to use the 5.x series of Pymunk until its possible to upgrade.

• Space.add() and Space.remove() no longer accept lists of objects (shapes, bodies or constraints), only the objects directly. Existing code can be updated to unpack the arguments: space.add(list_of_stuff) becomes space.add(*list_of_stuff).

• ShapeFilter.ALL_MASKS and CATEGORIES changed to static methods. ShapeFilter.ALL_MASKS() and ShapeFilter.CATEGORIES() becomes ShapeFilter.ALL_MASKS().
• Note: a tuple of 4 numbers are required when specifying a color (or use the `SpaceDebugColor` class directly). During testing it was found that some demos used a tuple of 3 instead which does not work in Pymunk 6.0 (or earlier version).

• Return a `PointQueryInfo` object from `Shape.point_query`, not the previous `(distance, PointQueryInfo)` tuple. Code that need the distance can access it from `PointQueryInfo.distance`.

• Removed `pymunk.inf`. Use standard Python `float('inf')` instead.

• Renamed package `pymunk.constraint` to `pymunk.constraints`. Code that imported the previous name should be updated to import from the new name instead.

• Changed `pygame_util.positive_y_is_up` default value to `False`. Existing code dependent on the old default should set the desired value (`True`). For new code it might be better to instead make the Pymunk simulation behave like the native pygame coordinates. See examples in examples folder for examples.

• It is now expected that places functions expecting a `Vec2d` or tuple of length 2 already are a tuple (or `Vec2d`). Previously a conversion happed by calling `tuple(argument)`. To fix old code simply wrap the argument in `tuple(...)` (Note: Due to no type checks a list of length 2 might also work, however, this is not supported and can change any time.

• `BB` base class changed to `NamedTuple`. They now has to be constructed with `left`, `bottom`, `right`, `top` as separate arguments.

• Repr of `BB` will return `BB(left=1, bottom=5, right=20, top=10)` instead of `(1, 5, 20, 10)`.

• `BB` is now immutable.

• New callbacks on `Constraint` object, `pre_solve` and `post_solve`, which can be used to run a function just before or after the solver on the constraint.

• Added code to make Pymunk work without extra config in PyInstaller, py2exe and probably other bundlers as well.

• Debug logging added to easier understand c memory issues. Uses logging.debug so should be easy to work around.

Minor changes unlikely to affect existing code:

• Removed `pymunk.chipmunk_path`.

• Changed `Shape.sensor` type to bool (from int).

• Add check that pickled objects were pickled by the same Pymunk version as the code loading it. The internal pickled format can change between major, minor and point releases of Pymunk.

• Slight change of format of `pymunk.chipmunk_version` version string.

• Small change to make the collision handler functions (begin, pre_step...) return the function assigned, not the wrapped function.

• Removed extra `*args` and `**kwargs` arguments to `CollisionHandler.__init__` method.

• Pymunk source code formatted with black & isort.
• `moment_for_poly()` and `area_for_poly()` now expects a Sequence (tuple/list like object) of tuples of length 2.

• Added default value of argument `point` to `apply_force_at_local_point`.

• Removed default value of argument `point` from `apply_impulse_at_world_point`. Just specify `point = (0, 0)` to mimic the old default.

• Added many asserts to check that whenever a tuple of length 2 or Vec2d is expected the length of the tuple is 2. Working code is unlikely affected, but bugs will be easier to find.

Behind the scenes:
• In order to allow adding some advanced features that are not available in Chipmunk today the method used to call C-code has changed to CFFI API mode. In addition to easier expansion it also provides increased performance.

**9.9.6 Pymunk 5.7.0 (2020-09-16)**

Fix release
This release contains a bunch of smaller fixes and improvements.

Changes:
• Fixed issue with PyInstaller onefile.
• Improved performance of Vec2d creation. Thanks Mikhail Simin!
• Handle debug drawing of springs with 0 length.
• Made bodies and constraints ordered when accessed from the space.
• Added `Space.use_spatial_hash` function to enable use of Spatial hash as its spatial index which can improve performance when there’s lots of similarly sized objects.
• Fixed case when Vec2d.projection get a tuple as other paramter.
• Fixed ZeroDivisionError for Vec2d.projection. Thanks Mohamed Saad Ibn Seddik!
• Fixed return type of Shape.center_of_gravity property (now returns Vec2d instead of cdata).
• Fixed issue when installing dev dependencies.
• Added chipmunk tank example (available in examples folder).
• Improved docs.

Heads up! A major update to Pymunk is on the way that will be released as Pymunk 6.0. It will contain big changes, some of them very API breaking, and it will also drop support for Python 2.

**9.9.7 Pymunk 5.6.0 (2019-11-02)**

Fix to avoid incompatible CFFI version
The main goal of this release is to ensure a compatible version of CFFI is installed when installing Pymunk though pip. Unfortunately there is a problem on Linux with CFFI 1.13.1. (Later and earlier versions will work fine)

Changes:
• Added a requirement on CFFI to not be 1.13.1 (since 1.13.1 doesnt work).
• Update cffi definitions to prevent deprecation warning in latest cffi.
• Added normal property to Arbiter object.
• Remove compiled docs from committed code.
• Removed build/test of CPython 3.4 from Travis and Appveyor configs since its not supported anymore.
• Update pyglet examples to work with pyglet 1.4.
• Fixed minor issue in platformer example.
• Improved docs.

9.9.8 Pymunk 5.5.0 (2019-05-03)

Updated Chipmunk version, FreeBSD, Android/Termux support and more!
This release contains a number of improvements. Chipmunk was updated to the latest version, and then a number of unmerged PRs were merged in. (The Chipmunk git repo is quite dead, so Pymunk will include unmerged PRs after manual review). Another major improvement is that now Pymunk can run on FreeBSD. It was also tested on Termux on Android, and several improvements to the installation process has been included. A bunch of smaller fixes are also included.

Changes:
• Update Chipmunk to 7.0.2 + unmerged PRs
• Pymunk can be installed and run on FreeBSD
• Pymunk can be installed and run on Termux on Android
• Fix debug drawing of polygons with radius
• Improved debug drawing of segments on pygame
• Fix problem when installing without wheel package installed
• New Constraints demo
• Improved docs

9.9.9 Pymunk 5.4.2 (2019-01-07)

Fix for chipmunk.dll load troubles on windows
This release fixes a problem on Windows that made the chipmunk.dll file depend on libwinpthread-1.dll which happened in Pymunk 5.4.1 because of the new build setup. The fix means that for now the threaded solver is disabled on Windows. In practice this should not be a big problem, the performance benefit of the threaded solver on a desktop running windows is unclear.

Changes:
• Disable threaded solver on Windows.
9.9.10 Pymunk 5.4.1 (2018-12-31)

Improved packaging
This release consists of a number of fixes to the packaging of Pymunk. One fix that will allow building for conda, and a number of changes to build binary wheels on linux.

Changes:
- Fixes to help Pymunk work with freezers such as cx_Freeze.
- Better wheels, now they contain the proper tags
- Fix problems using custom CFLAGS when compiling chipmunk

Enjoy!

9.9.11 Pymunk 5.4.0 (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!

9.9.12 Pymunk 5.3.2 (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
9.9.13 Pymunk 5.3.1 (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.

9.9.14 Pymunk 5.3.0 (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!

9.9.15 Pymunk 5.2.0 (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)
9.9.16 Pymunk 5.1.0 (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!

9.9.17 Pymunk 5.0.0 (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!

9.9.18 Pymunk 4.0.0 (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 dont 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. It’s 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 do 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!

**9.9.19 Pymunk 3.0.0 (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!

9.9.20 Pymunk 2.1.0 (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!

9.9.21 Pymunk 2.0.0 (2011-09-04)

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

New goodies in this release comes mainly form 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 msvcrtd.dll on windows now.
- Removed dependency on setuptools
- Minor updates on other API, added some missing properties and methods.

9.9.22 Pymunk 1.0.0 (2010-07-16)

9.9.23 Pymunk 0.8.3 (2009-07-26)

9.9.24 Pymunk 0.8.2 (2009-04-22)

9.9.25 Pymunk 0.8.1 (2008-11-02)

9.9.26 Pymunk 0.8 (2008-06-15)

First public release on Pypi.

9.9.27 Pymunk 0.1 (2007-08-01)

First public release. On the Pyweek game competition forum, and later used in our entry in Pyweek 5.

9.10 License

Copyright (c) 2007-2022 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
TEN

INDICES AND TABLES

• genindex
• modindex
• search
a
pymunk.autogeometry, 26

c
pymunk.constraints, 29

m
pymunk.matplotlib_util, 51

p
pymunk, 63
pymunk.pygame_util, 54
pymunk.pyglet_util, 59

v
pymunk.vec2d, 46
Symbols

__abs__() (pymunk.vec2d.Vec2d method), 47
__add__() (pymunk.vec2d.Vec2d method), 47
__floordiv__() (pymunk.vec2d.Vec2d method), 47
__init__() (pymunk.Arbiter method), 63
__init__() (pymunk.Body method), 66
__init__() (pymunk.Circle method), 71
__init__() (pymunk.CollisionHandler method), 74
__init__() (pymunk.ContactPoint method), 75
__init__() (pymunk.ContactPointSet method), 75
__init__() (pymunk.Poly method), 76
__init__() (pymunk.Segment method), 80
__init__() (pymunk.Shape method), 84
__init__() (pymunk.Space method), 88
__init__() (pymunk.SpaceDebugDrawOptions method), 95
__init__() (pymunk.autogeometry.PolylineSet method), 27
__init__() (pymunk.constraints.Constraint method), 30
__init__() (pymunk.constraints.DampedRotarySpring method), 31
__init__() (pymunk.constraints.DampedSpring method), 33
__init__() (pymunk.constraints.GearJoint method), 34
__init__() (pymunk.constraints.GrooveJoint method), 36
__init__() (pymunk.constraints.PinJoint method), 37
__init__() (pymunk.constraints.PivotJoint method), 39
__init__() (pymunk.constraints.RatchetJoint method), 40
__init__() (pymunk.constraints.RotaryLimitJoint method), 42
__init__() (pymunk.constraints.SimpleMotor method), 43
__init__() (pymunk.constraints.SlideJoint method), 45
__init__() (pymunk.matplotlib_util.DrawOptions method), 51
__init__() (pymunk.pygame_util.DrawOptions method), 98
__init__() (pymunk.Transform method), 99
__matmul__() (pymunk.vec2d.Vec2d method), 47
__neg__() (pymunk.vec2d.Vec2d method), 47
__pos__() (pymunk.vec2d.Vec2d method), 47
__sub__() (pymunk.vec2d.Vec2d method), 48
__truediv__() (pymunk.vec2d.Vec2d method), 48

A

a (pymunk.constraints.Constraint property), 30
a (pymunk.constraints.DampedRotarySpring property), 31
a (pymunk.constraints.DampedSpring property), 33
a (pymunk.constraints.GearJoint property), 34
a (pymunk.constraints.GrooveJoint property), 36
a (pymunk.constraints.PinJoint property), 37
a (pymunk.constraints.PivotJoint property), 39
a (pymunk.constraints.RatchetJoint property), 41
a (pymunk.constraints.RotaryLimitJoint property), 42
a (pymunk.constraints.SimpleMotor property), 43
a (pymunk.constraints.SlideJoint property), 45
a (pymunk.transform property), 80
activate() (pymunk.Body method), 67
activate_bodies() (pymunk.constraints.Constraint method), 30
activate_bodies() (pymunk.constraints.DampedRotarySpring method), 31
activate_bodies() (pymunk.constraints.DampedSpring method), 33
activate_bodies() (pymunk.constraints.GearJoint method), 34
activate_bodies() (pymunk.constraints.GrooveJoint method), 36
activate_bodies() (pymunk.constraints.PinJoint method), 37
activate_bodies() (pymunk.constraints.PivotJoint method), 39
activate_bodies() (pymunk.constraints.RatchetJoint method), 40
activate_bodies() (pymunk.constraints.RotaryLimitJoint method), 42
activate_bodies() (pymunk.constraints.SimpleMotor method), 43
activate_bodies() (pymunk.constraints.SlideJoint method), 45
activate_bodies() (pymunk.matplotlib_util.DrawOptions method), 51
activate_bodies() (pymunk.pygame_util.DrawOptions method), 98
activate_bodies() (pymunk.pyglet_util.DrawOptions method), 59
activate_bodies() (pymunk.Transform method), 99

INDEX
activate_bodies() (pymunk.constraints.SimpleMotor method), 43
activate_bodies() (pymunk.constraints.SlideJoint method), 45
add() (pymunk.Space method), 88
add_collision_handler() (pymunk.Space method), 89
add_default_collision_handler() (pymunk.Space method), 89
add_post_step_callback() (pymunk.Space method), 89
addWildcard_collision_handler() (pymunk.Space method), 89
ALL_CATEGORIES() (pymunk.ShapeFilter static method), 87
ALL_Masks() (pymunk.ShapeFilter static method), 87
alpha (pymunk.SymbolQueryInfo property), 83
anchor_a (pymunk.constraints.DampedSpring property), 33
anchor_a (pymunk.constraints.PinJoint property), 38
anchor_a (pymunk.constraints.PivotJoint property), 39
anchor_a (pymunk.constraints.SlideJoint property), 45
anchor_b (pymunk.constraints.DampedSpring property), 33
anchor_b (pymunk.constraints.GrooveJoint property), 36
anchor_b (pymunk.constraints.PinJoint property), 38
anchor_b (pymunk.constraints.PivotJoint property), 39
anchor_b (pymunk.constraints.SlideJoint property), 45
angle (pymunk.Body property), 67
angle (pymunk.constraints.RatchetJoint property), 41
angle (pymunk.vec2d property), 101
angle (pymunk.vec2d.Vec2d property), 48
angle_degrees (pymunk.Vec2d property), 101
angle_degrees (pymunk.vec2d.Vec2d property), 48
angular_velocity (pymunk.Body property), 68
apply_force_at_local_point() (pymunk.Body method), 68
apply_force_at_world_point() (pymunk.Body method), 68
apply_impulse_at_local_point() (pymunk.Body method), 68
apply_impulse_at_world_point() (pymunk.Body method), 68
Arbiter (class in pymunk), 63
area (pymunk.Circle property), 71
area (pymunk.Poly property), 77
area (pymunk.Segment property), 80
area (pymunk.Shape property), 84
area() (pymunk.BB method), 65

B
b (pymunk.constraints.Constraint property), 30
b (pymunk.constraints.DampedRotarySpring property), 31
b (pymunk.constraints.DampedSpring property), 33
b (pymunk.constraints.GearJoint property), 35
b (pymunk.constraints.GrooveJoint property), 36
b (pymunk.constraints.PinJoint property), 38
b (pymunk.constraints.PivotJoint property), 39
b (pymunk.constraints.RatchetJoint property), 41
b (pymunk.constraints.RotaryLimitJoint property), 42
b (pymunk.constraints.SimpleMotor property), 44
b (pymunk.constraints.SlideJoint property), 45
b (pymunk.Segment property), 81
b (pymunk.Transform property), 99
BB (class in pymunk), 64
bb (pymunk.Circle property), 71
bb (pymunk.Poly property), 77
bb (pymunk.Segment property), 81
bb (pymunk.Shape property), 84
bb_query() (pymunk.Space method), 90
begin (pymunk.CollisionHandler property), 74
bodies (pymunk.Space property), 90
Body (class in pymunk), 66
body (pymunk.Circle property), 71
body (pymunk.Poly property), 77
body (pymunk.Segment property), 81
body (pymunk.Shape property), 84
body_type (pymunk.Body property), 68
bottom (pymunk.BB property), 65

C
c (pymunk.Transform property), 99
cache_bb() (pymunk.Circle method), 72
cache_bb() (pymunk.Poly method), 77
cache_bb() (pymunk.Segment method), 81
cache_bb() (pymunk.Shape method), 84
categories (pymunk.ShapeFilter property), 87
center() (pymunk.BB method), 65
center_of_gravity (pymunk.Body property), 68
center_of_gravity (pymunk.Circle property), 72
center_of_gravity (pymunk.Poly property), 77
center_of_gravity (pymunk.Segment property), 81
center_of_gravity (pymunk.Shape property), 84
chipmunk_version (in module pymunk), 104
Circle (class in pymunk), 71
clamp_vect() (pymunk.BB method), 65
collect_segment() (pymunk.autogeometry.PolylineSet method), 27
collide_bodies (pymunk.constraints.Constraint property), 30
collide_bodies (pymunk.constraints.DampedRotarySpring property), 31
collide_bodies (pymunk.constraints.DampedSpring property), 33
D

d (pymunk.Transform property), 99
DampedRotarySpring (class in pymunk.constraints), 31
DampedSpring (class in pymunk.constraints), 32
damping (pymunk.constraints.DampedRotarySpring property), 32
damping (pymunk.constraints.DampedSpring property), 33
damping (pymunk.Space property), 91
data (pymunk.CollisionHandler property), 74
draw_circle() (pymunk.matplotlib_util.DrawOptions method), 52
draw_circle() (pymunk.pygame_util.DrawOptions method), 56
draw_circle() (pymunk.SpaceDebugDrawOptions method), 96
DYNAMIC (pymunk.Body attribute), 66
draw_dot() (pymunk.pyglet_util.DrawOptions method), 61
draw_dot() (pymunk.SpaceDebugDrawOptions method), 96
draw_fat_segment() (pymunk.matplotlib_util.DrawOptions method), 52
draw_fat_segment() (pymunk.pygame_util.DrawOptions method), 56
draw_fat_segment() (pymunk.pyglet_util.DrawOptions method), 61
draw_fat_segment() (pymunk.SpaceDebugDrawOptions method), 96
draw_polygon() (pymunk.matplotlib_util.DrawOptions method), 53
draw_polygon() (pymunk.pygame_util.DrawOptions method), 57
draw_polygon() (pymunk.pyglet_util.DrawOptions method), 61
draw_polygon() (pymunk.SpaceDebugDrawOptions method), 96
draw_segment() (pymunk.matplotlib_util.DrawOptions method), 53
draw_segment() (pymunk.pygame_util.DrawOptions method), 57
draw_segment() (pymunk.pyglet_util.DrawOptions method), 61
draw_segment() (pymunk.SpaceDebugDrawOptions method), 96
draw_shape() (pymunk.matplotlib_util.DrawOptions method), 53
draw_shape() (pymunk.pygame_util.DrawOptions method), 57
draw_shape() (pymunk.pyglet_util.DrawOptions method), 61
draw_shape() (pymunk.SpaceDebugDrawOptions method), 96
DRAW_COLLISION_POINTS (pymunk.matplotlib_util.DrawOptions attribute), 51
DRAW_COLLISION_POINTS (pymunk.pygame_util.DrawOptions attribute), 54
DRAW_COLLISION_POINTS (pymunk.pyglet_util.DrawOptions attribute), 59
DRAW_COLLISION_POINTS (pymunk.SpaceDebugDrawOptions attribute), 95
DRAW_CONSTRAINTS (pymunk.matplotlib_util.DrawOptions attribute), 51
DRAW_CONSTRAINTS (pymunk.pygame_util.DrawOptions attribute), 55
DRAW_CONSTRAINTS (pymunk.pyglet_util.DrawOptions attribute), 59
DRAW_CONSTRAINTS (pymunk.SpaceDebugDrawOptions attribute), 95
DRAW_SHAPES (pymunk.matplotlib_util.DrawOptions attribute), 51
DRAW_SHAPES (pymunk.pygame_util.DrawOptions attribute), 55
DRAW_SHAPES (pymunk.pyglet_util.DrawOptions attribute), 59
DRAW_SHAPES (pymunk.SpaceDebugDrawOptions attribute), 95
Dynamo (class in pymunk.matplotlib_util), 51
DrawOptions (class in pymunk.pyglet_util), 54
DrawOptions (class in pymunk.pygame_util), 59

Index
E

each_arbiter() (pymunk.Body method), 68
elasticity (pymunk.Circle property), 72
elasticity (pymunk.Poly property), 78
elasticity (pymunk_SEGMENT property), 81
elasticity (pymunk.Shape property), 84
error_bias (pymunk.constraints.Constraint property), 30
error_bias (pymunk.constraints.DampedRotarySpring property), 32
error_bias (pymunk.constraints.DampedSpring property), 33
error_bias (pymunk.constraints.GearJoint property), 35
error_bias (pymunk.constraints.GrooveJoint property), 36
error_bias (pymunk.constraints.PinJoint property), 38
error_bias (pymunk.constraints.PivotJoint property), 40
error_bias (pymunk.constraints.RatchetJoint property), 41
error_bias (pymunk.constraints.RotaryLimitJoint property), 42
error_bias (pymunk.constraints.SimpleMotor property), 44
error_bias (pymunk.constraints.SlideJoint property), 45
expand() (pymunk.BB method), 65

F

filter (pymunk.Circle property), 72
filter (pymunk.Poly property), 78
filter (pymunk_SEGMENT property), 81
filter (pymunk.Shape property), 85
flags (pymunk.matplotlib_util.DrawOptions property), 53
flags (pymunk.pygame_util.DrawOptions property), 57
flags (pymunk.pyglet_util.DrawOptions property), 61
force (pymunk.Body property), 69
friction (pymunk.Arbiter property), 63
friction (pymunk.Circle property), 72
friction (pymunk.Poly property), 78
friction (pymunk_SEGMENT property), 81
friction (pymunk.Shape property), 85
from_pygame() (in module pymunk.pygame_util), 58

G

GearJoint (class in pymunk.constraints), 34
get_angle_between() (pymunk_VEC2D method), 101
get_angle_between() (pymunk.Vec2d method), 48
get_angle_degrees_between() (pymunk.Vec2d method), 102
get_dist_sqrd() (pymunk.Vec2d method), 49
get_dist_sqrd() (pymunk.vec2d.Vec2d method), 49
get_distance() (pymunk.Vec2d method), 102
get_distance() (pymunk.vec2d.Vec2d method), 49
get_length_sqrd() (pymunk.Vec2d method), 102
get_length_sqrd() (pymunk.vec2d.Vec2d method), 49
get_mouse_pos() (in module pymunk.pygame_util), 58
get_vertices() (pymunk.Poly method), 78
gradient (pymunk.PointQueryInfo property), 75
groove_a (pymunk.constraints.GrooveJoint property), 36
groove_b (pymunk.constraints.GrooveJoint property), 36
GrooveJoint (class in pymunk.constraints), 36
group (pymunk.ShapeFilter property), 87

I

identity() (pymunk.Transform static method), 99
idle_speed_threshold (pymunk.Space property), 91
impulse (pymunk.constraints.Constaint property), 30
impulse (pymunk.constraints.DampedRotarySpring property), 32
impulse (pymunk.constraints.GearJoint property), 35
impulse (pymunk.constraints.GrooveJoint property), 36
impulse (pymunk.constraints.PinJoint property), 38
impulse (pymunk.constraints.PivotJoint property), 40
impulse (pymunk.constraints.RatchetJoint property), 41
impulse (pymunk.constraints.RotaryLimitJoint property), 42
impulse (pymunk.constraints.SimpleMotor property), 44
impulse (pymunk.constraints.SlideJoint property), 45
index() (pymunk.autogeometry.PolylineSet method), 27
index() (pymunk.BB method), 65
index() (pymunk.PointQueryInfo method), 75
index() (pymunk.SpaceDebugDrawQueryInfo method), 83
index() (pymunk.ShapeFilter method), 87
index() (pymunk.ShapeQueryInfo method), 88
index() (pymunk.Transform method), 99
index() (pymunk.Vec2D method), 102
index() (pymunk.vec2d.Vec2d method), 49
int_tuple (pymunk.Vec2d property), 102
int_tuple (pymunk.vec2d.Vec2d method), 49
interpolate_to() (pymunk.Vec2d method), 102
interpolate_to() (pymunk.vec2d.Vec2d method), 49
intersects() (pymunk.BB method), 65
intersects_segment() (pymunk.BB method), 65
is_closed() (in module pymunk.autogeometry), 28
is_first_contact (pymunk.Arbiter property), 64
is_removal (pymunk.Arbiter property), 64

Index
is_sleeping (pymunk.Body property), 69
iterations (pymunk.Space property), 91

K
KINEMATIC (pymunk.Body attribute), 66
kinetic_energy (pymunk.Body property), 69

L
left (pymunk.BB property), 65
length (pymunk.Vec2d property), 102
length (pymunk.vec2d.Vec2d property), 49
local_to_world() (pymunk.Body method), 69

M
march_hard() (in module pymunk.autogeometry), 28
march_soft() (in module pymunk.autogeometry), 28
mask (pymunk.ShapeFilter property), 87
mass (pymunk.Body property), 69
mass (pymunk.Circle property), 73
mass (pymunk.Poly property), 79
mass (pymunk.Segment property), 82
mass (pymunk.Shape property), 85
max (pymunk.constraints.RotaryLimitJoint property), 43
max (pymunk.constraints.SlideJoint property), 46
max_bias (pymunk.constraints.Constraint property), 30
max_bias (pymunk.constraints.DampedRotarySpring property), 32
max_bias (pymunk.constraints.DampedSpring property), 34
max_bias (pymunk.constraints.GearJoint property), 35
max_bias (pymunk.constraints.GrooveJoint property), 37
max_bias (pymunk.constraints.PinJoint property), 38
max_bias (pymunk.constraints.PivotJoint property), 40
max_bias (pymunk.constraints.RatchetJoint property), 41
max_bias (pymunk.constraints.RotaryLimitJoint property), 43
max_bias (pymunk.constraints.SimpleMotor property), 44
max_force (pymunk.constraints.RotaryLimitJoint property), 43
max_force (pymunk.constraints.SimpleMotor property), 44
max_force (pymunk.constraints.SlideJoint property), 46
merge() (pymunk.BB method), 65
merged_area() (pymunk.BB method), 65
min (pymunk.constraints.RotaryLimitJoint property), 43
min (pymunk.constraints.SlideJoint property), 46
module
pymunk, 63
pymunk.autogeometry, 26
pymunk.constraints, 29
pymunk.matplotlib_util, 51
pymunk.pygame_util, 54
pymunk.pyglet_util, 59
pymunk.vec2d, 46
moment (pymunk.Body property), 69
moment (pymunk.Circle property), 73
moment (pymunk.Poly property), 79
moment (pymunk.Segment property), 82
moment (pymunk.Shape property), 85
moment_for_box() (in module pymunk), 104
moment_for_circle() (in module pymunk), 104
moment_for_poly() (in module pymunk), 104
moment_for_segment() (in module pymunk), 104

N
newForCircle() (pymunk.BB static method), 65
normal (pymunk.Arbitrer property), 64
normal (pymunk.ContactPointSet attribute), 75
normal (pymunk.Segment property), 82
normal (pymunk.SegmentQueryInfo property), 84
normalized() (pymunk.Vec2d method), 103
normalized() (pymunk.vec2d.Vec2d method), 50
normalized_and_length() (pymunk.Vec2d method), 103
normalized_and_length() (pymunk.vec2d.Vec2d method), 50

O
offset (pymunk.Circle property), 73
ones() (pymunk.Vec2d static method), 103
ones() (pymunk.vec2d.Vec2d static method), 50

P
perpendicular() (pymunk.Vec2d method), 103
perpendicular() (pymunk.vec2d.Vec2d method), 50
perpendicular_normal() (pymunk.Vec2d method), 103
perpendicular_normal() (pymunk.vec2d.Vec2d method), 50
phase (pymunk.constraints.GearJoint property), 35
phase (pymunk.constraints.RatchetJoint property), 41
scale_to_length() (pymunk.vec2d.Vec2d method, 50)
scaled() (pymunk.Transform method, 100)
scaling() (pymunk.Transform static method, 100)
Segment (class in pymunk, 80)
segment_query() (pymunk.BB method, 65)
segment_query() (pymunk.Circle method, 73)
segment_query() (pymunk.Poly method, 79)
segment_query() (pymunk.Segment method, 82)
segment_query() (pymunk.Shape method, 85)
segment_query() (pymunk.Space method, 93)
segment_query_first() (pymunk.Space method, 93)
SegmentQueryInfo (class in pymunk, 83)
sensor (pymunk.Circle property, 73)
sensor (pymunk.Poly property, 79)
sensor (pymunk.Segment property, 82)
sensor (pymunk.Shape property, 86)
separate (pymunk.CollisionHandler property, 75)
set_neighbors() (pymunk.Segment method, 82)
Shape (class in pymunk, 84)
shape (pymunk.PointQueryInfo property, 76)
shape (pymunk.SegmentQueryInfo property, 84)
shape (pymunk.ShapeQueryInfo property, 88)
shape_dynamic_color (pymunk.matplotlib_util.DrawOptions attribute, 62)
shape_dynamic_color (pymunk.pygame_util.DrawOptions attribute, 53)
shape_dynamic_color (pymunk.pyglet_util.DrawOptions attribute, 57)
shape_dynamic_color (pymunk.SpaceDebugDrawOptions property, 97)
shape_query() (pymunk.Space method, 93)
shape_sleeping_color (pymunk.matplotlib_util.DrawOptions attribute, 54)
shape_sleeping_color (pymunk.pygame_util.DrawOptions attribute, 58)
shape_sleeping_color (pymunk.pyglet_util.DrawOptions attribute, 62)
shape_sleeping_color (pymunk.SpaceDebugDrawOptions attribute, 97)
shape_static_color (pymunk.matplotlib_util.DrawOptions attribute, 54)
shape_static_color (pymunk.pygame_util.DrawOptions attribute, 58)
shape_static_color (pymunk.SpaceDebugDrawOptions attribute, 62)
shape_static_color (pymunk.SpaceDebugDrawOptions attribute, 97)
ShapeFilter (class in pymunk, 86)
ShapeQueryInfo (class in pymunk, 88)
shapes (pymunk.Arbitrary property, 64)
shapes (pymunk.Body property, 69)
shapes (pymunk.Space property, 94)
shapes_collide() (pymunk.Circle method, 73)
shapes_collide() (pymunk.Poly method, 79)
shapes_collide() (pymunk.Segment method, 83)
shapes_collide() (pymunk.Shape method, 86)
simpleMotor (class in pymunk.constraints, 43)
simplify_curves() (in module pymunk.autogeometry), 43
simplify_vertexes() (in module pymunk.autogeometry), 29
sleep() (pymunk.Body method, 69)
sleep_time_threshold (pymunk.Space property, 94)
sleep_with_group() (pymunk.Body method, 70)
SlideJoint (class in pymunk.constraints, 45)
Space (class in pymunk, 88)
space (pymunk.Body property, 70)
space (pymunk.Circle property, 73)
space (pymunk.Poly property, 80)
space (pymunk.Segment property, 83)
space (pymunk.Shape property, 86)
SpaceDebugDrawOptions (class in pymunk), 95
STATIC (pymunk.Body attribute), 66
static_body (pymunk.Space property), 94
step() (pymunk.Space method), 94
stiffness (pymunk.constraints.DampedRotarySpring property), 32
stiffness (pymunk.constraints.DampedSpring property), 34
surface_velocity (pymunk.Arbiter property), 64
surface_velocity (pymunk.Circle property), 73
surface_velocity (pymunk.Segment property), 83
surface_velocity (pymunk.Shape property), 86

T
threads (pymunk.Space property), 94
to_convex_hull() (in module pymunk.autogeometry), 29
to_pygame() (in module pymunk.pygame_util), 59
top (pymunk.BB property), 65
torque (pymunk.Body property), 70
total_impulse (pymunk.Arbiter property), 64
total_ke (pymunk.Arbiter property), 64
Transform (class in pymunk), 98
transform (pymunk.matplotlib_util.DrawOptions property), 54
transform (pymunk.pygame_util.DrawOptions property), 58
transform (pymunk.pyglet_util.DrawOptions property), 62
transform (pymunk.SpaceDebugDrawOptions property), 97
translated() (pymunk.Transform method), 100
translation() (pymunk.Transform static method), 100
tx (pymunk.Transform property), 101
ty (pymunk.Transform property), 101

U
unit() (pymunk.Vec2d static method), 103
unit() (pymunk.vec2d.Vec2d static method), 50
unsafe_set_endpoints() (pymunk.Segment method), 83
unsafe_set_offset() (pymunk.Circle method), 73
unsafe_set_radius() (pymunk.Circle method), 73
unsafe_set_radius() (pymunk.Poly method), 80
unsafe_set_radius() (pymunk.Segment method), 83
unsafe_set_vertices() (pymunk.Poly method), 80
update() (pymunk.Circle method), 74
update() (pymunk.Poly method), 80
update() (pymunk.Segment method), 83
update() (pymunk.Shape method), 86
update_position() (pymunk.Body static method), 70
update_velocity() (pymunk.Body static method), 70
use_spatial_hash() (pymunk.Space method), 94

V
Vec2d (class in pymunk), 101
Vec2d (class in pymunk.vec2d), 47
velocity (pymunk.Body property), 70
velocity_at_local_point() (pymunk.Body method), 70
velocity_at_world_point() (pymunk.Body method), 70
velocity_func (pymunk.Body property), 70
version (in module pymunk), 104

W
world_to_local() (pymunk.Body method), 71

X
x (pymunk.Vec2d property), 103
x (pymunk.vec2d.Vec2d property), 50

Y
y (pymunk.Vec2d property), 103
y (pymunk.vec2d.Vec2d property), 50

Z
zero() (pymunk.Vec2d static method), 103
zero() (pymunk.vec2d.Vec2d static method), 50