# Horton Documentation

***Release 0.1***

**James King**

January 16, 2014

Contents

Horton is a library for making amusing things. It provides a Grid datastructure (*and various sub-classes thereof*) and some useful modules and functions for doing neat things with them. You can create cellular automata simulations or you can generate mazes. Grids are very flexible and Horton gives them a nice, simple, *Pythonic* API. What you do with them is left to your imagination.

Contents:

CHAPTER 1

# Introduction

Grids are very neat. Horton gives them a nice Python API:

```
>>> from horton import grid
>>> g = grid.Grid(3, 3)
>>> g[0, 0] = 1
>>> g[2, 2] = 2
>>> grid.Grid.pprint(g)
1 0 0
0 0 0
0 0 2
```

The grid dimensions are inclusive but notice that the indices start at 0. Trying to access a location in the grid that isn't there will result in a KeyError:

```
>>> g[100, 200]
Traceback (most recent call last):
  ...
KeyError: '(100, 100) is an invalid co-ordinate'
```

However there are grids for which that wouldn't be a problem:

```
>>> t = grid.Torus(3, 3)
>>> t[0, 0] = 1
>>> grid.Grid.pprint(t)
1 0 0
0 0 0
0 0 0
>>> t[3, 0]
1
```

This is because a Torus grid wraps around at the poles:

```
>>> t[9, 0]
1
```

Grids provide the Mapping interface from the *collections.abc* module. You can iterate over them in all the usual ways:

```
>>> for coordinate, value in g.items():
...     print("X: %d, Y: %d is %d" % (coordinate[0],
...                                  coordinate[1],
...                                  value))
...
X: 0, Y: 0 is 1
X: 1, Y: 0 is 0
```

```
X: 2, Y: 0 is 0
X: 0, Y: 1 is 0
X: 1, Y: 1 is 0
X: 2, Y: 1 is 0
X: 0, Y: 2 is 0
X: 1, Y: 2 is 0
X: 2, Y: 2 is 2
```

```
>>> print(" ".join(cell for cell in g))
1 0 0 0 0 0 0 0 2
```

Grids also have some extra attributes that are useful when working with them:

```
>>> small_grid = Grid(2, 2)
>>> small_grid.coordinates
[(0, 0), (1, 0), (0, 1), (1, 1)]
>>> small_grid.dimensions
(2, 2)
```

You can select a region of a Grid using slice notation:

```
>>> grid = Grid(10, 10)
>>> small_grid = grid[0:0, 4:4]
>>> small_grid[0, 0] = 1 # Note that they do not share structure
>>> Grid.pprint(small_grid)
1 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
>>> print(grid[0, 0])
0
```

Assuming you've installed the *optional* dependency, *pygame*, you can easily start rendering your Grid objects. See *Pygame Integration* for more information.

CHAPTER 2

# API

| | |
|---|---|
| horton.grid.Grid(width, height[, value]) | A Grid is a two-dimensional data-structure. |
| horton.grid.Torus(width, height[, value]) | A Grid whose edges are connected. |

**class** horton.grid.**Grid**(*width*, *height*, *value=0*)

A Grid is a two-dimensional data-structure.

It provides the Python Mapping interface whose keys are tuples representing co-ordinates in the Grid.

**__add__**(*other*)

Return a grid whose values are comprised by adding the values of two grids together.

**__contains__**(*value*)

Return True of *value* can be found in the grid.

**__eq__**(*other*)

Return True if equal to *other*.

Two grids are considered equal if every value in the grids are equal.

**__getitem__**(**args*)

Return something from the grid.

The first argument of args is a tuple. If the elements of the tuple are integers then fetch the value at the coordinate. If the elements are slices then return a Grid from the region defined by them.

**__iter__**()

Return an iterator over the values.

**__len__**()

Return the total size.

**__setitem__**(**args*)

Set an item in the grid to a value.

*The first argument is an (x, y) tuple and the second is the value.*

**__sub__**(*other*)

Return a grid whose values are comprised by subtracting the values from one by the other.

**coordinates**

Return the list of coordinates.

*This value is cached internally after the initial call.*

**classmethod copy**(*other*)

Return a new Grid as a copy of *other*.

**dimensions**
Return the dimensions tuple.

**classmethod from_array**(*width*, *height*, *arr*, *copy=True*)
Create a Grid from an array.

**get**(*x*, *y*, *default=None*)
Return a value at *x*, *y*.

*Return a default value if the key cannot be found.*

**items**()
Return a list of co-ordinate, value pairs.

**iter_items**()
Yield successive co-ordinate, value pairs.

**static pprint**(*grid*)
Pretty print a Grid object.

**values**
Return a copy of the grid values.

**class** horton.grid.**Torus**(*width*, *height*, *value=0*)
A Grid whose edges are connected.

**__getitem__**(**args*)
Return an item from the grid.

*The first argument is an (x, y) tuple.*

**__setitem__**(**args*)
Set an item in the grid to a value.

*The first argument is an (x, y) tuple and the second is a value.*

CHAPTER 3

# Pygame Integration

Pictures speak a thousand words. Horton comes with an optional module for rendering grids using Pygame. The goal is to make it super-easy to start getting something on the screen and scale up as your project gets a little more sophisticated.

The main function you should be aware of is `pygame.render.pg.render_grid()`.

**render_grid**(*surface*, *grid*, *x*, *y*, *width*, *height*[, *padding=0*, *render_cell=draw_cell*])

Render a `horton.grid.Grid` instance to the given *surface*.

**Parameters**

- **surface** – A pygame.Surface object
- **grid** – A horton.grid.Grid instance
- **x** – The surface x-coordinate to place the grid at
- **y** – The surface y-coordinate to place the grid at
- **width** – The desired width of the rendered grid
- **height** – The desired height of the rendered grid
- **padding** – The optional padding to apply to the cells of the grid
- **render_cell** – The function to call when rendering an individual cell.

All you need is this function and a `horton.grid.Grid` instance to draw a grid to the screen. The default `horton.render.pg.draw_cell()` will simply draw a filled-in black box if the cell evaluates to `True`. The minimal amount of code you need to get a grid on the screen is:

```
import pygame
import sys

from horton.grid import Grid
from horton.render.pg import render_grid
from pygame.locals import *

g = Grid.from_array([1, 0, 1,
                     1, 1, 1,
                     1, 0, 1])
pygame.init()

screen = pygame.display.set_mode((640, 480))
screen.fill((255, 255, 255))
render_grid(screen, g, 10, 10, 100, 300)
```

```
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            sys.exit()

    pygame.display.flip()
```

The *padding* parameter behaves much like padding in the CSS box-model. The position of the cell is relative to its position in the grid and the padding is applied to the content. In other words, it will *reduce* the size of the content of your cell.

If you want to customize your grid beyond the defaults provided you will have to supply your own `draw_cell()` function with the following signature:

**draw_cell**(*surface*, *cell*, *x*, *y*, *width*, *height*)

Render a cell from a `horton.grid.Grid` instance to the given *surface*.

**Parameters**

- **surface** – A pygame.Surface instance
- **cell** – The value of the cell to draw
- **x** – The screen x-coordinate of the cell to draw
- **y** – The screen y-coordinate of the cell to draw
- **width** – The calculated width of the cell
- **height** – The calculated height of the cell

You then pass your function to the *render_cell* parameter of the `horton.render.pg.render_grid()` function and let it do the rest:

```
import pygame
import random
import sys

from horton.grid import Grid
from horton.render.pg import render_grid
from pygame.locals import *

def random_colour_cell(surf, cell, x, y, w, h):
    if cell:
        colour = (random.randint(0, 255),
                  random.randint(0, 255),
                  random.randint(0, 255))
    else:
        colour = (255, 255, 255)

    pygame.draw.rect(surf, colour, pygame.Rect(x, y, w, h))

g = Grid.from_array(3, 3,
                    [1, 0, 1,
                    1, 1, 1,
                    1, 0, 1])
pygame.init()

screen = pygame.display.set_mode((640, 480))
screen.fill((255, 255, 255))

while True:
```

```
    for event in pygame.event.get():
        if event.type == QUIT:
            sys.exit()

    render_grid(screen, g, 10, 10, 100, 300,
                padding=2,
                render_cell=random_colour_cell)

    pygame.display.flip()
```

These two functions alone can get you pretty far. Just check out the `examples/` folder in your horton distribution to see what is possible.

CHAPTER 4

# Indices and tables

- *genindex*
- *modindex*
- *search*