Picraft Documentation

Release 0.4

Dave Jones

July 18, 2015

Contents

This package provides an alternate Python API for Minecraft Pi edition on the Raspberry Pi, or Raspberry Juice on the PC for Python 2.7 (or above), or Python 3.2 (or above).

CHAPTER 1

Links

- The code is licensed under the BSD license
- The source code can be obtained from GitHub, which also hosts the bug tracker
- The documentation (which includes installation, quick-start examples, and lots of code recipes) can be read on ReadTheDocs
- Packages can be downloaded from PyPI, but reading the installation instructions is more likely to be useful

CHAPTER 2

Table of Contents

2.1 Python 2.7+ Installation

There are several ways to install picraft under Python 2.7 (or above), each with their own advantages and disadvantages. Have a read of the sections below and select an installation method which conforms to your needs.

2.1.1 Raspbian installation

If you are using the Raspbian distro, it is best to install picraft using the system's package manager: apt. This will ensure that picraft is easy to keep up to date, and easy to remove should you wish to do so. It will also make picraft available for all users on the system. To install picraft using apt simply:

```
$ sudo apt-get update
$ sudo apt-get install python-picraft
```

To upgrade your installation when new releases are made you can simply use apt's normal upgrade procedure:

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

If you ever need to remove your installation:

```
$ sudo apt-get remove python-picraft
```

2.1.2 User installation

This is the simplest (non-apt) form of installation, but bear in mind that it will only work for the user you install under. For example, if you install as the `pi` user, you will only be able to use picraft as the `pi` user. If you run python as root (e.g. with `sudo python`) it will not find the module. See *System installation* below if you require a root installation.

To install as your current user:

```
$ sudo apt-get install python-pip
$ pip install --user picraft
```

Note that `pip` is **not** run with `sudo`; this is deliberate. To upgrade your installation when new releases are made:

```
$ pip install --user -U picraft
```

If you ever need to remove your installation:

```
$ pip uninstall picraft
```

2.1.3 System installation

A system installation will make picraft accessible to all users (in contrast to the user installation). It is as simple to perform as the user installation and equally easy to keep updated. To perform the installation:

```
$ sudo apt-get install python-pip
$ sudo pip install picraft
```

To upgrade your installation when new releases are made:

```
$ sudo pip install -U picraft
```

If you ever need to remove your installation:

```
$ sudo pip uninstall picraft
```

2.1.4 Virtualenv installation

If you wish to install picraft within a virtualenv (useful if you're working on several Python projects with potentially conflicting dependencies, or you just like keeping things separate and easily removable):

```
$ sudo apt-get install python-pip python-virtualenv
$ virtualenv sandbox
$ source sandbox/bin/activate
(sandbox) $ pip install picraft
```

Bear in mind that each time you want to use picraft you will need to activate the virtualenv before running Python:

```
$ source sandbox/bin/activate
(sandbox) $ python
>>> import picraft
```

To upgrade your installation, make sure the virtualenv is activated and just use pip:

```
$ source sandbox/bin/activate
(sandbox) $ pip install -U picraft
```

To remove your installation simply blow away the virtualenv:

```
$ rm -fr ~/sandbox/
```

2.1.5 Development installation

If you wish to develop picraft itself, it is easiest to obtain the source by cloning the GitHub repository and then use the "develop" target of the Makefile which will install the package as a link to the cloned repository allowing in-place development (it also builds a tags file for use with vim/emacs with Exuberant's ctags utility). The following example demonstrates this method within a virtual Python environment:

```
$ sudo apt-get install build-essential git git-core exuberant-ctags \
    python-virtualenv
$ virtualenv sandbox
$ source sandbox/bin/activate
(sandbox) $ git clone https://github.com/waveform80/picraft.git
(sandbox) $ cd picraft
(sandbox) $ make develop
```

To pull the latest changes from git into your clone and update your installation:

```
$ source sandbox/bin/activate
(sandbox) $ cd picraft
(sandbox) $ git pull
(sandbox) $ make develop
```

To remove your installation blow away the sandbox and the clone:

```
$ rm -fr ~/sandbox/ ~/picraft/
```

Even if you don't feel up to hacking on the code, I'd love to hear suggestions from people of what you'd like the API to look like (even if the code itself isn't particularly pythonic, the interface should be)!

2.1.6 Test suite

If you wish to run the picraft test suite, follow the instructions in *Development installation* above and then execute the following command:

```
(sandbox) $ make test
```

2.2 Python 3.2+ Installation

There are several ways to install picraft under Python 3.2 (or above), each with their own advantages and disadvantages. Have a read of the sections below and select an installation method which conforms to your needs.

2.2.1 Raspbian installation

If you are using the Raspbian distro, it is best to install picraft using the system's package manager: apt. This will ensure that picraft is easy to keep up to date, and easy to remove should you wish to do so. It will also make picraft available for all users on the system. To install picraft using apt simply:

```
$ sudo apt-get update
$ sudo apt-get install python3-picraft
```

To upgrade your installation when new releases are made you can simply use apt's normal upgrade procedure:

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

If you ever need to remove your installation:

```
$ sudo apt-get remove python3-picraft
```

2.2.2 User installation

This is the simplest (non-apt) form of installation, but bear in mind that it will only work for the user you install under. For example, if you install as the `pi` user, you will only be able to use picraft as the `pi` user. If you run python as root (e.g. with `sudo python`) it will not find the module. See *System installation* below if you require a root installation.

To install as your current user:

```
$ sudo apt-get install python3-pip
$ pip-3.2 install --user picraft
```

Note that `pip-3.2` is **not** run with `sudo`; this is deliberate. To upgrade your installation when new releases are made:

```
$ pip-3.2 install --user -U picraft
```

If you ever need to remove your installation:

```
$ pip-3.2 uninstall picraft
```

2.2.3 System installation

A system installation will make picraft accessible to all users (in contrast to the user installation). It is as simple to perform as the user installation and equally easy to keep updated. To perform the installation:

```
$ sudo apt-get install python3-pip
$ sudo pip-3.2 install picraft
```

To upgrade your installation when new releases are made:

```
$ sudo pip-3.2 install -U picraft
```

If you ever need to remove your installation:

```
$ sudo pip-3.2 uninstall picraft
```

2.2.4 Virtualenv installation

If you wish to install picraft within a virtualenv (useful if you're working on several Python projects with potentially conflicting dependencies, or you just like keeping things separate and easily removable):

```
$ sudo apt-get install python3-pip python-virtualenv
$ virtualenv -p python3 sandbox
$ source sandbox/bin/activate
(sandbox) $ pip-3.2 install picraft
```

Bear in mind that each time you want to use picraft you will need to activate the virtualenv before running Python:

```
$ source sandbox/bin/activate
(sandbox) $ python
>>> import picraft
```

To upgrade your installation, make sure the virtualenv is activated and just use pip:

```
$ source sandbox/bin/activate
(sandbox) $ pip-3.2 install -U picraft
```

To remove your installation simply blow away the virtualenv:

```
$ rm -fr ~/sandbox/
```

2.2.5 Development installation

If you wish to develop picraft itself, it is easiest to obtain the source by cloning the GitHub repository and then use the "develop" target of the Makefile which will install the package as a link to the cloned repository allowing in-place development (it also builds a tags file for use with vim/emacs with Exuberant's ctags utility). The following example demonstrates this method within a virtual Python environment:

```
$ sudo apt-get install build-essential git git-core exuberant-ctags \
    python-virtualenv
$ virtualenv -p python3 sandbox
$ source sandbox/bin/activate
(sandbox) $ git clone https://github.com/waveform80/picraft.git
(sandbox) $ cd picraft
(sandbox) $ make develop
```

To pull the latest changes from git into your clone and update your installation:

```
$ source sandbox/bin/activate
(sandbox) $ cd picraft
(sandbox) $ git pull
(sandbox) $ make develop
```

To remove your installation blow away the sandbox and the clone:

```
$ rm -fr ~/sandbox/ ~/picraft/
```

Even if you don't feel up to hacking on the code, I'd love to hear suggestions from people of what you'd like the API to look like (even if the code itself isn't particularly pythonic, the interface should be)!

2.2.6 Test suite

If you wish to run the picraft test suite, follow the instructions in *Development installation* above and then execute the following command:

```
(sandbox) $ make test
```

2.3 Quick Start

The first thing you need to learn in picraft is vectors, and vector ranges. Er, the two things you need to learn in picraft are vectors, vector ranges, and blocks. The three things ... look, I'll just come in again.

Firstly, ensure that you have a Minecraft game running on your Pi. Now start a terminal, start Python within the terminal, import the picraft library and start a connection to the Minecraft world:

```
>>> from picraft import *
>>> world = World()
```

The `World` class is the usual starting point for picraft scripts. It provides access to the blocks that make up the world, the players within the world, methods to save and restore the state of the world, and the ability to print things to the chat console. Let's start by printing something to the console:

```
>>> world.say('Hello, world!')
```

You should see "Hello, world!" appear in the chat console of the Minecraft game. Next, we can query where we're standing with the `pos` attribute of the `player` attribute:

```
>>> world.player.pos
Vector(x=-2.49725, y=18.0, z=-4.21989)
```

This tells us that our character is standing at the 3-dimensional coordinates -2.49, 18.0, -4.22 (approximately). In the Minecraft world, the X and Z coordinates (the first and last) form the "ground plane". In other words you can think of X as going left to right, and Z as going further to nearer. The Y axis represents height (it goes up and down). We can find out our player's coordinates rounded to the nearest block with the `tile_pos` attribute:

```
>>> world.player.tile_pos
Vector(x=-3, y=18, z=-5)
```

Therefore, we can make our character jump in the air by adding a certain amount to the player's Y coordinate. To do this we need to construct a `Vector` with a positive Y value and add it to the `tile_pos` attribute:

```
>>> world.player.tile_pos = world.player.tile_pos + Vector(y=5)
```

We can also use a Python short-hand for this:

```
>>> world.player.tile_pos += Vector(y=5)
```

This demonstrates one way of constructing a `Vector`. We can also construct one by listing all 3 coordinates explicitly:

```
>>> Vector(y=5)
Vector(x=0, y=5, z=0)
>>> Vector(0, 5, 0)
Vector(x=0, y=5, z=0)
```

We can use the *blocks* attribute to discover the type of each block in the world. For example, we can find out what sort of block we're currently standing on:

```
>>> world.blocks[world.player.tile_pos - Vector(y=1)]
<Block "grass" id=2 data=0>
```

We can assign values to this property to change the sort of block we're standing on. In order to do this we need to construct a new *Block* instance which can be done by specifying the id manually, or by name:

```
>>> Block(1)
<Block "stone" id=1 data=0>
>>> Block('stone')
<Block "stone" id=1 data=0>
```

Now we'll change the block beneath our feet:

```
>>> world.blocks[world.player.tile_pos - Vector(y=1)] = Block('stone')
```

We can query the state of many blocks surrounding us by providing a vector slice to the *blocks* attribute. To make things a little easier we'll store the base position first:

```
>>> v = world.player.tile_pos - Vector(y=1)
>>> world.blocks[v - Vector(1, 0, 1):v + Vector(2, 1, 2)]
[<Block "grass" id=2 data=0>,
 <Block "grass" id=2 data=0>,
 <Block "grass" id=2 data=0>,
 <Block "grass" id=2 data=0>,
 <Block "stone" id=1 data=0>,
 <Block "grass" id=2 data=0>,
 <Block "grass" id=2 data=0>,
 <Block "grass" id=2 data=0>,
 <Block "grass" id=2 data=0>]
```

Note that the range provided (as with all ranges in Python) is half-open, which is to say that the lower end of the range is *inclusive* while the upper end is *exclusive*. You can see this explicitly with the *vector_range()* function:

```
>>> v
Vector(x=-2, y=14, z=3)
>>> list(vector_range(v - Vector(1, 0, 1), v + Vector(2, 1, 2)))
[Vector(x=-3, y=14, z=2),
 Vector(x=-3, y=14, z=3),
 Vector(x=-3, y=14, z=4),
 Vector(x=-2, y=14, z=2),
 Vector(x=-2, y=14, z=3),
 Vector(x=-2, y=14, z=4),
 Vector(x=-1, y=14, z=2),
 Vector(x=-1, y=14, z=3),
 Vector(x=-1, y=14, z=4)]
```

This may seem a clunky way of specifying a range and, in the manner shown above it is. However, most standard infix arithmetic operations applied to a vector are applied to *all* its elements:

```
>>> Vector()
Vector(x=0, y=0, z=0)
>>> Vector() + 1
Vector(x=1, y=1, z=1)
>>> 2 * (Vector() + 1)
Vector(x=2, y=2, z=2)
```

This makes construction of such ranges or slices considerably easier. For example, to construct a vertical range of vectors from the origin (0, 0, 0) to (0, 10, 0) we first assign the origin to `v` which we use for the start of the range, then add `Vector(y=10)` to it, and finally add one to compensate for the half-open nature of the range:

```
>>> v = Vector()
>>> list(vector_range(v, v + Vector(y=10) + 1))
[Vector(x=0, y=0, z=0),
 Vector(x=0, y=1, z=0),
 Vector(x=0, y=2, z=0),
 Vector(x=0, y=3, z=0),
 Vector(x=0, y=4, z=0),
 Vector(x=0, y=5, z=0),
 Vector(x=0, y=6, z=0),
 Vector(x=0, y=7, z=0),
 Vector(x=0, y=8, z=0),
 Vector(x=0, y=9, z=0),
 Vector(x=0, y=10, z=0)]
```

We can also re-write the example before this (the blocks surrounding the one the player is standing on) in several different ways:

```
>>> v = world.player.tile_pos
>>> list(vector_range(v - 1, v + 2 - Vector(y=2)))
[Vector(x=-3, y=14, z=2),
 Vector(x=-3, y=14, z=3),
 Vector(x=-3, y=14, z=4),
 Vector(x=-2, y=14, z=2),
 Vector(x=-2, y=14, z=3),
 Vector(x=-2, y=14, z=4),
 Vector(x=-1, y=14, z=2),
 Vector(x=-1, y=14, z=3),
 Vector(x=-1, y=14, z=4)]
```

We can change the state of many blocks at once similarly by assigning a new `Block` object to a slice of blocks:

```
>>> v = world.player.tile_pos
>>> world.blocks[v - 1:v + 2 - Vector(y=2)] = Block('stone')
```

This is a relatively quick operation, as it only involves a single network call. However, re-writing the state of multiple blocks with different values is more time consuming:

```
>>> world.blocks[v - 1:v + 2 - Vector(y=2)] = [
...     Block('wool', data=i) for i in range(9)]
```

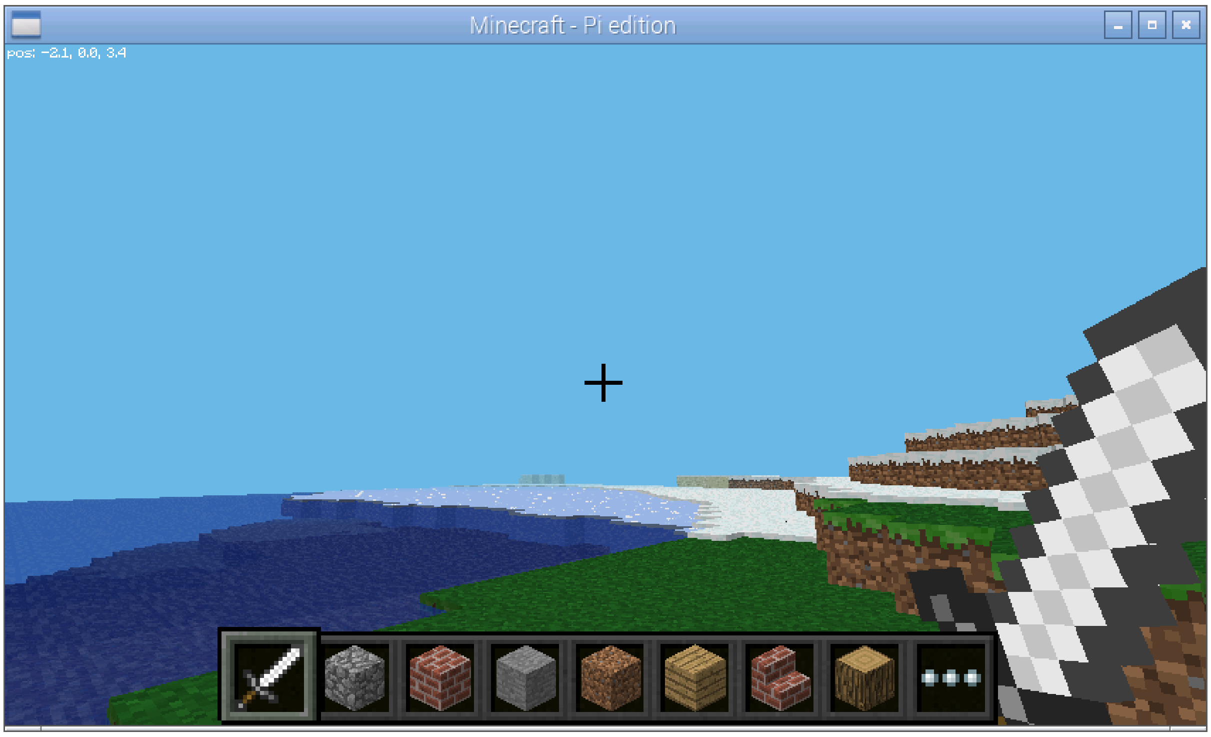
You should notice that the example above takes a few seconds to process (each block requires a separate network transaction and due to deficiencies in the *Minecraft network protocol*, each transaction takes a while to execute). This can be accomplished considerably more quickly by batching multiple requests together:

```
>>> world.blocks[v - 1:v + 2 - Vector(y=2)] = Block('stone')
>>> with world.connection.batch_start():
...     world.blocks[v - 1:v + 2 - Vector(y=2)] = [
...         Block('wool', data=i) for i in range(9)]
```

Finally, the state of the Minecraft world can be saved and restored easily with the `checkpoint` object:

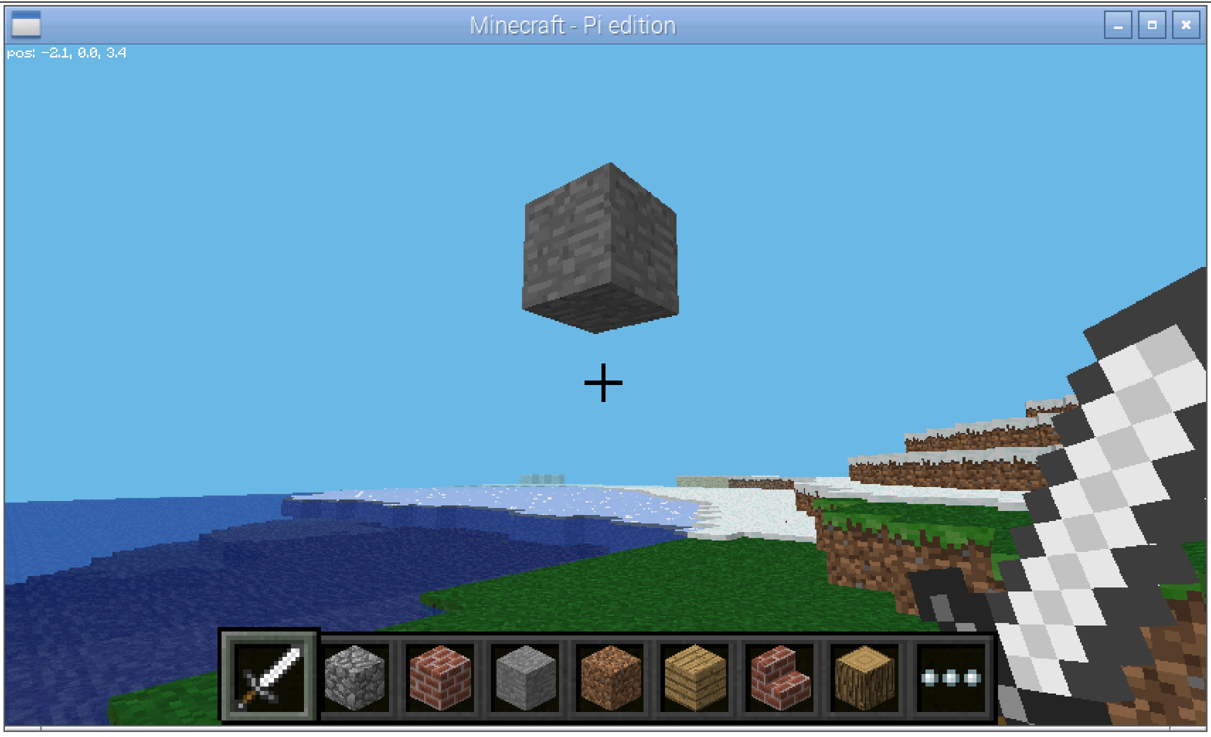
```
>>> world.checkpoint.save()
>>> world.blocks[v - 1:v + 2 - Vector(y=2)] = Block('stone')
>>> world.checkpoint.restore()
```

In order to understand vectors, it can help to visualize them. Pick a relatively open area in the game world.



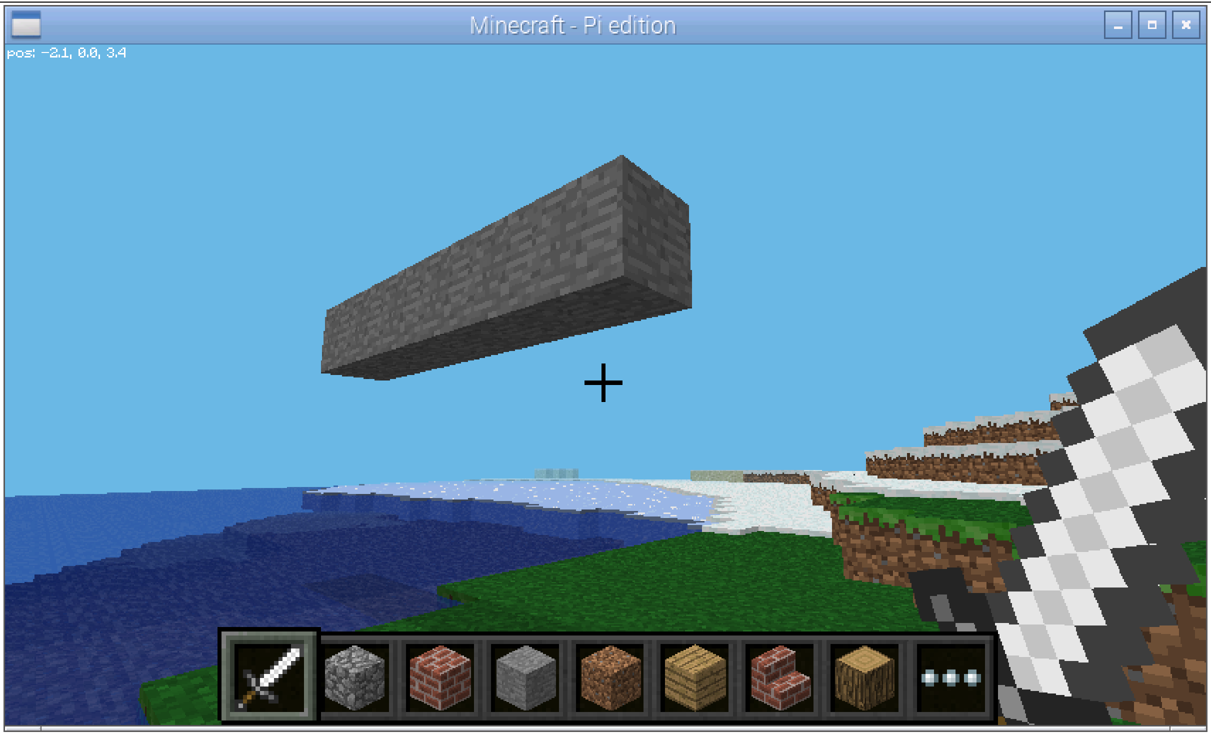
We'll save the vector of your player's position as `v` then add 3 to it. This moves the vector 3 along each axis (X, Y, and Z). Next, we'll make the block at `v` into stone:

```
>>> v = world.player.tile_pos
>>> v = v + 3
>>> world.blocks[v] = Block('stone')
```



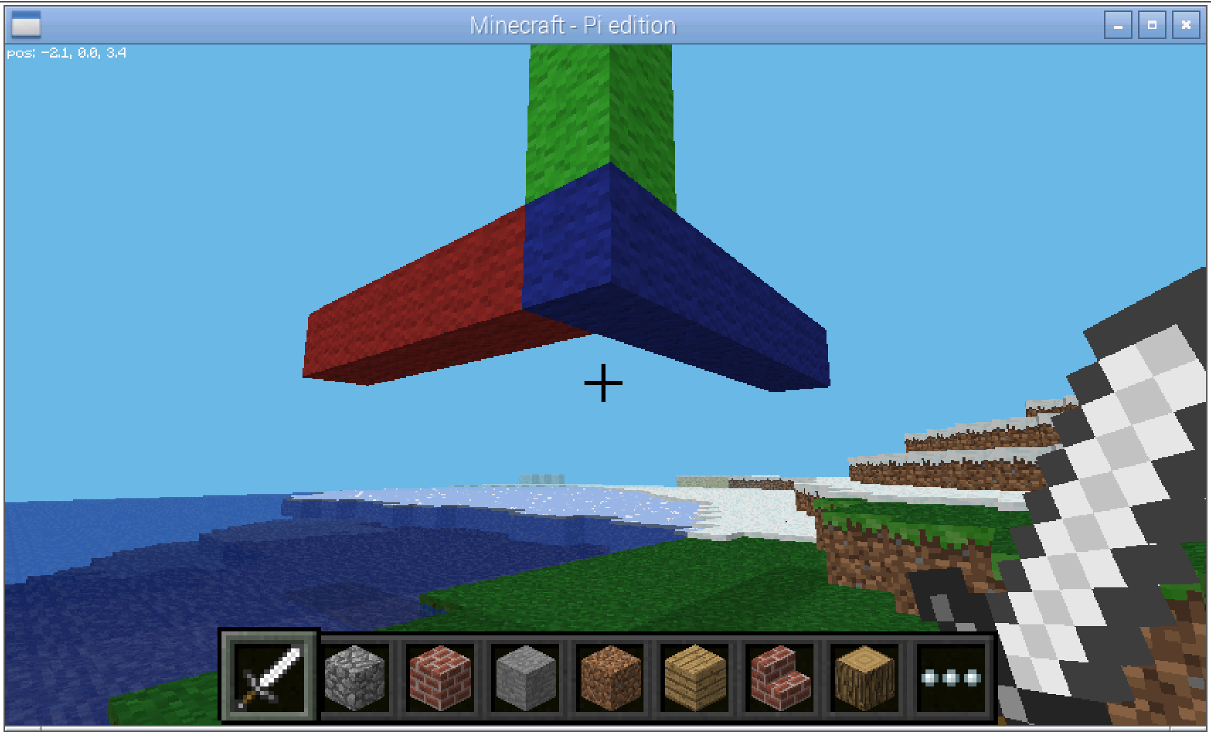
Now we'll explore vector slices a bit by making a line along X+5 into stone. Remember that slices (and ranges) are half-open so we need to add an extra 1 to the end of the slice:

```
>>> world.blocks[v:v + Vector(x=5) + 1] = Block('stone')
```



In order to visualize the three different axes of vectors we'll now draw them each. Here we also use a capability of the `Block` constructor to create a block with a particular color:

```
>>> world.blocks[v:v + Vector(x=5) + 1] = Block('#ff0000')
>>> world.blocks[v:v + Vector(y=5) + 1] = Block('#00ff00')
>>> world.blocks[v:v + Vector(z=5) + 1] = Block('#0000ff')
```

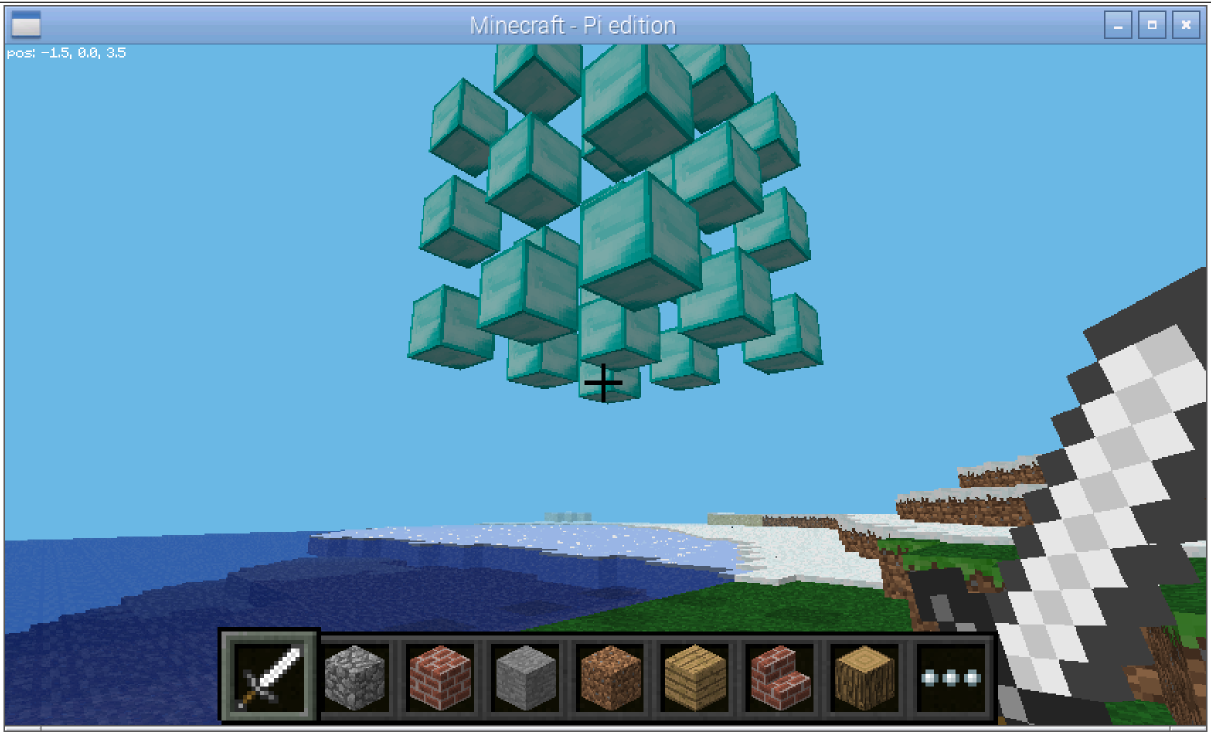


Finally, we can use a vector range to demonstrate patterns. Firstly we wipe out our axes by setting the entire block to "air". Then we define a vector range over the same block with a step of 2, and iterate over each vector within setting it to diamond:

```
>>> world.blocks[v:v + 6] = Block('air')
>>> r = vector_range(v, v + 6, Vector() + 2)
>>> for rv in r:
...     world.blocks[rv] = Block('diamond_block')
```

Once again, we can make use of a batch to speed this up:

```
>>> world.blocks[v:v + 6] = Block('air')
>>> with world.connection.batch_start():
...     for rv in r:
...         world.blocks[rv] = Block('diamond_block')
```



This concludes the quick tour of the picraft library. Conversion instructions from mcpi can be found in the next chapter, followed by picraft recipes in the chapter after that. Finally, the API reference can be found at the end of the manual.

2.4 Conversion from mcpi

If you have existing scripts that use the reference implementation (minecraft-pi aka mcpi), and you wish to convert them to using the picraft library, this section contains details and examples covering equivalent functionality between the libraries.

2.4.1 Minecraft.create

Equivalent: *World*

To create a connection using default settings is similar in both libraries:

```
>>> import mcpi.minecraft as minecraft
>>> mc = minecraft.Minecraft.create()

>>> from picraft import World
>>> w = World()
```

Creating a connection with an explicit hostname and port is also similar:

```
>>> import mcpi.minecraft as minecraft
>>> mc = minecraft.Minecraft.create('localhost', 4711)

>>> from picraft import World
>>> w = World('localhost', 4711)
```

2.4.2 Minecraft.getBlock

See *Minecraft.getBlockWithData* below.

2.4.3 Minecraft.getBlockWithData

Equivalent: `blocks`

Accessing the id of a block is rather different. There is no direct equivalent to `getBlock`, just `getBlockWithData` (as there's no difference in operational cost so there's little point in retrieving a block id without the data). In mcpi this is done by executing a method; in picraft this is done by querying an attribute with a `Vector`:

```
>>> import mcpi.minecraft as minecraft
>>> mc = minecraft.Minecraft.create()
>>> mc.getBlock(0, -1, 0)
2
>>> mc.getBlockWithData(0, -1, 0)
Block(2, 0)

>>> from picraft import World, Vector
>>> w = World()
>>> w.blocks[Vector(0, -1, 0)]
<Block "grass" id=2 data=0>
```

The id and data can be extracted from the `Block` tuple that is returned:

```
>>> w.blocks[Vector(0, -1, 0)].id
2
>>> w.blocks[Vector(0, -1, 0)].data
0
```

2.4.4 Minecraft.setBlock

Equivalent: `blocks`

Setting the id (and optionally data) of a block is also rather different. In picraft the same attribute is used as for accessing block ids; just *assign* a `Block` instance to the attribute, instead of querying it:

```
>>> import mcpi.minecraft as minecraft
>>> mc = minecraft.Minecraft.create()
>>> mc.getBlock(0, -1, 0)
2
>>> mc.setBlock(0, -1, 0, 1, 0)

>>> from picraft import World, Vector, Block
>>> w = World()
>>> w.blocks[Vector(0, -1, 0)]
<Block "grass" id=2 data=0>
>>> w.blocks[Vector(0, -1, 0)] = Block(1, 0)
```

2.4.5 Minecraft.setBlocks

Equivalent: *blocks*

Again, the same attribute as for `setBlock` is used for `setBlocks`; just pass a slice of *vectors* instead of a single vector (the example below shows an easy method of generating such a slice by adding two vectors together for the upper end of the slice):

```
>>> import mcpi.minecraft as minecraft
>>> mc = minecraft.Minecraft.create()
>>> mc.getBlock(0, -1, 0)
2
>>> mc.setBlocks(0, -1, 0, 0, 5, 0, 1, 0)

>>> from picraft import World, Vector, Block
>>> w = World()
>>> v = Vector(0, -1, 0)
>>> w.blocks[v]
<Block "grass" id=2 data=0>
>>> w.blocks[v:v + Vector(1, 7, 1)] = Block(1, 0)
```

2.4.6 Minecraft.getHeight

Equivalent: *height*

Retrieving the height of the world in a specific location is done with an attribute (like retrieving the id and type of blocks). Unlike mcpi, you pass a full vector (of which the Y-coordinate is ignored), and the property returns a full vector with the same X- and Z-coordinates, but the Y-coordinate of the first non-air block from the top of the world:

```
>>> import mcpi.minecraft as minecraft
>>> mc = minecraft.Minecraft.create()
>>> mc.getHeight(0, 0)
0

>>> from picraft import World, Vector
>>> w = World()
>>> w.height[Vector(0, -10, 0)]
Vector(x=0, y=0, z=0)
```

2.4.7 Minecraft.getPlayerEntityIds

Equivalent: *players*

The connected player's entity ids can be retrieved by iterating over the *players* attribute which acts as a mapping from player id to *Player* instances:

```
>>> import mcpi.minecraft as minecraft
>>> mc = minecraft.Minecraft.create()
>>> mc.getPlayerEntityIds()
[1]

>>> from picraft import World
>>> w = World()
>>> list(w.players)
[1]
```

2.4.8 Minecraft.saveCheckpoint

Equivalent: *save()*

Checkpoints can be saved in a couple of ways with picraft. Either you can explicitly call the `save()` method, or you can use the `checkpoint` attribute as a context manager:

```
>>> import mcpi.minecraft as minecraft
>>> mc = minecraft.Minecraft.create()
>>> mc.saveCheckpoint()

>>> from picraft import World
>>> w = World()
>>> w.checkpoint.save()
```

In the context manager case, the checkpoint will be saved upon entry to the context and will only be restored if an exception occurs within the context:

```
>>> from picraft import World, Vector, Block
>>> w = World()
>>> with w.checkpoint:
...     # Do something with blocks...
...     w.blocks[Vector()] = Block.from_name('stone')
```

2.4.9 Minecraft.restoreCheckpoint

Equivalent: `restore()`

As with saving a checkpoint, either you can call `restore()` directly:

```
>>> import mcpi.minecraft as minecraft
>>> mc = minecraft.Minecraft.create()
>>> mc.saveCheckpoint()
>>> mc.restoreCheckpoint()

>>> from picraft import World
>>> w = World()
>>> w.checkpoint.save()
>>> w.checkpoint.restore()
```

Or you can use the context manager to restore the checkpoint automatically in the case of an exception:

```
>>> from picraft import World, Vector, Block
>>> w = World()
>>> with w.checkpoint:
...     # Do something with blocks
...     w.blocks[Vector()] = Block.from_name('stone')
...     # Raising an exception within the block will implicitly
...     # cause the checkpoint to restore
...     raise Exception('roll back to the checkpoint')
```

2.4.10 Minecraft.postToChat

Equivalent: `say()`

The `postToChat` method is simply replaced with the `say()` method with the one exception that the latter correctly recognizes line breaks in the message:

```
>>> import mcpi.minecraft as minecraft
>>> mc = minecraft.Minecraft.create()
>>> mc.postToChat('Hello world!')

>>> from picraft import World
>>> w = World()
>>> w.say('Hello world!')
```

2.4.11 Minecraft.setting

Equivalent: *immutable* and *nametags_visible*

The `setting` method is replaced with (write-only) properties with the equivalent names to the settings that can be used:

```
>>> import mcpi.minecraft as minecraft
>>> mc = minecraft.Minecraft.create()
>>> mc.setting('world_immutable', True)
>>> mc.setting('nametags_visible', True)

>>> from picraft import World
>>> w = World()
>>> w.immutable = True
>>> w.nametags_visible = True
```

2.4.12 Minecraft.player.getPos

Equivalent: *pos*

The `player.getPos` and `player.setPos` methods are replaced with the *pos* attribute which returns a *Vector* of floats and accepts the same to move the host player:

```
>>> import mcpi.minecraft as minecraft
>>> mc = minecraft.Minecraft.create()
>>> mc.player.getPos()
Vec3(12.7743,12.0,-8.39158)
>>> mc.player.setPos(12,12,-8)

>>> from picraft import World, Vector
>>> w = World()
>>> w.player.pos
Vector(x=12.7743, y=12.0, z=-8.39158)
>>> w.player.pos = Vector(12, 12, -8)
```

One advantage of this implementation is that adjusting the player's position relatively to their current one becomes simple:

```
>>> w.player.pos += Vector(y=20)
```

2.4.13 Minecraft.player.setPos

See *Minecraft.player.getPos* above.

2.4.14 Minecraft.player.getTilePos

Equivalent: *tile_pos*

The `player.getTilePos` and `player.setTilePos` methods are replaced with the *tile_pos* attribute which returns a *Vector* of ints, and accepts the same to move the host player:

```
>>> import mcpi.minecraft as minecraft
>>> mc = minecraft.Minecraft.create()
>>> mc.player.getTilePos()
Vec3(12,12,-9)
>>> mc.player.setTilePos(12, 12, -8)

>>> from picraft import World, Vector
>>> w = World()
>>> w.player.tile_pos
Vector(x=12, y=12, z=-9)
>>> w.player.tile_pos += Vector(y=20)
```

2.4.15 Minecraft.player.setTilePos

See *Minecraft.player.getTilePos* above.

2.4.16 Minecraft.player.setting

Equivalent: `autojump`

The `player.setting` method is replaced with the write-only `autojump` attribute:

```
>>> import mcpi.minecraft as minecraft
>>> mc = minecraft.Minecraft.create()
>>> mc.player.setting('autojump', False)

>>> from picraft import World
>>> w = World()
>>> w.player.autojump = False
```

2.4.17 Minecraft.player.getRotation

Equivalent: `heading`

The `player.getRotation` method is replaced with the read-only `heading` attribute:

```
>>> import mcpi.minecraft as minecraft
>>> mc = minecraft.Minecraft.create()
>>> mc.player.getRotation()
49.048615

>>> from picraft import World
>>> w = World()
>>> w.player.heading
49.048615
```

2.4.18 Minecraft.player.getPitch

Equivalent: `pitch`

The `player.getPitch` method is replaced with the read-only `pitch` attribute:

```
>>> import mcpi.minecraft as minecraft
>>> mc = minecraft.Minecraft.create()
>>> mc.player.getPitch()
4.3500223

>>> from picraft import World
>>> w = World()
>>> w.player.pitch
4.3500223
```

2.4.19 Minecraft.player.getDirection

Equivalent: *direction*

The `player.getDuration` method is replaced with the read-only `duration` attribute:

```
>>> import mcpi.minecraft as minecraft
>>> mc = minecraft.Minecraft.create()
>>> mc.player.getDirection()
Vec3(0.1429840348766887,-0.3263934845430674,0.934356922711132)

>>> from picraft import World
>>> w = World()
>>> w.player.direction
Vector(x=0.1429840348766887, y=-0.3263934845430674, z=0.934356922711132)
```

2.4.20 Minecraft.entity.getPos

Equivalent: *pos*

The `entity.getPos` and `entity.setPos` methods are replaced with the *pos* attribute. Access the relevant *Player* instance by indexing the *players* attribute:

```
>>> import mcpi.minecraft as minecraft
>>> mc = minecraft.Minecraft.create()
>>> mc.entity.getPos(1)
Vec3(12.7743,12.0,-8.39158)
>>> mc.entity.setPos(1, 12, 12, -8)

>>> from picraft import World, Vector
>>> w = World()
>>> w.players[1].pos
Vector(x=12.7743, y=12.0, z=-8.39158)
>>> w.players[1].pos = Vector(12, 12, -8)
```

2.4.21 Minecraft.entity.setPos

See *Minecraft.entity.getPos* above.

2.4.22 Minecraft.entity.getTilePos

Equivalent: *tile_pos*

The `entity.getTilePos` and `entity.setTilePos` methods are replaced with the *tile_pos* attribute. Access the relevant *Player* instance by indexing the *players* attribute:

```
>>> import mcpi.minecraft as minecraft
>>> mc = minecraft.Minecraft.create()
>>> mc.entity.getTilePos(1)
Vec3(12,12,-9)
>>> mc.entity.setTilePos(1, 12, 12, -8)

>>> from picraft import World, Vector
>>> w = World()
>>> w.players[1].tile_pos
Vector(x=12, y=12, z=-9)
>>> w.players[1].tile_pos += Vector(y=20)
```

2.4.23 Minecraft.entity.setTilePos

See *Minecraft.entity.getTilePos* above.

2.4.24 Minecraft.entity.getRotation

Equivalent: `heading`

The `entity.getRotation` method is replaced with the read-only `heading` attribute:

```
>>> import mcpi.minecraft as minecraft
>>> mc = minecraft.Minecraft.create()
>>> mc.entity.getRotation(213)
49.048615

>>> from picraft import World
>>> w = World()
>>> w.players[213].heading
49.048615
```

2.4.25 Minecraft.entity.getPitch

Equivalent: `pitch`

The `entity.getPitch` method is replaced with the read-only `pitch` attribute:

```
>>> import mcpi.minecraft as minecraft
>>> mc = minecraft.Minecraft.create()
>>> mc.entity.getPitch(213)
4.3500223

>>> from picraft import World
>>> w = World()
>>> w.players[213].pitch
4.3500223
```

2.4.26 Minecraft.entity.getDirection

Equivalent: `direction`

The `entity.getDuration` method is replaced with the read-only `duration` attribute:

```
>>> import mcpi.minecraft as minecraft
>>> mc = minecraft.Minecraft.create()
>>> mc.entity.getDirection(213)
Vec3(0.1429840348766887,-0.3263934845430674,0.934356922711132)

>>> from picraft import World
>>> w = World()
>>> w.players[213].direction
Vector(x=0.1429840348766887, y=-0.3263934845430674, z=0.934356922711132)
```

2.4.27 Minecraft.camera.setNormal

Equivalent: `first_person()`

The `camera` attribute in picraft holds a `Camera` instance which controls the camera in the Minecraft world. The `first_person()` method can be used to set the camera to view the world through the eyes of the specified player. The player is specified as the world's `player` attribute, or as a player retrieved from the `players` attribute:

```
>>> import mcpi.minecraft as minecraft
>>> mc = minecraft.Minecraft.create()
>>> mc.camera.setNormal()
>>> mc.camera.setNormal(2)

>>> from picraft import World
>>> w = World()
>>> w.camera.first_person(w.player)
>>> w.camera.first_person(w.players[2])
```

2.4.28 Minecraft.camera.setFollow

Equivalent: `third_person()`

The `camera` attribute in picraft holds a `Camera` instance which controls the camera in the Minecraft world. The `third_person()` method can be used to set the camera to view the specified player from above. The player is specified as the world's `player` attribute, or as a player retrieved from the `players` attribute:

```
>>> import mcpi.minecraft as minecraft
>>> mc = minecraft.Minecraft.create()
>>> mc.camera.setFollow()
>>> mc.camera.setNormal(1)

>>> from picraft import World
>>> w = World()
>>> w.camera.third_person(w.player)
>>> w.camera.third_person(w.players[1])
```

2.4.29 Minecraft.camera.setFixed

Equivalent: `pos`

The `pos` attribute can be passed a `Vector` instance to specify the absolute position of the camera. The camera will be pointing straight down (y=-1) from the given position and will not move to follow any entity:

```
>>> import mcpi.minecraft as minecraft
>>> mc = minecraft.Minecraft.create()
>>> mc.camera.setFixed()
>>> mc.camera.setPos(0,20,0)

>>> from picraft import World, Vector
>>> w = World()
>>> w.camera.pos = Vector(0, 20, 0)
```

2.4.30 Minecraft.camera.setPos

See *Minecraft.camera.setFixed* above.

2.4.31 Minecraft.block.Block

Equivalent: `Block`

The `Block` class in picraft is similar to the `Block` class in mcpi but with one major difference: in picraft a `Block` instance is a tuple descendent and therefore immutable (you cannot change the id or data attributes of a `Block` instance).

This may seem like an arbitrary barrier, but firstly its quite rare to adjust the the id or data attribute (it's rather more common to just overwrite a block in the world with an entirely new type), and secondly this change permits blocks to be used as keys in a Python dictionary, or to be stored in a set.

The *Block* class also provides several means of construction, and additional properties:

```
>>> from picraft import Block
>>> Block(1, 0)
<Block "stone" id=1 data=0>
>>> Block(35, 1)
<Block "wool" id=35 data=1>
>>> Block.from_name('wool', data=1).description
u'Orange Wool'
>>> Block.from_color('#ffffff').description
u'White Wool'
```

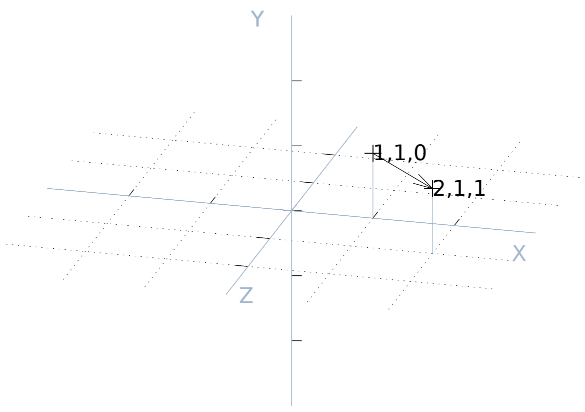
2.5 Vectors

Vectors are a crucial part of working with picraft; sufficiently important to demand their own section. This chapter introduces all the major vector operations with simple examples and diagrams illustrating the results.

2.5.1 Vector-vector operations

The picraft *Vector* class is extremely flexible and supports a wide variety of operations. All Python's built-in operations (addition, subtraction, division, multiplication, modulus, absolute, bitwise operations, etc.) are supported between two vectors, in which case the operation is performed element-wise. In other words, adding two vectors `A` and `B` produces a new vector with its `x` attribute set to `A.x + B.x`, its `y` attribute set to `A.y + B.y` and so on:

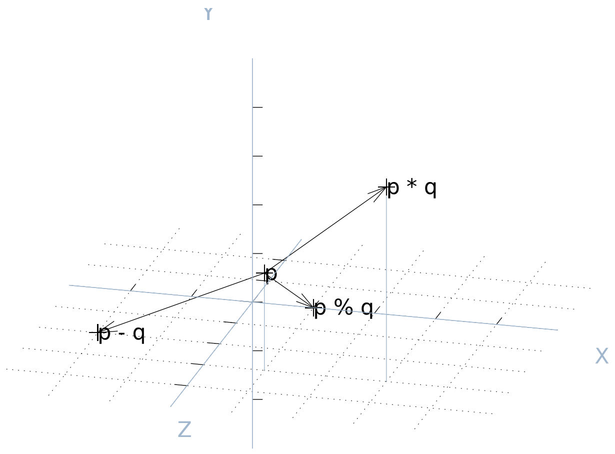
```
>>> from picraft import *
>>> Vector(1, 1, 0) + Vector(1, 0, 1)
Vector(x=2, y=1, z=1)
```



Likewise for subtraction, multiplication, etc.:

```
>>> p = Vector(1, 2, 3)
>>> q = Vector(3, 2, 1)
>>> p - q
Vector(x=-2, y=0, z=2)
>>> p * q
```

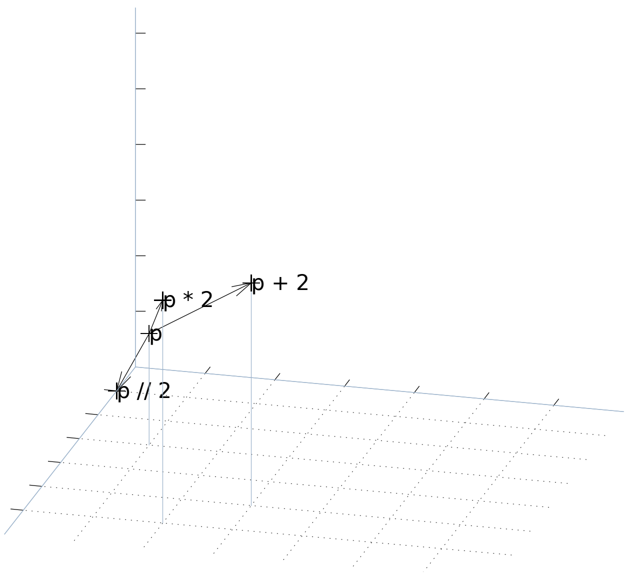
```
Vector(x=3, y=4, z=3)
>>> p % q
Vector(x=1, y=0, z=0)
```



2.5.2 Vector-scalar operations

Vectors also support several operations between themselves and a scalar value. In this case the operation with the scalar is applied to each element of the vector. For example, multiplying a vector by the number 2 will return a new vector with every element of the original multiplied by 2:

```
>>> p * 2
Vector(x=2, y=4, z=6)
>>> p + 2
Vector(x=3, y=4, z=5)
>>> p // 2
Vector(x=0, y=1, z=1)
```



2.5.3 Miscellaneous function support

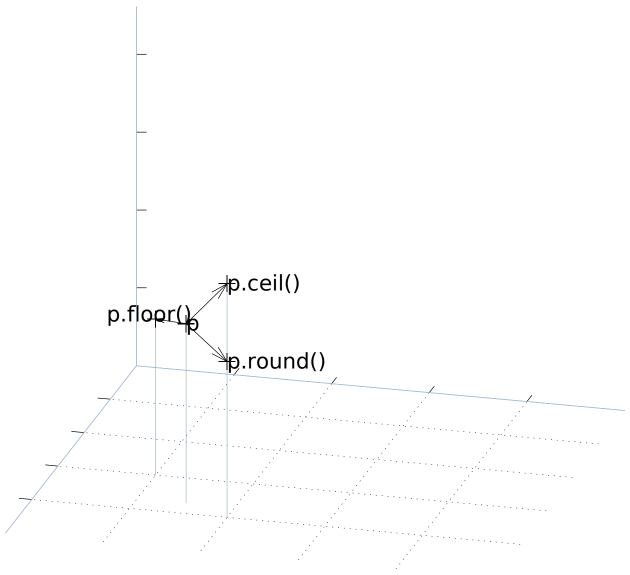
Vectors also support several of Python's built-in functions:

```
>>> abs(Vector(-1, 0, 1))
Vector(x=1, y=0, z=1)
>>> pow(Vector(1, 2, 3), 2)
Vector(x=1, y=4, z=9)
>>> import math
>>> math.trunc(Vector(1.5, 2.3, 3.7))
Vector(x=1, y=2, z=3)
```

2.5.4 Vector rounding

Some built-in functions can't be directly supported, in which case equivalently named methods are provided:

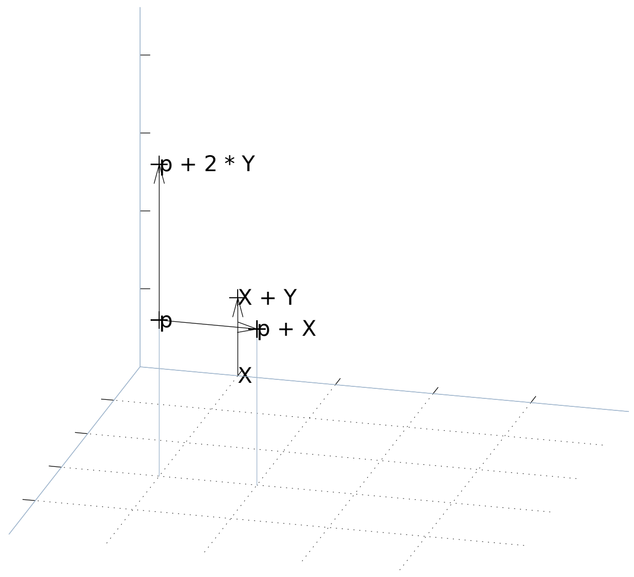
```
>>> p = Vector(1.5, 2.3, 3.7)
>>> p.round()
Vector(x=2, y=2, z=4)
>>> p.ceil()
Vector(x=2, y=3, z=4)
>>> p.floor()
Vector(x=1, y=2, z=3)
```



2.5.5 Short-cuts

Several vector short-hands are also provided. One for the unit vector along each of the three axes (X, Y, and Z), one for the origin (O), and finally V which is simply a short-hand for Vector itself. Obviously, these can be used to simplify many vector-related operations:

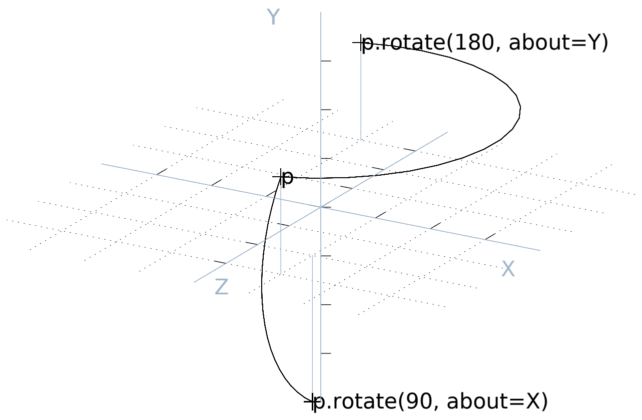
```
>>> X
Vector(x=1, y=0, z=0)
>>> X + Y
Vector(x=1, y=1, z=0)
>>> p = V(1, 2, 3)
>>> p + X
Vector(x=2, y=2, z=3)
>>> p + 2 * Y
Vector(x=1, y=6, z=3)
```



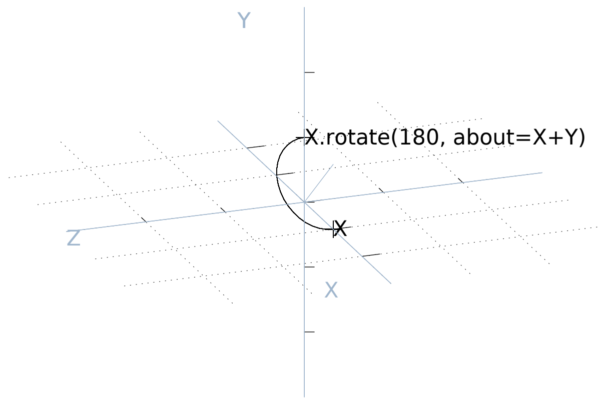
2.5.6 Rotation

From the paragraphs above it should be relatively easy to see how one can implement vector translation and vector scaling using everyday operations like addition, subtraction, multiplication and divsion. The third major transformation usually required of vectors, rotation, is a little harder. For this, the *rotate()* method is provided. This takes two mandatory arguments: the number of degrees to rotate, and a vector specifying the axis about which to rotate (it is recommended that this is specified as a keyword argument for code clarity). For example:

```
>>> p = V(1, 2, 3)
>>> p.rotate(90, about=X)
Vector(x=1.0, y=-3.0, z=2.0)
>>> p.rotate(180, about=Y)
Vector(x=-0.9999999999999997, y=2, z=-3.0)
>>> p.rotate(180, about=Y).round()
Vector(x=-1.0, y=2.0, z=-3.0)
```

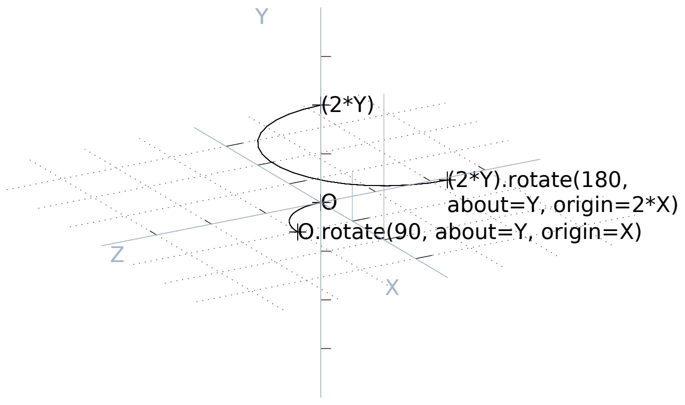


```
>>> X.rotate(180, about=X + Y).round()
Vector(x=-0.0, y=1.0, z=-0.0)
```



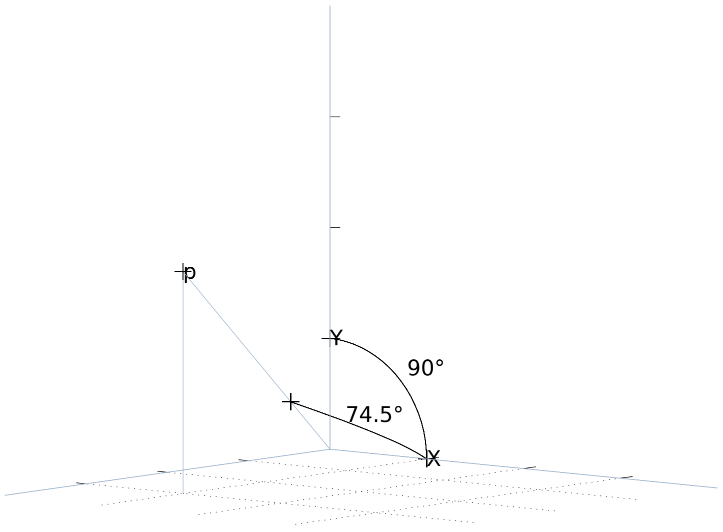
A third optional argument to rotate, *origin*, permits rotation about an arbitrary line. When specified, the axis of rotation passes through the point specified by *origin* and runs in the direction of the axis specified by *about*. Naturally, *origin* defaults to the origin (0, 0, 0):

```
>>> (2 * Y).rotate(180, about=Y, origin=2 * X).round()
Vector(x=4.0, y=2.0, z=0.0)
>>> O.rotate(90, about=Y, origin=X).round()
Vector(x=1.0, y=0.0, z=1.0)
```


Y
(2*Y)
(2*Y).rotate(180, about=Y, origin=2*X)
O
O.rotate(90, about=Y, origin=X)
Z
X

To aid in certain kinds of rotation, the `angle_between()` method can be used to determine the angle between two vectors (in the plane common to both):

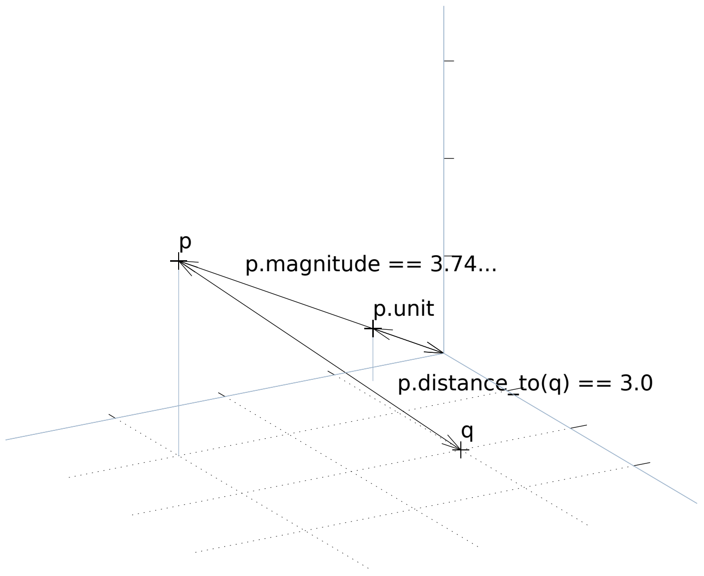
```
>>> X.angle_between(Y)
90.0
>>> p = V(1, 2, 3)
>>> X.angle_between(p)
74.498640433063
```



2.5.7 Magnitudes

The `magnitude` attribute can be used to determine the length of a vector (via Pythagoras' theorem, while the `unit` attribute can be used to obtain a vector in the same direction with a magnitude (length) of 1.0. The `distance_to()` method can also be used to calculate the distance between two vectors (this is simply equivalent to the magnitude of the vector obtained by subtracting one vector from the other):

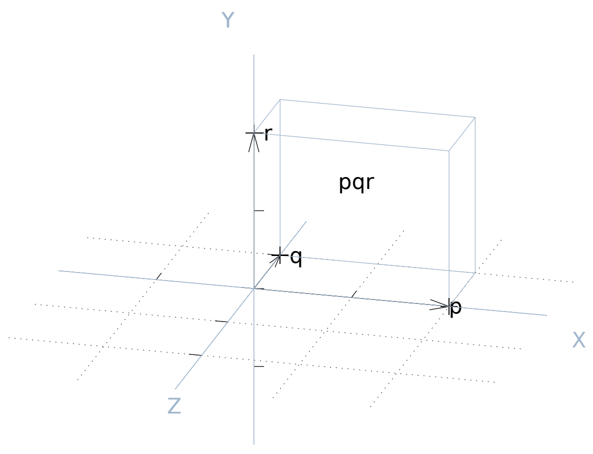
```
>>> p = V(1, 2, 3)
>>> p.magnitude
3.7416573867739413
>>> p.unit
Vector(x=0.2672612419124244, y=0.5345224838248488, z=0.8017837257372732)
>>> p.unit.magnitude
1.0
>>> q = V(2, 0, 1)
>>> p.distance_to(q)
3.0
```



2.5.8 Dot and cross products

The dot and cross products of a vector with another can be calculated using the `dot()` and `cross()` methods respectively. These are useful for determining whether vectors are orthogonal (the dot product of orthogonal vectors is always 0), for finding a vector perpendicular to the plane of two vectors (via the cross product), or for finding the volume of a parallelepiped defined by three vectors, via the triple product:

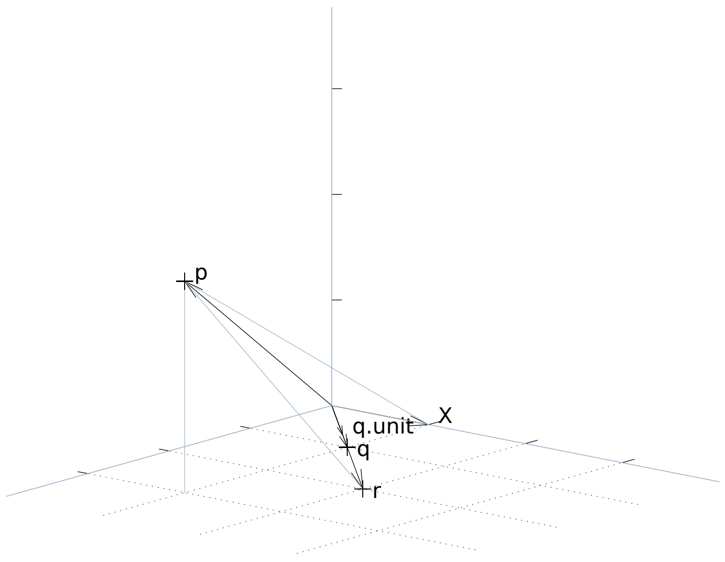
```
>>> p = V(x=2)
>>> q = V(z=-1)
>>> p.dot(q)
0
>>> r = p.cross(q)
>>> r
Vector(x=0, y=2, z=0)
>>> area_of_pqr = p.cross(q).dot(r)
>>> area_of_pqr
4
```



2.5.9 Projection

The final method provided by the `Vector` class is `project()` which implements scalar projection. You might think of this as calculating the length of the shadow one vector casts upon another. Or, put another way, this is the length of one vector in the direction of another (unit) vector:

```
>>> p = V(1, 2, 3)
>>> p.project(X)
1.0
>>> q = X + Z
>>> p.project(q)
2.82842712474619
>>> r = q.unit * p.project(q)
>>> r.round(4)
Vector(x=2.0, y=0.0, z=2.0)
```



2.6 Recipes

2.6.1 Player Position

The player's position can be easily queried with the *pos* attribute. The value is a *Vector*. For example, on the command line:

```
>>> world = World()
>>> world.player.pos
Vector(x=2.3, y=1.1, z=-0.81)
```

Teleporting the player is as simple as assigning a new vector to the player position. Here we teleport the player into the air by adding 50 to the Y-axis of the player's current position (remember that in the Minecraft world, the Y-axis goes up/down):

```
>>> world.player.pos = world.player.pos + Vector(y=50)
```

Or we can use a bit of Python short-hand for this:

```
>>> world.player.pos += Vector(y=50)
```

If you want the player position to the nearest block use the *tile_pos* instead:

```
>>> world.player.pos
Vector(x=2, y=1, z=-1)
```

2.6.2 Blocks

The state of blocks in the world can be queried and changed by reading and writing to the *blocks* attribute. This is indexed with a *Vector* (or slice of vectors) and returns or accepts a *Block* instance. For example, on the command line we can find out the type of block we're standing on like so:

```
>>> world = World()
>>> p = world.player.tile_pos
>>> world.blocks[p - Y]
<Block "dirt" id=3 data=0>
```

We can modify the block we're standing on by assigning a new block type to it:

```
>>> world.blocks[p - Y] = Block('stone')
```

We can modify several blocks surrounding the one we're standing on by assigning to a slice of blocks. Remember that Python slices are half-open so the easiest way to specify the slice is to specify the start and the end inclusively and then simply add one to the end. Here we'll change `p` to represent the vector of the block beneath our feet, then set it and all immediately surrounding blocks to stone:

```
>>> p -= Y
>>> world.blocks[p - (X + Z):p + (X + Z) + 1] = Block('stone')
```

2.6.3 Auto Bridge

This recipe (and several others in this chapter) was shamelessly stolen from Martin O'Hanlon's excellent site which includes lots of recipes (although at the time of writing they're all for the mcpi API). In this case the original script can be found in Martin's auto-bridge project.

The script tracks the position and likely future position of the player as they walk through the world. If the script detects the player is about to walk onto air it changes the block to diamond:

```
from __future__ import unicode_literals

import time
from picraft import World, Vector, Block
from collections import deque

world = World(ignore_errors=True)
world.say('Auto-bridge active')
try:
    bridge = deque()
    last_pos = None
    while True:
        this_pos = world.player.pos
        if last_pos is not None:
            # Has the player moved more than 0.1 units in a horizontal direction?
            movement = (this_pos - last_pos).replace(y=0.0)
            if movement.magnitude > 0.1:
                # Find the next tile they're going to step on
                next_pos = (this_pos + movement.unit).floor() - Vector(y=1)
                if world.blocks[next_pos] == Block('air'):
                    with world.connection.batch_start():
                        bridge.append(next_pos)
                        world.blocks[next_pos] = Block('diamond_block')
                        while len(bridge) > 10:
                            world.blocks[bridge.popleft()] = Block('air')
        last_pos = this_pos
        time.sleep(0.01)
except KeyboardInterrupt:
    world.say('Auto-bridge deactivated')
    with world.connection.batch_start():
        while bridge:
            world.blocks[bridge.popleft()] = Block('air')
```

Note that the script starts by initializing the connection with the `ignore_errors=True` parameter. This causes the picraft library to act like the mcpi library: errors in "set" calls are ignored, but the library reacts faster because of this. This is necessary in a script like this where rapid reaction to player behaviour is required.

2.6.4 Events

The auto-bridge recipe above demonstrates a form of reacting to changes, in that case player position changing. There is a formal event handling mechanism in Minecraft, but at the time of writing the API only exposes the "block hit" event which occurs when a player hits a block with their sword (by right clicking).

The picraft library provides two different ways of working with events; you can select whichever one suits your particular application. The basic way of reacting to events is to periodically "poll" Minecraft for them (with the *poll()* method). This will return a list of all events that occurred since the last time your script polled the server. For example, the following script prints a message to the console when you hit a block, detailing the block's coordinates and the face that you hit:

```
from time import sleep
from picraft import World

world = World()

while True:
    for event in world.events.poll():
        world.say('Player %d hit face %s of block at %d,%d,%d' % (
            event.player.player_id, event.face,
            event.pos.x, event.pos.y, event.pos.z))
    sleep(0.1)
```

This is fine for simple scripts but you can probably see how more complex scripts that check exactly which block has been hit start to involve long series of `if` statements which look a bit ugly in code. The following script creates a couple of blocks near the player on startup: a black block (which ends the script when hit), and a white block (which makes multi-colored blocks fall from the sky):

```
from random import randint
from picraft import World, X, Y, Z, Vector, Block

world = World()

p = world.player.tile_pos
white_pos = p - 2 * X
black_pos = p - 3 * X

world.blocks[white_pos] = Block('#ffffff')
world.blocks[black_pos] = Block('#000000')

running = True
while running:
    for event in world.events.poll():
        if event.pos == white_pos:
            rain = Vector(p.x + randint(-10, 10), p.y + 20, p.z + randint(-10, 10))
            rain_end = world.height[rain]
            world.blocks[rain] = Block('wool', randint(1, 15))
            while rain != rain_end:
                with world.connection.batch_start():
                    world.blocks[rain] = Block('air')
                    rain -= Y
                    world.blocks[rain] = Block('wool', randint(1, 15))
        elif event.pos == black_pos:
            running = False
```

The alternate method of event handling in picraft is to rely on picraft's built-in event loop. This involves "tagging" functions which will react to block hits with the *on_block_hit()* decorator, then running the *main_loop()* method. This causes picraft to continually poll the server and call the tagged functions when their criteria are matched by a block-hit event:

```
from random import randint
from picraft import World, X, Y, Z, Vector, Block

world = World()

p = world.player.tile_pos
```

```
white_pos = p - 2 * X
black_pos = p - 3 * X

world.blocks[white_pos] = Block('#ffffff')
world.blocks[black_pos] = Block('#000000')

@world.events.on_block_hit(pos=black_pos)
def stop_script(event):
    world.connection.close()

@world.events.on_block_hit(pos=white_pos)
def make_it_rain(event):
    rain = Vector(p.x + randint(-10, 10), p.y + 20, p.z + randint(-10, 10))
    rain_end = world.height[rain]
    world.blocks[rain] = Block('wool', randint(1, 15))
    while rain != rain_end:
        with world.connection.batch_start():
            world.blocks[rain] = Block('air')
            rain -= Y
            world.blocks[rain] = Block('wool', randint(1, 15))

world.events.main_loop()
```

One advantage of this method (other than slightly cleaner code) is that event handlers can easily be made multi-threaded (to run in parallel with each other) simply by modifying the decorator used:

```
from random import randint
from picraft import World, X, Y, Z, Vector, Block

world = World()

p = world.player.tile_pos
white_pos = p - 2 * X
black_pos = p - 3 * X

world.blocks[white_pos] = Block('#ffffff')
world.blocks[black_pos] = Block('#000000')

@world.events.on_block_hit(pos=black_pos)
def stop_script(event):
    world.connection.close()

@world.events.on_block_hit(pos=white_pos, thread=True)
def make_it_rain(event):
    rain = Vector(p.x + randint(-10, 10), p.y + 20, p.z + randint(-10, 10))
    rain_end = world.height[rain]
    world.blocks[rain] = Block('wool', randint(1, 15))
    while rain != rain_end:
        with world.connection.batch_start():
            world.blocks[rain] = Block('air')
            rain -= Y
            world.blocks[rain] = Block('wool', randint(1, 15))

world.events.main_loop()
```

Now you should find that the rain all falls simultaneously (more or less, given the constraints of the Pi's bandwidth!) when you hit the white block multiple times.

2.6.5 Shapes

This recipe demonstrates drawing shapes with blocks in the Minecraft world. The picraft library includes a couple of rudimentary routines for calculating the points necessary for drawing lines:

- *line()* which can be used to calculate the positions along a single line
- *lines()* which calculates the positions along a series of lines

Here we will attempt to construct a script which draws each regular polygon from an equilateral triangle up to a regular octagon. First we start by defining a function which will generate the points of a regular polygon. This is relatively simple: the interior angles of a polygon always add up to 180 degrees so the angle to turn each time is 180 divided by the number of sides. Given an origin and a side-length it's a simple matter to iterate over each side generating the necessary point:

```
from __future__ import division

import math
from picraft import World, Vector, O, X, Y, Z, lines

def polygon(sides, center=O, radius=5):
    angle = 2 * math.pi / sides
    for side in range(sides):
        yield Vector(
                center.x + radius * math.cos(side * angle),
                center.y + radius * math.sin(side * angle))

print(list(polygon(3, center=3*Y)))
print(list(polygon(4, center=3*Y)))
print(list(polygon(5, center=3*Y)))
```

Next we need a function which will iterate over the number of sides for each required polygon, using the *lines()* function to generate the points required to draw the shape. Then it's a simple matter to draw each polygon in turn, wiping it before displaying the next one:

```
from __future__ import division

import math
from picraft import World, Vector, O, X, Y, Z, lines

def polygon(sides, center=O, radius=5):
    angle = 2 * math.pi / sides
    for side in range(sides):
        yield Vector(
                center.x + radius * math.cos(side * angle),
                center.y + radius * math.sin(side * angle))

def shapes():
    for sides in range(3, 9):
        yield lines(polygon(sides, center=3*Y))

w = World()
for shape in shapes():
    # Draw the shape
    with w.connection.batch_start():
        for p in shape:
            w.blocks[p] = Block('stone')
    sleep(0.5)
    # Wipe the shape
    with w.connection.batch_start():
        for p in shape:
            w.blocks[p] = Block('air')
```

2.6.6 Animation

This recipe demonstrates, in a series of steps, the construction of a simplistic animation system in Minecraft. Our aim is to create a simple stone cube which rotates about the X axis somewhere in the air. Our first script uses `vector_range()` to obtain the coordinates of all blocks within the cube, then uses the `rotate()` method to rotate them about the X axis:

```
from __future__ import division

from time import sleep
from picraft import World, Vector, X, Y, Z, vector_range, Block

world = World()
world.checkpoint.save()
try:
    cube_range = vector_range(Vector() - 2, Vector() + 2 + 1)
    # Draw frame 1
    state = {}
    for v in cube_range:
        state[v + (5 * Y)] = Block('stone')
    with world.connection.batch_start():
        for v, b in state.items():
            world.blocks[v] = b
    sleep(0.2)
    # Wipe frame 1
    with world.connection.batch_start():
        for v in state:
            world.blocks[v] = Block('air')
    # Draw frame 2
    state = {}
    for v in cube_range:
        state[v.rotate(15, about=X).round() + (5 * Y)] = Block('stone')
    with world.connection.batch_start():
        for v, b in state.items():
            world.blocks[v] = b
    sleep(0.2)
    # and so on...
finally:
    world.checkpoint.restore()
```

As you can see in the script above we draw the first frame, wait for a bit, then wipe the frame by setting all coordinates in that frame's state back to "air". Then we draw the second frame and wait for a bit.

Although this approach works, it's obviously very long winded for lots of frames. What we want to do is calculate the state of each frame in a function. This next version demonstrates this approach; we use a generator function to yield the state of each frame in turn so we can iterate over the frames with a simple `for` loop.

We represent the state of a frame of our animation as a dict which maps coordinates (in the form of `Vector` instances) to `Block` instances:

```
from __future__ import division

from time import sleep
from picraft import World, Vector, X, Y, Z, vector_range, Block


def animation_frames(count):
    cube_range = vector_range(Vector() - 2, Vector() + 2 + 1)
    for frame in range(count):
        state = {}
        for v in cube_range:
            state[v.rotate(15 * frame, about=X).round() + (5 * Y)] = Block('stone')
        yield state


world = World()
world.checkpoint.save()
try:
    for frame in animation_frames(10):
        # Draw frame
        with world.connection.batch_start():
            for v, b in frame.items():
                world.blocks[v] = b
        sleep(0.2)
        # Wipe frame
        with world.connection.batch_start():
            for v, b in frame.items():
                world.blocks[v] = Block('air')
finally:
    world.checkpoint.restore()
```

That's more like it, but the updates aren't terribly fast despite using the batch functionality. In order to improve this we should only update those blocks which have actually changed between each frame. Thankfully, because we're storing the state of each as a dict, this is quite easy:

```
from __future__ import division

from time import sleep
from picraft import World, Vector, X, Y, Z, vector_range, Block


def animation_frames(count):
    cube_range = vector_range(Vector() - 2, Vector() + 2 + 1)
    for frame in range(count):
        yield {
            v.rotate(15 * frame, about=X).round() + (5 * Y): Block('stone')
            for v in cube_range}


def track_changes(states, default=Block('air')):
    old_state = None
    for state in states:
        # Assume the initial state of the blocks is the default ('air')
        if old_state is None:
            old_state = {v: default for v in state}
        # Build a dict of those blocks which changed from old_state to state
        changes = {v: b for v, b in state.items() if old_state.get(v) != b}
        # Blank out blocks which were in old_state but aren't in state
        changes.update({v: default for v in old_state if v not in state})
        yield changes
        old_state = state


world = World()
world.checkpoint.save()
try:
    for state in track_changes(animation_frames(20)):
        with world.connection.batch_start():
            for v, b in state.items():
                world.blocks[v] = b
        sleep(0.2)
finally:
    world.checkpoint.restore()
```

Note: this still isn't perfect. Ideally, we would identify contiguous blocks of coordinates to be updated which have the same block and set them all at the same time (which will utilize the *world.setBlocks* call for efficiency). However, this is relatively complex to do well so I shall leave it as an exercise for you, dear reader!

2.6.7 Minecraft TV

If you've got a Raspberry Pi camera module, you can build a TV to view a live feed from the camera in the Minecraft world. Firstly we need to construct a class which will accept JPEGs from the camera's MJPEG stream, and render them as blocks in the Minecraft world. Then we need a class to construct the TV model itself and enable interaction with it:

```python
from __future__ import division

import io
import time
import picamera
from picraft import World, V, Block
from picraft.block import _BLOCKS_BY_COLOR
from PIL import Image


class MinecraftTVScreen(object):
    def __init__(self, world, origin, size):
        self.world = world
        self.origin = origin
        self.size = size
        self.jpeg = None
        # Construct a palette for PIL
        self.palette = Image.new('P', (1, 1))
        self.palette_len = len(_BLOCKS_BY_COLOR)
        PALETTE = {data: color for color, (id, data) in _BLOCKS_BY_COLOR.items()}
        PALETTE = [PALETTE[i] for i in range(16)]
        self.palette.putpalette(
            [c for rgb in PALETTE for c in rgb] +
            list(PALETTE[0]) * (256 - len(PALETTE))
            )

    def write(self, buf):
        if buf.startswith(b'\xff\xd8'):
            if self.jpeg:
                self.jpeg.seek(0)
                self.render(self.jpeg)
            self.jpeg = io.BytesIO()
        self.jpeg.write(buf)

    def close(self):
        self.jpeg = None

    def render(self, jpeg):
        o = self.origin
        img = Image.open(jpeg)
        img = img.resize(self.size, Image.BILINEAR)
        img = img.quantize(self.palette_len, palette=self.palette)
        with self.world.connection.batch_start():
            for x in range(img.size[0]):
                for y in range(img.size[1]):
                    self.world.blocks[o + V(0, y, x)] = Block.from_id(35, img.getpixel((x, y)))


class MinecraftTV(object):
    def __init__(self, origin=V(), size=(12, 8)):
        self.camera = picamera.PiCamera()
        self.camera.resolution = (64, int(64 / size[0] * size[1]))
        self.camera.framerate = 2
        self.world = World(ignore_errors=True)
        self.origin = origin
        self.size = V(0, size[1], size[0])
        self.button_vec = None
        self.screen = MinecraftTVScreen(
            self.world, origin + V(0, 1, 1), (size[0] - 2, size[1] - 2))

    def main_loop(self):
        self.create_tv()
        try:
            while True:
                for event in self.world.events.poll():
                    if event.pos == self.button_vec:
                        if self.camera.recording:
                            self.switch_off()
                        else:
                            self.switch_on()
                time.sleep(0.1)
        finally:
            if self.camera.recording:
                self.switch_off()
            self.destroy_tv()

    def create_tv(self):
        o = self.origin
        self.world.blocks[o:o + self.size + 1] = Block('#ffffff')
        self.world.blocks[
            o + V(0, 1, 1):o + self.size - V(0, 1, 1) + 1] = Block('#000000')
        self.button_vec = o + V(z=2)
        self.world.blocks[self.button_vec] = Block('#800000')

    def destroy_tv(self):
        o = self.origin
        self.world.blocks[o:o + self.size + 1] = Block('air')

    def switch_on(self):
        self.camera.start_recording(self.screen, format='mjpeg')

    def switch_off(self):
        self.camera.stop_recording()
        o = self.origin
        self.world.blocks[
            o + V(0, 1, 1):o + self.size - V(0, 2, 2) + 1] = Block('#000000')


tv = MinecraftTV(origin=V(2, 0, 5), size=(24,16))
tv.main_loop()
```

Don't expect to be able to recognize much in the Minecraft TV; the resolution is extremely low and the color matching is far from perfect. Still, if you point the camera at obvious blocks of primary colors and move it around slowly you should see a similar result on the in-game display.

The script includes the ability to position and size the TV as you like, and you may like to experiment with adding new controls to it!

2.7 Frequently Asked Questions

None yet, but then it's the first release! Feel free to ask the author, or add questions to the issue tracker on GitHub, or even edit this document yourself and add frequently asked questions you've seen on other forums!

2.8 API Reference

The picraft package consists of several modules which permit access to and modification of a Minecraft world. The package is intended as an alternative Python API to the "official" Minecraft Python API (for reasons explained in the *Frequently Asked Questions*).

The classes defined in most modules of this package are available directly from the `picraft` namespace. In other words, the following code is typically all that is required to access classes in this package:

```
import picraft
```

For convenience on the command line you may prefer to simply do the following:

```
from picraft import *
```

However, this is frowned upon in code as it pulls everything into the global namespace, so you may prefer to do something like this:

```
from picraft import World, Vector, Block
```

This is the style used in the *Recipes* chapter. Sometimes, if you are using the `Vector` class extensively, you may wish to use the short-cuts for it:

```
from picraft import World, V, O, X, Y, Z, Block
```

The following sections document the various modules available within the package:

- *API - The World class*
- *API - The Block class*
- *API - Vector, vector_range, etc.*
- *API - Events*
- *API - Connections and Batches*
- *API - Players*
- *API - Exceptions*

2.9 API - The World class

The world module defines the `World` class, which is the usual way of starting a connection to a Minecraft server and which then provides various attributes allowing the user to query and manipulate that world.

Note: All items in this module are available from the `picraft` namespace without having to import `picraft.world` directly.

The following items are defined in the module:

2.9.1 World

class `picraft.world.`**`World`**(*host=u'localhost'*, *port=4711*, *timeout=0.3*, *ignore_errors=False*)

Represents a Minecraft world.

This is the primary class that users interact with. Construct an instance of this class, optionally specifying the *host* and *port* of the server (which default to "localhost" and 4711 respectively). Afterward, the instance can be used to query and manipulate the minecraft world of the connected game.

The `say()` method can be used to send commands to the console, while the `player` attribute can be used to manipulate or query the status of the player character in the world. The `players` attribute can be used to manipulate or query other players within the world (this object can be iterated over to discover players):

```
>>> from picraft import *
>>> world = World()
>>> len(world.players)
1
>>> world.say('Hello, world!')
```

say (*message*)

Displays *message* in the game's chat console.

The *message* parameter must be a string (which may contain multiple lines). Each line of the message will be sent to the game's chat console and displayed immediately. For example:

```
>>> world.say('Hello, world!')
>>> world.say('The following player IDs exist:\n%s' %
...     '\n'.join(str(p) for p in world.players))
```

blocks

Represents the state of blocks in the Minecraft world.

This property can be queried to determine the type of a block in the world, or can be set to alter the type of a block. The property can be indexed with a single `Vector`, in which case the state of a single block is returned (or updated) as a `Block` object:

```
>>> world.blocks[g.player.tile_pos]
<Block "grass" id=2 data=0>
```

Alternatively, a slice of vectors can be used. In this case, when querying the property, a sequence of `Block` objects is returned, When setting a slice of vectors you can either pass a sequence of `Block` objects or a single `Block` object. The sequence must be equal to the number of blocks represented by the slice:

```
>>> world.blocks[Vector(0,0,0):Vector(2,1,1)]
[<Block "grass" id=2 data=0>,<Block "grass" id=2 data=0>]
>>> world.blocks[Vector(0,0,0):Vector(5,1,5)] = Block.from_name('grass')
```

As with normal Python slices, the interval specified is half-open. That is to say, it is inclusive of the lower vector, *exclusive* of the upper one. Hence, `Vector():Vector(x=5,1,1)` represents the coordinates (0,0,0) to (4,0,0). It is usually useful to specify the upper bound as the vector you want and then add one to it:

```
>>> world.blocks[Vector():Vector(x=1) + 1]
[<Block "grass" id=2 data=0>,<Block "grass" id=2 data=0>]
>>> world.blocks[Vector():Vector(4,0,4) + 1] = Block.from_name('grass')
```

Warning: Querying or setting sequences of blocks can be extremely slow as a network transaction must be executed for each individual block. When setting a slice of blocks, this can be speeded up by specifying a single `Block` in which case one network transaction will occur to set all blocks in the slice. The Raspberry Juice server also supports querying sequences of blocks with a single command (picraft will automatically use this). Additionally, `batch_start()` can be used to speed up setting sequences of blocks (though not querying).

camera

Represents the camera of the Minecraft world.

The `Camera` object contained in this property permits control of the position of the virtual camera in the Minecraft world. For example, to position the camera directly above the host player:

```
>>> world.camera.third_person(world.player)
```

Alternatively, to see through the eyes of a specific player:

```
>>> world.camera.first_person(world.players[2])
```

Warning: Camera control is only supported on Minecraft Pi edition.

checkpoint

Represents the Minecraft world checkpoint system.

The *Checkpoint* object contained in this attribute provides the ability to save and restore the state of the world at any time:

```
>>> world.checkpoint.save()
>>> world.blocks[Vector()] = Block.from_name('stone')
>>> world.checkpoint.restore()
```

connection

Represents the connection to the Minecraft server.

The *Connection* object contained in this attribute represents the connection to the Minecraft server and provides various methods for communicating with it. Users will very rarely need to access this attribute, except to use the *batch_start()* method.

events

Provides an interface to poll events that occur in the Minecraft world.

The *Events* object contained in this property provides methods for determining what is happening in the Minecraft world:

```
>>> events = world.events.poll()
>>> len(events)
3
>>> events[0]
<BlockHitEvent pos=1,1,1 face="x+" player=1>
>>> events[0].player.pos
<Vector x=0.5, y=0.0, z=0.5>
```

height

Represents the height of the Minecraft world.

This property can be queried to determine the height of the world at any location. The property can be indexed with a single *Vector*, in which case the height will be returned as a vector with the same X and Z coordinates, but a Y coordinate adjusted to the first non-air block from the top of the world:

```
>>> world.height[Vector(0, -10, 0)]
Vector(x=0, y=0, z=0)
```

Alternatively, a slice of two vectors can be used. In this case, the property returns a sequence of *Vector* objects each with their Y coordinates adjusted to the height of the world at the respective X and Z coordinates.

immutable

Write-only property which sets whether the world is changeable.

Warning: World settings are only supported on Minecraft Pi edition.

Note: Unfortunately, the underlying protocol provides no means of reading a world setting, so this property is write-only (attempting to query it will result in an `AttributeError` being raised).

nametags_visible

Write-only property which sets whether players' nametags are visible.

Warning: World settings are only supported on Minecraft Pi edition.

Note: Unfortunately, the underlying protocol provides no means of reading a world setting, so this property is write-only (attempting to query it will result in an `AttributeError` being raised).

player

Represents the host player in the Minecraft world.

The *HostPlayer* object returned by this attribute provides properties which can be used to query the status of, and manipulate the state of, the host player in the Minecraft world:

```
>>> world.player.pos
Vector(x=-2.49725, y=18.0, z=-4.21989)
>>> world.player.tile_pos += Vector(y=50)
```

players

Represents all player entities in the Minecraft world.

This property can be queried to determine which players are currently in the Minecraft world. The property is a mapping of player id (an integer number) to a *Player* object which permits querying and manipulation of the player. The property supports many of the methods of dicts and can be iterated over like a dict:

```
>>> len(world.players)
1
>>> list(world.players)
[1]
>>> world.players.keys()
[1]
>>> world.players[1]
<picraft.player.Player at 0x7f2f91f38cd0>
>>> world.players.values()
[<picraft.player.Player at 0x7f2f91f38cd0>]
>>> world.players.items()
[(1, <picraft.player.Player at 0x7f2f91f38cd0>)]
>>> for player in world.players:
...     print(player.tile_pos)
...
-3,18,-5
```

On the Raspberry Juice platform, you can also use player name to reference players:

```
>>> world.players['my_player']
<picraft.player.Player at 0x7f2f91f38cd0>
```

2.9.2 Checkpoint

class `picraft.world.`**`Checkpoint`**(*connection*)

Permits restoring the world state from a prior save.

This class provides methods for storing the state of the Minecraft world, and restoring the saved state at a later time. The *save()* method saves the state of the world, and the *restore()* method restores the saved state.

This class can be used as a context manager to take a checkpoint, make modifications to the world, and roll them back if an exception occurs. For example, the following code will ultimately do nothing because an exception occurs after the alteration:

```
>>> from picraft import *
>>> w = World()
>>> with w.checkpoint:
```

```
...     w.blocks[w.player.tile_pos - Vector(y=1)] = Block.from_name('stone')
...     raise Exception()
```

Warning: Checkpoints are only supported on Minecraft Pi edition.

Warning: Minecraft only permits a single checkpoint to be stored at any given time. There is no capability to save multiple checkpoints and no way of checking whether one currently exists. Therefore, storing a checkpoint may overwrite an older checkpoint without warning.

Note: Checkpoints don't work *within* batches as the checkpoint save will be batched along with everything else. That said, a checkpoint can be used *outside* a batch to roll the entire thing back if it fails:

```
>>> v = w.player.tile_pos - Vector(y=1)
>>> with w.checkpoint:
...     with w.connection.batch_start():
...         w.blocks[v - Vector(2, 0, 2):v + Vector(2, 1, 2)] = [
...             Block.from_name('wool', data=i) for i in range(16)]
```

restore()
Restore the state of the Minecraft world from a previously saved checkpoint. No facility is provided to determine whether a prior checkpoint is available (the underlying network protocol doesn't permit this).

save()
Save the state of the Minecraft world, overwriting any prior checkpoint state.

2.9.3 Camera

class picraft.world.**Camera**(*connection*)
This class implements the `camera` attribute.

first_person(*player*)
Causes the camera to view the world through the eyes of the specified player. The *player* can be the `player` attribute (representing the host player) or an attribute retrieved from the `players` list. For example:

```
>>> from picraft import World
>>> w = World()
>>> w.camera.first_person(w.player)
>>> w.camera.first_person(w.players[1])
```

third_person(*player*)
Causes the camera to follow the specified player from above. The *player* can be the `player` attribute (representing the host player) or an attribute retrieved from the `players` list. For example:

```
>>> from picraft import World
>>> w = World()
>>> w.camera.third_person(w.player)
>>> w.camera.third_person(w.players[1])
```

pos
Write-only property which sets the camera's absolute position in the world.

Note: Unfortunately, the underlying protocol provides no means of reading this setting, so this property is write-only (attempting to query it will result in an `AttributeError` being raised).

2.10 API - The Block class

The block module defines the `Block` class, which is used to represent the type of a block and any associated data it may have, and the `Blocks` class which is used to implement the `blocks` attribute on the `World` class.

Note: All items in this module, except the compatibility constants, are available from the `picraft` namespace without having to import `picraft.block` directly.

The following items are defined in the module:

2.10.1 Block

class `picraft.block.`**`Block`**`(`*id*, *data*`)`

Represents a block within the Minecraft world.

Blocks within the Minecraft world are represented by two values: an *id* which defines the type of the block (air, stone, grass, wool, etc.) and an optional *data* value (defaults to 0) which means different things for different block types (e.g. for wool it defines the color of the wool).

Blocks are represented by this library as a `namedtuple()` of the *id* and the *data*. Calculated properties are provided to look up the `name` and `description` of the block from a database derived from the Minecraft wiki, and classmethods are defined to construct a block definition from an `id` or from alternate things like a `name` or an `RGB color`:

```
>>> Block.from_id(0, 0)
<Block "air" id=0 data=0>
>>> Block.from_id(2, 0)
<Block "grass" id=2 data=0>
>>> Block.from_name('stone')
<Block "stone" id=1 data=0>
>>> Block.from_color('#ffffff')
<Block "wool" id=35 data=0>
```

The default constructor attempts to guess which classmethod to call based on the following rules (in the order given):

1. If the constructor is passed a string beginning with '#' that is 7 characters long, it calls `from_color()`
2. If the constructor is passed a tuple containing 3 values, it calls `from_color()`
3. If the constructor is passed a string (not matching the case above) it calls `from_name()`
4. Otherwise the constructor calls `from_id()` with all given parameters

This means that the examples above can be more easily written:

```
>>> Block(0, 0)
<Block "air" id=0 data=0>
>>> Block(2, 0)
<Block "grass" id=2 data=0>
>>> Block('stone')
<Block "stone" id=1 data=0>
>>> Block('#ffffff')
<Block "wool" id=35 data=0>
```

Aliases are provided for compatibility with the official reference implementation (AIR, GRASS, STONE, etc):

```
>>> import picraft.block
>>> picraft.block.WATER
<Block "flowing_water" id=8 data=0>
```

classmethod `from_color`(*color*, *exact=False*)

Construct a *`Block`* instance from a *color* which can be represented as:

- A tuple of `(red, green, blue)` integer byte values between 0 and 255
- A tuple of `(red, green, blue)` float values between 0.0 and 1.0
- A string in the format '#rrggbb' where rr, gg, and bb are hexadecimal representations of byte values.

If *exact* is `False` (the default), and an exact match for the requested color cannot be found, the nearest color (determined simply by Euclidian distance) is returned. If *exact* is `True` and an exact match cannot be found, a `ValueError` will be raised:

```
>>> from picraft import *
>>> Block.from_color('#ffffff')
<Block "wool" id=35 data=0>
>>> Block.from_color('#ffffff', exact=True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "picraft/block.py", line 351, in from_color
    if exact:
ValueError: no blocks match color #ffffff
>>> Block.from_color((1, 0, 0))
<Block "wool" id=35 data=14>
```

Note that calling the default constructor with any of the formats accepted by this method is equivalent to calling this method:

```
>>> Block('#ffffff')
<Block "wool" id=35 data=0>
```

classmethod `from_id`(*id*, *data=0*)

Construct a *`Block`* instance from an *id* integer. This may be used to construct blocks in the classic manner; by specifying a number representing the block's type, and optionally a data value. For example:

```
>>> from picraft import *
>>> Block.from_id(1)
<Block "stone" id=1 data=0>
>>> Block.from_id(2, 0)
<Block "grass" id=2 data=0>
```

The optional *data* parameter defaults to 0. Note that calling the default constructor with an integer parameter is equivalent to calling this method:

```
>>> Block(1)
<Block "stone" id=1" data=0>
```

classmethod `from_name`(*name*, *data=0*)

Construct a *`Block`* instance from a *name*, as returned by the *`name`* property. This may be used to construct blocks in a more "friendly" way within code. For example:

```
>>> from picraft import *
>>> Block.from_name('stone')
<Block "stone" id=1 data=0>
>>> Block.from_name('wool', data=2)
<Block "wool" id=35 data=2>
```

The optional *data* parameter can be used to specify the data component of the new *`Block`* instance; it defaults to 0. Note that calling the default constructor with a string that doesn't start with "#" is equivalent to calling this method:

```
>>> Block('stone')
<Block "stone" id=1 data=0>
```

id
The "id" or type of the block. Each block type in Minecraft has a unique value. For example, air blocks have the id 0, stone, has id 1, and so forth. Generally it is clearer in code to refer to a block's `name` but it may be quicker to use the id.

data
Certain types of blocks have variants which are dictated by the data value associated with them. For example, the color of a wool block is determined by the *data* attribute (e.g. white is 0, red is 14, and so on).

pi
Returns a bool indicating whether the block is present in the Pi Edition of Minecraft.

pocket
Returns a bool indicating whether the block is present in the Pocket Edition of Minecraft.

name
Return the name of the block. This is a unique identifier string which can be used to construct a `Block` instance with `from_name()`.

description
Return a description of the block. This string is not guaranteed to be unique and is only intended for human use.

2.10.2 BLOCK_COLORS

`picraft.block.`**`BLOCK_COLORS`**
A set of the available colors that can be used with `Block.from_color()`. Each color is represented as `(red, green, blue)` tuple where each component is an integer between 0 and 255.

2.10.3 Compatibility

Finally, the module also contains compatibility values equivalent to those in the mcpi.block module of the reference implementation. Each value represents the type of a block with no associated data:

AIR	FURNACE_ACTIVE	MUSHROOM_RED
BED	FURNACE_INACTIVE	NETHER_REACTOR_CORE
BEDROCK	GLASS	OBSIDIAN
BEDROCK_INVISIBLE	GLASS_PANE	REDSTONE_ORE
BOOKSHELF	GLOWING_OBSIDIAN	SAND
BRICK_BLOCK	GLOWSTONE_BLOCK	SANDSTONE
CACTUS	GOLD_BLOCK	SAPLING
CHEST	GOLD_ORE	SNOW
CLAY	GRASS	SNOW_BLOCK
COAL_ORE	GRASS_TALL	STAIRS_COBBLESTONE
COBBLESTONE	GRAVEL	STAIRS_WOOD
COBWEB	ICE	STONE
CRAFTING_TABLE	IRON_BLOCK	STONE_BRICK
DIAMOND_BLOCK	IRON_ORE	STONE_SLAB
DIAMOND_ORE	LADDER	STONE_SLAB_DOUBLE
DIRT	LAPIS_LAZULI_BLOCK	SUGAR_CANE
DOOR_IRON	LAPIS_LAZULI_ORE	TNT
DOOR_WOOD	LAVA	TORCH
FARMLAND	LAVA_FLOWING	WATER
FENCE	LAVA_STATIONARY	WATER_FLOWING
FENCE_GATE	LEAVES	WATER_STATIONARY
FIRE	MELON	WOOD
FLOWER_CYAN	MOSS_STONE	WOOD_PLANKS
FLOWER_YELLOW	MUSHROOM_BROWN	WOOL

Use these compatibility constants by importing the block module explicitly. For example:

```
>>> from picraft import block
>>> block.AIR
<Block "air" id=0 data=0>
>>> block.TNT
<Block "tnt" id=46 data=0>
```

2.11 API - Vector, vector_range, etc.

The vector module defines the *Vector* class, which is the usual method of represent coordinates or vectors when dealing with the Minecraft world. It also provides functions like *vector_range()* for generating sequences of vectors.

Note: All items in this module are available from the *picraft* namespace without having to import *picraft.vector* directly.

The following items are defined in the module:

2.11.1 Vector

class picraft.vector.**Vector**(*x=0*, *y=0*, *z=0*)

Represents a 3-dimensional vector.

This tuple derivative represents a 3-dimensional vector with x, y, z components. Instances can be constructed in a number of ways. By explicitly specifying the x, y, and z components (optionally with keyword identifiers), or leaving the empty to default to 0:

```
>>> Vector(1, 1, 1)
Vector(x=1, y=1, z=1)
>>> Vector(x=2, y=0, z=0)
Vector(x=2, y=0, z=0)
>>> Vector()
Vector(x=0, y=0, z=0)
>>> Vector(y=10)
Vector(x=0, y=10, z=0)
```

Shortcuts are available for the X, Y, and Z axes:

```
>>> X
Vector(x=1, y=0, z=0)
>>> Y
Vector(x=0, y=1, z=0)
```

Note that vectors don't much care whether their components are integers, floating point values, or `None`:

```
>>> Vector(1.0, 1, 1)
Vector(x=1.0, y=1, z=1)
>>> Vector(2, None, None)
Vector(x=2, y=None, z=None)
```

The class supports simple arithmetic operations with other vectors such as addition and subtraction, along with multiplication and division with scalars, raising to powers, bit-shifting, and so on. Such operations are performed element-wise [1]:

```
>>> v1 = Vector(1, 1, 1)
>>> v2 = Vector(2, 2, 2)
```

[1] I realize math purists will hate this (and demand that abs() should be magnitude and * should invoke matrix multiplication), but the element wise operations are sufficiently useful to warrant the short-hand syntax.

```
>>> v1 + v2
Vector(x=3, y=3, z=3)
>>> 2 * v2
Vector(x=4, y=4, z=4)
```

Simple arithmetic operations with scalars return a new vector with that operation performed on all elements of the original. For example:

```
>>> v = Vector()
>>> v
Vector(x=0, y=0, z=0)
>>> v + 1
Vector(x=1, y=1, z=1)
>>> 2 * (v + 2)
Vector(x=4, y=4, z=4)
>>> Vector(y=2) ** 2
Vector(x=0, y=4, z=0)
```

Within the Minecraft world, the X,Z plane represents the ground, while the Y vector represents height.

Note: Note that, as a derivative of tuple, instances of this class are immutable. That is, you cannot directly manipulate the x, y, and z attributes; instead you must create a new vector (for example, by adding two vectors together). The advantage of this is that vector instances can be used in sets or as dictionary keys.

replace(*x=None*, *y=None*, *z=None*)
Return the vector with the x, y, or z axes replaced with the specified values. For example:

```
>>> Vector(1, 2, 3).replace(z=4)
Vector(x=1, y=2, z=4)
```

ceil()
Return the vector with the ceiling of each component. This is only useful for vectors containing floating point components:

```
>>> Vector(0.5, -0.5, 1.2)
Vector(1.0, 0.0, 2.0)
```

floor()
Return the vector with the floor of each component. This is only useful for vectors containing floating point components:

```
>>> Vector(0.5, -0.5, 1.9)
Vector(0.0, -1.0, 1.0)
```

dot(*other*)
Return the dot product of the vector with the *other* vector. The result is a scalar value. For example:

```
>>> Vector(1, 2, 3).dot(Vector(2, 2, 2))
12
>>> Vector(1, 2, 3).dot(X)
1
```

cross(*other*)
Return the cross product of the vector with the *other* vector. The result is another vector. For example:

```
>>> Vector(1, 2, 3).cross(Vector(2, 2, 2))
Vector(x=-2, y=4, z=-2)
>>> Vector(1, 2, 3).cross(X)
Vector(x=0, y=3, z=-2)
```

distance_to(*other*)
Return the Euclidian distance between two three dimensional points (represented as vectors), calculated according to Pythagoras' theorem. For example:

```
>>> Vector(1, 2, 3).distance_to(Vector(2, 2, 2))
1.4142135623730951
>>> Vector().distance_to(X)
1.0
```

angle_between(*other*)

Returns the angle between this vector and the *other* vector on a plane that contains both vectors. The result is measured in degrees. For example:

```
>>> X.angle_between(Y)
90.0
>>> (X + Y).angle_between(X)
45.00000000000001
```

project(*other*)

Return the scalar projection of this vector onto the *other* vector. This is a scalar indicating the length of this vector in the direction of the *other* vector. For example:

```
>>> Vector(1, 2, 3).project(2 * Y)
2.0
>>> Vector(3, 4, 5).project(Vector(3, 4, 0))
5.0
```

rotate(*angle*, *about*, *origin=None*)

Return this vector after rotation of *angle* degrees about the line passing through *origin* in the direction *about*. Origin defaults to the vector 0, 0, 0. Hence, if this parameter is omitted this method calculates rotation about the axis (through the origin) defined by *about*. For example:

```
>>> Y.rotate(90, about=X)
Vector(x=0, y=6.123233995736766e-17, z=1.0)
>>> Vector(3, 4, 5).rotate(30, about=X, origin=10 * Y)
Vector(x=3.0, y=2.3038475772933684, z=1.330127018922194)
```

Information about rotation around arbitrary lines was obtained from Glenn Murray's informative site.

x

The position or length of the vector along the X-axis. In the Minecraft world this can be considered to run left-to-right.

y

The position or length of the vector along the Y-axis. In the Minecraft world this can be considered to run vertically up and down.

z

The position or length of the vector along the Z-axis. In the Minecraft world this can be considered as depth (in or out of the screen).

magnitude

Returns the magnitude of the vector. This could also be considered the distance of the vector from the origin, i.e. `v.magnitude` is equivalent to `Vector().distance_to(v)`. For example:

```
>>> Vector(2, 4, 4).magnitude
6.0
>>> Vector().distance_to(Vector(2, 4, 4))
6.0
```

unit

Return a unit vector (a vector with a magnitude of one) with the same direction as this vector:

```
>>> X.unit
Vector(x=1.0, y=0.0, z=0.0)
>>> (2 * Y).unit
Vector(x=0.0, y=1.0, z=0.0)
```

Note: If the vector's magnitude is zero, this property returns the original vector.

2.11.2 Short-hand variants

The *Vector* class is used sufficiently often to justify the inclusion of some shortcuts. The class itself is also available as `V`, and vectors representing the three axes are each available as `X`, `Y`, and `Z`. Finally, a vector representing the origin is available as `O`:

```
>>> from picraft import V, O, X, Y, Z
>>> O
Vector(x=0, y=0, z=0)
>>> 2 * X
Vector(x=2, y=0, z=0)
>>> X + Y
Vector(x=1, y=1, z=0)
>>> (X + Y).angle_between(X)
45.00000000000001
>>> V(3, 4, 5).projection(X)
3.0
>>> X.rotate(90, about=Y)
Vector(x=0.0, y=0.0, z=1.0)
```

2.11.3 vector_range

class `picraft.vector.`**`vector_range`**(*start*, *stop=None*, *step=None*, *order=u'zxy'*)

Like `range()`, *vector_range* is actually a type which efficiently represents a range of vectors. The arguments to the constructor must be *Vector* instances (or objects which have integer `x`, `y`, and `z` attributes).

If *step* is omitted, it defaults to `Vector(1, 1, 1)`. If the *start* argument is omitted, it defaults to `Vector(0, 0, 0)`. If any element of the *step* vector is zero, `ValueError` is raised.

The contents of the range are largely determined by the *step* and *order* which specifies the order in which the axes of the range will be incremented. For example, with the order `'xyz'`, the X-axis will be incremented first, followed by the Y-axis, and finally the Z-axis. So, for a range with the default *start*, *step*, and *stop* set to `Vector(3, 3, 3)`, the contents of the range will be:

```
>>> list(vector_range(Vector(3, 3, 3), order='xyz'))
[Vector(0, 0, 0), Vector(1, 0, 0), Vector(2, 0, 0),
 Vector(0, 1, 0), Vector(1, 1, 0), Vector(2, 1, 0),
 Vector(0, 2, 0), Vector(1, 2, 0), Vector(2, 2, 0),
 Vector(0, 0, 1), Vector(1, 0, 1), Vector(2, 0, 1),
 Vector(0, 1, 1), Vector(1, 1, 1), Vector(2, 1, 1),
 Vector(0, 2, 1), Vector(1, 2, 1), Vector(2, 2, 1),
 Vector(0, 0, 2), Vector(1, 0, 2), Vector(2, 0, 2),
 Vector(0, 1, 2), Vector(1, 1, 2), Vector(2, 1, 2),
 Vector(0, 2, 2), Vector(1, 2, 2), Vector(2, 2, 2)]
```

Vector ranges implement all common sequence operations except concatenation and repetition (due to the fact that range objects can only represent sequences that follow a strict pattern and repetition and concatenation usually cause the resulting sequence to violate that pattern).

Vector ranges are extremely efficient compared to an equivalent `list()` or `tuple()` as they take a small (fixed) amount of memory, storing only the arguments passed in its construction and calculating individual items and sub-ranges as requested.

Vector range objects implement the `collections.Sequence` ABC, and provide features such as containment tests, element index lookup, slicing and support for negative indices.

The default order (`'zxy'`) may seem an odd choice. This is primarily used as it's the order used by the Raspberry Juice server when returning results from the world.getBlocks call. In turn, Raspberry Juice probably uses this order as it results in returning a horizontal layer of vectors at a time (given the Y-axis is used for height in the Minecraft world).

Warning: Bear in mind that the ordering of a vector range may have a bearing on tests for its ordering and equality. Two ranges with different orders are unlikely to test equal even though they may have the same *start*, *stop*, and *step* attributes (and thus contain the same vectors, but in a different order).

Vector ranges can be accessed by integer index, by `Vector` index, or by a slice of vectors. For example:

```
>>> v = vector_range(Vector() + 1, Vector() + 3)
>>> list(v)
[Vector(x=1, y=1, z=1),
 Vector(x=1, y=1, z=2),
 Vector(x=2, y=1, z=1),
 Vector(x=2, y=1, z=2),
 Vector(x=1, y=2, z=1),
 Vector(x=1, y=2, z=2),
 Vector(x=2, y=2, z=1),
 Vector(x=2, y=2, z=2)]
>>> v[0]
Vector(x=1, y=1, z=1)
>>> v[Vector(0, 0, 0)]
Vector(x=1, y=1, z=1)
>>> v[Vector(1, 0, 0)]
Vector(x=2, y=1, z=1)
>>> v[-1]
Vector(x=2, y=2, z=2)
>>> v[Vector() - 1]
Vector(x=2, y=2, z=2)
>>> v[Vector(x=1):]
vector_range(Vector(x=2, y=1, z=1), Vector(x=3, y=3, z=3),
        Vector(x=1, y=1, z=1), order='zxy')
>>> list(v[Vector(x=1):])
[Vector(x=2, y=1, z=1),
 Vector(x=2, y=1, z=2),
 Vector(x=2, y=2, z=1),
 Vector(x=2, y=2, z=2)]
```

However, integer slices are not currently permitted.

count(*value*)
Return the count of instances of *value* within the range (note this can only be 0 or 1 in the case of a range, and thus is equivalent to testing membership with `in`).

index(*value*)
Return the zero-based index of *value* within the range, or raise `ValueError` if *value* does not exist in the range.

2.11.4 line

`picraft.vector.`**line**(*start*, *end*)
A three-dimensional implementation of Bresenham's line algorithm, derived largely from Bob Pendelton's implementation (public domain). Given the end points of the line as the *start* and *end* vectors, this generator function yields the coordinate of each block (inclusive of the *start* and *end* vectors) that should be filled in to render the line.

For example:

```
>>> list(line(O, V(10, 5, 0)))
[Vector(x=0, y=0, z=0),
 Vector(x=1, y=1, z=0),
 Vector(x=2, y=1, z=0),
 Vector(x=3, y=2, z=0),
 Vector(x=4, y=2, z=0),
 Vector(x=5, y=3, z=0),
 Vector(x=6, y=3, z=0),
 Vector(x=7, y=4, z=0),
 Vector(x=8, y=4, z=0),
 Vector(x=9, y=5, z=0),
 Vector(x=10, y=5, z=0)]
```

2.11.5 lines

`picraft.vector.`**`lines`**`(`*points*, *closed=True*`)`

Extension of the `line()` function which returns all vectors necessary to render the lines connecting the specified *points* (which is an iterable of `Vector` instances).

If the optional *closed* parameter is `True` (the default) the last point in the *points* sequence will be connected to the first point. Otherwise, the lines will be left disconnected (assuming the last point is not coincident with the first). For example:

```
>>> points = [O, 4*X, 4*Z]
>>> list(lines(points))
[Vector(x=0, y=0, z=0),
 Vector(x=1, y=0, z=0),
 Vector(x=2, y=0, z=0),
 Vector(x=3, y=0, z=0),
 Vector(x=4, y=0, z=0),
 Vector(x=3, y=0, z=1),
 Vector(x=2, y=0, z=2),
 Vector(x=1, y=0, z=3),
 Vector(x=0, y=0, z=4),
 Vector(x=0, y=0, z=3),
 Vector(x=0, y=0, z=2),
 Vector(x=0, y=0, z=1),
 Vector(x=0, y=0, z=0)]
```

2.12 API - Events

The events module defines the `Events` class, which provides methods for querying events in the Minecraft world, and `BlockHitEvent` which is the only event type currently supported.

Note: All items in this module are available from the `picraft` namespace without having to import `picraft.events` directly.

The following items are defined in the module:

2.12.1 Events

class `picraft.events.`**`Events`**`(`*connection*`)`

This class implements the `events` attribute.

There are two ways of responding to picraft's events: the first is to `poll()` for them manually, and process each event in the resulting list:

```
>>> for event in world.events.poll():
...     print(repr(event))
...
```

```
<BlockHitEvent pos=1,1,1 face="y+" player=1>,
<PlayerPosEvent old_pos=0.2,1.0,0.7 new_pos=0.3,1.0,0.7 player=1>
```

The second is to "tag" functions as event handlers with the decorators provided and then call the `main_loop()` function which will handle polling the server for you, and call all the relevant functions as needed:

```
@world.events.on_block_hit(pos=Vector(1,1,1))
def hit_block(event):
    print('You hit the block at %s' % event.pos)

world.events.main_loop()
```

By default, only block hit events will be tracked. This is because it is the only type of event that the Minecraft server provides information about itself, and thus the only type of event that can be processed relatively efficiently. If you wish to track player positions, assign a set of player ids to the `track_players` attribute. If you wish to include idle events (which fire when nothing else is produced in response to `poll()`) then set `include_idle` to `True`.

Finally, the `poll_gap` attribute specifies how long to pause during each iteration of `main_loop()` to permit event handlers some time to interact with the server. Setting this to 0 will provide the fastest response to events, but will result in event handlers having to fight with event polling for access to the server.

clear()

Forget all pending events that have not yet been retrieved with `poll()`.

This method is used to clear the list of events that have occurred since the last call to `poll()` without retrieving them. This is useful for ensuring that events subsequently retrieved definitely occurred *after* the call to `clear()`.

main_loop()

Starts the event polling loop when using the decorator style of event handling (see `on_block_hit()`).

This method will not return, so be sure that you have specified all your event handlers before calling it. The event loop can only be broken by an unhandled exception, or by closing the world's connection (in the latter case the resulting `ConnectionClosed` exception will be suppressed as it is assumed that you want to end the script cleanly).

on_block_hit(*thread=False*, *multi=True*, *pos=None*, *face=None*)

Decorator for registering a function as an event handler.

This decorator is used to mark a function as an event handler which will be called for any events indicating a block has been hit while `main_loop()` is executing. The function will be called with the corresponding `BlockHitEvent` as the only argument.

The *pos* attribute can be used to specify a vector or sequence of vectors (including a `vector_range`); in this case the event handler will only be called for block hits on matching vectors.

The *face* attribute can be used to specify a face or sequence of faces for which the handler will be called.

For example, to specify that one handler should be called for hits on the top of any blocks, and another should be called only for hits on any face of block at the origin one could use the following code:

```
from picraft import World, Vector

world = World()

@world.events.on_block_hit(pos=Vector(0, 0, 0))
def origin_hit(event):
    world.say('You hit the block at the origin')

@world.events.on_block_hit(face="y+")
def top_hit(event):
    world.say('You hit the top of a block at %d,%d,%d' % event.pos)

world.events.main_loop()
```

The *thread* parameter (which defaults to `False`) can be used to specify that the handler should be executed in its own background thread, in parallel with other handlers.

Finally, the *multi* parameter (which only applies when *thread* is `True`) specifies whether multi-threaded handlers should be allowed to execute in parallel. When `True` (the default), threaded handlers execute as many times as activated in parallel. When `False`, a single instance of a threaded handler is allowed to execute at any given time; simultaneous activations are ignored (but not queued, as with unthreaded handlers).

`on_idle`(*thread=False*, *multi=True*)
Decorator for registering a function as an idle handler.

This decorator is used to mark a function as an event handler which will be called when no other event handlers have been called in an iteration of *`main_loop()`*. The function will be called with the corresponding *`IdleEvent`* as the only argument.

Note that idle events will only be generated if *`include_idle`* is set to `True`.

`on_player_pos`(*thread=False*, *multi=True*, *old_pos=None*, *new_pos=None*)
Decorator for registering a function as a position change handler.

This decorator is used to mark a function as an event handler which will be called for any events indicating that a player's position has changed while *`main_loop()`* is executing. The function will be called with the corresponding *`PlayerPosEvent`* as the only argument.

The *old_pos* and *new_pos* attributes can be used to specify vectors or sequences of vectors (including a *`vector_range`*) that the player position events must match in order to activate the associated handler. For example, to fire a handler every time any player enters or walks over blocks within (-10, 0, -10) to (10, 0, 10):

```
from picraft import World, Vector, vector_range

world = World()
world.events.track_players = world.players

from_pos = Vector(-10, 0, -10)
to_pos = Vector(10, 0, 10)
@world.events.on_player_pos(new_pos=vector_range(from_pos, to_pos + 1))
def in_box(event):
    world.say('Player %d stepped in the box' % event.player.player_id)

world.events.main_loop()
```

Various effects can be achieved by combining *old_pos* and *new_pos* filters. For example, one could detect when a player crosses a boundary in a particular direction, or decide when a player enters or leaves a particular area.

Note that only players specified in *`track_players`* will generate player position events.

`poll`()
Return a list of all events that have occurred since the last call to *`poll()`*.

For example:

```
>>> w = World()
>>> w.events.track_players = w.players
>>> w.events.include_idle = True
>>> w.events.poll()
[<PlayerPosEvent old_pos=0.2,1.0,0.7 new_pos=0.3,1.0,0.7 player=1>,
 <BlockHitEvent pos=1,1,1 face="x+" player=1>,
 <BlockHitEvent pos=1,1,1 face="x+" player=1>]
```

```
>>> w.events.poll()
[<IdleEvent>]
```

process()

Poll the server for events and call any relevant event handlers registered with `on_block_hit()`.

This method is called repeatedly the event handler loop implemented by `main_loop()`; developers should only call this method when their (presumably non-threaded) event handler is engaged in a long operation and they wish to permit events to be processed in the meantime.

include_idle

If `True`, generate an idle event when no other events would be generated by `poll()`. This attribute defaults to `False`.

poll_gap

The length of time (in seconds) to pause during `main_loop()`.

This property specifies the length of time to wait at the end of each iteration of `main_loop()`. By default this is 0.1 seconds.

The purpose of the pause is to give event handlers executing in the background time to communicate with the Minecraft server. Setting this to 0.0 will result in faster response to events, but also starves threaded event handlers of time to communicate with the server, resulting in "choppy" performance.

track_players

The set of player ids for which movement should be tracked.

By default the `poll()` method will not produce player position events (`PlayerPosEvent`). Producing these events requires extra interactions with the Minecraft server (one for each player tracked) which slow down response to block hit events.

If you wish to track player positions, set this attribute to the set of player ids you wish to track and their positions will be stored. The next time `poll()` is called it will query the positions for all specified players and fire player position events if they have changed.

Given that the `players` attribute represents a dictionary mapping player ids to players, if you wish to track all players you can simply do:

```
>>> world.events.track_players = world.players
```

2.12.2 BlockHitEvent

class `picraft.events.`**`BlockHitEvent`**(*pos*, *face*, *player*)

Event representing a block being hit by a player.

This tuple derivative represents the event resulting from a player striking a block with their sword in the Minecraft world. Users will not normally need to construct instances of this class, rather they are constructed and returned by calls to `poll()`.

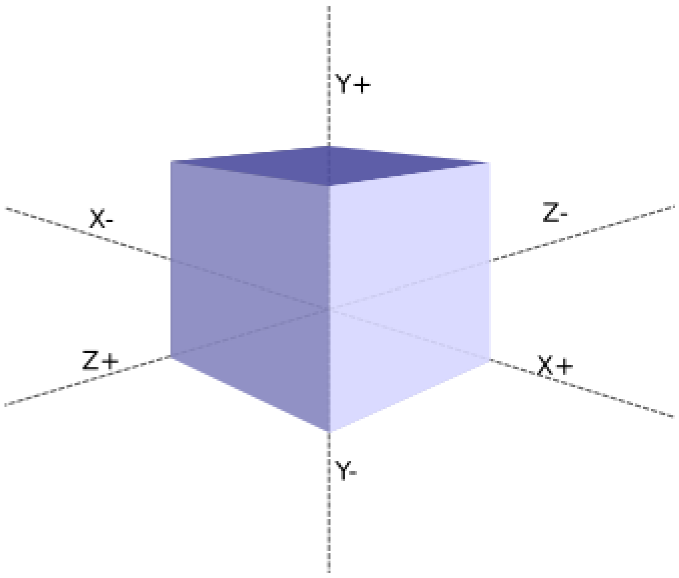
Note: Please note that the block hit event only registers when the player *right clicks* with the sword. For some reason, left clicks do not count.

pos

A `Vector` indicating the position of the block which was struck.

face

A string indicating which side of the block was struck. This can be one of six values: 'x+', 'x-', 'y+', 'y-', 'z+', or 'z-'. The value indicates the axis, and direction along that axis, that the side faces:



player
A *Player* instance representing the player that hit the block.

2.12.3 PlayerPosEvent

class `picraft.events.`**`PlayerPosEvent`**(*old_pos*, *new_pos*, *player*)
Event representing a player moving.

This tuple derivative represents the event resulting from a player moving within the Minecraft world. Users will not normally need to construct instances of this class, rather they are constructed and returned by calls to *poll()*.

old_pos
A *Vector* indicating the location of the player prior to this event. The location includes decimal places (it is not the tile-position, but the actual position).

new_pos
A *Vector* indicating the location of the player as of this event. The location includes decimal places (it is not the tile-position, but the actual position).

player
A *Player* instance representing the player that moved.

2.12.4 IdleEvent

class `picraft.events.`**`IdleEvent`**
Event that fires in the event that no other events have occurred since the last poll. This is only used if *Events.include_idle* is `True`.

2.13 API - Connections and Batches

The connection module defines the *Connection* class, which represents the network connection to the Minecraft server. Its primary purpose for users of the library is to initiate batch sending via the *Connection.batch_start()* method.

Note: All items in this module are available from the *picraft* namespace without having to import *picraft.connection* directly.

The following items are defined in the module:

2.13.1 Connection

class `picraft.connection.`**`Connection`**(*host*, *port*, *timeout=0.3*, *ignore_errors=False*, *encoding=u'ascii'*)

Represents the connection to the Minecraft server.

The *host* parameter specifies the hostname or IP address of the Minecraft server, while *port* specifies the port to connect to (these typically take the values "127.0.0.1" and 4711 respectively).

The *timeout* parameter specifies the maximum time that the client will wait after sending a command before assuming that the command has succeeded (see the *The Minecraft network protocol* section for more information). If *ignore_errors* is `True`, act like the official reference implementation and ignore all errors for commands which do not return data.

Users will rarely need to construct a `Connection` object themselves. An instance of this class is constructed by `World` to handle communication with the game server (`connection`).

The most important aspect of this class is its ability to "batch" transmissions together. Typically, the `send()` method is used to transmit requests to the Minecraft server. When this is called normally (outside of a batch), it immediately transmits the requested data. However, if `batch_start()` has been called first, the data is *not* sent immediately, but merely appended to the batch. The `batch_send()` method can then be used to transmit all requests simultaneously (or alternatively, `batch_forget()` can be used to discard the list). See the docs of these methods for more information.

`close`()

Closes the connection.

This method can be used to close down the connection to the game server. After this method is called, any further requests will raise a `ConnectionClosed` exception.

`send`(*buf*)

Transmits the contents of *buf* to the connected server.

If no batch has been initiated (with `batch_start()`), this method immediately communicates the contents of *buf* to the connected Minecraft server. If *buf* is a unicode string, the method attempts to encode the content in a byte-encoding prior to transmission (the encoding used is the `encoding` attribute of the class which defaults to "ascii").

If a batch has been initiated, the contents of *buf* are appended to the latest batch that was started (batches can be nested; see `batch_start()` for more information).

`transact`(*buf*)

Transmits the contents of *buf*, and returns the reply string.

This method immediately communicates the contents of *buf* to the connected server, then reads a line of data in reply and returns it.

Note: This method ignores the batch mechanism entirely as transmission is required in order to obtain the response. As this method is typically used to implement "getters", this is not usually an issue but it is worth bearing in mind.

`batch_start`()

Starts a new batch transmission.

When called, this method starts a new batch transmission. All subsequent calls to `send()` will append data to the batch buffer instead of actually sending the data.

To terminate the batch transmission, call `batch_send()` or `batch_forget()`. If a batch has already been started, a `BatchStarted` exception is raised.

Note: This method can be used as a context manager (the-with-statement) which will cause a batch to be started, and implicitly terminated with `batch_send()` or `batch_forget()` depending on whether an exception is raised within the enclosed block.

`batch_send()`
Sends the batch transmission.

This method is called after `batch_start()` and `send()` have been used to build up a list of batch commands. All the commands will be combined and sent to the server as a single transmission.

If no batch is currently in progress, a `BatchNotStarted` exception will be raised.

`batch_forget()`
Terminates a batch transmission without sending anything.

This method is called after `batch_start()` and `send()` have been used to build up a list of batch commands. All commands in the batch will be cleared without sending anything to the server.

If no batch is currently in progress, a `BatchNotStarted` exception will be raised.

`ignore_errors`
If `False` (the default), use the `timeout` to determine when responses have been successful. If `True` assume any response without an expected reply is successful (this is the behaviour of the reference implementation; it is faster but less "safe").

`timeout`
The length of time in seconds to wait for a response (positive or negative) from the server when `ignore_errors` is `False`.

`encoding`
The encoding that will be used for messages transmitted to, and received from the server. Defaults to `'ascii'`.

`server_version`
Returns an object (currently just a string) representing the version of the Minecraft server we're talking to. Presently this is just `'minecraft-pi'` or `'raspberry-juice'`.

2.14 API - Players

The player module defines the `Players` class, which is available via the `players` attribute, the `Player` class, which represents an arbitrary player in the world, and the `HostPlayer` class which represents the player on the host machine (accessible via the `player` attribute).

Note: All items in this module are available from the `picraft` namespace without having to import `picraft.player` directly.

The following items are defined in the module:

2.14.1 Player

class `picraft.player.`**`Player`**(*connection*, *player_id*)
Represents a player within the game world.

Players are uniquely identified by their *player_id*. Instances of this class are available from the `players` mapping. It provides properties to query and manipulate the position and settings of the player.

`direction`
The direction the player is facing as a unit vector.

This property can be queried to retrieve a unit `Vector` pointing in the direction of the player's view.

Warning: Player direction is only supported on Raspberry Juice.

heading

The direction the player is facing in clockwise degrees from South.

This property can be queried to determine the direction that the player is facing. The value is returned as a floating-point number of degrees from North (i.e. 180 is North, 270 is East, 0 is South, and 90 is West).

Warning: Player heading is only supported on Raspberry Juice.

pitch

The elevation of the player's view in degrees from the horizontal.

This property can be queried to determine whether the player is looking up (values from 0 to -90) or down (values from 0 down to 90). The value is returned as floating-point number of degrees from the horizontal.

Warning: Player pitch is only supported on Raspberry Juice.

player_id

Returns the integer ID of the player on the server.

pos

The precise position of the player within the world.

This property returns the position of the selected player within the Minecraft world, as a `Vector` instance. This is the *precise* position of the player including decimal places (representing portions of a tile). You can assign to this property to reposition the player.

tile_pos

The position of the player within the world to the nearest block.

This property returns the position of the selected player in the Minecraft world to the nearest block, as a `Vector` instance. You can assign to this property to reposition the player.

2.14.2 HostPlayer

class `picraft.player.`**`HostPlayer`**(*connection*)

Represents the host player within the game world.

An instance of this class is accessible as the `Game.player` attribute. It provides properties to query and manipulate the position and settings of the host player.

autojump

Write-only property which sets whether the host player autojumps.

When this property is set to True (which is the default), the host player will automatically jump onto blocks when it runs into them (unless the blocks are too high to jump onto).

Warning: Player settings are only supported on Minecraft Pi edition.

Note: Unfortunately, the underlying protocol provides no means of reading a world setting, so this property is write-only (attempting to query it will result in an `AttributeError` being raised).

direction

The direction the player is facing as a unit vector.

This property can be queried to retrieve a unit `Vector` pointing in the direction of the player's view.

Warning: Player direction is only supported on Raspberry Juice.

`heading`

The direction the player is facing in clockwise degrees from South.

This property can be queried to determine the direction that the player is facing. The value is returned as a floating-point number of degrees from North (i.e. 180 is North, 270 is East, 0 is South, and 90 is West).

Warning: Player heading is only supported on Raspberry Juice.

`pitch`

The elevation of the player's view in degrees from the horizontal.

This property can be queried to determine whether the player is looking up (values from 0 to -90) or down (values from 0 down to 90). The value is returned as floating-point number of degrees from the horizontal.

Warning: Player pitch is only supported on Raspberry Juice.

`pos`

The precise position of the player within the world.

This property returns the position of the selected player within the Minecraft world, as a `Vector` instance. This is the *precise* position of the player including decimal places (representing portions of a tile). You can assign to this property to reposition the player.

`tile_pos`

The position of the player within the world to the nearest block.

This property returns the position of the selected player in the Minecraft world to the nearest block, as a `Vector` instance. You can assign to this property to reposition the player.

2.15 API - Exceptions

The exc module defines the various exception classes specific to picraft.

Note: All items in this module are available from the *`picraft`* namespace without having to import *`picraft.exc`* directly.

The following items are defined in the module:

2.15.1 Exceptions

exception `picraft.exc.`**`Error`**

Base class for all PiCraft exceptions

exception `picraft.exc.`**`NotSupported`**

Exception raised for unimplemented methods / properties

exception `picraft.exc.`**`ConnectionError`**

Base class for PiCraft errors relating to network communications

exception `picraft.exc.`**`ConnectionClosed`**

Exception raised when an operation is attempted against a closed connection

exception `picraft.exc.`**`CommandError`**

Exception raised when a network command fails

exception picraft.exc.**NoResponse**
Exception raised when a network command expects a response but gets none

exception picraft.exc.**BatchStarted**
Exception raised when a batch is started before a prior one is complete

exception picraft.exc.**BatchNotStarted**
Exception raised when a batch is terminated when none has been started

exception picraft.exc.**EmptySliceWarning**
Warning raised when a zero-length vector slice is passed to blocks

2.16 The Minecraft network protocol

This chapter contains details of the network protocol used by the library to communicate with the Minecraft game. Although this is primarily intended to inform future developers of this (or other) libraries, it may prove interesting reading for users to understand some of the decisions in the design of the library.

2.16.1 Specification

Requirements

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this section are to be interpreted as defined in RFC 2119.

Overall Operation

The Minecraft protocol is a text-based "interactive" line oriented protocol. All communication is initiated by the client and consists of single lines of text which MAY generate a single line of text in response. Lines MUST terminate with ASCII character 10 (line feed, usually shortened to LF or \n).

Protocol implementations MUST use the ASCII encoding (non-ASCII characters are not ignored, or an error, but their effect is undefined).

A Minecraft network session begins by connecting a TCP stream socket to the server, which defaults to listening on port 4711. Protocol implementations SHOULD disable Nagle's algorithm (TCP_NODELAY) on the socket as the protocol is effectively interactive and relies on many small packets. No "hello" message is transmitted by the client, and no "banner" message is sent by the server. A Minecraft session ends simply by disconnecting the socket.

Commands and responses MUST consist of a single line. The typical form of a command, described in the augmented Backus-Naur Form (ABNF) defined by RFC 5234 is as follows:

```
command = command-name "(" [ option *( "," option ) ] ")" LF

command-name = 1*ALPHA "." 1*ALPHA [ "." 1*ALPHA ]
option = int-val / float-val / str-val

bool-val = "0" / "1"
int-val = 1*DIGIT
float-val = 1*DIGIT [ "." 1*DIGIT ]
str-val = *CHAR
```

Note: Note that the ABNF specified by RFC 5234 does not provide for implicit specification of linear white space. In other words, unless whitespace is explicitly specified in ABNF constructions, it is not permitted by the specification.

The typical form of a response (if one is given) is as follows:

```
response = ( success-response / fail-response ) LF

success-response = int-vector / float-vector
fail-response = "Fail"

int-vector = int-val "," int-val "," int-val
float-vector = float-val "," float-val "," float-val
```

The general character classes utilised in the ABNF definitions above are as follows:

```
ALPHA = %x41-5A / %x61-7A ; A-Z / a-z
DIGIT = %x30-39           ; 0-9
CHAR = %x01-09 / %x0B-FF  ; any character except LF
SP = %x20                 ; space
LF = %x0A                 ; line-feed
```

Client Notes

Successful commands either make no response, or provide a single line of data as a response. Unsuccessful commands either make no response, or provide a single line response containing the string “Fail” (without the quotation marks). The lack of positive (and sometimes negative) acknowledgements provides a conundrum for client implementations: how long to wait before deciding that a command has succeeded? If “Fail” is returned, the client can immediately conclude the preceding command failed. However, if nothing is returned, the client must decide whether the command succeeded, or whether the network or server is simply being slow in responding.

The longer the client waits, the more likely it is to correctly report failed operations (in the case of slow systems). However, the longer the wait, the slower the response time (and performance) of the client.

The official reference implementation simply ignores errors in commands that produce no response (providing the best performance, but the least safety). The picraft implementation provides a configurable timeout (including the ability to ignore errors like the reference implementation).

Clients MAY either ignore errors (as the official API does) or implement some form or timeout to determine when operations are successful (as in this API by default).

Specific Commands

The following sections define the specific commands supported by the protocol.

camera.mode.setFixed

Syntax:

```
camera-fixed-command = "camera.mode.setFixed()" LF
```

The `camera.mode.setFixed` command fixes the camera’s position at the current location. The camera’s location can subsequently be updated with the `camera.setPos` command but will not move otherwise. The camera’s orientation is fixed facing down (parallel to a vector along Y=-1).

camera.mode.setFollow

Syntax:

```
camera-follow-command = "camera.mode.setFollow(" [int] ")" LF
```

The `camera.mode.setFollow` command fixes the camera’s position vertically above the player with the specified ID (if the optional integer is specified) or above the host player (if no integer is given). The camera’s position will follow the specified player's position, but the orientation will be fixed facing down (parallel to a vector along Y=-1).

camera.mode.setNormal

Syntax:

```
camera-normal-command = "camera.mode.setNormal(" [int] ")" LF
```

The `camera.mode.setNormal` command aligns the camera's position with the "head" of the player with the specified ID (if the optional integer is specified) or the host player (if no integer is given). The camera's position and orientation will subsequently track the player's head.

camera.setPos

Syntax:

```
camera-set-pos-command = "camera.mode.setPos(" float-vector ")" LF
```

When the camera position has been fixed with `camera.mode.setFixed()`, this command can be used to alter the position of the camera. The orientation of the camera will, however, remain fixed (parallel to a vector along Y=-1).

chat.post

Syntax:

```
world-chat-command = "chat.post(" str-val ")" LF
```

The `chat.post` command causes the server to echo the message provided as the only parameter to the in-game chat console. The message MUST NOT contain the LF character, but other control characters are (currently) permitted.

entity.getPos

Syntax:

```
entity-get-pos-command = "entity.getPos(" int-val ")" LF
entity-get-pos-response = player-get-pos-response
```

The `entity.getPos` command performs the same action as the *player.getPos* command for the player with the ID given by the sole parameter; refer to *player.getPos* for full details.

entity.getTile

Syntax:

```
entity-get-tile-command = "entity.getTile(" int-val ")" LF
entity-get-tile-command = player-get-tile-response
```

The `entity.getTile` command performs the same action as the *player.getTile* command for the player with the ID given by the sole parameter; refer to *player.getTile* for full details.

entity.setPos

Syntax:

```
entity-set-pos-command = "entity.setPos(" int-val "," float-vector ")" LF
```

The `entity.setPos` command performs the same action as the *player.setPos* command for the player with the ID given by the first parameter. The second parameter is equivalent to the first parameter for *player.setPos*; refer to that command for full details.

entity.setTile

Syntax:

```
entity-set-tile-command = "entity.setTile(" int-val "," int-vector ")" LF
```

The `entity.setTile` command performs the same action as the *player.setTile* command for the player with the ID given by the first parameter. The second parameter is equivalent to the first parameter for *player.setTile*; refer to that command for full details.

player.getPos

Syntax:

```
player-get-pos-command = "player.getPos()" LF
player-get-pos-response = float-vector LF
```

The `player.getPos` command returns the current location of the host player in the game world as an X, Y, Z vector of floating point values. The coordinates 0, 0, 0 represent the spawn point within the world.

player.getTile

Syntax:

```
player-get-tile-command = "player.getTile()" LF
player-get-tile-response = int-vector LF
```

The `player.getTile` command returns the current location of the host player in the game world, to the nearest block coordinates, as an X, Y, Z vector of integer values.

player.setPos

Syntax:

```
player-set-pos-command = "player.setPos(" float-vector ")" LF
```

The `player.setPos` command teleports the host player to the specified location in the game world. The floating point values given are the X, Y, and Z coordinates of the player's new position respectively.

player.setTile

Syntax:

```
player-set-tile-command = "player.setTile(" int-vector ")" LF
```

The `player.setTile` command teleports the host player to the specified location in the game world. The integer values given are the X, Y, and Z coordinates of the player's new position respectively.

player.setting

Syntax:

```
player-setting-command = "player.setting(" str-val "," bool-val ")" LF
```

The `player.setting` command alters a property of the host player. The property to alter is given as the *str-val* (note: this is unquoted) and the new value is given as the *bool-val* (where 0 means “off” and 1 means “on”). Valid properties are:

- `autojump` - when enabled, causes the player to automatically jump onto blocks that they run into.

world.checkpoint.restore

Syntax:

```
world-restore-command = "world.checkpoint.restore()" LF
```

The `world.checkpoint.restore` command restores the state of the world (i.e. the id and data of all blocks in the world) from a prior saved state (created by the `world.checkpoint.save` command). If no prior state exists, nothing is restored but no error is reported. Restoring a state does not wipe it; thus a saved state can be restored multiple times.

world.checkpoint.save

Syntax:

```
world-save-command = "world.checkpoint.save()" LF
```

The `world.checkpoint.save` command can be used to save the current state of the world (i.e. the id and data of all blocks in the world, but not the position or orientation of player entities). Only one state is stored at any given time; any save overwrites any existing state.

The state of the world can be restored with a subsequent `world.checkpoint.restore` command.

world.getBlock

Syntax:

```
world-get-block-command = "world.getBlock(" int-vector ")" LF
world-get-block-response = int-val LF
```

The `world.getBlock` command can be used to retrieve the current type of a block within the world. The result consists of an integer representing the block type.

See Data Values (Pocket Edition) for a list of block types.

world.getBlockWithData

Syntax:

```
world-get-blockdata-command = "world.getBlockWithData(" int-vector ")" LF
world-get-blockdata-response = int-val "," int-val LF
```

The `world.getBlockWithData` command can be used to retrieve the current type and associated data of a block within the world. The result consists of two comma-separated integers which represent the block type and the associated data respectively.

See Data Values (Pocket Edition) for further information.

world.getHeight

Syntax:

```
world-get-height-command = "world.getHeight(" int-val "," int-val ")" LF
world-get-height-response = int-val LF
```

In response to the `world.getHeight` command the server calculates the Y coordinate of the first non-air block for the given X and Z coordinates (first and second parameter respectively) from the top of the world, and returns this as the result.

world.getPlayerIds

Syntax:

```
world-enum-players-command = "world.getPlayerIds()" LF
world-enum-players-response = [ int-val *( "|" int-val ) LF ]
```

The `world.getPlayerIds` command causes the server to a return a pipe (`|`) separated list of the integer player IDs of all players currently connected to the server. These player IDs can subsequently be used in the commands qualified with `entity`.

world.setBlock

Syntax:

```
world-set-block-command = "world.setBlock(" int-vector "," int-val [ "," int-val ] ")" LF
```

The `world.setBlock` command can be used to alter the type and associated data of a block within the world. The first three integer values provide the X, Y, and Z coordinates of the block to alter. The fourth integer value provides the new type of the block. The optional fifth integer value provides the associated data of the block.

See Data Values (Pocket Edition) for further information.

world.setBlocks

Syntax:

```
world-set-blocks-command = "world.setBlock(" int-vector "," int-vector "," int-val [ "," int-val
```

The `world.setBlocks` command can be used to alter the type and associated data of a range of blocks within the world. The first three integer values provide the X, Y, and Z coordinates of the start of the range to alter. The next three integer values provide the X, Y, and Z coordinates of the end of the range to alter.

The seventh integer value provides the new type of the block. The optional eighth integer value provides the associated data of the block.

See Data Values (Pocket Edition) for further information.

world.setting

Syntax:

```
world-setting-command = "world.setting(" str-val "," bool-val ")" LF
```

The `world.setting` command is used to alter global aspects of the world. The setting to be altered is named by the first parameter (the setting name MUST NOT be surrounded by quotation marks), while the boolean value (the only type currently supported) is specified as the second parameter. The settings supported by the Minecraft Pi engine are:

- `world_immutable` - This controls whether or the player can alter the world (by placing or destroying blocks)
- `nametags_visible` - This controls whether the nametags of other players are visible

2.16.2 Critique

The Minecraft protocol is a text-based "interactive" line oriented protocol. By this, I mean that a single connection is opened from the client to the server and all commands and responses are transmitted over this connection. The completion of a command does *not* close the connection.

Despite text protocols being relatively inefficient compared to binary (non-human readable) protocols, a text-based protocol is an excellent choice in this case: the protocol isn't performance critical and besides, this makes it extremely easy to experiment with and debug using nothing more than a standard telnet client.

Unfortunately, this is where the good news ends. The following is a telnet session in which I experimented with various possibilities to see how "liberal" the server was in interpreting commands:

```
chat.post(foo)
Chat.post(foo)
chat.Post(foo)
chat.post (foo)
chat.post(foo))
chat.post(foo,bar)
chat.post(foo) bar baz
chat.post foo
Fail
```

- The first attempt (`chat.post(foo)`) succeeds and prints "foo" in the chat console within the game.
- The second, third and fourth attempts (`Chat.post(foo)`, `chat.Post(foo)`, and `chat.post (foo)`) all fail silently.
- The fifth attempt (`chat.post(foo))`) succeeds and prints "foo)" in the chat console within the game (this immediately raised my suspicions that the server is simply using regex matching instead of a proper parser).
- The sixth attempt (`chat.post(foo,bar)`) succeeds, and prints "foo,bar" in the chat console.
- The seventh attempt (`chat.post(foo) bar baz`) succeeds, and prints "foo" in the console.
- The eighth and final attempt (`chat.post foo`) also fails and actually elicits a "Fail" response from the server.

What can we conclude from the above? If one were being generous, we might conclude that the ignoring of trailing junk (`bar baz` in the final example) is an effort at conforming with Postel's Law. However, the fact that command name matching is done case insensitively, and that spaces leading the parenthesized arguments cause failure would indicate it's more likely an oversight in the (probably rather crude) command parser.

A more serious issue is that in certain cases positive acknowledgement, and even negative acknowledgement, are lacking from the protocol. This is a major oversight as it means a client has no reliable means of deciding when a command has succeeded or failed:

- If the client receives "Fail" in response to a command, it can immediately conclude the command has failed (and presumably raise some sort of exception in response).
- If nothing is received, the command *may* have succeeded.
- Alternatively, if nothing is received, the command *may* have failed (see the silent failures above).
- Finally, if nothing is received, the server or intervening network may simply be running slowly and the client should wait a bit longer for a response.

So, after sending a command a client needs to wait a certain period of time before deciding that a command has succeeded or failed. How long? This is impossible to decide given that it depends on the state of the remote system and intervening network.

The longer a client waits, the more likely it is to correctly notice failures in the event of slow systems/networks. However, the longer a client waits the longer it will be before another command can be sent (given that responses are not tied to commands by something like a sequence number), resulting in poorer performance.

The official reference implementation of the client (mcpi) doesn't wait at all and simply assumes that all commands which don't normally provide a response succeed. The picraft implementation provides a configurable timeout, or the option to ignore errors like the reference implementation (the default is to wait 0.2s in order to err on the side of safety).

What happens with unknown commands? Let's try another telnet session to find out:

```
foo
Fail
foo()
```

It appears that anything without parentheses is rejected as invalid, but anything with parentheses is accepted (even though it does nothing ... is that an error? I've no idea!).

What happens when we play with commands which accept numbers?

```
player.setPos(0.5,60,-60)
player.setPos(0.5,60.999999999999999999999999999999999999,-60)
player.setPos(0.5,0x3c,-60)
player.setPos(5e-1,60,-60)
player.setPos(0.5,inf,-60)
player.setPos(0.5,NaN,nan)
player.setPos(0.5,+60,-60)
player.setPos(0.5,--60,-60)
Fail
player.setPos(   0.5,60,-60)
player.setPos(0.5   ,60,-60)
Fail
player.setPos(0.5,60,-60
player.setPos(0.5,60,-60   foo
player.setPos(0.5  foo,60,-60)
Fail
```

In each case above, if nothing was returned, the command succeeded (albeit with interesting results in the case of NaN and inf values). So, we can conclude the following:

- The server doesn't seem to care if we use floating point literals, decimal integer literals, hex literals, exponent format, or silly amounts of decimals. This suggests to me it's just splitting the options on "," and throwing each resulting string at some generic str2num routine.
- Backing up the assumption that some generic str2num routine is being used, the server also accepts "NaN" and "inf" values as numbers (albeit with silly results).
- Leading spaces in options are fine, but trailing ones result in failure.
- Unless it's the last option in which case anything goes.
- Including the trailing parenthesis, apparently.

As we've seen above, the error reporting provided by the protocol is beyond minimal. The most we ever get is the message "Fail" which doesn't tell us whether it's a client side or server side error, a syntax error, an unknown command, or anything else. In several cases, we don't even get "Fail" despite nothing occurring on the server.

In conclusion, this is not a well thought out protocol, nor a terribly well implemented server.

A plea to the developers

I would dearly like to see this situation improved and be able to remove this section from the docs! To that end, I would be more than happy to discuss (backwards compatible) improvements in the protocol with the developers. It shouldn't be terribly hard to come up with something similarly structured (text-based, line-oriented), which doesn't break existing clients, but permits future clients to operate more reliably without sacrificing (much) performance.

2.17 Change log

2.17.1 Release 0.4 (2015-07-19)

Release 0.4 adds plenty of new features:

- The events system has been expanded considerably to include an event-driven programming paradigm (decorate functions to tell picraft when to call them, e.g. in response to player movement or block hits). This includes the ability to run event handlers in parallel with automatic threading
- Add support for circle drawing through an arbitrary plane. I'm still not happy with the implementation, and may revise it in future editions, but I am happy with the API so it's worth releasing for now (*#7_*)
- Add Raspbian packaging; we've probably got to the point where I need to start making guarantees about backward compatibililty in which case it's probably time to make this more generally accessible by including deb packaging (*#8_*)
- Lots of doc revisions including a new vectors chapter, more recipes, and so on!

2.17.2 Release 0.3 (2015-06-21)

Release 0.3 adds several new features:

- Add support for querying a range of blocks with one transaction on the Raspberry Juice server (#1)
- Add support for rotation of vectors about an arbitrary line (#6)
- Add bitwise operations and rounding of vectors
- Lots of documentation updates (fixes to links, new recipes, events documented properly, etc.)

2.17.3 Release 0.2 (2015-06-08)

Release 0.2 is largely a quick bug fix release to deal with a particularly stupid bug in 0.1 (but what are alphas for?). It also adds a couple of minor features:

- Fix a stupid error which caused `block.data` and `block.color` (which make up the block database) to be excluded from the PyPI build (#3)
- Fix being able to set empty block ranges (#2)
- Fix being able to set block ranges with non-unit steps (#4)
- Preliminary implementation of getBlocks support (#1)

2.17.4 Release 0.1 (2015-06-07)

Initial release. This is an alpha version of the library and the API is subject to change up until the 1.0 release at which point API stability will be enforced.

2.18 License

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index