Hive system manual
Sjoerd de Vries, March 2011

Introduction and design goals

This is the first manual of the hive system. It will definitely be replaced by another manual once it is more mature.

Although, 'more mature' might be the wrong term here. The hive system is not in alpha or anything; it is just that this is a huge project, and I am currently its only author. At the low level layer, the project is finished. I am now building higher-level stuff on top of it. This is much more fun, and there is already more than enough high-level stuff to demonstrate that the whole thing works. Still, there is a great whole lot of work still to be done, and I am looking for programmers to help me. Hence this manual.

The hive system is designed to be a. extremely flexible and b. to appeal to a wide range of users, from the Python programming experts who uses metaclasses and functools.partial on a daily basis, to artists and scientists who cannot program at all, but use editors and drag-and-drop GUIs to configure their game or protocol. This is impossible to achieve with a single program or API: generally, the more user-friendly a piece of software is, the more it limits your options; if you make it more powerful, it becomes harder to use. That's why the hive system has many layers.

Currently, the deep layers are finished (libcontext, bees); the middle layer (dragonfly) is quite usable, but the highest layer (the hivemap GUI and Spyder stuff) is still at the proof-of-principle level. This highest layer is aimed at non-programmers, so if you are one of those, I would say: stop reading right now. Just keep an eye on the project from time to time, until the more mature manual is here. You will love the hive system, I promise. It just isn't ready for you yet.

Good, now that all non-programmers have left the audience, we can start with a bit of code:

<manual-1.py>

Make sure that you have Panda 3D installed.
You will see a binary grid displayed with 50 % of its cells set to True (white). Press Escape to stop.

Looking at the code, a number of things become clear.
First of all, everything is done by creating a class, here called "myhive", which derives from some generic base class. The class is then instanced and run. This is not so different from other object-oriented Python libraries, even though the last part looks quite heavyweight with its six-step process. However, the "myhive" class looks positively weird. It doesn't contain an __init__ method, and only contains something that looks like a class attribute. This shows up even more clearly in myhive's base class, the pandahive:

Source code: dragonfly/pandahive/pandahive.py
<last 12 lines>

Serious abuse of the Python class statement is the single most defining feature of the hive system. Components called drones and workers (together: bees) are placed into hives and then connected. At the lower levels, a very elaborate system of metaclasses and wrappers makes sure that these bees become instance members instead of class members (so that communication between bees in one

instance of the hive doesn't affect other instances), and also that inheriting from hive classes works properly. Fortunately, it just works. You don't need to worry about the details, unless you really want to dive into the mechanics of the bee module. Which I do not recommend, lest your brain is eaten.

In short, the hive system is designed with maximum flexibility in mind. At the plus side, it is really easy to change something, as you can easily subclass a hive in two lines (and that's just the beginning!). At the down side, there are is quite a bit of ttwtwtd, "things that (just) work the way they do", by magic. The hive system does not hide all of its inner workings behind a clean API. Clean APIs produce elegant code that works perfectly, right up to the point that you want to do something special that is not the API. In the hive system, you always have the option to delve deeper and do something special. For the common case, you will have boilerplate code to deal with. Or, if you want, you can create your own layer on top to hide the boilerplate away (a method to do getinstance-build-place-close-init-run all at once, why not?). The hive system gives you the choice.

Summary
The hive system contains of many layers.
The high-level layers are aimed at non-programmers, but they are not ready yet.
The deeper layers are for those with Python programming skills. Flexibility is their most important goal.
You can easily design your own layers on top, with more elegance and user-friendliness.

A quick example

In the following example, we will hack into a lower level to change the values of the grid at random every frame, like a television that has no signal.

<manual-2.py>

We could also do a different hack by seizing control from pandahive's run() and doing everything ourselves:

<manual-3.py>

Good. Now that you know that you can do things quick and dirty, I will show you how to do it in a way that is more like playing nice and doing it the hive way.
As said above, the hive system works by the placement of bees into hive classes. There are two kinds of bees: drones, which have implicit connections, and workers, which are connected explicitly to each other. We have used one drone already, the pandacanvas that draws the grid. Now we will use some workers. First, let's just do the initial drawing of the grid:

<manual-4.py>

Let's go over the code.
The dragonfly library defines a startsensor worker. It triggers on the hive's first tick, which in case of the pandahive means on the first rendering frame. There is also a "draw" worker to draw the grid. However, we have to tell it what we are drawing: a dragonfly.grid.bgrid, which by convention is called ("object", "bgrid").
As you can see, there is a quite a bit of static typing in the hive system. This may be annoying to you if you are used to the nice duck-typing in Python. However, this is the only way to automatically build

higher layers, i.e. a GUI where people can fill in the variables and have them parsed appropriately. Hell, if you would make a GUI that knows about an ("object", "bgrid") editor, people could specify the grid's start pixels by point-and-click! More importantly, at the middle layer, if you connect workers that have multiple inputs or outputs, the hive system will automatically connect the right ones. That's why we can write *connect(grid, do_draw)* and *connect(start, do_draw)* instead of *connect(grid.outp, do_draw.inp)* and *connect(start.outp, do_draw.trig)*, which would work also. So, there is some boilerplate because of static typing, but some of it is removed, too.
In addition, there is no static type **checking** in the hive system: if you say that it is a duck, it will be treated as a duck.
Finally, we set up a variable worker called "grid" that holds our grid,box,parameters tuple, and we feed its value at startup to the "draw" worker. This feeding, i.e. pulling a value and pushing it forward upon receiving a trigger, is done by a transistor. Again, you must specify its type.
Pushing and pulling values is central to the hive system. Pushing is analogous to calling a function with some arguments. Pulling is what happens if you ask a function for its return value. Anyway, the example should make it clear.

So, let's implement the updating every frame. For this, we will use the ticksensor, which fires at every tick of the hive, i.e. once per frame. The call0 worker takes as a parameter a function, which it will call with zero arguments when triggered. Just connect the workers. As the function parameter, we will use the same randomize_grid() function that was used to initialize the grid.
The drawupdate will then update the grid, but it needs an identifier. Fortunately, the draw worker holds the identifier of the most recently drawn object. Like for the initial drawing, a transistor makes it all work.
There is one big catch. We have given our global grid variable as a parameter to the "grid" variable worker. We want the worker to hold a reference to this grid, so that we can mutate it in randomize_grid() and see the changes on screen. Holding a reference is the default behavior in Python. However, in the hive system, workers and hives will make their own independent copy of their parameters, so that hive instancing and subclassing works properly. Usually, that's good, but now we want to overrule it. This can be done by replacing "grid" with "bee.reference(grid)" in the parameters to the worker.
Here is the code:

<manual-5.py>

By the way, if you want to see something more exciting than white noise, make the call0 worker point to the following function:

<manual-6.py, lines 17-31>

In case you have never seen it, it is called Conway's Game of Life. Just sit back and watch the blinkers and gliders move over the screen.
Here is the final code.

<manual-6.py>

Summary
Bees (drones and workers) are placed into a hive, using the Python class statement.
Low-level Python hacks are also possible.
Drones are connected implicitly, workers are connected explicitly.

Through connections, workers can push a value to other workers, or pull a value from another worker.
Workers often use static types. This is necessary to build high-level GUIs automatically.
However, there is no static type checking: if you tell it is a duck, it is treated as a duck.

Tutorial 1: Tetris

In this tutorial we will build a minimal Tetris game. *(Tetris was originally developed by Alexey Pajitnov. The rights are currently held by Tetris Company. This tutorial and its code are purely for demonstrative purposes and no infringement is intended.)*
This tutorial is meant to be relatively "fast". Hive system features will be only discussed for their direct practical application. The underlying principles and mechanisms are introduced in more depth in the chess and moving panda tutorials.

We will be using the pandahive for this tutorial, which relies on Panda3d for visualization. We could as well have used a pygamehive or whatever, except that I didn't build a pygamehive yet. (If you want, you can build a pygamehive of your own, taking the pandahive as an example. Canvas and keyboard bindings are not that difficult.)

In setting up a Tetris game, there are a number of parameters: the size of the main grid, the shape of the tetromino blocks, the screen area where we display the main grid, the screen area for displaying the score.
Hives are designed for flexibility. Parameters should be class attributes that can be overridden by subclassing the hive. Here is a hive that contains these parameters:

<tetris-0.py>

To specify the tetromino blocks, it uses dragonfly's bgrid (binary grid, black/white grid) class. The specified values are (x,y) coordinates of True (filled) grid elements/pixels.
box2d describes a 2D bounding box in the format (upperleft_position_x, upperleft_position_y, size_x, size_y).
The other parameters are self-evident.

Here are some more parameters added:

<tetris-0a.py>

Both screen areas have now an ID. This is because dragonfly's canvas system works with IDs. First you do an initial canvas.draw, specifying the drawn object and the ID. Then, when the object has been altered, you update the draw area with canvas.update(ID).
There is now also a dummy parameter class to hold arbitrary parameter values. We use it to store the desired color of the grid lines.
Next, there is a whole section of "someparameter_ = attribute('someparameter') ". This boilerplate has a good reason. If you create a bee (or any object) in the Python class statement, it will always take the *current value* of the parameter. Overriding the parameter in a subclass doesn't change this. Bee.attribute solves this problem.

For example:

```
class main_wrong(pandahive):
  mainarea_id = "main"
  update = dragonfly.canvas.update3(mainarea_id)

# The update worker's ID parameter is "main"
```

```
class main_wrong2(main_wrong):
  mainarea_id = "main2"

# main_wrong2 now has an update worker, but the worker's ID
parameter is still "main", not "main2"

class main_right(pandahive):
  mainarea_id = "main"
  attr_mainarea_id = bee.attribute("mainarea_id")
  update = dragonfly.canvas.update3(attr_mainarea_id)

class main_right2(main_right):
  mainarea_id = "main2"

# main_right2 now has an update worker with "main2"
# main_right's update worker still has "main"
```

Finally, there are two variable workers that hold the main grid and the current tetromino block grid. Attached to each of these variables is a gridcontrol worker. By connecting to this worker, we can update the values of the grid. We could of course also connect to the variable directly, but this would not work if something refers to the variable's original object:
To see what I mean, look at the following Python snippet:

```
obj = [1,2,3]

class holder:
  def __init__(self, obj):
    self.obj = obj
  def update(self):
    print "UPDATE:", self.obj

h = holder(obj)
h.update()
>>> UPDATE: [1,2,3]
obj = [4,5,6]
h.update()
>>> UPDATE: [1,2,3]   #No effect!
obj[:] = [4,5,6]
h.update()
>>> UPDATE: [4,5,6]   #Now it works...
```

In short, the gridcontrol worker can do the equivalent of "obj[:] = [4,5,6]"

The next version of our Tetris hive actually does something:

<tetris-1.py>

It contains a subhive "tetris_select_block" that accepts the "blocks" tuple as parameter. The value of the parameter is stored in a gentuple2 worker, which accepts a tuple whose value is not yet known at the time of the class definition. When the hive is built, it accesses the actual value of the tuple using "get_parameter".
When triggered on the "select" antenna, the hive will select a block at random. The value of the selected block can be retrieved from the hive's "selected" output.

But of course, we can also retrieve it using Python. In the main hive, we connect to a "tetris_select_block" a trigger worker "test". We build the hive, retrieve the initial value of the selected block (an empty tuple), then we trigger "test", and retrieve the value again. It will now print out the elements of a random block.
To make sure that we did the parametrization correctly, we create a derived class "main2" that only uses the first three Tetris blocks. As you can see, when we repeat the procedure, it will select a different block than the first hive, and the selected block will always be one of the first three Tetris blocks. Unit test passed.

The following Tetris hive will display the selected block on screen, in the lower left corner, until you press Escape:

<tetris-2.py>

Let's look at the main hive and discuss the new code starting from the top.

- First, there is now a canvas drone that takes care of the binding of our dragonfly.canvas areas to Panda3d.
Then, there comes a configure bee. A configure bee is a way to "bake" initialization commands into a hive.

```
class main(dragonfly.pandahive.pandahive):
  ...
  c0 = configure("canvas")
  c0.reserve(mainarea_id_,("object","bgrid"), box=mainarea_,
parameters = mainarea_parameters_)
  ...

m = main().getinstance()
m.build("main")
m.place()
```

is equivalent to:

```
class main(dragonfly.pandahive.pandahive):
  ...

m = main().getinstance()
m.build("main")
m.canvas.reserve(m.mainarea_id, ("object","bgrid"), box=m.mainarea,
parameters = m.mainarea_parameters)
m.place()
```

These commands are also baked into any hive that derives from main.
There is one caveat in this particular case. canvas.reserve reserves a canvas identifier, which translates into a libcontext plugin when the canvas is placed. Therefore, canvas.reserve must be called before canvas.place, which means that the configure bee must be placed before the canvas drone. Therefore, the configure bee must have name that comes before "canvas", because bees are placed in alphabetical order.

- The select_block subhive is now connected to the blockgridcontrol. There is now also an init_main worker, which can be triggered and then returns an empty grid with the specified dimensions. This worker is connected to the maingridcontrol.

- The drawing is drone in the tetris_draw subhive. It retrieves from the parent hive the main area's bounding box and identifier. It holds a drawgrid variable and an associated grid control. This drawgrid variable is the superposition of the main grid (containing all the previously placed blocks) and the current block grid (which can still move), and this is what is shown on screen.
Upon "start", the drone makes an initial drawing of an empty grid. Upon "draw", it pulls in the main grid, makes a copy of it, and pushes the copy into the drawgrid. Then, it pulls in the block grid and merges it with the drawgrid. Finally, it updates the main area, using the identifier, so that the current content of the drawgrid is shown in the mainarea.

- Back to the main hive, you can see a start sensor, which is activated on the first rendered frame. The start sensor selects a block, initializes the main grid, starts the tetris_draw subhive and makes the first drawing.

- Finally, the main hive now contains a raiser, which formats any exceptions that may arise.

In the next version, the tetris_select_block subhive is updated so that a random rotation is applied to the block.

<tetris-3.py>

The selected block is first copied into a "chosen" variable. The "chosen" block is then rotated 90 degrees, up to four times. The "select" trigger now first calls "do_select", that copies a selected block into chosen. "rotate" applies the random rotation, and "do_select2" pushes the grid as an output.

Now we will place the block at the top of the grid:

<tetris-4.py>

The tetris_control worker has two pull antennas, for the maingrid and the blockgrid. When triggered on "place_init", it activates the "m_place_init" function. This function triggers the grid1 and grid2 buffers to pull in the maingrid and blockgrid and to store them in the buffers. Then, it computes the place-at-the-top translation and applies it to the blockgrid. If the maingrid and blockgrid then overlap, the "lost" output trigger is activated, meaning that the game is lost.
In the main hive, the tetris_control worker is triggered at the start.

Now it is time to make the blocks fall:

<tetris-5.py>

The tetris_control worker has now been expanded with a "move_down" trigger antenna and a "dropped" trigger output. The "move_down" trigger activates the "m_move_down" method, which is quite simple: first, the grids are pulled in and a copy of the blockgrid is made. Then, the blockgrid copy is translated one unit down. If this results in an overlap, the maingrid is merged with the original blockgrid (before the translation), and the "dropped" trigger is activated. Else, the translation is applied to the original blockgrid.
In the main hive, there is now a tick sensor, which fires every tick (in the pandahive: every rendered frame). This tick sensor is connected to a cycle worker, which is connected so that it fires every 10 ticks.
The cycle activates the tetris_control's "move_down" trigger, and then updates the drawing. The tetris_control's "dropped" trigger is connected to the selection/rotation and initial placement of a new block.
Finally, the tetris_control's "lost" trigger is connected to an exitactuator, which stops the game. The exitactuator is between quotes because it was not defined in the current class, but in a base class (the pandahive, where it was already connected to the Escape key).

Now our game is starting to look like Tetris. Of course, we should be able to move the blocks left and right. To do this, we will use keyboard sensors, which will be connected to two new trigger antennas in the tetris_control: move_left and move_right. These triggers call a method move_sideways(), which applies the translation first on a copy, then checks if it's legal, and then applies it for real. Of course, after we move a block, we must update the drawing.

<tetris-6.py>

Now the code for rotation and dropping. For rotation, we make it bounce from the wall if necessary, and if the rotation can be made without causing overlap, we apply it, else we do nothing. We bind clockwise rotation to RETURN and counter-clockwise rotation to SPACE. For dropping, we simply move down the block until we clash or go through the bottom; then we move back up one unit, merge and activate the "dropped" trigger. The cycle has also been slowed down to once per 15 ticks.

<tetris-7.py>

Whenever a block has been dropped, full lines should be removed. The following version has a remove_lines method implemented in the tetris_control worker:

<tetris-8.py>

There is only thing left to do: the score. Let's design a subhive that draws a string in the scorearea, in the same way that the tetris_draw hive draws a grid in the mainarea. Again, we use a variable to hold the string, and a controller to modify it.
One little problem: strings are not mutable in Python, so controllers don't work on them. For this reason, the hive system has a mutable string called "mstr" (the implementation is ridiculously simple, see bee/mstr.py), and dragonfly has a controller mutable.mstrcontroller that takes a normal string and uses it to set the value of the mstr.
The logic of the score subhive is then as follows. The score count (int variable) is stored in the parent hive. The parent hive also reserves the scorearea ID, telling that an mstr is going to be drawn into it. The score hive knows this ID and the scorearea parameters from the parent hive.

The mstr that is drawn is held in scorestr. The score hive has a "start" trigger antenna, which causes an initial draw of the scorestr to be made. It also has a ("pull","int") antenna "score", that must be connected to the parent hive's score count. When the "draw" trigger is activated, the "set_score" transistor is activated, which pulls the score count in, converts it to str and pushes it into the controller. Also, whenever "set_score" fires, the "update" trigger is activated, causing the draw area to be updated with the new value of the string.

In the main hive, the score is simply initialized at zero and drawn once at startup. It will remain at zero forever. Here is the code:

<tetris-9.py>

Finally, we want to increase the score when a block has dropped, and when one or more lines are cleared.
To define the reward given for a block and for clearing 1-4 lines, five variables are defined in the main hive. They are connected to five new ("pull","int") antenna's in the tetris_control worker. In the worker, a triggerfunc "get_score_and_rewards" is defined, which pulls in the current value of the score and of all rewards. Whenever a drop has occurred, this method is called. The remove_lines() method is modified so that it returns the proper line reward. This is then added to the block reward and to the old score to compute the new score, which is then pushed as an output.
The main hive connects this new score output directly to the score variable. Also, whenever a new score is pushed out, this is detected by the update_score trigger, which connects to the "draw" trigger of the score hive, causing the score draw area to be updated.

Here is then the final code of a fully functional Tetris game:

<tetris-10.py>

Enjoy!

Extras
Here are some things that a real Tetris has, but our game does not. If you wish, you can add them as an exercise:
- Play a sound when a button is pressed or a line is cleared, or throughout the game (ti tadada tadadum tadadi ...). I didn't add hive bindings to Panda's sound system yet, but you can invoke it directly.
- Keep track of a high score
- Intro/game-over screens
- Create a level system. A level table defines at which score the next level is reached, and what the value of the cycle period then will be. You can also adjust the rewards based on the level.
- Create a third drawing area where the next upcoming block(s) are displayed. This requires some rewiring! If you want a challenge, try to do it by inheriting from the current Tetris hive and subhives, without copy-pasting or changing their files.
- I am planning to implement "toggle" keyboard sensors, which fire on both key presses (key down) and key releases (key up). Once they are there, you can use them to implement soft dropping.

Tutorial 2: Chess

This tutorial is about using the hive system as a glue language. We will use a hive to connect pre-existing Python components, creating a chess game where we can play against the computer or against another person.

This tutorial is a "slow" tutorial, meaning that concepts and mechanisms will be explained in depth. For a more practically-oriented tutorial, look at the Tetris tutorial. For a *very* practical example, take a look at the screencast.

2a. The components

All components are in /chess/components/ .

The Panda3d chessboard GUI
The chessboard is one of the demo examples of the Panda3d engine. I have copied the file and put it into the chess/components directory. Because we will use it as a Python module and not as a stand-alone script, the file name was changed from "Tut-Chessboard.py" into "TutChessboard.py", and the final two lines, "w = World; w.run()", were commented out.
Note that the chessboard is just a GUI: it has no notion whatsoever of the rules of chess.

I wrapped the chessboard GUI in "chessboard.py". It subclasses TutChessboard.World, and adds the following functionality:
- The class keeps track whose turn it is (self.turn is "Black" or "White")
- The "make_move" method takes as argument a long chess notation string (e.g. "Ng1-f3") and updates the pieces on the 3D board accordingly. It assumes that the move is valid.
- When the user makes a move on the 3D board, the swapPieces method is invoked (that's how the Panda3d example authors made it). The swapPieces method is re-defined in "chessboard": first, the move is converted to a long chess notation string using the "unparse_move" function. Then, the "move" method is called. If it raises a ValueError, the move was invalid: the pieces are restored to their previous position, and the "invalid_move" method is called. By default, calling "invalid_move" raises again a ValueError.
By default, the "move" method always raises a ValueError, i.e. every move is invalid by default. Therefore, the "move" method must be overridden by the application.

Note: unlike the Tetris tutorial and the other tutorials, we will not use the pandahive here. The hive system has no knowledge that the chessboard GUI uses Panda3d, it could as well have been Pygame or any other visual library.

The Glaurung chess engine
Glaurung (also known as Stockfish) is a very strong open source chess engine that uses the Universal Chess Interface (UCI) to communicate. Basically, after you start up the engine, you get a command line, where you can type in chess moves and other commands, and the engine prints a response. You can download the Glaurung engine for Windows or for OSX and it is available as an Ubuntu package, too.

I have made a simple Python UCIChessEngine wrapper class in the chess/components directory. To initialize it, you must provide the executable and directory of the Glaurung engine (or any UCI engine). On Ubuntu, "glaurung" is in the system path so "engine = UCIChessEngine('glaurung')" will do. On

Windows, it depends on the installation folder: it should be something like "engine = UCIChessEngine('glaurung.exe', 'C:\\Program Files\\Glaurung') ".
The easiest solution on Windows is to copy the Glaurung/Stockfish executable to the current directory (manual/chess) and name it "glaurung.exe".
After that, you can call "engine.make_move(move)" to make a move on the board. You must do this for all moves, both black and white. Again, the move should be a valid move in long chess notation (e.g. "Ng1-f3"). You can ask Glaurung to suggest a move by calling "engine.get_move()".

Note: the UCIChessEngine wrapper is very basic. It doesn't allow the chess engine to do searching or pondering in the background, so if you play carefully, you may be able to beat it. When invoked *properly*, the Glaurung engine is too strong for any human player.

The chesskeeper
The chesskeeper (chess/components/chesskeeper.py) is a Python class that I wrote as a chess game keeper. It keeps track of the position of the pieces, it knows the chess rules, and it will complain loudly (raise a ValueError) if a move is illegal. The most important method is "make_move", which does just that. There is also "new" for a new game and "copy" and "restore_copy" to save and load board positions.

"chesskeeper.py" also contains an utility function "parse_move", which converts a long notation chess string into a 3-element tuple *(start coordinate, end coordinate, special)*:
"e2-e4", "White" => ((4, 1), (4, 3), None)
"O-O", "White" => ((4, 0), (7, 0), "O-O")
"O-O", "Black" => ((4, 7), (7, 7), "O-O")
This function is used by the other components.

2b. Gluing a game together: general design issues

Note: This section is very theoretical. If you don't like that, you can gloss over it quickly or skip to the next section.

Suppose we want to implement a generic "classical" game (turn-based, two or more players, using dice, cards and/or a board) in a clean and flexible way. In general, such a game would consist of three parts. Players are "producers" of moves or actions. A central gamekeeper keeps track of the game and determines if the player moves are legal. Finally, the moves are "consumed" by the board, or some some other visual component that shows the current state of the game. This model is valid for something as simple as tic-tac-toe (naughts and crosses) or something as complicated as Settlers of Catan or Magic: The Gathering (or even Civilization, which is far too complicated for a physical board).

Let's consider first a game with two players, white and black.
Good object-oriented practice is to put everything in a class. For example:

```
class game:
  def __init__(self):
    ...
  def get_move_white(self):
     ...
  def get_move_black(self):
```

```
    ...
  def evaluate_move(self):
    ...
  def show_move(self):
    ...
```

Unfortunately, this code violates two design principles. First, separation of concerns: to make things more flexible, we should make separate classes for the gamekeeper, the board and the players. Second, don't repeat yourself: since the two players are equal, we should define the code for a player only once.

We could solve the first problem by using mixins:

```
class player_black:
  def get_move_black(self):
    ...

class player_white:
  def get_move_white(self):
    ...

class gamekeeper:
  def evaluate_move(self):
    ...

class gameboard:
  def show_move(self):
    ...

class game(player_black, player_white, gamekeeper, gameboard):
  def __init__(self):
    ...
```

However, we are still duplicating the code of the player class.
The solution to this is to embed the other classes, instead of mixing them in. The naive approach would be this:

```
class player:
  def get_move(self):
    ...

class gamekeeper:
  def evaluate_move(self):
    ...

class gameboard:
  def show_move(self):
    ...
```

```
class game:
  def __init__(self):
    self.black = player()
    self.white = player()
    self.gamekeeper = gamekeeper()
    self.gameboard = gameboard()
```

But what if the other player is on the internet, or a chess AI? We can easily give them their own classes:

```
class player_base:
  ...

class player(player_base):
  def get_move(self):
     ...

class player_AI(player_base):
  def get_move(self):
     ...

class player_internet(player_base):
  def get_move(self):
     ...
```

but to create an Internet game or a versus-AI game, we would have to re-define the __init__ method completely, without the possibility of code reuse. The solution is to use class attributes:

```
class game:
  player_black_class = player
  player_white_class = player
  gamekeeperclass = gamekeeper
  gameboardclass = gameboard
  def __init__(self):
    self.black = player_black_class()
    self.white = player_white_class()
    self.gamekeeper = gamekeeperclass()
    self.gameboard = gameboardclass()

class game_vs_internet(game):
  player_black_class = player_internet

class game_vs_AI(game):
  player_black_class = player_AI
```

This design is now a lot better, but it still has a couple of limitations:
- For a game with an arbitrary number of players, e.g. a poker game, the class attribute would be a list of players. In that case, the class attribute doesn't work so well:

```
class twomanpoker:
  players = [player, player]

class threemanpoker(twomanpoker):
  players.append(player)
#=> NameError

class threemanpoker(twomanpoker):
  players = twomanpoker.players
  players.append(player)
#=> modifies parent class!!!

class threemanpoker(twomanpoker):
  players = list(twomanpoker.players)
  players.append(player)
#=> correct, but tedious
```

- Objects may have more than one function. In practice, components may be both move producers and consumers. A chessboard GUI will not only show the game board, the player can also use it to make a move. An Internet connection to another player will produce moves, but the other player would like to know your moves as well. To work properly, components may need to access the parent, or other components. Although there are workarounds, all of this complicates the design considerably.

Design problems like these are exactly what the hive system attempts to solve. At the lowest level, it consists of plugins and sockets that are connected to each other. Components may place as many sockets and plugins as they wish, but a socket or plugin has one and only one function. At the medium level, it is based on hives, which look very much like the game class above.
The next section introduces libcontext, the low-level library that deals with sockets and plugins.

2c. The libcontext library

Libcontext is the low-level library upon which everything in the hive system is based. Contexts have a name, and they have sockets and plugins in them. Sockets are always functions, but plugins can be any object. When you close the context, every plugin is connected to every socket with the same name: the socket function is called with the plugin value as argument. See the example below:

```
def printvalue(value):
  print "Value:", value

import libcontext
c = libcontext.context("mycontext")
libcontext.push("mycontext")

s = libcontext.socketclasses.socket_single_required(printvalue)
p = libcontext.pluginclasses.plugin_single_required(42)
libcontext.socket("value", s)
libcontext.plugin("value", p)
```

```
libcontext.pop()
c.close()

# >>> Value: 42
```

As you can see, libcontext is a state machine. It knows which is the current context, because contexts have a name. This allows for a hierarchical organization of contexts:

```
def printvalue(value):
  print "Value:", value

import libcontext
c = libcontext.context("mycontext")
cc = libcontext.context("mycontext", "child")
cc.import_socket(c, "value")
libcontext.push("mycontext")

s = libcontext.socketclasses.socket_container(printvalue)
p = libcontext.pluginclasses.plugin_single_required(42)
libcontext.socket("value", s)
libcontext.plugin("value", p)

libcontext.push("child")
#current context name is now ("mycontext", "child)

pp = libcontext.pluginclasses.plugin_single_required(43)
libcontext.plugin("value", pp)

libcontext.pop()

libcontext.pop()
cc.close()
# >>> Value: 43
c.close()
# >>> Value: 42
```

Note that the plugin declared in the child context is connected to the socket of the parent context, because we imported it from the parent (even before it was actually defined!). Also note that we changed the socket class from "socket_single_required" to "socket_container". This is so that we can accommodate more than one plugin.

In practice, you will not directly use libcontext.context, push/pop or import_socket. However, libcontext.socket and libcontext.plugin are ubiquitous in bees, in both drones and workers. So to program your own bees, it is handy to know what types of plugins and sockets there are:

libcontext.pluginclasses:
  plugin_single_required: must be matched with exactly one socket (of the same name)
  plugin_single_optional: must be matched with zero or one sockets

plugin_multi_required: must be matched with one or more sockets
plugin_multi_optional: must be matched with zero or more sockets
plugin_supplier: alias for plugin_multi_optional

libcontext.socketclasses:
socket_single_required: must be matched with exactly one plugin
socket_single_optional: must be matched with zero or one plugins
socket_multi_required: must be matched with one or more plugins
socket_multi_optional: must be matched with zero or more plugins
socket_container: alias for socket_multi_optional

Now let's take our original libcontext example and make a class out of it:

```
def printvalue(value):
  print "Value:", value

import libcontext
from libcontext.pluginclasses import *
from libcontext.socketclasses import *

class valueclass:
  def __init__(self, name, value):
    self.name = name
    self.value = value
    self.context = libcontext.context(name)
  def place(self):
    libcontext.push(self.name)
    s = libcontext.socketclasses.socket_single_required(printvalue)
    p = libcontext.pluginclasses.plugin_single_required(self.value)
    libcontext.socket("value", s)
    libcontext.plugin("value", p)
    libcontext.pop()
  def close(self):
    self.context.close()

c = valueclass("mycontext", 42)
c.place()
c.close()
```

Nice. However, the subject of this tutorial is gluing components together. So, let's glue together multiple valueclasses into a single class:

```
class hive:
  def __init__(self, names, values):
    self.children = []
    for name, value in zip(names, values):
      child = valueclass(name,value)
      self.children.append(child)
```

```
  def place(self):
    for child in self.children: child.place()
  def close(self):
    for child in self.children: child.close()

h = hive(["mycontext1","mycontext2"],[42, 43])
h.place()
h.close()
```

However, we can create only one hive in this way. If we create another, libcontext will complain:

```
h = hive(["mycontext1","mycontext2"],[42, 43])
h.place()
h.close()

h2 = hive(["mycontext1","mycontext2"],[42, 43])
>>> Exception: Cannot register context mycontext1: context already
exists
```

The solution is to give the hive a context of its own:

```
class hive:
  def __init__(self, name, names, values):
    self.name = name
    self.context = libcontext.context(name)
    libcontext.push(self.name)
    self.children = []
    for name, value in zip(names, values):
      child = valueclass(name,value)
      self.children.append(child)
    libcontext.pop()
  def place(self):
    libcontext.push(self.name)
    for child in self.children: child.place()
    libcontext.pop()
  def close(self):
    self.context.close()

h = hive("h1", ["mycontext1","mycontext2"],[42, 43])
h.place()
h.close()

h2 = hive("h2", ["mycontext1","mycontext2"],[42, 43])
h2.place()
h2.close()
```

This will create two hive contexts "h1" and "h2", and four child contexts, ("h1","mycontext1"), ("h1","mycontext2"), ("h2","mycontext1") and ("h2","mycontext2"). Whenever we close a hive

context, its children are automatically closed, too: libcontext takes care of that.

2d. The bee module: drones and hives

We ended the previous section with a hive class, taking the names and parameters of its children as list arguments to __init__.

You *could* build a node system this way. Generalizing to other children than just valueclasses, you would store three lists: the names, the types of the children (in this example: valueclass) and the list of arguments-to-the-children (in this example: the values). You could create some kind of inheritance mechanism where you copy the lists and then add or remove children.

Or, you could just use the drones and hives provided by the bee module:

```
def printvalue(value):
  print "Value:", value

import libcontext
from libcontext.pluginclasses import *
from libcontext.socketclasses import *

import bee

class printvalueclass(bee.drone):
  def place(self):
    s = libcontext.socketclasses.socket_container(printvalue)
    libcontext.socket("value", s)

class valueclass(bee.drone):
  def __init__(self, value):
    self.value = value
  def place(self):
    p = libcontext.pluginclasses.plugin_single_required(self.value)
    libcontext.plugin("value", p)

class hive(bee.hive):
  p = printvalueclass()
  mycontext1 = valueclass(42)
  mycontext2 = valueclass(43)

h = hive().getinstance()
h.build("h1")
h.place()
h.close()

h2 = hive().getinstance()
h2.build("h2")
h2.place()
```

```
h2.close()

class hive2(hive): #adds a drone
  mycontext3 = valueclass(44)

class hive3(hive): #removes a drone
  mycontext2 = None
```

This code has the same semantics as our previous "hive". Although it looks otherwise, the drone children are NOT created when the class is defined. When building the hive, a wrapper is created for each drone, "buffering" the arguments. Only when getinstance() is called on the hive, the drones are actually created. In this way, each hive instance has children that are independent from the same children in other hive instances:

```
h3 = hive().getinstance()
h3.build("h3")
h3.place()
h3.close()

h4 = hive().getinstance()
h4.build("h4")
h4.place()
h4.close()

print h3.mycontext1.value, h4.mycontext1.value
>>> 42, 42
h4.mycontext1.value = 0
print h3.mycontext1.value, h4.mycontext1.value
>>> 42, 0
```

Also, note that we were able to decouple the "value" socket from the the "value" plugin. This is because drones do not have a context of their own: they use the context of their parent hive, and all sockets and plugins are placed in that context. Workers and hives do have contexts of their own. Workers import all plugins and sockets from their parent hive, while hives do not. So, if you really need independent contexts, it is best to use hive children, a.k.a. subhives:

```
class child1(bee.hive):
  p = printvalueclass()
  value = valueclass(42)

class child2(bee.hive):
  p = printvalueclass()
  value = valueclass(43)

class hive(bee.hive):
  mycontext1 = child1()
  mycontext2 = child2()
```

Or you can use a type of hive that is called a frame. Frames do automatically import the sockets and plugins from the parent hive. Like any other hive, you can put drones, workers and other hives inside a frame. This allows your hive to be more modular:

```
class child1(bee.frame):
  value = valueclass(42)

class child2(bee.frame):
  value = valueclass(43)

class hive(bee.hive):
  p = printvalueclass()
  mycontext1 = child1()
  mycontext2 = child2()
```

Finally, parameterization of a subhive is also possible:

```
from bee import parameter, get_parameter

class child(bee.frame):
  v = parameter("object")
  value = valueclass(get_parameter("v"))

class hive(bee.hive):
  p = printvalueclass()
  mycontext1 = child(42)   #works
  mycontext2 = child(v=43) #also works
```

These simple examples demonstrate the flexibility of the bee module in designing a program and its components.
Now let's move on to do some real work.

2e. A game of chess, using drones

We will now build our chess game using drones. All of the drones are in /chess/components/drones/.

The base hive that we will use is the command hive. The command hive creates a command line prompt where you can type text, just like the Python interpreter:

<tut-drone-0.py>

If you run the script, you can type commands and nothing happens, until you type "exit" or "quit".

Step 1: command line chess
By adding the appropriate drones, you can enter chess moves on the command line:

<tut-drone-1.py>

>>> e2-e4

e2-e4
>>> e7-e5
e7-e5
>>> Qd1xd8
ValueError: Qd1xd8

Bummer. There are pawns between the queens, so the move is illegal, but it was worth a try. A ValueError is what you deserve for trying to cheat!

Now let's have a look at our hive. We have placed five drones into the hive: two keyboardmove drones, a chessprocessor, a chesskeeper and a movereporter. For every drone, we will go over the code (in chess/components/drones):

keyboardmove
The keyboardmove drone requires a player argument, which must be "Black" or "White". In addition, it contains a list of moveprocessors. A moveprocessor is a downstream processor of a move: it must be a function that can be called with one argument, a long chess notation string. By default, the list of moveprocessors is empty; however, in the keyboardmove drone's place() method, a socket ("game","process_move") is declared. When this socket receives a moveprocessor plugin,it is added to the list. Therefore, any other drone that wants to declare a moveprocessor should wrap it in a plugin and declare it as ("game","process_move").
The *name* ("game","process_move") is just a convention; we could have called it "process_move" or ("chess","process_move") or ("chess","moveprocessor") or whatever, as long as we consistently stick to it.

The drone also declares a socket ("game", "turn"). This socket should be filled by an external drone, with a function that can be called with zero arguments, and which tells us whose turn it currently is, black or white. This allows the keyboardmove drone to ignore keyboard input if it's the other player's turn: *if self.turn() != self.player: return*

Now comes the heart of the drone: capturing keyboard input and pushing it forward.
As you know, all system processes contain three standard streams that connect them to the command line terminal and to other processes: in Python, they are available as sys.stdin, sys.stdout and sys.stderr. Likewise, in the hive system, every hive and worker contains four *event streams*: evin, evout, everr and evexc (the fourth deals with events caused by a Python exception). Events are just tuples (with a bit of extra, you can find the source code in bee/event.py). To deal with events, every hive has a drone called an eventhandler (bee/eventhandler.py), which provides certain sockets and plugins for listening to events in a hive's evin.
In addition, the command hive is set up so that its evin receives input from the terminal: if you type *spam*, the command hive will receive an event with the value ("command", "spam").
So what we want is to listen for events that start with "command". To do that, we register an ("evin","listener") plugin with the value ("leader", self.move, "command"). This has the effect of: *if event.startswith(("command",)): self.move(event[1:])*

The self.move() method is then quite simple. Since we already chopped off the leading element of the event tuple, the remaining tuple will just contain one string: the chess move that was typed in. The hive system wants us to mark the event as processed (boilerplate again...), but for the rest, all we need to do is to extract the chess move and to send it to all registered moveprocessors.

</chess/components/drones/keyboardmove.py, line 11-16>

```
  def move(self, event):
    if self.turn() != self.player: return
    if len(event) != 1: return
    event.processed = True
    move = event[0]
    for f in self.moveprocessors: f(move)
```

chessprocessor

The chessprocessor is the coordinating drone. It has a number of functions, which we will discuss one by one.

It declares a ("game", "process_move") plugin, so it will receive moves from both keyboardmove drones. In other words: self.process_move is registered as a moveprocessor. The moveprocessor first re-formats the move, so that small inconsistencies are smoothened out (e.g. Ke1-g1 vs 0-0 vs O-O). To do so, it relies on an function declared as a plugin by an external drone as ("game", "format_move"). Then, it pushes the move forward to registered movemakers. As you can see, movemakers can be registered by external drones under ("game", "make_move").

After calling the movemakers, the chessprocessor calls a function that tells us if the game has finished. This function must be declared as a plugin by an external drone as ("game", "finished"). If the game is finished, the exit function is called. All top-level hives in dragonfly (commandhive, consolehive and pandahive) declare an exit function as an "exit" plugin: therefore, declaring an "exit" socket is all that a drone needs to do to get access to this function.

Finally, if the game is not finished, it will flip the self.turn attribute from "Black" to "White" or vice versa. In the place() method, you see that the self.turn attribute is wrapped in a getter function and declared as the ("game","turn") plugin needed by the keyboardmove drones.

chesskeeper

The chesskeeper drone is a thin wrapper around the chesskeeper class. Upon initialization, the drone creates an embedded chesskeeper object. The chessprocessor drone requires a move formatter and the chesskeeper class provides that functionality, so the chesskeeper drone wraps it in a plugin and declares it as ("game", "format_move"). Again, we could have chosen another name than ("game", "format_move"), as long as we had used it consistently in both the chesskeeper drone and the chessprocessor drone.

The same goes for the function that detects if the game is finished or not. It is created as a getter function for the chesskeeper's "finished" attribute, and then declared as a ("game", "finished") plugin.

Finally, the make_move function of the chesskeeper is registered as a movemaker under ("game","make_move"), so that it is called by the chessprocessor when it receives a move from the keyboardmove drone.

movereporter

The movereporter is the simplest drone of all. It registers as a movemaker its print_move function, which does just that.

Placement order

As you can see, the chessprocessor will have two registered movemakers, one from the chesskeeper and one from the movereporter. It will call first the chesskeeper, and then the movereporter. In other words, the move is first made on the board, and printed only afterwards. If the move was illegal, it never gets printed at all.

This is because both drones where declared without a name, and all bees without a name are placed

into the hive in the order they were created in the class. This causes the place() method of the chesskeeper to be called before the place() method of the movereporter, so that the first registered moveprocessor comes from the chesskeeper.

We could have given the movereporter a name, and then the movereporter would receive the move first. Named bees are always placed before unnamed bees:

```
class myhive(commandhive):
  keyboardmove("White")
  keyboardmove("Black")
  chessprocessor()
  chesskeeper()
  m = movereporter() #placed 1st
```

When there are multiple named bees (or subhives), they are placed **not** in order of creation but **by alphabet**, before the unnamed bees:

```
class myhive(commandhive):
  keyboardmove("White") #placed 3rd
  keyboardmove("Black") #placed 4th
  chessprocessor()      #placed 5th
  m = movereporter()    #placed 2nd
  c = chesskeeper()     #placed 1st
```

It is good to pay attention to these rules when building your hives. Fortunately, the order of placement doesn't matter most of the time; but when it does, it can lead to bugs that are very hard to find.

Step 2: Adding a GUI

Now we will add our external chessboard GUI. Because it is external, it is a little harder to implement than if we had used the pandahive, where hive and GUI are already integrated. Both Panda and the hive system require some special setup and they both like to have full control, so to let them cooperate we need to take a bit care. On the other hand, once you know how to do it, you can easily adapt it for a GUI written in Pygame, Tkinter, wx, GTK, Qt or any other library.

On the Panda side, we need to tell Panda where the required models for the chessboard are located:

```
from panda3d.core import getModelPath
import os
getModelPath().prependPath(os.getcwd())
```

Also, Panda has an object called the taskMgr, which needs to be called every frame:

```
from direct.showbase.ShowBase import taskMgr
# This should be called every frame:
#   taskMgr.step()
```

To integrate this into the hive system, we have to modify the commandhive. The code is simple enough, but to know what it does requires quite a bit of explanation.

At runtime, i.e. after close(), every hive consists of a modified libcontext.context. The class of this context is stored in the _hivecontext class attribute.
This context contains, to begin with, all named drones, workers and subhives. If you replace "chesskeeper()" with "c = chesskeeper()", as in the example above, you can then retrieve the current turn:

```
...
m.init()
print m.c.keeper.turn
...
```

Or, if you similarly adapt with "white = keyboardmove('White')", "black = keyboardmove('Black')", you can even set up an initial board position:

```
...
m.init()
m.white.move(bee.event("e2-e4"))
m.black.move(bee.event("e7-e5"))
m.run()
```

In addition, the hive's runtime context can have methods, most notably init() and run(). These are implemented by a specific drone that is wrapped into the runtime context. Such a drone is called an app, and you can wrap it into a context using bee.hivemodule.appcontext:

```
import bee, bee.hivemodule

class spamdrone(bee.drone):
  def spam(self):
    print "SPAM!"
  def place(self):
    pass

class myhive(bee.hive):
  _hivecontext = bee.hivemodule.appcontext(spamdrone)

m = myhive().getinstance()
m.build("m")
m.place()
m.close()

m.spam()

>>> SPAM!
```

Now we are getting there.
The commandhive's app is (surprisingly...) called commandapp. It contains a method on_tick() which is called every tick of the hive. By default, it does nothing. All we need to do is to override it, so that it

calls Panda's taskMgr.step(), and then to specify our modified commandapp as the new app of our main hive:

```
from bee.commandhive import commandhive, commandapp

class myapp(commandapp):
  def on_tick(self):
    taskMgr.step()
    taskMgr.step()

class myhive(commandhive):
  _hivecontext = bee.hivemodule.appcontext(myapp)
```

That's all!
(I know, it calls taskMgr.step() twice, but for some reason, it doesn't work well otherwise. Don't ask me why.)

The only other thing we have to do is to place a chessboard drone into the hive. Again, the code of the drone is in chess/components/drones/chessboard.py.
The drone embeds a pandachessboard object. As you can see, the pandachessboard class is just the normal chessboard class with its move() method overridden: it explicitly calls the parent drone's move() method.

The chessboard drone's move() method works as follows:
First, it checks for the turn, just like the keyboardmove drone does. A player is not allowed to make a move on the board if it is not his/her turn: doing so raises a ValueError. Remember that the chessboard component will move back the piece to its original position if a ValueError is raised.
Then comes a bit of boilerplate. Normally, all bees in the hive system are called in response to events, and all events pass through the event handler before they go into the hive. The ("command", "e2-e4") events that we generated from the commandline also went through the event handler. However, now we are calling bees directly. To be sure that everything works fine, we must first lock the event handler and then release it again. For now, we could get away with forgetting it, but then we would have serious problems in connecting the chess AI in the next section.
Finally, the chessboard drone's move() method pushes moves forward to downstream moveprocessors just as the keyboardmove drone does.
The chessboard drone also acts as a movemaker, just like the movereporter does. Therefore, it receives all moves from the chessprocessor, both its own moves and moves typed in from the command line. In all cases, it simply updates the board position as implemented by the chessboard class.

This gives us the following final code:

<tut-drone-2.py>

As you start the program, you see that you can now make moves using the chessboard GUI. You can still type them in, in which case the chessboard GUI is updated. The GUI auto-promotes pawns to a queen, if you want another piece you must type it, e.g "e7-e8N" for a knight.
If you make an illegal move, the program will abort with a ValueError.

Step 3: Adding a chess AI

Here is the code for playing against the Glaurung engine.

<tut-drone-3.py>

Compared with the previous code, it has the following differences:
- The keyboardmove("Black") is gone. You can no longer type moves for black on the commandline.
- Likewise, the chessboard drone has now an argument "White", so that you cannot make moves for black on the chessboard.
- Finally, a chessUCI drone has been added. The first argument is the player, the second and (optional) third argument are the invocation of the Glaurung engine; if you are on Windows, you probably have to change it into something like "chessUCI('Black','glaurung.exe','C:\\Program Files\\Glaurung')". An easier solution on Windows is to copy the Glaurung/Stockfish executable to the current directory (manual/chess) and name it "glaurung.exe".

You can try if you can beat the engine, just don't misclick, or the program will abort!

The code for the chessUCI drone is in /chess/components/drones/chessUCI.py. It has a lot in common with the chessboard drone: it also has an embedded object that provides most of the functionality, it is also both a move generator and a movemaker, and it pushes moves forward to downstream moveprocessors in the same manner. However, there is one important difference. The chessboard drone (and also the keyboardmove drone) is a spontaneous move generator: it pushes moves towards the moveprocessors. The chess engine, on the other hand, has to be asked for a move: you must pull it out. We will deal with this difference in a much more explicit way in the next part, when we will use workers to build a chess game. However, in both cases, the solution is the same: the event_next plugin. When do we want to ask the engine for a move? That's easy, when the other player has made a move. Since the chessUCI drone is also a moveprocessor, it will be notified: so, we should pull a move from the engine in the make_move() method, if the move comes from the other player (i.e. it is the other player's turn). However, we don't want to make the move right away. If we would have two AI's playing against each other, black would respond to white's move, who would respond to black's move, etc., creating an infinite loop that plays the whole game in a single tick, until either the game is over or Python runs out of stack space. Instead, we want to make the move after the current event has been processed. Therefore, we create a event that we send to a convenient plugin called ("evin", "event","next"), which appends the event to the event queue and processes it when the current event has been processed. We also specify a listener so that the drone can detect its own event emitted one step earlier. In the chessUCI drone, the event is called ("game","engine",id(self.UCIengine),"get_move"); since every drone has its own Python id, multiple chessUCI drones don't get in each other's way. Of course, any other equivalent naming scheme will work just as well.

The chessUCI drone also has a start() method, which listens for the ("start",) event. This event is emitted by the commandhive on the first tick. The start method makes a move if the chessUCI is white, so we can play against it as black:

<tut-drone-3a.py>

Finally, two instances of the engine can play against each other, without chessboard GUI:

<tut-drone-3c.py>

Note that these are really two independent drones: if you want to let two different UCI engines play against each other, all you have to do is change the arguments.

There is just one problem when two engines play against each other: all the events still take place in one tick. If you try to add a chessboard, it will not even get rendered:

<tut-drone-3d.py>

The solution lies in slightly modifying the chessUCI drone. You can see the source code for this chessUCI2 drone in the same directory. Here you see for the first time a drone inheriting from another drone and calling its parent drone's method. It is nothing special, but you just have to know: chessUCI.move(self) won't work, you have to do chessUCI._wrapped_hive.move(self). The move function is modified to include a 1 sec delay. Also, the place() method is modified so that ("evin","event", "next") is replaced with ("evin","event", "next_tick"). In case you hadn't guessed, this function will emit the event only when the next tick has arrived. Here is the final code:

<tut-drone-3e.py>

You can sit back and watch the computer play against itself.

2f. A game of chess, using workers

In the previous section, we built a chess game using drones. Now, we will do the same using workers. Workers can do the same things that drones can: they can declare sockets and plugins in a place() method, which will be automatically connected to any other plugins and sockets of the same name. However, workers don't always have a place() method for connections, because they can use an additional mechanism: explicit connection using antennas and outputs. The principle is simple: one worker declares an output, a second worker declares an antenna, and within the hive, they are connected with bee.connect. For example, the following code will echo whatever you type (until you type "quit" or "exit"):

<tut-worker-0.py>

Both of these workers are defined in dragonfly, which contains over 100 of them. Let's go over the source code of the commandsensor:

<dragonfly/io/commandsensor.py>

The place() method looks a lot like the keyboardmove drone we used in the previous section. Like the keyboardmove drone, it registers a method as a listener for command events. However, the rest of the worker doesn't look anything like a drone: rather, it looks like a miniature hive.
Workers consist of so-called segments that are added to the worker class in the same way as bees are added to a hive. There are 18 different segments: the common ones *connect, antenna, output, modifier, buffer, variable, transistor, parameter, trigger, pretrigger* and *triggerfunc* and the less common ones *startvalue, operator, weaver, unweaver, toggler, untoggler* and *test*. The *connect* segment is the most important one, since it connects two other segments to each other. You shouldn't confuse the *connect* segment (bee.segments.connect) with bee.connect: the first is used to connect segments within workers,

the second is used to connect workers within hives.

Most segments have both a mode and a type. Let's first discuss the types. Types can be "trigger", "str", "int", "float", "bool", a few other ones, or "object". Segments can be connected with "connect(<source segment>, <target segment>)". You can only connect segments if the type and mode match.
You can add some more information to a type if you want: Instead of "int", you can say ("int", "count") . The number of types is fixed, but subtype information is just an extra. The first element(s) of two type tuples must match: you can always connect ("int","count") to "int" or to ("int","count","small"), but you cannot connect ("int", "count") to ("int", "negative").

The mode of a segment can be "push" or "pull". Especially for antenna and output segments, this is a very important choice. A "push" output actively sends its output to all connected target workers. All of these target workers have a "push" antenna (of the same type as the output), waiting patiently for input. In contrast, with "pull" data, it is the output that waits patiently until its value is actively requested by the "pull" antenna of the other worker. To make it do so, the pull buffer or transistor to which the antenna is connected must be triggered.

Let's go back to the commandsensor. As you can see, it contains a buffer segment called b_outp. A buffer segment receives and sends data of the same mode and type. In addition, you can set its value in Python code. A push buffer can also be triggered to push its value forward. In the commandsensor worker, a triggerfunc is used for that, so that the buffer can be triggered from Python code. If we wanted to trigger a segment from another segment, we would use "trigger" or "pretrigger".
So, what happens is that the send_event() method receives the event from evin, it sets the value of b_outp, and then triggers b_outp to push its value to the outp output segment. Thus, the commandsensor has a single output of type ("push", "str"), which contains the value of the command that was typed in.
This output can be then connected in the hive to any worker that has a ("push", "str") antenna. The display worker has such an antenna. Actually, it is a bit more complicated: dragonfly.io.display is a so-called metaworker: a callable object that takes some arguments and then returns a worker:

<dragonfly/io/display.py>

The actual worker definition starts at "class display(bee.worker)". Since "type_inp" is "str", you can see that it has indeed an antenna "inp" of type ("push", "str"). This antenna is then connected to a variable "v_inp". A *variable* segment is just like a buffer, except that it takes push input and gives pull output. Since v_inp is not connected further to any other segment, a push buffer could also have been used here.

Next comes a modifier segment called "display". A modifier is a segment that decorates a Python method so that the method can be triggered like any other segment. The method cannot have any arguments other than self. In the code, the modifier segment is followed by a trigger segment, which makes sure that the modifier is called whenever v_inp is updated by inp. First, v_inp is updated and then display() is called: if we had wanted it the other way around, a pretrigger segment would have been used instead.
The display() modifier calls a displayfunc() function, which comes from a "display" plugin supplied by the top level hive: in case of the commandhive, it is just the Python print function.
In other words, the inp antenna updates the v_inp variable, which triggers the display() modifier, which prints the v_inp variable using the hive's "display" plugin.
Finally, the display worker has a variable called "header", which has a "str" type. It is followed by a

parameter segment. A parameter segment makes a variable segment or buffer segment definable in the __init__ function: instead of "display('str')()" we could say . "display('str')('Value:')" or something. The second argument provides a default value None for the header. If the header is not None, it gets printed in front of every value.

To summarize everything what happens in the hive: when you type "e2-e4", the hive receives an event ("command","e2-e4") in its evin. This event is then processed by the eventhandler. The commandsensor worker has registered with the eventhandler a listener for events starting with "command", so its send_event() method receives an event ("e2-e4",). It then sets the value of the b_outp buffer to "e2-e4" and then triggers it, which outputs the value through outp. The value is received by the display worker on its inp antenna, which updates the v_inp variable. This triggers the display() modifier, which prints the value using the hive's display plugin.

Instead of stating "connect(com, d)", we could have stated explicitly "connect(com.outp, d.inp)", but it is not necessary. The hive system is smart enough to figure out that there is only one possible combination of output and antenna.
You have to remember that an antenna segment or an output segment is just syntactic sugar around a socket or plugin. At runtime, they do not exist: you cannot ask commandsensor.outp and display.inp for their value, nor can they send or receive triggers. All that they do is to make sure that commandsensor.b_outp and display.v_inp are connected with each other.

Implementation details
In case you want to know, this is how it works "under the hood":

Like hives, workers have their own libcontext.context. Within the worker context, each output and antenna declares a single socket or plugin.

A push output declares a socket_container called ("bee","output",<name>,<type>). The socket expects to receive function objects that take a <type> argument, and the socket appends these function objects to an output list.
A push antenna declares such a function object as a plugin_supplier called ("bee","antenna",<name>,<type>).
At runtime, every function in the output list is called with the output value.

A pull antenna declares a socket_single_optional called ("bee","antenna",<name>,<type>). It expects to receive a function object that takes zero arguments, and that returns a <type>.
A pull output declares such a function object as a plugin_supplier called ("bee","output",<name>,<type>).
At runtime, when triggered, a pull antenna calls the function object to get a <type> object, or it raises a RuntimeError if it didn't receive a function object when the hive was closed.

bee.connect is a wrapper around a special connection context that contains one socket and one plugin, one of them coming from an output (or event stream) and the other from an antenna (or event stream). It does some searching and some checking that their types match. It imports the socket and the plugin from their worker contexts under the same name, so that they get matched when the connection context is closed.
(end of implementation details)

Step 1: Command line chess

With the following code, you can type in chess moves at the command line. It works just like the drone version, but now it uses workers:

<tut-worker-1.py>

The commandsensor worker is connected to the chessprocessor worker with "connect(com, g)". That means that the chessprocessor worker must have a ("push", "str") antenna. Let's have a look at the code of the chessprocessor worker (chess/components/workers/chessprocessor.py):
You can see that it has indeed an antenna of type ("push", ("str", "chess")), which is compatible with ("push", "str"). The name of the antenna is "inp_move": so, instead of "connect(com, g)", we could also have stated "connect(com.outp, g.inp_move)", "connect(com, g.inp_move)", or "connect(com.outp, g)".

You can also see that the chessprocessor worker contains a "str" variable "v_turn" that describes whose turn it is. Obviously, its start value is "White". v_turn is also exported as a pull output, so that workers connected to this output can retrieve the turn. This is an alternative to the "turn" plugin used in the drone version. It is not yet connected to any worker in the main hive.

The chessprocessor worker also contains a pull antenna "inp_gamekeeper" for an ("object", "gamekeeper"). Nor surprisingly, this object is supplied as an output by the chesskeeper worker:

<components/workers/chesskeeper.py>

If we look at the code, we see that the chesskeeper worker contains an ("object", "gamekeeper") variable, which is assigned to a new embedded chesskeeper object in the place() function. The object is exported through an output called "gamekeeper". Therefore, in the main hive, the statement "connect(k,g)" actually means "connect(k.gamekeeper, g.inp_gamekeeper)".

Going back to the code of the chessprocessor, there is an output called "finished", of type ("push","trigger"), meaning that it just sends a signal without providing a value (like providing zero arguments to a function).
The "finished" output is triggered when the game is over. It is connected in the hive to an exitactuator worker, which exits the hive. In the drone version, this exiting functionality was provided by the chessprocessor drone, using the commandhive's "exit" plugin. Knowing that, you can now guess how the exitactuator looks like:

<dragonfly/sys/exitactuator.py>

As you can see, there is the ("push", "trigger") antenna, which triggers a modifier. [Normally, antennas cannot send triggers, but ("push", "trigger") antennas can. Likewise, ("push", trigger") outputs can be triggered] . Then, the triggered modifier calls the top hive's "exit" plugin in the same way that the chessprocessor drone does, and the hive exits.

Now we are coming to the most important part of the chessprocessor: processing the moves and pushing them forward.

<components/workers/chessprocessor.py>, lines 21-49

The inp_move antenna has been discussed above: it receives the move from the commandsensor and

then stores it in v_move.
On the output side, there is a ("push", ("str", "chess")) output called "outp_move". The value it outputs is stored in the v_outp_move variable. However, a variable gives pull output only. To convert it to push, a transistor t_outp_move is used. Triggering the transistor pulls the value from v_outp_move and pushes it into the outp_move output.
(We could have saved one segment here by using a push buffer instead of a variable, and connecting it directly to the output. We would then have triggered the push buffer instead of the transistor.)

The actual processing is done by the process_move modifier. At the very bottom you can see "trigger(v_move, process_move)", which means that process_move() is called whenever v_move has been updated (by the inp_move antenna). The process_move() method triggers the retrieval of the gamekeeper object through the inp_gamekeeper antenna. Then it uses the gamekeeper to format the move that has just been received. The formatted move is stored in the v_outp_move variable, and the t_outp_move transistor is then triggered so that the move is sent over the outp_move output.

One of the workers that thus receives the move is the chesskeeper worker: in the main hive, through "connect(g,k)", the outp_move output is connected to the chesskeeper's make_move antenna. This antenna stores its value in v_make_move. The update of v_make_move triggers the do_make_move() modifier, which calls the make_move() method on the embedded chesskeeper object, with v_make_move as the argument.
The other worker worker that receives the move is the display worker.

Back to the chessprocessor worker: after calling all movemakers, if the gamekeeper says the game is over, the "finished" output is triggered. If not, the turn is changed from "White" to "Black" or vice versa. Finally, the "made_move" output is triggered, indicating that a move has been made successfully.

Hopefully, workers are now starting to make sense to you. If not, it is better to read again the current section up to this point. From now on, we will go a little bit faster.

Two command line players
The first step towards adding a GUI or an AI is to check that a player can only make moves if it is his/her turn. In the drone version, the drones for command line input, GUI input and AI input already had this check implemented. Now we will implement this check at the hive level, using workers.

<tut-worker-1a.py>

The module "logic" of the dragonfly library contains a metaworker called "filter", which takes one type argument to create a worker. The worker contains a ("push", <type>) antenna "inp" and a ("pull", "bool") antenna "filter". When it receives input on "inp", it pulls the value of "filter". If the value is True, the input is pushed to the ("push",<type>) output segment "true". If the value is False, it goes instead to the ("push",<type>) output segment "false".
We create two filter workers, one for white and one for black. The commandsensor is connected to the "inp" antenna of each filter. The "true" output of each filter is connected to the chessprocessor.
To each "filter" antenna we connect a metaworker called "equal2", which takes one type argument to create a worker, which then takes one parameter argument. Obviously, the worker has an output ("pull","bool"), or else we couldn't connect it to the filter. Whenever this output is pulled for a value, the "equal2" worker pulls its input, which is a ("pull", <type>) antenna. If the value of the input is equal to the value of the parameter (in our case: "Black" or "White"), it returns True, else it returns False. The input of the filter is connected to the "turn" output of the chesskeeper.

So, what happens is the following. Whenever the command sensor receives a move, it first sends it to the white filter. The filter first pulls from the 'equal2("White")' worker. This worker pulls the "turn" output value of the chessprocessor, and returns True or False depending on whether it is "White" or not. If it is False, the white filter outputs the move on its "false" output, which is not connected to anything. Only if the results is True, the move is pushed to the chessprocessor over the "true" output segment, and the move is made.
Then, the command sensor sends the move to the black filter. The filter pulls the 'equal2("Black")' worker, which returns True if the chessprocessor's turn is "Black". Only then, the black filter pushes the move forward.
Sounds good, right? Unfortunately, it doesn't work. If you type in "e2-e4", you will get an exception. First, the commandsensor calls the white filter and the white move is made. Because of the white move, the chessprocessor's turn is immediately updated to "Black". So, when the commandsensor then calls the black filter, the move also passes the filter and the chessprocessor tries to make the move "e2-e4" for black, which raises an exception. So, the turn is updated too early.

The solution is the same as for the two AIs in the drone version: to delay the update of the turn until the next event. Here is the fixed version of the code:

<tut-worker-1b.py>

As you see, there is now an "on_next" worker added. It takes a ("push","trigger") input, which will be held until the next event and then forwarded. It is triggered whenever a move has successfully been made. After the current event has been processed, it will trigger a transistor "t_turn" that pulls the current turn and pushes it into a variable "v_turn". This "v_turn" variable is now where the filters are reading from, instead of directly from g.turn.

Note: We are building a hive, but this part looks extremely similar to building a worker! Although the transistor and variable are metaworkers (dragonfly.std.transistor and dragonfly.std.variable), and not segments, the usage is almost the same. If you happen to forget the () at the end, the hive system will tell you.

Note 2: the drone version for command line chess avoided this "turn problem" by marking the command event as "processed". We could do the same here, connecting the filters to some hacked version of the command sensor that has an additional ("push","trigger")-"mark_last_event_as_processed"-antenna. However, this would seriously restrict the flexibility for adding another command sensor later, e.g. for logging. That's why we use another way in this tutorial, but the choice is yours.
Or, if you want, you can create a worker that has this antenna and for the rest just transmits any event it receives. Combined with some workers in dragonfly.event, you can tap into the evin stream without using the command sensor, marking events as processed while still maintaining full flexibility. Again, the choice is yours.

Exceptions
Before moving on to the GUI, there is one more thing that we should take care of. The Python exception we received was very uninformative. Exceptions will be discussed in more detail later, but for now it is enough to know that adding the following snippet will tell you in which context (hive, subhive(s), worker, segment) the exception was raised:

```
  raiser = bee.raiser()
  connect("evexc", raiser)
```

Here is the final version of the code:

<tut-worker-1c.py>

Step 2: Adding a GUI

Now we will add a Panda GUI to the hive, just like in the drone version:

<tut-worker-2.py>

The hive code is straightforward. It contains the same stuff to make Panda and the hive system cooperate as the drone version. The chessboard is now a worker instead of a drone. As a move generator, it is connected to the chessprocessor. As a move maker, it also receives a connection back from the chessprocessor. Finally, it is connected to the current turn in the same way the black/white filters are. You can look at the source code and compare it to the code of the chessboard drone, and it should be pretty obvious to you.

Exceptions
The drone version suffered from the limitation that you couldn't make any misclicks or typos, or the game would crash with an exception. The current version still crashes, but at least tells you where the exception comes from. Now it's the time to go a bit deeper into detail about exceptions in the hive system.

All workers and hives contain an event stream "evexc". Through this stream, objects of the type bee.exception are sent. Bee.exception is a subclass of bee.event, which is itself a subclass of tuple. All event streams can be connected to workers: in case of evin, a worker with an output ("push", "event") can connect to it; evout and everr can connect to an antenna ("push","event"), and evexc can connect to ("push", "exception"). Event streams can also be connected to one another.
A bee.exception object is always a tuple of two, and both of the elements are tuples themselves. The second element is a tuple of two, containing the Python exception class and its value. The first element is the *context tuple*. The context tuple contains initially two elements, the worker name and its segment. This exception object will then be forwarded to all workers connected to the worker's evexc.

Every connected worker that receives a bee.exception object has the possibility of clearing it by setting its "cleared" attribute to True. If no connected worker does this, the exception is re-raised. However, in addition, every evexc from a worker or subhive has a "last-resort" connection to the parent hive's evexc. The parent will then put its own context name in front of the context tuple and send it through its own evexc.
Let's consider the following example:

```
class someworker(bee.worker):
  @modifier
  def some_modifier(self):
    raise ValueError("somevalue")

class childhive(bee.hive)
```

```
  w = someworker()

class tophive(bee.hive)
  c = childhive()

t = tophive()
t.build("t")
t.place()
t.close()
```

Now, suppose that ValueError("somevalue") is raised in "some_modifier". This creates a bee.exception object ((("w","some_modifier"), (ValueError, "somevalue")) in t.c.w.evexc. This exception will then appear in t.c.evexc as ((("c", "w","some_modifier"), (ValueError, "somevalue")). Finally, it appears in t.evexc as ((("t","c","w","some_modifier"), (ValueError, "somevalue")). However, the tophive has nowhere to send it to, so the Python exception will just be re-raised.

Now you may have also an idea what the raiser snippet does:

```
  raiser = bee.raiser()
  connect("evexc", raiser)
```

The bee.raiser is a worker with a ("push","exception") antenna. Upon receiving a bee.exception, it will raise a WorkerError Python exception, which contains both the context tuple and the original Python exception, giving us useful information for debugging. The WorkerError is raised with the original traceback.
The word "evexc" is in quotes, because there is no "evexc" symbol name in the current class definition, but it is instead defined in the parent class (commandhive). Bee.connect can take as argument either a bee (raiser or raiser.raisin) or a bee name ("raiser" or "('raiser','raisin')"). If you delete a bee (raiser = None), all associated connections are deleted as well, unless they are by name (of the deleted bee: raiser = None deletes connect("evexc", raiser) but not connect(evexc, "raiser") or connect("evexc", "raiser")).

Stack-based exception handling versus hierarchical exception handling
A major piece of trouble is coming up the horizon: catching exceptions properly. And I really mean *catching* exceptions: as you can see, the hive system has no problem in re-formatting and re-raising exceptions into something more informative. With *catching*, I mean converting exceptions into some message or action and then go on with the program.
Any kind of node system has a fundamental problem with this. If an exception arises in a worker X, ***who is responsible of dealing with it***? Python has one answer: the worker that called X, i.e. tried to push a value into X or pull a value out of X. This is stack-based exception handling, based on the call stack (or traceback stack, if you prefer). The alternative answer is that the parent hive should be responsible for dealing with exceptions in its children, in worker X or subhive Y or whatever. Such hierarchical exception handling is great if you want to use the evexc stream for exception handling, as we want to do here.

Here comes our first shot at catching exceptions, using hierarchical exception handling:

<tut-worker-2a.py>

We connect the evexc of the chesskeeper, the chessprocessor and the chessboard to an exception reporter worker called "except_valueerror". If you look at the source code of "except_valueerror", you see that it contains a ("push","exception") antenna which takes the exception, stores it and triggers a modifier that prints it and marks it as cleared.
However, this doesn't do what we want. For example, if we type in "Qd1xd8", it gets properly reported as invalid move, but the move is nevertheless made on the chessboard (ha! got your queen!) , and is also printed out by the display worker. This is because the ValueError is raised by the chesskeeper, and reported by except_valueerror, but then it is swallowed, so the chessprocessor afterwards calls the chessboard with the illegal move, which it happily carries out.

Here comes the trouble: to solve the problem, we need to mix the two kinds of exception handling. We want to propagate, stack-wise, up to the chessprocessor's process_move() function, all exceptions in the chesskeeper and the movemakers, i.e. in all functions that the processor calls in self.gamekeeper.format_move and self.trig_outp_move. This we can do with a Python "try" block. As always in Python, an exception will abort the "try" block, so that the chessboard will never be called with a move that raised an exception in the chesskeeper.
Then, we want to deal with the exception in a hierarchical way: if it is a ValueError, as an exception raised in the chessprocessor worker, that gets reported by except_valueerror; else, as an exception in the original worker, that finally makes it to the raiser.

As mentioned above, the hive system forwards all exceptions to evexc, and every worker's and hive's evexc is connected to the parent hive's evexc. We have connected the main hive's evexc to a raiser, whose function is to flag the exception as "cleared" and then raise a new WorkerError exception, containing the original exception and the context tuple. However, the raiser also makes sure that the "try" statement does not work, because there will be no ValueError to catch. Fortunately, if there is no raiser or other worker that flags the exception as cleared, the hive system will fall back to Python's stack-based exception handling. So the key to switching to stack-based exception handling is to disable the raiser.
To facilitate this, there are a couple of plugins provided by the eventhandler. Every raiser must call the function provided by ("eventhandler","raiser","active") and return immediately if this function returns False. Bee.raiser does this and so does our except_valueerror worker. In addition, there are plugins ("eventhandler","raiser","activate") and ("eventhandler","raiser","deactivate").
So, in the chessprocessor, we must disable the raiser, forward the moves, and re-activate the raiser (more precisely, restore it to whatever state it was before) The chessprocessor2 worker is a version of the chessprocessor worker with these adaptations implemented.
The hive code is then as follows:

<tut-worker-2b.py>

Now we are almost there. The only thing still missing is that if we make an illegal move using the chessboard, the pieces should return to their original position. The component has it implemented already, but here too, we must switch to stack-based exception handling and back again. In the chessboard2 worker, this is dealt with correctly. When a move is made, the raiser is first disabled so that ValueErrors can be caught in a "try" statement. If the move turns out to be illegal, the chessboard component will now correctly move back the pieces and then call the invalid_move() method. In the invalid_move() method, we simply re-enable the raiser and make the move again, knowing that a new ValueError will be raised, caught and reported by the chessprocessor2. (Don't try to simply call "raise": the old ValueError has already gone through all the evexc'es and will not go through again; instead, it will go straight to the main hive evexc, stopping the hive.)

Here is the final hive code:

<tut-worker-2c.py>

As you can see, you can make moves on the chessboard, and for illegal moves the pieces are moved back to their original positions. All illegal moves are reported, no matter if they came from the chessboard or from the command line.

Note: The section above contains a lot of explaining. However, once you know what to do, the actual code is quite simple: a few lines of code to get the raiser control plugins, two more lines to call them. If you still find it too complicated for your taste, you can simply choose not to use a raiser at all, and to rely completely on stack-based exception handling. As always, the hive system gives you the choice.

Step 3: Adding a chess AI

Adding the chess AI is then the same as in the drone example. Again, the chessboard is re-configured to allow only moves for white, the black filter is gone, and there is now a new chessUCI worker.
Like the chessboard, its output is connected to the chessprocessor, and it also receives input from the chessboard as a movemaker. It is also connected to the on_next worker, to generate a new move whenever an old move has been made. Of course, this is only done on the correct turn: for this, it uses the chesskeeper's turn directly (the use of the on_next worker makes that no delay mechanism is necessary).

<tut-worker-3.py>

The version where the engine plays as white is equivalent, except that the engine is now connected to a startsensor to make the first move.

<tut-worker-3a.py>

Worker-like subhives
The next version of our code will have the same functionality (human playing black against a computer), but it will have a slightly different form. It is possible to make the hive considerably more readable by grouping some workers into subhives that behave like one big worker.

In components/workers, you can find two of these worker-like subhives, called "human" and "computer".

<components/workers/human.py>
<components/workers/computer.py>
<tut-worker-3b.py>

Looking at the source code of "human", you see that the base class is bee.frame, a type of hive that imports and exports all sockets and plugins from and to the parent hive. A frame is perfect for grouping workers and drones to make the main hive more readable. The bee.parameter makes that "human" needs a string to be initialized: as you can see, this string is used as the parameter for the equal2 worker.
Bee.antenna and bee.output work a bit differently than their counterparts in bee.segments. You don't

need to declare a type or mode: what bee.antenna and bee.output do is create an entry point at the hive level for an existing antenna/output segment. You can connect to human.com as if you were connecting to human.p.inp: the mode/type of human.com is the mode/type of human.p.inp, namely ("push", ("str","chess")). The "turn" antenna and the "move" output are likewise exporting/exposing a worker antenna/output segment to the outside world.

Looking at the code for the "computer" hive, you see that it has three parameters: par1_player, par2_engine_binary and par3_engine_dir, with the default value of None. Remember that named bees are evaluated in alphabetical order: bee.parameter is no exception to this rule. By naming the parameters alphabetically, the first parameter is the player, followed by the engine binary and (optionally) the directory, just like the chessUCI worker.
Note that we have also included a startsensor, which will respond to the ("start",) event. This only works because frames do not have an eventhandler of their own, and instead use the parent hive's eventhandler. Therefore, events received by the parent hive's evin also reach subhives, if they are frames.

Finally, here is the code to let two computers play chess against each other:

<tut-worker-3c.py>

You see that the on_next worker has been replaced by an on_next_tick worker, similar to the drone version. There are two of these workers, one for each computer. Also, instead of the "computer" subhive, it uses "computer2", which adds a delay of 1 second. You can look at the "computer2" code to see how it is done. You can also look at "computer2a", which is a different implementation of "computer2": instead of starting over from scratch, it inherits from "computer" and makes some changes.

Step 4: Building a chess game by visual programming

This tutorial has covered a great deal of the lower and middle layers of the hive system. The last part consists of a quick demo of the high level. It will be discussed in much more detail in the next tutorial. All that will be said now is that the hive system can build hives from arbitrary data. Power of this kind is not often found outside Lisp-family languages.
Tut-worker-4.py contains yet another hive for playing chess as white against the computer. It is the same as tut-worker-3.py, except that the hive has been reduced from 36 lines to 6 lines. The code in tut-worker-4a.py is different in one letter, and lets you play as black. Tut-worker-4b.py differs also in one letter, and lets two computers play.
The one-letter difference is because they load different data files: tut-worker-4.web, tut-worker-4a.web and tut-worker-4b.web. These web files are hivemaps that describe the whole hive: the workers, their parameters and their connections.

There is a GUI to edit these hivemap files. It is still very rudimentary, I have not yet found the time or the skill to make it more attractive. But it works.
First, install wxPython if you don't have it already. Go to the hivegui folder. On a command line, type "python hivegui.py /../manual/chess/tut-worker-4.web". On Windows, just click on hivegui.py to start the program, and open the web file with the File menu.
Finally, click "Zoom to fit".
You can play around with left and right mouse clicks and change the hives in whatever way you like.

Enjoy!

Note: if you are on Windows, you will need to adapt the Glaurung path before the hivemaps will work as executable hive code. In the GUI, click on the "computer" or "computer2" worker, and change the text fields. Again, an easier solution might be to copy the Glaurung/Stockfish executable to the current directory (manual/chess) and name it "glaurung.exe".

Note2: When you open the hivemap, you will see a black Panda 3D screen appear. This is because all workers are imported, including the chessboard worker, which will import the chessboard component and start Panda's directgui. I didn't find a workaround for this, yet. Please ignore it.

Note3: You could also use a text editor to edit the hivemap files, if you prefer.

2g. What to use: drones or workers?

We have reached the end of this long tutorial, where we have glued a chess game together, first with drones, then with workers (and finally in a GUI). Both drones and workers have their uses, depending on your style and on the situation. Here is a final summary of the differences between them:

- Workers have explicit connections, drones do not
- Workers have static typing, drones do not
- Workers can be placed and connected in a GUI
- Drones have an __init__ method, workers do not
- Both workers and drones (can) have a place() method for sockets and plugins
- You can subclass drones (with a bit of difficulty); you cannot subclass workers
- Both workers and drones can be grouped into a subhive that behaves like one big worker or one big drone.

Tutorial 3: moving pandas in 3D

This tutorial has two purposes. First, to show the hive system's 3D capabilities. Currently, the hive system has bindings to Panda 3D; in the future, bindings to the Blender Game Engine and Python-Ogre will be added, so that all hives will run under those 3D engines as well. Second, to demonstrate the hive system's high levels, allowing complete configuration and customization by data files and visual programming.
This is again a "slow" tutorial, explaining all concepts in detail, so that can extend them to your own projects. I will assume that you are familiar with workers and drones and other lower-level stuff of the hive system. For a quick practical introduction, have a look at the Tetris tutorial. For a much deeper understanding, study the chess tutorial.
If you want just a quick demo of 3D and visual programming in the hive system, this tutorial isn't for you. Take a look at the screencast instead.

3D and high-level stuff mix well together. No sane person programs a 3D game by hard-coding all vertex coordinates. You store them in a data file, created with some high-level modeling program, and in your code you invoke a high-level engine that reads in the file and breaks it down to low-level OpenGL or DirectX commands. Even that is only done in a tutorial: in a real application, you don't hard-code the file names of your models, but you store them in a high-level configuration file (XML or whatever), using different configuration files for different areas of your 3D world.

So, here's the plan: first, I will show the hive system's 3D stuff at a fairly low level. It is flexible but crude, and the code looks ugly. Then, I will move on and explain how the high level is built on top of that, producing nice and clean code and data files.
As a basis, I will use Panda 3D's "Hello World" panda tutorial (http://www.panda3d.org/manual/index.php/A_Panda3D_Hello_World_Tutorial), where it is explained how to place a panda in a grass scenery and make it walk back and forth. I will assume that you have some familiarity with 3D programs and programming; if not, you should first do the original Panda 3D tutorial before continuing here. You should also check out the wonderful game tutorial at http://www.mygamefast.com, which does an excellent job of teaching Panda3D programming and general game design to non-programmers.

Showing a panda

Here is the code that shows the panda and the scenery. Bindings to Panda 3D are through the pandahive. For now, the camera is still controlled directly from Python, using code from the Panda 3D tutorial.

<panda-1.py>

pandahive.scene is a pandascene drone that takes commands that define a 3D model. You can find it in pandascene.py in the /dragonfly/pandahive directory. As you can see, the pandascene API is very ugly to call directly from source code: the underlying Panda 3D API is much better suited for that. Instead, the pandascene is designed with two other goals in mind:
First, you can use it before the 3D scene is actually initialized. Because it is called before place(), calls to the pandascene can be mixed with code that causes the placement of additional plugins and sockets.
Second, all calls are completely independent from each other, they don't need any variables returned by a previous call. This is a crucial requirement for baking them into the hive.
Both of these features are essential for building high-level layers, as we will see later on.

Making the panda walk on the spot

In the following code, we use the "W" key to make the panda walk on the spot, and "S" to make it stop again:

<panda-2.py>

As you can see, we have added a parameter 'entityname="mypanda"' to the add_actor_MATRIX call.
First, a little bit of the terminology used in the hive system. First, every object that loads 3D information from a file, in a certain format (e.g. Panda3D .egg) is called a **mesh**. Second, as in Panda3D, a **model** is an object without animation, while an **actor** has animations attached. A model consists of a single mesh whereas an actor has meshes for the coordinates and for the animations. Finally, as in Ogre, an **entity** is any movable object; all actors are (also) entities but not all entities are actors: an entity can also be a model or a group of other entities (in Panda3D, an entity is called a node).
In the pandascene, you can provide an entityname,which serves as an ID for the created entity in the hive system. This is used by the animation worker, to retrieve the actor that is to be animated.
We have stored the entity ID and the name of the animation in variables, and both variables are connected to the animation worker. The animation worker is then controlled using the two keyboard sensors.

The animation worker is from the directory dragonfly.scene.unbound, which also contains other workers to manipulate entities (position, orientation, etc.). There is also a directory dragonfly.scene.bound, which contains the same workers, but they don't use an entity ID: instead, an entity must first be bound to the hive. We will come to that later.

Spawning additional pandas

From this point, the Panda 3D tutorial proceeds by explaining how to move the panda in space. Here, we will delay that for a while. For now, we will focus instead on the spawning of additional pandas. The hive system also has the concept of **modelclasses** and **actorclasses**. These are defined in the same way as normal actors and models, but they are not rendered right away. Instead, they are spawned by the spawn worker, that accepts a matrix. The following code spawns a second panda in front of the first, when you press the Z key:

<panda-3.py>

The spawn worker contains two antennas. A pull antenna is connected to the ID of the actorclass ("pandaclass"). A push antenna spawns an instance of that actorclass, with the entity ID ("panda2") and entity matrix (6 units forward) that is pushed into it.

Note: As you can see, dragonfly.scene provides a matrix class that is a bit nicer than the bare-bone matrix tuple used by the pandascene drone. It can be initialized from a Panda3D NodePath; in this case, it encodes a position of y=-6 (don't ask me why, but -y is "forward" in the coordinate system of the panda model).

If you press the Z key another time, no new panda will appear. This has two reasons. First, we try to add an entity with the same ID, which isn't correct (right now, this doesn't raise an error in the hive system, but maybe it will in the future). Second, it places the panda at exactly the same spot, so you won't see it anyway.
So, we need to set up some workers to generate a new ID and matrix. The hive system has two workers that are useful for this:

- dragonfly.std.iterator(<type>)(<iterable>). <iterable> can be any Python iterable: a list, a tuple, any object with an __iter__ method, a generator expression, or a call to a generator function.
- dragonfly.std.generator(<type>,<generator>)(). <generator> can be a generator function, or any other object that returns an iterable when called.

Generator functions are Python functions that have "yield" instead of "return". For more information on generator functions and iterables, consult the Python documentation.

The following code uses two generator functions: an ID generator that returns "spawnedpanda1", "spawnedpanda2", ..., and a random matrix generator that returns a random orientation and position of the panda.

<panda-4.py>

Now, if you press the Z key repeatedly, a whole herd of pandas will be created. Note that these are really independent actors, not Panda3D instance nodes: the W and S keys affect only the initial panda.

Configuring the hive

When programming, it is good practice to put as much code as you can into a class. This makes the code much more flexible and reusable, because someone else can embed or inherit from your class and then adapt it to their needs by overriding. However, we have put a lot of code outside of the hive class. We could put it into a method, but doing this can make things messy if you do it too much: for example, method A must be called before method B, which first calls a base method B*, which calls C... Also, if you want to re-use some part of a method but not another part, you may end up copy-pasting code anyway.
The hive system has a much cleaner alternative: configure bees. You have to realize, though, configure bees are not for every situation. For simple parameterization you can just use class attributes in combination with bee.attribute and bee.get_parameter. In addition, code that is called at run-time, or heavily communicates with other code, can be better implemented by a drone or a worker. However, initialization consisting of extensive, data-heavy descriptions and declarations is where you want to use configure bees. In other words, configure bees are great for stuff like defining actors and models.

Configure bees are created inside a hive, using 'c = bee.configure(<target>)'. <target> can be any bee: any drone, worker or subhive that is part of the same hive. Alternatively, <target> can be a string that describes the name of such a bee. In that case, it can refer to a bee that was defined in a base class, or even to a bee that was supplied as a named parameter to the hive (bee injection).
After you have created a configure bee, you can call methods on it as if it were the target. So if your drone "spam" implements a method "eggs", you can do the following:

```
class hive:
  s = spam()
  c = bee.configure(s)
  c.eggs()
```

However, the call to "eggs" is not made immediately, but it's buffered, along with any arguments passed to "eggs". These buffered calls are evaluated upon the placement of the configure bee: when a hive's place() method is called, the hive will call the place() method of all its children (see the "placement order" section in the chess tutorial to see in which order). When the configure bee's place() method is called, its buffered calls are executed in the same order that they were made. For debugging convenience, any exceptions are caught and re-raised in the original code frame, as if the call was executed immediately. The only limitation is that any value returned by the buffered call will be ignored.

If you want to make multiple calls to spam, there are three ways to do it:

```
class hive:
  s = spam()
  c = bee.configure(s)
  c.eggs()
  c.eggs()

class hive:
  s = spam()
  c = bee.configure(s)
  c.eggs()
  c2 = bee.configure(s)
```

```
  c2.eggs()

class hive:
  s = spam()
  c = bee.configure(s)
  c.eggs().eggs()
```

The first method is the most obvious one. The second method makes most sense if you are subclassing. Here, the name of the new configure bee c2 will make sure that it will be executed after the original one: with a name like c0 it would be executed in front. The third method exploits the fact that the buffered call always returns *self*, allowing you to chain configuration calls indefinitely.

In addition to configure bees, there are also init bees, which work exactly the same way, except that they are evaluated just after hive initialization: hive.init() will first initialize all workers' startvalues and then evaluate any init bees placed into the hive.

In the following code, the model, actor and actorclass definitions have been made using configure bees. For readability, they are defined not in the main hive, but in a subhive frame called "myscene".

<panda-5.py>

In a derived class of "myscene", either of them can be easily removed by setting its configure bee to None. For example, the following code will remove the grassy scenery from the hive:

```
class myscene2(myscene):
  c1 = None

class myhive2(myhive):
  #override the placement, replacing "myscene" with "myscene2"
  pandaname_ = bee.attribute("pandaname")
  pandaclassname_ = bee.attribute("pandaclassname")
  myscene = myscene2(
          scene="scene",
          pandaname=pandaname_,
          pandaclassname=pandaclassname_,
         )
```

In addition, we have now parameterized the name of the panda entity "mypanda" and of the actorclass "pandaclass". You can create a derived class where these names have been changed and it will still work:

```
class myhive2(myhive):
  pandaname = "mypanda2"
  pandaclassname = "pandaclass2"
```

You could parameterize other variables (the name "walk" of the animation, the file names of the Panda eggs) in the same manner.

Spawning pandas with a mouse click

Now, we will add a 2D icon of a panda. Clicking on the icon will spawn a new panda as if we had pressed the Z key. To do this, we add the following elements:

- A pandacanvas drone to support the drawing of images and other 2D elements
- A mousearea drone to register mouse clicks on specified screen areas
- A identifier "pandaicon" that identifies both the image and the clicking area
- A mouseareasensor worker to listen for clicks on the "pandaicon" area

<panda-6.py>

As you can see, the drawing and registration of the area are done in the myscene frame using init bees. The area's left upper corner is at (50,470) and the area is 96x96 pixels, as specified in the box2d. We also define a dummy parameter class, which we use to tell the pandacanvas to enable transparency for this image.

Note: if you compare this code with the Tetris example (or the Game of Life example in the introduction), you see that in those examples, a canvas.draw worker is used instead to draw on the canvas. This gives equivalent results: pandacanvas.draw_image is exposed as a ("canvas","draw", ("object","image")) plugin, which is automatically detected and used by the canvas.draw worker. So, the canvas.draw worker is fully capable of drawing an image if we store the file name in an ("object","image") variable. We could connect the worker to this variable, also with the box and the parameter values, and to a start sensor.
If you plan to update the icon appearance regularly at run-time (animated icon, recharge time after activation, ...), this is the way to go. However, since the icon is never updated in our example, it would just be overkill: using bee.init is more straightforward and requires fewer lines of code. The hive system gives you the choice.

The Spyder framework

Up to here, we have been doing fairly direct, low-level stuff. Now we will make our first turn towards high-level hive programming, using the Spyder framework.

The Spyder framework is by far the oldest part of the hive system. In fact, the whole hive system was started as an extension of Spyder, to make it more powerful: now it is more like the other way around. Spyder is a data modelling framework, to define, create, validate, load, save and convert between data models. This can be used for example to auto-generate scientific web servers (http://haddock.chem.uu.nl/Haddock , http://haddock.chem.uu.nl/services/CPORT and http://haddock.chem.uu.nl/enmr/csrosetta.php).
Data models can also be parametric 3D objects, such as tubes, screws, or even houses. Some time ago, I made a Blender 2.4x plugin where you can edit data models in Blender, adjusting their parameters and seeing their 3D representation update automatically (and the other way around). Once I have time, I will adapt it to the new Blender 2.5. Since you can add Spyder objects directly to a hive, it should be quite useful for the hive system.

To discuss all the features of Spyder is out of the scope of this manual. There are some old tutorials for it, and dusting them off is next on my todo list. Spyder-within-the-hive-system also has a few glitches

still. I added workarounds to bee and dragonfly so that the existing data models are loaded correctly, but another data model would require another workaround. However, once these glitches are fixed, you can then easily create your own data models (monsters with levels and hit points and such) to use within the hive system. Separate tutorials for that will be written! But for now, we will just use Spyder by example.

The first Spyder data model we will use is Spyder.AxisSystem. An AxisSystem describes an orientation and a position, in the same way as Panda's NodePath does. An AxisSystem contains four Spyder.Coordinate data models: these are called origin, x, y and z, describing the origin and axes of the coordinate system. You can set them directly: every Coordinate has three Spyder.Float attributes x,y and z. So, to scale an AxisSystem's z axis, you could say:

```
a = AxisSystem()
scale = 2
a.z.x *= scale
a.z.y *= scale
a.z.z *= scale
```

and also

```
a.z *= scale
```

AxisSystem also contains several utility functions like rotateZ to manipulate the axes.
Finally, dragonfly.scene.matrix can be constructed from an AxisSystem, and the pandascene has several add_XXX_SPYDER methods to deal with entities whose orientation is defined as a Spyder.AxisSystem.
In the following code, NodePath has been replaced by AxisSystem throughout the code:

<panda-7.py>

Nice, we have removed one dependency on Panda3D.
But of course, this is not high-level programming at all.
That comes in this code:

<panda-8.py>

What has happened here?
First, myscene does no longer derive from bee.frame, but from dragonfly.pandahive.spyderframe.
More importantly, all the configure bees are gone. All of the 3D information is provided in the form of Spyder objects.
We are no longer configuring by giving explicit commands, not even delayed commands. Instead, we are only declaring data.
Welcome to the high level.

The following sections will explain how the high level works.
First, let me explain how it does ***not*** work: there is no global inspector function that looks for Spyder object class attributes at run-time and then performs some action. The <panda-8.py> code is completely equivalent to the <panda-7.py> code. Even though the configure bees are gone from the original class definition, they are ***generated*** by the Spyder objects in the spyderframe hive. The only difference

between spyderframes/spyderhives and normal hives is that spyderhives recognize Spyder objects as special bee-generating objects.
Second, Spyder objects are not treated as dumb data. They are not passed around as arguments to some global configurebee-generating function. Instead, they are treated as bee-generating objects: a method make_bee() is called on them. The make_bee() method must return a bee, and that bee is placed into the hive. It's as simple as that.
Still, Spyder classes do not usually have a make_bee() method. In that case, the spyderhive must provide one for them. This is done by a bee called a SpyderMethod, which takes three arguments: the name of the method (usually make_bee), the name of the Spyder class, and a function that makes the bee.
Let's have a look in the source code of the pandahive's spyderframe (dragonfly/pandahive/spyderframe.py). As you can see at the end of the file, it inherits from the standard bee.spyderhive.spyderframe, and then adds SpyderMethods to it. In this manner, Spyder.Model3D, Spyder.Actor3D and other 3D Spyder classes are given a make_bee() method, so that they can be placed in the spyderframe. Looking at the actual functions that are registered as make_bee methods, you see that they are virtually the same as the code in <panda-7.py>.

Now we will do the same for the image and the mouse area:

<panda-8a.py>

We have replaced the init bees by a Spyder.Image object and a Spyder.MouseArea object. Now that you know how a spyderhive works, it should be easy to understand. Looking in dragonfly/pandahive/spyderframe.py, you see that a make_bee() method is defined for Spyder.Image and Spyder.MouseArea, generating the init bees necessary to define an icon.

However, the following change will probably be a surprise to you:

<panda-8b.py>

We have replaced the Image and MouseArea objects with a Spyder.Icon object, but there is no make_bee() SpyderMethod for Spyder.Icon! How can this be?
Spyder is not just for the construction for data models, but also for their conversion. Somewhere in the directories of the Spyder framework, it is written that a Spyder.Icon is equivalent to a Spyder.Image plus a Spyder.MouseArea, and how this conversion takes place (in case you want to know: Spyder/canvas/canvas.spy, at the end of the file).
In addition, Spyder has a special feature: X.spam() is equivalent to X.convert(Y).spam() [if X has no method spam, but Y has, and X can be converted into Y]. This is so for every Spyder object X, and it is always valid, not only within spyderhives. All that the spyderhive does is to assign a method make_bee() to Image and MouseArea, and then to call Icon.make_bee().
In this particular case, there is a converter from Icon into [Image, MouseArea]. Therefore, Icon.make_bee() results in [Image.make_bee(), MouseArea.make_bee()], which is a list of two init bees. The spyderhive then recognizes that a list of bees was returned. It wraps these bees into one big composite bee, which is then placed into the hive, and the icon is shown.

Let's put some more of this theory into practice. In the following custom spyderhive, we give Spyder.Coordinate a make_bee() method that displays a panda on the spot:

<panda-8c.py>

As you can see, a small panda will now show up at (5,-3,0). Since our spyderhive derives from pandahive.spyderframe, the other Spyder objects are displayed normally.

The Spyder framework contains many converters, but we can dynamically add new ones to our spyderhive, too. The following code displays two small pandas, but not by generating configure bees directly. Instead, a SpyderConverter is defined, that converts a Coordinate into a Model3D. Therefore, within the context of the spyderhive, Coordinate.make_bee() is evaluated as Coordinate.convert(Model3D).make_bee(), which displays the panda.

<panda-8d.py>

As you can see, you can completely customize 3D object creation from the high level, without any knowledge of the pandascene or configure bees.

More extensive conversions are also possible. Spyder contains a data model called Object3D, which contains faces, vertices and low-level 3D stuff like that. On top of that, there are many parametric objects (spheres, cylinders, tubes etc.) for which a converter to Object3D has been defined.
Panda3D allows for the definition of custom geometry, and pandahive.spyderframe contains a make_bee() method for Spyder.Object3D. This means that *automatically* every Spyder parametric object can be rendered in the pandahive, because they are convertible to Object3D. The same goes for *anything* that converts into parametric objects.
In the following code, two AxisSystems are displayed as cylinders that indicate the axes, with a panda at the end.

<panda-8e.py>

And here, the pandas are replaced by small white spheres:

<panda-8f.py>

Finally, it is possible to store Spyder objects in a file using the tofile() method. The following code does the same as the previous version, but in addition it stores the data in a so-called World3D object, which contains both the 3D objects and the color materials. The World3D object is stored in a Spyder file called "m.web".

<panda-8g.py>

You can open "m.web" in a text editor and have a look at it. You can say that it is a pretty-printed, very verbose call to World3D's__init__ function. Or you can call it an XML-like or JSON-like data format. Anyway, the same text is what you see if you print a Spyder object.

Finally, the following code does not define the AxisSystems or materials, but simply reads them back from "m.web", and displays them:

<panda-8h.py>

That is enough Spyder objects for now. We will come back to them later when we come to visual programming. For now, you can revert to <panda-8b.py>. We will continue with spawning more

pandas.

Note: If you want to play around with Spyder data models, you can do it as follows. Fire up the Python interpreter, then type "import dragonfly" and then "import Spyder" (even if you don't use dragonfly). Then, to get more info about a data model, call help() on it: for example, "Spyder.AxisSystem.help()". However, keep in mind that XArray.help() is not very useful, since XArray is always auto-generated from X: use X.help() instead.

Note 2: The Spyder framework gives you considerable freedom in constructing a Spyder object. A Spyder object can be parsed from non-keyword and keyword arguments, but also from another Spyder object, a dictionary, a list, or even a string. With nested data models, this freedom becomes much greater. For example. an AxisSystem, which contains four Coordinates, can be constructed as:

```
AxisSystem(a) #where a is an AxisSystem object
AxisSystem(c,c,c,c)  #where c is a Coordinate object
AxisSystem([c,c,c,c])
AxisSystem({"origin":c,"x":c,"y":c,"z":c})
AxisSystem((1,2,3),c,c,c)
AxisSystem((1,2,3),{x:1,y:2,z:3},c,c)
...
```

just to name a few.

The big advantage is that your object construction code will look incredibly expressive and beautiful, even if the corresponding Spyder data model is quite complex.
The downside is that you really shouldn't make mistakes. Spyder will try *each* of the constructors; if they all fail, *each* of their error messages will be reported. Also, since the error may occur at a member of a member of a member, the error message can be long indeed! Combine this with the multiple chained conversions that Spyder can do, and you will be in for a bit of troubleshooting.
So you have been warned: if you use Spyder a lot, your final code will be powerful and beautiful, but it may take you some time to get there. Power has a price, but the hive system gives you the choice.

Spawning more pandas

We will proceed with the pandas, adding more and more of them and also of different kinds. First, we have to refactor the code a bit:

<panda-9.py>

Now, with a bit of copy-pasting, we can add a second type of panda. This one is a smaller panda. It will have a smaller model and a smaller icon:

<panda-9a.py>

And here comes the third one. It is a panda standing upright, from a model that is also distributed with Panda3D. Unlike the other panda, it has no animations attached to it. Therefore, we declare it as an EntityClass3D instead of an ActorClass3D, and we spawn it with dragonfly.scene.spawn instead of spawn_actor:

<panda-9b.py>

Well, the code is simple, but it is long and ugly. We're basically copy-pasting, which is bad design: we should generalize the problem. But how can we define icons and workers for a whole bunch of pandas at once?

The answer is really quite obvious. Still, even if you are an experienced Python programmer, the following code is likely to pop your eyes in shock:

<panda-9c.py>

A *for loop and an if statement inside a class definition*?! Can you do that with Python? Yes you can! Unlike functions, code in a class statement is executed immediately. Also, class attributes are just local variables, and assigning local variables is equivalent to modifying the locals() dictionary:

```
a = 1
locals()["a"] = 1 #equivalent
```

Therefore, instead of doing

```
class pandascenehive(dragonfly.pandahive.spyderframe):
  icon = Spyder.Icon(...)
  icon2 = Spyder.Icon(...)
  icon3 = Spyder.Icon(...)
```

we can store all our parameters in a separate dictionary, and loop over it:

```
class pandascenehive(dragonfly.pandahive.spyderframe):
  for name in pandadict:
    locals()[name+"_icon"] = Spyder.Icon(...)
```

And, more importantly, we can pull off the same trick with workers:

<panda-9d.py>

Now we have the advantage that we store all our panda-related parameters in one dict, instead of all over the class. We can focus on populating the dict with additional pandas, setting their parameters, without worrying about implementation details: the hive has become a configurable black box.

Note: With some imagination, you may also start to guess how visual programming is achieved in the hive system. After all, workers are just bees, and with Spyder you can build bees out of anything. All made possible by this crazy programming language that allows for loops in class statements: Python.

Of course, the current code still has some limitations. The pandadict is an ugly global variable, because we cannot bake it into the hive (the for loop is evaluated too soon). Also, we define the data in one place, but it is still interpreted in two places (two for loops). These limitations can be removed by using combohives, which will be discussed in the very end of the tutorial.
Another limitation is more fundamental: we cannot use bee.attribute and bee.get_parameter anymore. If you want to modify the hive, modify the data, don't override the class attribute names.

Oh yeah, and the keyboard sensor is gone. Makes a nice exercise for the reader: add an additional field to the pandadict value tuple that indicates the keycode, and build a keyboard sensor for it. If it is None, don't add a keyboard sensor at all. With a language that allows if statements in a class definition, it's not so hard to do.

The next section will be a bit more 3D-oriented: we will finally move the panda from its spot. For now, one type of panda is quite enough, so you can revert again to <panda-8b.py>. We will come back to multiple pandas later.

Moving the panda around

In the following code, the panda finally moves from its spot:

<panda-10.py>

When you press the W key, the panda will slowly move forward to another spot. You can stop it in its tracks by pressing S. Once it reaches its destination, it stays walking on the spot forever, until you freeze it with the S key.

How is this done? A few new workers have been added to the scene. First, there is the setPos worker, which sets the position of an entity. It takes as push input a Spyder Coordinate (which can be built automatically from a 3-tuple). It is fed by an interpolation worker. This works as follows: when started, the interpolation worker pulls in at every frame a fraction variable (a value between 0 and 1). Using that fraction, it interpolates between the start and end coordinates (which were given as parameters), and pushes out the result (to the setPos worker). Finally, the fraction itself is computed by an interval_time worker. When started, the interval_time worker changes its value gradually from 0 to 1, within the interval (in seconds) that was supplied as parameter. This process can be paused by the S key.

In the following mode, the Panda will start moving back to its origin after 4 seconds, reaching it after another 4 seconds:

<panda-10a.py>

Moving back and forth is managed by two interpolation workers. But how are the fractions managed? The fraction sent to the first interpolation worker should increase from 0 to 1 in 4 seconds and then stay there. The second fraction should stay zero for 4 seconds, and then go from 0 to 1 in the final 4 seconds. This is managed by the sequence worker. It takes a fraction and it splits it into two fractions in the way described above.

The back and forth movement will just occur once. We can change it into a loop by restarting the sequence worker as soon as the second interval worker reaches its end:

<panda-10b.py>

Finally, we will make the panda turn. The sequence worker is configured for splitting into four outputs, of 8, 1, 8 and 1 second. The 1 second fractions are connected to an interpolation connected to a setHpr worker, which sets the orientation.

<panda-10c.py>

Note: the interval worker has no relation to the Panda3D interval class. The same goes for the sequence worker.

Hive binding

The next part discusses an important technique: hive binding. You can organize your workers into subhives and frames, and they will share or not share their sockets and plugins as you choose. However, hive binding gives you more flexibility, controlling exactly how sockets and plugins go into the bound hive. This is not a tutorial on hive binding, and it will only be shown by example. Still, it is useful to know that hive binding allows you to control when a hive receives ticks, what access it has to keyboard and mouse input, to exit functions, and what will be its sense of time. In other words, hive binding creates a sandbox. Finally, it allows you to bind an entity to a hive, and that is what the code below does:

<panda-11.py>

The code above statically binds a hive to the initial panda. We create a bind class "pandawalkbind", derived from bee.staticbind.staticbind_baseclass and a whole bunch of mixins. It has a class attribute "hive", which has the value "pandawalkhive". This pandawalkhive contains all the panda-walking code from above, except that all dragonfly.scene.unbound workers (animation, setPos and setZ) have now been replaced with the same workers from dragonfly.scene.bound. They do not receive the name of the actor or entity as a pull antenna, because this has been bound into the hive.
This actual binding is done in myhive, line 121-123. We create an instance, not of "pandawalkbind", but of "pandawalkbind().worker". This worker receives as input the name of the panda entity, on its pull antenna "bindname". No matter how you configure your static hive binder, the main antenna "bindname" is always there. Because we included entity binding (the dragonfly.scene.bind mixin), it expects to receive there an ID with the name of the entity.

Static hive binding is done at run-time, when the hive starts up. This has the advantage that you can make the hive binding process, including the creation of sockets and plugins, depend on init bees or even on run-time data, which you cannot do with subhives. On the other hand, if some plugins and sockets are mismatched in your hive, you will receive the error message only at startup. Since the hive system, Spyder and Panda3D each take a few seconds to start up, this may slow down your debugging cycle. Still, the ability to abstract away the name of the entity in your hive may be worth it. The choice is yours.

We have still one remnant left of the original Panda3D tutorial code: the spinning camera task. We will now implement it using hive binding, and without any of those nasty sines and cosines. We will do it step by step.

<panda-11a.py>

In the code above, the spinning has been removed. The camera position has been set in front of the panda using an init bee, and a white cube is rendered at the origin (half of it is underground). The cube is defined in myscene, and it's called "camcenter".

In the next version, the camcenter cube is spinning constantly:

<panda-11b.py>

This is achieved in the same way as with the pandawalk, except that the binder and its hive are called "camerabind" and "camerabindhive". The camerabindhive has a sequence of two intervals, with the rotation angle first going from 180 to 360 and then from 0 to 180, with the pitch fixed at -20. The fact that the cube starts spinning immediately is achieved with a startsensor, and the sequence is looped in the same way as the walking panda.

However, although the cube is spinning, the camera is still fixed in front of the panda. Fixing this is easy: parenting the camera to the camcenter! dragonfly.scene.unbound has a parent worker, which takes an entityname and a parent's entityname, and parents the first entity to the second. We connect it to a start sensor, tweak the initial camera position a bit, and then we are finished:

<panda-11c.py>

Well, except for the ugly white cube, which keeps facing us like a billboard. We will just use a simple hide worker to hide it at startup:

<panda-11d.py>

All functionality is now completely contained within the hive.

Dynamic hive binding

Up to now, there has always been an initial panda, and a class of pandas that could be spawned by Z or by clicking on the icon. The initial panda could walk, but pandas from the class would just stand there forever.
We will now remove that limitation. We will make all spawned pandas able to walk with W and S, just as we have done before with one panda.
How do we achieve that? We have already bound a pandawalk hive statically, to a single initial panda entity. What we have to do is to create new copies of the pandawalk hive at run time, and to bind them to a new panda entity whenever one is spawned. In other words, we have to do dynamic hive binding. This is achieved in the following code:

<panda-12.py>

Let's go through the changes one by one.
First, the initial panda is gone. It has disappeared from myscene, and there are no more pandaname attributes in the main hive. Likewise, there is no more static binding to it.
The binder "pandawalkbind" is still there. However, the staticbind_baseclass is gone, now it completely consists of mixins. This makes it a dynamic binder: in the hive system, all binders are dynamic unless you add the staticbind_baseclass in front.

The next change is to the panda ID generator. A generator changes its value whenever you pull it. However, if you want to use this value in more than one place (spawning and binding), you have to buffer it. In the new code, the generator has changed names from "panda_id" to "panda_id_gen", and "panda_id" is now the buffering variable. Refreshing the buffer with a newly generated ID is done by

triggering the t_panda_id_gen transistor.

Again, we instantiate a worker from the pandawalkbind binder. However, unlike a static bind worker, a dynamic bind worker doesn't have a pull antenna "bindname". Instead, it has a push antenna "bind": whenever an entity ID is pushed into "bind", a new hive instance is generated and the pushed entity ID is bound into the hive. From then on, the hive instance is known under the same name as the entity: spawnedpanda1, spawnedpanda2, ...

Finally, we create a connector "trig_spawn". Whenever this connector is triggered, it generates a new entity ID, then spawns the entity, and finally binds the entity to a new pandawalk hive. The connector is triggered by both the Z key and the panda icon.

Now, let's see the new hive at work. Create a few pandas, and then press W to make it walk.

The result isn't quite what we want. As soon as we press W, our herd of pandas changes into a single, walking, many-headed panda-monster. After a few seconds, the monster changes into a single panda turning around and moving back. Why?
When we created the panda's, they all had a random orientation and position. However, when we pressed the W key, *all* panda positions were set to (0,0,0), slowly changing to (0,-10,0). The initial orientations were preserved, that's why the monster had many heads. After 8 seconds, the setHpr interpolation gets activated and then all the pandas have the same orientation, too, showing up as a single panda.
The problem is that we are making changes in terms of the *global* coordinate frame: (0,0,0) means in the middle of the grass. What we want here is to work in the *local* coordinate frame: (0,0,0) should mean the panda's initial position, and moving in the Y direction should move the panda forward, not sideways in the global Y direction.
We can achieve this by adding a single line to the binder class: *bind_entity = "local"*

<panda-12a.py>

Now, pressing W will cause all pandas to walk, and S will cause them all to stop.

Let's avoid a possible misunderstanding here. Panda3D contains an example about line dancers, where triggering the animation causes all of them to move. This is because those line dancers were really instances of one single actor, with one single animation state. This is not so in the hive system (you *can* do it also in the hive system, but you would use a Spyder MultiInstance3D object). In our hive, our pandas and their hives are really independent and don't share any state. The reason that all of the pandas started to walk, is that *all* of their hives received a keyboard event. By controlling the keyboard events that go into the hives, you can control which panda(s) respond to the W key.

For example, try and execute the following code:

<panda-12b.py>

You can create a panda, press W, and nothing happens. Second panda, same thing. However, after creating the third panda, W and S will cause it to walk and stop. The third panda is the one that responds, no matter how many pandas are created afterwards.

Looking at the code, you can see that another line is added to the pandawalkbind class: *bind_keyboard*

= *"indirect"*. This means that the hive in principle has keyboard support, so that keyboard sensors are allowed, but that keyboard events are not directly received from the parent hive: they must be sent to the hive's evin.
Every dynamic bind worker has a mechanism to dispatch events to bound hives. Event tuples are received on the push antenna "event". The event is then split into a head (the first field) and the tail (the second field). The head must be an identifier, and the tail must be another event. If the head corresponds to the identifier of one of the bound hives, the tail event is sent to the evin of the bound hive.
So, what happens in the main hive, is that all keyboard events are detected by the keyboardevents worker. All keyboard events are of the form ("keyboard", <key>), so looking for a leading element "keyboard" will capture them all. Then, a head is added, and its value is "spawnedpanda3": the identifier of the third panda that is spawned. Finally, the new event is sent to pandawalk.event, causing the keyboard event to be dispatched to the "spawnedpanda3" hive, if it exists.

In the following code, it is always the last created panda that receives keyboard input:

<panda-12c.py>

Instead of a separate "head" event variable, there is now a "selected" ID variable. This variable is receives input from t_bind, so it contains the last panda that was created.

We cannot connect "selected" directly to "add_head", because the types do not match: ID versus event. However, an ID is also a valid event, and can be used directly by the add_head worker. For this, the hive system has a duck converter. This takes in an object of one type and re-declares exactly the same object as a different type. By duck-connecting the "selected" variable to the add_head worker, we are feeding the keys to a single, selected hive. (If only the *value* of the object would be compatible, you would use the cast converter, for example to convert push-int value 4 to push-float value 4.0).

We will move on and make it possible to change the selected panda using the TAB and BACKSPACE keys. For that, we will use the selector worker. This worker is quite simple: it has a register_and_select push antenna, which takes an identifier and does just that. Then, there is the ID pull output "selected", and the trigger antennas "select_next" and "select_prev". Those last two we connect to the TAB and BACKSPACE key sensors. The register_and_select antenna receive the identifier of the hive/entity from t_bind.
Finally, there is now a display worker "disp_sel", that displays the selected identifier, whenever it is pushed into disp_sel by t_get_selected. Here is the code:

<panda-12d.py>

Note: In Panda3D, it is possible to pick 3D objects by the mouse. Using a pick sensor worker, you could select any panda you want, instead of just the next or previous one. However, I didn't implement such a sensor yet. You can try to do it yourself, if you want.

Now let's add something to get rid of too many pandas. In the following code, pressing the K key will kill the selected panda:

<panda-12e.py>

Killing the panda consists of six parts. First, the current selected identifier is retrieved using the

transistor t_kill. Second, this ID is sent to pandaworker.stop, which terminates the corresponding pandawalk hive. Third, the panda entity is removed from the scene. Fourth, the identifier is sent to a display sensor that says "Killed:". Fifth, the identifier is removed from the selector. Finally, the selection is updated.

There are still a few bugs and limitations in the code. First, pressing TAB or BACKSPACE while there are no pandas, or killing the last panda, will crash the program. That can be fixed by adapting the t_get_selected transistor: only if the selector is not empty, the "selected" variable is updated from the value held by the selector:

<panda-12f.py>

In addition, pressing K while there are no pandas will also crash the program. Again, this can be prevented by only executing the kill if the selector is non-empty:

<panda-12g.py>

Finally, it would be nice to have a visual marker that indicates which panda is selected. For example, a blue circle on the ground:

<panda-12h.py>

Now, we can let this circle be a marker by adding the following workers:
- A variable v_marker that holds the entity name of the marker
- hide_marker, which is triggered upon startup and by the kill connector
- show_marker, which is triggered whenever the selection is updated
- parent_marker, which parents the marker to the selected panda entity

Here is the final code:

<panda-12i.py>

Hopefully you have now an idea of what hive binding can do for you. As I said, this is not a tutorial about hive binding, and many options have not been mentioned, but you should realize that you have total control. If you feel like it, you can make some weird binder that remaps the W key to the S key or vice versa, or where time runs backward. You can do whatever you want. So can everybody else, by deriving from your hive or by hivebinding your hive inside their own, changing anything they like. The hive system gives you the choice.

Hivemaps

We will continue to our path towards the high levels of the hive system. Now that we can spawn and bind one kind of hive, we will now learn how to spawn arbitrary hives. At the end of this section, we will build those spawned hives by visual programming.

First, we have to refactor the final code of the last section:

<panda-13.py>

In the refactored code, the pandawalk worker has been renamed to pandabinder. More importantly, in the pandawalkbind class, the attribute "hive" is gone: we are no longer hardcoding the hive "pandawalkhive" into the binder. Instead, another mixin "dragonfly.bind.bind" has been added to the binder class.
In the main hive, the t_bind transistor, which carries the spawned panda's ID, no longer connects to the pandabinder worker. Instead, it connects to a weaver. A weaver is actually a bee.segment, one of the less common ones: it takes multiple pull inputs <type1>, <type2>, ..., and wraps them into a single pull output (<type1>,<type2>, ...). As you can see, it is also available as a (meta-)worker. This is because pandabinder.bind no longer accepts the single ID provided by t_bind: because of the changes we made, it requires now a tuple of two IDs, with the second one being the ID of the hive (more precisely: ID of the *hive type*). We have also added a configurable drone that can register hives under an ID: it is called the hiveregister drone. Finally, a configure bee is added that registers our pandawalkhive under the ID "pandawalk". The v_hivename variable holds this ID, and it is connected to the weaver, so that the binding process goes correctly.

Here is another bit of refactoring. The pandaclass class attribute has gone, and so has the keyboard sensor. We have moved all code that defines the panda logic down in the file. The v_panda variable contains no longer the pandaclass value, and v_hivename is also empty. Instead, clicking the icon now sets the v_panda and v_hivename variables to "pandaclass" and "pandawalk", respectively:

<panda-13a.py>

We can extend this code easily towards additional pandas. In the following code, the second, smaller panda and its icon have been added again:

<panda-13b.py>

Most of this new code is nothing special: just copy-paste, taking the parameters from earlier sections. However, as you can see, a different hive is bound to the small panda: the original pandawalkhive is re-wired, so that the panda walks on E instead of W.

However, the following change in the code is much more radical:

<panda-13c.py>

70 lines are gone. Still, the functionality is the same: the hives are now read from an external file. Not a Python file, but a *data file*. No programming syntax at all. Just a complete Spyder description of workers, connections and parameters.

This is the second time that this trick is pulled off. First at the end of the chess tutorial, and now here. This time, however, the magic behind the trick will be revealed.

A data description of a hive is called a hivemap. A Spyder hivemap file contains a single Spyder.Hivemap object, defined in hivemap/hivemap.spy. A Hivemap contains two attributes, "workers" and "connections".
"workers" are an array of Spyder.Worker objects. Every worker has a unique ID, and a string that describes which type it is (e.g. "dragonfly.scene.unbound.setZ"). A worker also contain two lists of (name,value) pairs: the parameters and the metaparameters. For normal workers, the list of metaparameters is always empty. Metaparameters have a non-empty metaparameter list, and the

parameter list may be empty or not. For example, dragonfly.std.transistor("str")() will have a metaparameterlist [("type","str")] and an empty parameter list. dragonfly.std.transistor("int")(10) will have metaparameterlist[("type","int")] and parameterlist [("value",10)].

"connections" are an array of WorkerConnection objects, which are quite simple: a start connector, consisting of the identifier of the worker and the name of the output; an end connector, with the identifier of the other worker and the name of its antenna.

The trick is really simple then. The hivemaphive, which translates a hivemap into a hive, is defined in bee/spyderhive/hivemaphive, in less than 50 lines.
A hivemaphive is a special kind of spyderhive, where Spyder.Hivemap is given a make_bee() method. The make_bee() method first imports the hivemap's workers: a worker of type "dragonfly.scene.bound.setZ" will lead to the import of dragonfly, then of dragonfly.scene, etc., and finally of the worker itself.
Then, a new hive is constructed, with two for loops in it: one for the workers, and one for the connections. Workers are built normally, using the (meta)parameters, and so are the connections. The make_bee() method simply returns an instance of the constructed hive.

Take your time to appreciate the power of this. The hive system is your perfect platform for domain-specific modeling or scripting. The workers in this tutorial are fairly direct in manipulating 3D coordinates, but with workers, you can easily generate bindings to any Python code. Expose your RPG engine to Python, and suddenly you have high-level components like "attack nearest enemy" or "cast fireball". Also, you are not stuck with Spyder.Hivemap as the data format of your hives. You can write code to create bees and hives from *anything*, including from a string like "if health < 10 then run away". The whole environment is under your control, and it is automatically editable in a GUI. Finally, all code is data, so a security checker can validate user-generated content, allowing only the bees that you permit on your server.

Hivegui

I have created a crude and simple GUI to edit hivemaps. It is in the hivegui/ directory, and requires wxPython to run. It uses Chris Barker's FloatCanvas. It is just a proof-of-principle GUI: I am sure that a programmer with some talent for visual design could build a better one. However, the current GUI should be sufficient for basic usage. In the following section, I will just explain how it works.

You can open hivegui.py with Python. On a command-line, you can provide a hivemap as command line argument.
In the Create menu you see a very long list of workers. You will fruitlessly search for a configuration file containing this list: all bees are discovered at run time. There is a little file called "locations.conf" in the hivegui directory. In addition, when you open a hivemap file, the hivegui will look for a "hivegui.conf" in the directory of the hivemap. If one exists, it will append it to "locations.conf", and re-discover the bees.

When a locations.conf/hivegui.conf file is read in, all lines starting with @ are added to sys.path, and then the module names on all other lines are imported. Directories in the imported modules are further imported.
All imported modules are searched for objects that have a "guiparams" or "metaguiparams" dictionary. Objects with "metaguiparams" are considered meta-workers, and objects with just "guiparams" are considered as workers. For workers, a "guiparams" dict is automatically created by bee.worker.

Metaworker objects have to define their own "metaguiparams" in order to work properly, see dragonfly.std.variable for an example.
Worker-like hives also have their guiparams dict: this means they too are recognized as workers. You can even "fake" a (meta-)worker by defining a custom object with a guiparams or metaguiparams dict, and hivegui will recognize it.

The guiparams/metaguiparams dicts are further interpreted into a set of editable parameters. A guiparams dict must always contain two keys "antennas" and "outputs": they contain the names, modes and data types of the antennas and outputs. Again, all of this is taken care of automatically by bee.worker. The remaining keys are (mostly) for the initialization parameters, these are generated by bee.worker based on the worker's bee.segment.parameter segments. The hivegui translates these parameters into a GUI with editable fields.
After creating a worker, you can click on the worker and then you can edit these fields.

Parameters for metaworkers always consist of fields and their names: creating a metaworker will pop up a window where you are asked to enter them immediately. The metaworker object is actually called with these parameters, and the returned worker object is then inspected for its guiparams dict, and built.

The hivemap data model contains some simple layout properties. Every worker contains a Coordinate2D, describing the position of the worker on a 2D canvas. Every connection has a list of 2D coordinates that describe the shape of the curve between them. These properties are editable in the hivegui.

The basic interface of hivegui is quite simple. Antennas are black, outputs are red. Antennas are on the left, outputs on the right. In case of many antennas or outputs, some will be at the bottom. Event streams are on top.
Push antennas and outputs are solid lines, pull antennas and outputs are dashed. You can move over an an antenna or output and see its type in the left bottom. You can click on a worker to select it, double-click to edit its parameters. If you want to change metaparameters, sorry: delete and re-create the worker. A selected worker can be deleted. By clicking on an antenna or output of a selected worker, you can connect it to something else. Workers are connected by curves.
You can also select connections. For a selected connection, you can double-click anywhere to add a point to the curve. These points appear as purple dots: double-clicking on them will delete them.

Finally, you can save your hivemap as a Spyder.Hivemap file, which can be directly executed when placed in a hivemaphive.

Making a panda jump

The following section is about letting a panda jump. With all the features introduced in the previous tutorials and the current tutorial up to here, it isn't hard to do. Therefore, "making a panda jump" will be treated as a kind of model problem: I will show you several strategies of accomplishing it, so that you can choose for yourself which solution is the most appropriate for your own problem.

First, let's add a class of pandas standing upright, as we have already done before. Upon spawning, they will be bound to a pandajumphive, which is initially empty:

<panda-14.py>

Now, we will make the panda jump on pressing the SPACE bar. The panda will jump in a frog-like manner. That is, the upward speed of the panda will go from zero to maximum in a single thrust, in zero time, and afterwards it's just gravity doing its work.
For such a frog jump, the laws of physics will give you the following formulas:

average speed = 4 * maximum_height * (1 - progress)
height = average_speed * progress
(where progress goes from 0 to 1, with the panda jumping at 0 and landing at 1)

Here comes the first way of implementing a jump: one big jumpworker.

<jumpworker.py>
<panda-14a.py>

The jumpworker does everything by itself, with minimal interaction with the rest of the hive system. It just hooks itself into the hive, using the hive system's functions to register listeners and to receive parameters. Once activated, it starts the timer, computes the height on every tick, and sets the entity's Z position. Once the panda has landed again, it unregisters itself with the hive system. The hive system's only responsibility is to detect the Z key press and to start the worker.

Personally, I think that for the current problem, this worker is too heavy. Still, there may be cases where it is the way to go. The most obvious case would be when the physics is already taken care of by some external engine, which expects a link to a 3D matrix of some sort. In that case, you would replace the formula part by a call to the physics engine.
Even then, it might be better to bind to that engine using a drone and to create a lighter worker that simply interfaces with the drone through some socket ("physics", <whatever>). If you then want to swap physics engines, you just replace the drone and the hive will still work.
Another case where you may want to go for heavy workers like this is when you are setting up a sandbox for people who are not very good at programming, and who prefer one big worker over splitting the problem into flexible sub-parts. But then, you may also consider the third solution below.

The second solution balances the load between Python and the hive system. A custom worker (jumpworker2) is created, but it is very simple: it has a pull output that will contain the height, and a pull antenna for the progress. Whenever the output is requested, the worker pulls in the progress and computes the height, using the formulas above. The worker has no side effects. Detecting the key press, making the progress go from 0 to 1, pulling the height, and setting the Z position is all delegated to the hive.

<jumpworker2.py>
<panda-14b.py>

The third solution is a combination of the previous two. It uses the simple jumpworker2 of the second solution, but it is embedded in a worker-like hive (jumpworkerhive) that behaves exactly the same as the big worker of the first solution: just connect to the start antenna.

<panda-14c.py>

Here is another variant, where the height and duration have been parameterized. The usage of the worker-like hive is now completely identical to the big worker of the first solution.

<panda-14d.py>

Personally, I think that this third solution is the one that is best-suited to the problem. It combines the vastly greater expressive power of Python when it comes to formulas, with the flexibility and clean result of connecting the components in a hive. It offers access at two different levels: the jumpworkerhive can be embedded as a worker for the standard case, and for more customized use, the jumpworker2 can be used directly.

However, it is also possible to implement the jump completely in the hive system, without creating a custom worker in Python:

<panda-14e.py>

This fourth solution is a pure visual programming solution, even though it written as text. Perhaps you would choose this solution if your environment does not allow custom workers, or if you absolutely hate formulas. Personally, I think that this particular problem is better represented by a formula in Python, but for other problems this may not be the case.
On the other hand, you may also consider the next solution of specifying your code as data. However, by representing the hive as source code, instead of hivemap data, you keep all the flexibility of object-oriented programming. For example, you can easily add some bees that move the panda forward for as long as it is in the air:

<panda-14f.py>

Finally, there is the ultimate solution of representing everything as a hivemap:

<pandajump.web>
<panda-14g.py>

The implementation is exactly the same as the visual programming solution above, except that it is, well, visual. The obvious advantage is that you don't have to be a programmer to create such a hivemap. For small problems such as this one, it can be a great and beautiful solution (although, to make it beautiful, a much better GUI needs to be developed than the one I cooked up). But beware: bigger hivemaps can quickly turn into complicated spaghetti.
A lot also depends on the quality of the bees: with the jumpworker2 available to your GUI, your hivemap will be much smaller and cleaner than the one above. It would also be possible for you to create a generic "formula worker" where you can enter some formula as text, within a visual programming environment. The tools and options are there, now it is up to you.

It is well known that visual programming languages scale up very poorly. Modularization and decoupling is important in any programming context, but far more so for visual languages. There is the famous so-called Deutsch limit, which says that a visual program can consist of no more than 50 visual elements on-screen before it becomes unreadable.

The jumping panda section is meant to demonstrate what techniques are available to you within the hive system to prevent this from happening. First of all, you can shift small or large parts of the burden to Python by writing custom workers and drones. You can use worker-like hives and similar kinds of

nesting and modularization, and you can use hive inheritance to decouple features. Finally, you can use hive binding to create multi-instance sandboxes that have well-defined interactions with the environment. If you feel that the task has been fragmented into small enough pieces, you can use (or let others use) visual programming to tackle those pieces individually. Or not. The hive system gives you the choice.

Combohives

We are approaching the very end of this tutorial, and of the manual as a whole. Throughout the text, a bottom-up approach has been followed: first explaining the low level, and then how higher levels are built on top of them. A top-down approach would also have been possible, but it would have been much harder to explain to programmers. The hive system is special enough as it is.

We will now make our final turn on our climbing way up to the highest level. In the following section, we will define a whole program, data and code, in a single place. To do so, we will use a powerful but somewhat scary feature: combohives.

One of the rules of programming is that you should separate the concerns. Game logic should be separated from 3D data. Physics should be separated from rendering. For an RPG or FPS game, monsters stats such as hitpoints should be implemented in a different module than the appearance of monsters on screen: in particular for an online game, where the hitpoints and AI are taken care of by the server while the appearance is the sole responsibility of the client.

However, from a data perspective, this makes no sense at all. It's not natural to specify a list of monster stats, a second list of monster AI settings and a third list of monster models. Instead, you want to specify a single list of monster class objects, with each object having properties for the 3D model, animations, physics, monster stats, AI, and whatever else.

With combohives, you can have both. You can build a hive where the concerns have been properly separated, and on top of that, a configuration layer where everything can be defined in one place. However, combohives really change the hive properties towards the data-centered perspective: while your hive has become very elegant to configure, it becomes harder to extend and its mechanics much more difficult to grasp. When I came back from a long break in developing the hive system, I could still immediately understand all of my test examples, except for the combohives. So, you have been warned. The hive system gives you the choice.

We achieved already a point where all data was provided in a single dict: <panda-9d.py>. The following is an updated version of that code, with the hive binding of hivemaps of the previous sections added:

<panda-15.py>
The limitations of such a design where already discussed: an ugly global variable, multiple for loops over the same dict, and we cannot bake it into the hive. Now, we will fix those limitations step by step.

First, let's do some refactoring. The code above creates a single hive of panda actorclasses and icons, with a for loop in it to declare multiple models. Now, we will move the for loop to the main myscene hive: for every panda, we will create a separate hive (a pandasceneframe), with a single actorclass and icon in it.

<panda-15a.py>

Next, we will do the same for the panda logic (the spawning, registration and hive binding). For every panda, we load one hivemap, we register it, and we build one pandalogicframe to build it. All pandalogicframes are together in a single pandalogichive, which contains the for loop and some connections to the outside world.

<panda-15b.py>

Next, we will split the code into a constant part and a configuration part:

<pandalib.py>
<panda-15c.py>

The constant part (pandalib) contains a hive with only the basic machinery for spawning, marking, selecting and killing pandas. It contains a 3D myscene with just the grass, the camera center and the marker; and a camerabind binder that has no hive in it. Finally, there is a pandalogichive that is empty except for its connections to the outside world. Neither the myscene nor the camerabind nor the pandalogichive has yet been added to the hive (although some attribute references assume that they will be present in the final hive).

The configuration part consists of the global pandadict. In addition, it overrides myscene (as myscene2), adding pandasceneframes in the same way as in the previous version. The same goes for the pandalogichive (pandalogichive2). Finally, it contains a main hive that derives from the pandalib main hive and adds the new myscene2, the pandalogichive2 and its connections, and binds the camera.

Now, the constant part is fine. It's the configuration part that needs to be adapted. Let's have a different look at the problem.
We start with a myscene2 hive that is empty, and a pandalogichive2 that is empty. For every panda in the pandadict, we want to add an icon and a model to the pandascene2 hive, and a pandalogicframe to the pandalogichive2. We want to do this placement in one place.
So, what we need is a special super-hive that can target other hives, placing bees and Spyder objects into them. That is exactly what a combohive does.

Combohives cannot contain normal bees, such as drones, workers, init bees, configure bees or subhives. Instead, they can only contain so-called combodrones, which are drones that have a make_combo() method instead of a place() method; and they can also contain Spyder objects that have a make_combo() method. The make_combo() method should return a bee dict: the value of a dict entry should be a bee or a list of unnamed bees (drones, workers, configure/init bees, Spyder objects, hives, all are allowed). The key of the bee dict entry should be the name of a hive that was provided to the combobee as keyword argument (injection).

In the following code, the pandasceneframes are placed into myscene2 by a combohive:

<panda-15d.py>

The combo hive generates a bee dict with a single entry: the key is "myscene", the name under which the myscene2 hive is injected into the combohive, and the value is a list of pandasceneframes. The bee dict is wrapped into a combodrone (with a method make_combo() that returns the bee dict), using

bee.combodronewrapper.

In the following, the combodrone is expanded to include the game logic:

<panda-15e.py>

Here you see that the pandalogichive2 also receives a list of bees: the first one is a hiveregister configure bee, and for every panda it receives one pandalogicframe and two connect bees.

It is now required that pandalogichive2 is injected into the combohive as "pandalogic", and also that the hiveregister drone is injected into the pandalogichive2 as "hivereg".

In the following code, combohives are put to full use:

<panda-15f.py>

Here, the camera hive is also added to the camerabind binder from the combohive. This is done using a Spyder.Combobee, but a combodronewrapper could be used as well.

More importantly, we finally got rid of pandadict as a global variable. The pandadict is still constructed as before, but only as a temporary variable. It is baked into the hive as a class attribute, and received as a parameter by a new combodrone called pandadictdrone. This combodrone generates the same bee dict as above, but now it is constructed and returned by a make_combo method.
This means that the value of the pandadict does not need to be known at hive definition time. You can now create a hive that derives from mycombohive where the pandadict is re-defined, or has a different name, and the hive will work perfectly.
We have finally created a fully configurable 3D program within a single class.

The final code
<panda-finalweb.py>
<panda-final.web>
<panda-final.py>
<pandalib.py>

Here it is then: the final code, the same that is used in the screencast.
The main difference from <panda-15f.py> is that the pandadict is no longer constructed in code, but read from a Spyder file.
I have created a custom Spyder data model called MovingPanda, It is defined in movingpandademo/movingpanda.spy in the Spyder directory.
The pandadict-constructing code has been moved to <panda-finalweb.py>, and in the final step, an Array of MovingPanda objects is constructed out of the pandadict, and written to file as <panda-final.web>. Of course, you can now also edit this web file directly in a text editor. Or you could design an editor GUI for it, like the hivegui is for Spyder.Hivemap.
In the main code, a combohive is defined, adding a method make_combo() to MovingPanda. The implementation of this method, the load_panda() function, is basically identical to the pandadictdrone above, except that builds only a single panda class.

Finally, a MovingPandaArray is read in from panda-final.web and placed into the hive. The rest works by magic.