FoxDot Documentation

Release 0.5.9

Ryan Kirkbride

Jun 03, 2021

Contents:

FoxDot is a Python library and programming environment that provides a fast and user-friendly abstraction to the powerful audio-engine, SuperCollider. It comes with its own IDE, which means it can be used straight out of the box; all you need is Python and SuperCollider and you're ready to go!

For more information on installation, check out the *Installation Guide*, or if you're already set up, you can also find a useful *Getting Started* guide that introduces the key components of FoxDot.

If you still need help, check out the FAQ, or you can drop by the `#foxdot` channel on slack, drop by the discussion board, or file a bug in the issue tracker. Please let us know where you think this documentation can be improved.

CHAPTER 1

Contents

1.1 Guides

This section contains a couple of walkthroughs that will help you get familiar with FoxDot. If you're new to Foxdot, you'll want to begin with the *Getting Started* guide.

1.1.1 Installation Guide

Downloads

- Python (version 2 and 3 are supported)
- SuperCollider 3.8 and above
- sc3 plugins (Some cool extra features for SuperCollider – recommended but not required)

Installing

Follow the installation instructions for your downloads of Python and SuperCollider. When installing Python, click **yes** when asked if you want to add Python to your system path and **yes** if you want to install pip – this is used for automatically downloading/installing Python libraries such as FoxDot.

Install the latest version of FoxDot from the Python Package Index using pip from your command line (command prompt in Windows, terminal in MacOS and Linux) by executing:

```
pip install FoxDot
```

Alternatively, you can build from the source on GitHub and keep up to date with the development version:

```
git clone https://github.com/Qirky/FoxDot.git
cd FoxDot
python setup.py install
```

Open SuperCollder and install the FoxDot Quark (this allows FoxDot to communicate with SuperCollider – requires Git to be installed on your machine) by entering the following in the editor and pressing Ctrl+Return which "runs" a line of code:

```
Quarks.install("FoxDot")
```

Recompile the SuperCollider class library by going to Language -> Recompile Class Library or pressing Ctrl+Shift+L.

If you don't have Git installed, you can download an install the necessary SuperCollider Quarks directly from GitHub by running the 2 lines of code below in SuperCollider:

```
Quarks.install("https://github.com/Qirky/FoxDotQuark.git")
Quarks.install("https://github.com/supercollider-quarks/BatLib.git")
```

Startup

Open SuperCollider and evaluate the following (this needs to be done before opening FoxDot):

```
FoxDot.start
```

SuperCollider is now listening for messages from FoxDot. To start FoxDot from the command line just type:

```
python -m FoxDot
```

The FoxDot interface should open up and you're ready to start jamming! Check out the *Getting Started* guide for some useful tips on getting to know the basics of FoxDot. Happy coding!

Installing SC3 Plugins

The SC3 Plugins are a collections of classes that extend the already massive SuperCollider library. Some of these are used for certain "effects" in FoxDot (such as bitcrush) and will give you an error in SuperCollider if you try to use them without installing the plugins. The SC3 Plugins can be downloaded from here. Once downloaded place the folder into your SuperCollider "Extensions" folder and then restart SuperCollider. To find the location of the "Extensions" folder, open SuperCollider and evaluate the following line of code:

```
Platform.userExtensionDir
```

This will display the location of the "Extensions" folder in the SuperCollider "post window", usually on the right hand side of the screen. If this directory doesn't exist, just create it and put the SC3 plugins in there and restart SuperCollider. When you next open FoxDot, go to the "Language" drop-down menu and tick "Use SC3 Plugins". Restart FoxDot and you're all set!

Installing with SuperCollider 3.7 or earlier

If you are having trouble installing the FoxDot Quark in SuperCollider, it's usually because the version of SuperCollider you are installing doesn't have the functionality for installing Quarks or it doesn't work properly. If this is the case, you can download the contents of the following SuperCollider script: foxdot.scd. Once downloaded, open the file in SuperCollider and press Ctrl+Return to run it. This will make SuperCollider start listening for messages from FoxDot.

Installing on Linux

Much of the installation (including Python & SuperCollider) has been automated into a simple shell script written by Noisk8. You can download the Linux Install script from their GitHub which contains some information on what the script is doing and how to run it. (note: it is in Spanish but modern web browsers will translate that for you)

For more answers to other frequently asked questions, check out the FAQ post on the discussion forum.

Please report any issues or bugs on the project's GitHub or if you have any questions, feel free to leave a message on the discussion forum.

1.1.2 Getting Started

Also available in Spanish.

Python is an object-oriented programming language that focusses on flexibility and readability. It also contains a large library of functions and serves a large user base. So it's about time we were about to live code music with it.

If you're ever stuck, or want to know more about a function or class – just type `help` followed by the name of that Python object in brackets:

```
help(object)
```

FoxDot provides a Python interface to SuperCollider – mainly as a quick and easy to use abstraction for SuperCollider classes, Pbind and SynthDef. Please read the SuperCollider documentation if you'd like to know more.

A SynthDef is essentially your digital instrument and FoxDot creates players that use these instruments with your guidance. To execute code in FoxDot, make sure your text cursor is in the 'block' of code (sections of text not separated by blank lines) and press `Ctrl+Return`. The output of any executed code is displayed in the console in the bottom half of the window.

Try `print(2+2)` *and see what you get.*

Now try something that spans multiple lines:

```
for n in range(10):
    sq = n*n
    print(n, sq)
```

1. Player Objects

It is, in fact, possible to create SuperCollider SynthDefs using FoxDot, but that's outside of the scope of this starter guide. To have a look at the existing (but quite small, unfortunately) library of FoxDot SynthDefs, just execute:

```
print(SynthDefs)
```

Choose one and create a FoxDot player object using the double arrow syntax like in the example below. If my chosen SynthDef was "pluck" then I could create an object "p1":

```
p1 >> pluck()
```

To stop an individual player object, simply execute `p1.stop()`. To stop all player objects, you can press `Ctrl+.`, which is a shortcut for the command `Clock.clear()`.

The >> in Python is usually reserved for a type of operation, like + or -, but it is not the case in FoxDot, and the reason will become clear shortly. If you now give your player object some arguments, you can change the notes being played

back. The first argument, the note degree, doesn't need explicit naming, but you'll need to specify whatever else you want to change – such as note durations or amplitudes.

```
p1 >> pluck([0,2,4], dur=[1,1/2,1/2], amp=[1,3/4,3/4])
```

These keyword arguments relate to the corresponding SynthDef ("pluck" in this case) but with a few exceptions:

- degree
- dur
- scale
- oct

These are used by FoxDot to calculate the frequencies of notes to be played and when to play them. You can view the SuperCollider code (although not easy to read) by executing `print(pluck)` for example. You can group together sounds by putting multiple values within an argument in round brackets like so:

```
b1 >> bass([(0,9),(3,7)], dur=4, pan=(-1,1))
```

Notice how you don't need to put single values in square brackets? FoxDot takes care of that for you. You can even have a Player Object follow another!

```
b1 >> bass([0,2,3,4], dur=4)
p1 >> pluck(dur=1/2).follow(b) + (0,2,4) # This adds a triad to the bass notes
```

Sound samples can also be played back using the Sample Player object, which is created like so:

```
d1 >> play("x-o-")
```

Each character refers to a different audio file. To play samples simultaneously, simply create a new player object:

```
d1 >> play("xxox")
hh >> play("---(-=)", pan=0.5)
```

Characters in round brackets are alternated in each loop (know as lacing) such that the above player, `hh`, would be writtern literally as `hh >> play('--------=')` which save you a lot of typing! Putting characters in square brackets will play them twice as fast and can be put in round brackets as if they were one character themselves. Try it out:

```
d1 >> play("x[--]o(=[-o])")
```

2. Patterns

Player Objects use Python lists, known more commonly as arrays in other languages, to sequence themselves. You've already used these previously, but they aren't exactly flexible for manipulation. For example, try multiplying a list by two like so:

```
print([1, 2, 3] * 2)
```

Is the result what you expected? If you want to manipulate the internal values in Python you have to use a for loop:

```
l = []
for i in [1, 2, 3]:
    l.append(i * 2)
print(l)
```

or list comprehension:

```
print([i*2 for i in [1,2,3]])
```

But what if you want to multiply the values in a list by 2 and 3 in an alternating way? That requires quite a lot of messing around actually, especially if you don't know what values you'll be using. FoxDot uses a container type called a 'Pattern' to help solve this problem. They act like regular lists but any mathematical operation performed on it is done to each item in the list and done so pair-wise if using a second pattern. The base Pattern can be created like so:

```
print(P[1,2,3] * 2)
>>> P[2, 4, 6]
print(P[1,2,3] + [3,4])
>>> P[4, 6, 6, 5, 5, 7]
```

Notice how in the second operation, the output consists of all the combinations of the two patterns i.e. `[1+3, 2+4, 3+3, 1+4, 2+3, 3+4]`.

- Try some other mathematical operators and see what results you get.
- What happens when you group numbers in brackets, like P[1,2,3] * (1,2)?

There are several other Pattern classes in FoxDot that help you generate arrays of numbers but also behave in the same way as the base Pattern. We'll just look at two types here, but execute `print(classes(Patterns.Sequences))` to see what others exist and have a go at using them.

In Python, you can generate a range of integers with the syntax `range(start, stop, step)`. By default, start is 0 and step is 1. You can use `PRange(start, stop, step)` to create a Pattern object with the equivalent values:

```
print(range(10))
>>> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(PRange(10))
>>> P[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(PRange(10) * [1, 2])            # Pattern class behaviour
>>> P[0, 2, 2, 6, 4, 10, 6, 14, 8, 18]
```

But what about combining patterns? In Python, you can concatenate two lists (append one to another) by using the + operator but FoxDot Patterns use this to perform addition to the data within the list. To connect two Pattern objects together, you can use the pipe symbol, `|`, which Linux users might be familiar with – it is used to connect command line programs by sending output from one process as input to another.

```
print(PRange(4) | [1,7,6])
>>> P[0, 1, 2, 3, 1, 7, 6]
```

FoxDot automatically converts any object being piped to a Pattern to the base Pattern class so you don't have to worry about making sure everything is the right type. There exists several types of Pattern sequences in FoxDot (and the list is still growing) that make generating these numbers a little easier. For example, to play the first octave of a pentatonic scale from bottom to top and back again, you might use two `PRange` objects:

```
p1 >> pluck(PRange(5) | PRange(5,0,-1), scale=Scale.default.pentatonic)
```

The `PTri` class does this for you:

```
p1 >> pluck(PTri(5), scale=Scale.default.pentatonic)
```

3. TimeVars

A `TimeVar` is an abbreviation of "Time Dependent Variable" and is a key feature of FoxDot. A TimeVar has a series of values that it changes between after a pre-defined number of beats and is created using a `var` object with the syntax

`var([list_of_values],[list_of_durations])`. Example:

```
a = var([0,3],4)              # Duration can be single value
print(int(Clock.now()), a)    # 'a' initally has a value of 0
>>> 0, 0
print(int(Clock.now()), a)    # After 4 beats, the value changes to 3
>>> 4, 3
print(int(Clock.now()), a)    # After another 4 beats, the value changes to 0
>>> 8, 0
```

When a TimeVar is used in a mathematical operation, the values it affects also become TimeVars that change state when the original TimeVar changes state – this can even be used with patterns:

```
a = var([0,3], 4)
print(a + 5)          # beat = 0
>>> 5
print(a + 5)          # beat = 4
>>> 8
b = PRange(4) + a
print(b)               # beat = 8 and a has a value of 0
>>> P[0, 1, 2, 3]
print(b)               # beat = 12 and a has a value of 3
>>> P[3, 4, 5, 6]
```

1.2 References

This section contains the modules references.

1.2.1 FoxDot

FoxDot package

Subpackages

FoxDot.lib package

Subpackages

FoxDot.lib.Code package

Submodules

FoxDot.lib.Code.foxdot_func_cmp module

FoxDot.lib.Code.foxdot_live_function module

FoxDot.lib.Code.foxdot_tokenize module

FoxDot.lib.Code.foxdot_when_statement module

FoxDot.lib.Code.main_lib module

Module contents

FoxDot.lib.GhostCoder package

Submodules

FoxDot.lib.GhostCoder.Grammar module

TODO: Fixme

#.. automodule:: FoxDot.lib.GhostCoder.Grammar

members

undoc-members

show-inheritance

FoxDot.lib.GhostCoder.Writer module

TODO: Fixme

#.. automodule:: FoxDot.lib.GhostCoder.Writer

members

undoc-members

show-inheritance

Module contents

TODO: Fixme

#.. automodule:: FoxDot.lib.GhostCoder

members

undoc-members

show-inheritance

FoxDot.lib.Patterns package

Submodules

FoxDot.lib.Patterns.Generators module

FoxDot.lib.Patterns.Main module

FoxDot.lib.Patterns.Operations module

FoxDot.lib.Workspace.Console module

FoxDot.lib.Workspace.Editor module

FoxDot.lib.Workspace.Format module

FoxDot.lib.Workspace.LineNumbers module

FoxDot.lib.Workspace.MenuBar module

FoxDot.lib.Workspace.Prompt module

FoxDot.lib.Workspace.TextBox module

Module contents

Submodules

FoxDot.lib.Bang module

FoxDot.lib.Buffers module

FoxDot.lib.Constants module

FoxDot.lib.Effects module

FoxDot.lib.Key module

FoxDot.lib.Logging module

FoxDot.lib.Midi module

FoxDot.lib.OSC module

TODO: Fixme

#.. automodule:: FoxDot.lib.OSC

members

undoc-members

show-inheritance

FoxDot.lib.OSC3 module

TODO: Fixme

#.. automodule:: FoxDot.lib.OSC3

members

undoc-members

show-inheritance

FoxDot.lib.Players module

FoxDot.lib.Repeat module

FoxDot.lib.Root module

FoxDot.lib.Scale module

FoxDot.lib.ServerManager module

FoxDot.lib.TempoClock module

FoxDot.lib.TimeVar module

Module contents

Module contents

1.3 Changelog

1.3.1 v0.5.6 fixes and updates

- Running FoxDot with a `--pipe` flag is compatible with Python 2
- Updated the "pads" SynthDef
- Old unidirectional server manager class can be used by editing config file.

1.3.2 v0.5.5 fixes and updates

- Fix `TimeVar` class so it no longer inherits from `Repeatable` i.e. no longer has access to the "every" method.
- Fix pattern bug when creating a pattern using the `P` generator; `P[P(0,2)]` no longer returns `P[0, 2]`. However, `P[(0,2)]` is interpreted exactly as `P[0, 2]` and will return that instead.
- Added more variation to the "formantFilter" effect
- "loop" samples are added "on-the-fly" as opposed to loaded at start up. These can be loaded by filepath (with or without extension):

```
a1 >> loop("/path/to/sample.wav")
a2 >> loop("/path/to/sample")

a1 >> loop("yeah")              # Searches recursively for yeah.
↪(wav|wave|aif|aiff|flac)
a1 >> loop("perc/kick")         # Supports directories in the path
```

(continues on next page)

(continued from previous page)

```
a1 >> loop("*kick*", sample=2) # Supports * ? [chars] and [!chars]
a1 >> loop("**/*_Em")          # Supports ** as 'recursively all subdirectories'
```

1.3.3 v0.5.4 fixes and updates

- Better communication from external processes. Running FoxDot with a `--pipe` flag (e.g. `python -m FoxDot --pipe`) allows commands to be written via the stdin. Each command should end with a blank line.

1.3.4 v0.5.3 fixes and updates

- Player attribute aliases added. Using `pitch` and `char` will return a player's `degree` attribute.
- Player Key behaviour improved. Using multiple conditions e.g. `4 < p1.pitch < 7` will hold the value 1 while `p1.pitch` is between 4 and 7, and a 0 otherwise. These conditions can be "mapped" to values other than 1 by using the `map` method to map values, or results of functions, to other values/functions (which are applied to the values):

```
b1 >> bass(var([0,4,5,3]))
# Takes a dictionary of values / functions
p1 >> pads(b1.pitch.map(
    { 0: 2,
        4: lambda x: x + P(0,2),
            lambda x: x in (5,3): lambda y: y + PRand([0,2,4,7])
    }))
```

- Known issue: mapping to a pattern of values for a Player's duration does not work as expected so be careful.
- The `Player.every` method can now take `Pattern` methods, which affect the degree of the `Player` (specifying attributes will be added later). Instead of applying the function every time it is called, it has a switch that applies the function then "un-applies" the function.

```
p1 >> play("x-i-").every(6, "amen").every(8, "palindrome")
```

1.3.5 v0.5.2 fixes and updates

- Improved behaviour of `TimeVar`, `Pvar`, and `PvarGenerator` classes when created via mathematical operators.
- SynthDefs can be read loaded into FoxDot from SuperCollider using the `FileSynthDef` and `CompiledSynthDef` classes (see `SynthDef.py`).
- `DefaultServer` instance has a `forward` attribute that, when not `None`, sends any outgoing OSC message to. Example:

```
# Sends any OSC message going to SuperCollider to the address
DefaultServer.forward = OSCClient(("localhost", 57890))
```

1.3.6 v0.5.0 fixes and updates

- Pattern "zipping" behaviour changed. A `PGroup` within a sequence is extended when zipped with another instead of nesting it. e.g.

```
# Old style
>>> P[(0,1),(2,3)].zip([(4,5)])
P[P(0, 1, P(4, 5)), P(2, 3, P(4, 5))]
# New style
>>> P[(0,1),(2,3)].zip([(4,5)])
P[P(0, 1, 4, 5), P(2, 3, 4, 5)]
```

- Consequently, sample player strings can use the <> arrows to play multiple sequences together using one string.

```
# This plays three patterns together
d1 >> play("<x ><  o[ o]>< -(-=)>", sample=(0,1,2))
```

- To use a different sample value for each pattern use a group of values as in the example above. Each value in relates to each pattern e.g. the "x" used sample 0, the "o" pattern uses sample 1 and the "-" pattern uses sample 2. If you want to use multiple values just use a group within a group:

```
# Plays the snare drum in both channels at different rates
d1 >> play("<x x><  o >", pan=(0, (-1,1)), rate=(1, (1,.9))
```

- Network synchronisation introduced! This is still quite a beta feature and feedback would be appreciated if you come across any issues. Here's how to do it:
- To connect to another instance of FoxDot over the network you need one user to be the master clock. The master clock user needs to go from the menu to "Language" then "Listen for connections". This will start listening for connections from other FoxDot instances. It will print the IP address and port number to the console; give this information to your live coding partner. They need to run the following code using the IP address on the master clock machine:

```
Clock.connect("<ip address>")
```

- This will copy some data, e.g. tempo, from the master clock and also adjust for the differences in local machine time (if your clocks are out of sync). The latter will depend on the latency of the connection between your machines. If you are out of time slightly, set the `Clock.nudge` value to a small value (+-0.01) until the clocks are in sync. Now whenever you change the `Clock.bpm` value, the change will propagate to everyone on the next bar.

1.3.7 v0.4.14 fixes and updates

- Pattern getitem method now allows Patterns to be indexed using a Player Key e.g. `P[0,1,2,3][p1.degree]` that will return the item in the Pattern based on the integer values of the key (`p1.degree` in this example).
- Added `future` method. Like `schedule` it adds a callable object to the queue but doesn't need the exact beat occurrence, just how many beats in the future from "now". First argument is time, followed by the object, arguments, and keyword arguments.
- Player object `stop` method properly removes the player from `Clock.playing` list.
- Using Player Key `__getitem__` returns a player key whose calculation function is `__getitem__`. This is useful if you want to use just one of the values of another player if they are in a group. e.g.

```
p1 >> pads((0,2,4) + var([0,4,5,3]))
b1 >> bass(p1.degree[0]) # Only plays the "root" note of the chord
```

- SuperCollider bus number resets to 4 instead of 1 to prevent feedback loops when using reverb.
- Changed "verb" keyword to "mix" for reverb effect. Default is changed from 0.25 to 0.1.

- `GeneratorPattern new` method converts the `other` argument to a Pattern so you can use lists/tuples as opposed to just Patterns/PGroups when performing operations e.g. `PWalk() + (0,4)`.
- Fixed `newline` method to only add an indent if the INSERT index was in brackets or following a colon instead of doing so if the line had open brackets / colon. Evaluated code no longer highlights any empty preceeding lines.
- Python 3 uses xrange as range

1.3.8 v0.4.13 fixes and updates

- Moved demo files into main package to fix install from pip.

1.3.9 v0.4.12 fixes and updates

- `Player.stop_calling` is now `Player.never`. If a `Player` is calling its own method (implemented by the `every` method e.g. `p1 >> pads().every(4, "reverse")` you can now stop the repeated call by using `p1.never("reverse")`.
- Fixed circular referencing bug when using PGroups e.g. `p1 >> pads(p1.degree + (0,4))`
- Window transparency can now be toggled from the "Edit" menu
- Added Tutorial files that can be loaded from the menu
- Multiple uses of the `every` method with the same method name can be used together by specifying an `ident` keyword, which can be any hashable value i.e. a string or integer.

```
# The second "stutter" no longer overrides the first
d1 >> play("x-u-").every(8, "stutter", 8).every(3, "stutter", 4, dur=1, degree="y
↪", ident=1)
```

- Fix `group_modi` function to test for `TimeVar` instances instead of trying and failing to index their contents so that `TimeVar`'s with strings in their contents don't get into an infinite recursive call.

1.3.10 v0.4.11 fixes and updates

- Removed `sys.maxint` to conform with Python 3

1.3.11 v0.4.10 fixes and updates

- Fixed negative pitch bug
- `PGroupMod` replaces `PGroupStar` when using square brackets in a "play" string. This "flattens" the values so that many nested `PGroups` don't create exponentially larger loops when sending events to SuperCollider.
- Fixed `stutter` so that delays caused by `PGroups` are no longer lost.
- `PRand`, `PwRand`, and `PxRand` choose from a random index instead of a random element so that any "nested" `GeneratorPatterns` generate a new item instead of returning the same one i.e. at index 0.
- Fixed `Player.degrade`
- `slidedelay` default value changed from 0.75 to 0
- Replaced "Control" with "Command" for menu short-cuts on Mac OS (Thanks ianb)

- Improved documentation layout
- Player methods such `shuffle` no longer affect the text of a `play` Player as it would overload the undo heap. This may be added back in at a layer date.
- Infinite recursion errors caused by circular referencing no longer seem to occur.
- Improved printing of Players to include identifier e.g. `<p1 - pluck>`.

1.3.12 v0.4.9 fixes and updates

- Fixed issues with indexing `GeneratorPattern` and using `var` in Player methods.
- Random `GeneratorPattern` objects, such as `PRand` can take a `seed` keyword that will give you the same sequence of values for the same value of seed (must be an integer).

1.3.13 v0.4.8 fixes and updates

- Unsaved work is stored in a temporary filed that can be loaded on the next startup.
- Player objects can now take tuples as an argument, which delays the next event (similar to the `delay` argument but works with the following event):

```
# The Player object uses the smallest duration in the tuple to move to the next␣
↪event
p1 >> pluck([0,1,2], dur=[1,1,(0.5,1)])
```

- Pattern function `PRhythm` takes a list of single durations and tuples that contain values that can be supplied to the `PDur` e.g.:

```
# The following plays the hi hat with a Euclidean Rhythm of 3 pulses in 8 steps
d1 >> play("x-o-", dur=PRhythm([2,(3,8)]))
```

1.3.14 v0.4.7 fixes and updates

- FoxDot is now Python 3 compatible, so make sure you treat your print statements as functions i.e. use `print("Hello, World!")`
- Added `audioin` SynthDef for processing audio from the default recording device.
- Fixed bugs relating to chaining multiple `every` methods and ending their call cycle when the parent player is stopped
- Improved flexibility of referencing player attributes e.g.

```
p1 >> pads([0,1,2,3], dur=2).every(8, "stutter", 4, degree=p1.degree+[2,4,7])
```

1.3.15 v0.4 fixes and updates

- FoxDot is now Python 3 compatible, so make sure you treat your print statements as functions i.e. use `print("Hello, World!")`

1.3.16 v0.3.7 fixes and updates

- Nested pattern bug fixed so that they no longer cause patterns to loop
- Improved clock scheduling after proper "latency" implementation
- Added a new SynthDef, `loop`, to play longer samples:

```
# First argument is the name of the file minus the extension

p1 >> loop("billions")

# Use the dur keyword to specify when to loop the file

p1 >> loop("billions", dur=8)

# The second argument is the starting point in beats such that the following 2␣
↪lines are equivalent

p1 >> loop("billions", dur=16)

p1 >> loop("billions", [0,8], dur=8)
```

- Added ability to use the lambda symbol in place of the word lambda. Insert it by using `Ctrl+L`.
- Put `slide`, `slidefrom`, `coarse`, `pshift` into their own effects

1.3.17 v0.3.6 fixes and updates

- Any delay or stutter behaviour in Players is now handed over to SuperCollider by timestamping the OSCBundle, which should make FoxDot a lot more efficient & removed `send_delay` and `func_delay` classes.
- Using a `TimeVar` in a pattern function, such as `PDur`, now creates a time-varying pattern as opposed to a pattern that uses the `TimeVar`'s current value. e.g.

```
>>> test = PDur(var([3,5], 4), 8)
>>> print test # time is 0
P[0.75, 0.75, 0.5]
>>> print test # time is 4
P[0.5, 0.25, 0.5, 0.25, 0.5]
```

- Adding values to a player performs the whole operation as opposed to adding each value in turn when the Player is called. This improves efficiency when using data structures such as `TimeVar``s as it only creates a new once ``TimeVar` when the addition is done.
- Improved usability of `PlayerKey` class, accessed when get the attribute of a Player e.g. `p1.degree`.
- Sleep time set to small value. 0 sleep time would crash FoxDot on startup on some systems.
- Made the behaviour of the `every` method more consistent rather than just starting the cycle at the next bar.

1.3.18 v0.3.5 fixes and updates

- In addition to P*, P+, P/, and P** have been added. P+ refers uses the sustain values in a player to derive its delay. P/ delays the events every other time it is accessed, and P** shuffles the order the values are delayed.
- Added `PWalk` generator pattern.
- TimeVars are easier to update once created.

```
# Creates a named instance called foo
var.foo = var([0,1],4)

# Reassigning a var to a named var updates the values instead of creating a new␣
↪var
var.foo = var([2,3,4,5],2)
```

- Removed `sleep` from scheduling clock loop to increase performance. If you want to decrease the amount of CPU FoxDot uses, change the sleep duration to a small number around 0.001 like so

```
Clock.sleep_time = 0.001
```

- Added pitch shift (`pshift`) to Sample Players, which increases the pitch of a sample by the number of semitones. You can use `Scale.default.semitones()` to generate semitones from the current scale.

1.3.19 v0.3.3 fixes and updates

- Added a new `Pattern` type data structure called a P-Star or `P*`. It is a subclass of `PGroup` but it has a "behaviour" that effects the current event of Player object, which, in this instance, adds a delay to each value based on the current Player's duration. e.g.

```
# Plays the first note, 0, for 4 beats then the pitches 2, 4, and 6 at 4/3 beats␣
↪each.
p1 >> pluck([0, P*(2,4,6)], dur=4)

# The can be nested
p1 >> pluck([0, P*(2,4,P*(6,7)], dur=4)

# Work in the same way that a SamplePlayer uses square brackets
p2 >> play("x[--]o[-o]")
```

- Frequency and buffer number calculation is done per OSCmessage which means these values can be modified in any delayed message i.e. when using the Player `stutter` method like so:

```
p1 >> pluck([0,1,2,3], dur=1).every(4, 'stutter', 4, degree=[10,12], pan=[-1,1] )

d1 >> play("x-o-").every(5, 'stutter', 2, cycle=8, degree="S")
```

- Using as `linvar` as the Clock tempo will no longer crash the Clock.
- New effects have been added; `shape` which is a value between 0 and 1 (can be higher) that relates to a level of distortion, and `formant` which is a value between 0 and 8 and applies different formant filters to the audio.
- `hpf` and `lpf` have resonance values now: `hpr` and `lpr`
- You can open the config file directly from FoxDot by using the "Help & Settings" menu. Likewise you can open the directory that holds where your samples are kept. This can be changed in the config file.

1.3.20 v0.3.2 fixes and updates

- `PlayerKey` data type can handle `PGroup` transformations without crashing, which improves performance when using `follow`
- `PlayerKey` data type greater than and less than functions fixed and now works with amplitudes.
- Better handling of scheduled functions that are "late"

- Experimental: `play` SynthDef can have a rate of -1 to be played in reverse and also uses a keyword `coarse` similar in function to `chop`
- Added `Pattern` method, palindrome that appends a mirrored version of the pattern to itself.
- Removed visual feedback for shuffling, rotating, etc patterns in Players as it did not work correctly with nested patterns.

1.3.21 v0.3.1 fixes and updates

- `TempoClock` uses a `start_time` value that, when used on multiple instances of FoxDot, should synchronise the timings. This is a work in progress
- Added a "use SC3 Plugins" tick-box on the "Code" drop down menu to allow for easier configuration
- `piano` SynthDef added using th SC3 Plugin "MdaPiano"

1.3.22 v0.3 fixes and update

- `var` type can be used with Player `delay` and nested groups in the `oct` attribute.
- Increased `TempoClock` latency to 0.2 seconds for improved performance.
- Better handling for auto-completed quotation marks

1.3.23 v0.2.11 fixes and updates

- Caught `ImportError` if the user does not have `rtmidi` installed.
- Improved `Player.stutter`

1.3.24 v0.2.10 fixes and updates

- New SynthDefs added. Use `print SynthDefs` to view.
- Improved timing in the `TempoClock` class through use of threading and a latency value. Thanks to Yaxu and Charlie Roberts for the help.
- Dubstep samples added to the 'K' character.
- Sample banks re-arranged. Use `print Samples` for more information.
- Sample Player argument, `scrub` removed. You can now use `slide`/`slidefrom` and `vib` as you would do with a normal Player object to manipulate playback rate.
- `Pattern` class now has a `layer` method that takes a name of a `Pattern` method as its first argument and then arguments and keyword arguments for that method and creates a pattern of `PGroups` with their values zipped together.

```
>>>  print P[1,2,3,4].layer("reverse")
P[P(1, 4), P(2, 3), P(3, 2), P(4, 1)]

>>>  print P[1,2,3,4].layer("rotate", 2)
P[P(1, 3), P(2, 4), P(3, 1), P(4, 2)]
```

- New nested `PGroup` behaviour added for players. Each value in each `PGroup` in an event relates to the values in any other `PGroup` in the same index, even if that value is also a `PGroup`. This concept is better described through an example:

```
p1 >> pluck((0,2), pan=(0,(-1,1)), vib=(0,(0,12)), dur=4, chop=(0,4))
```

- The first note, 0, is played with a pan of 0, chop of 0, and with no vibrator added. The second note, 2, is played with a chop of 4 and with no vibrato with a pan of -1 (left) but with a vibrato value of 12 with a pan of 1 (right).
- Experimental: Players can "follow" other Players' attributes over time by referencing their attributes.

```
p1 >> pads([4,5,6,7], dur=2, chop=4)

p2 >> pluck(p1.degree + 2, vib=p1.chop*3)
```

1.3.25 v0.2.9 fixes and updates

- Improved automatic bracket handling and formatting
- Colour scheme update
- "Upper-case" samples now read properly
- `cycle` argument added to the `.every()` player method to denote the cycle length of which to execute the specified method, e.g.

```
# Shuffles the samples on the 5th beat of each 8 beat cycle
bd >> play("x-o-").every(5, 'shuffle', cycle=8)
```

1.3.26 v0.2.8 fixes and updates

- Minor bug fixes
- Improved automatic bracket handling and formatting
- Console is now resizable
- Scale and root can be assigned using the equals operator e.g. `Scale.default = "minor"` and `Root.default = var([0,4])`

1.3.27 v0.2.7 fixes and updates

- Rest class added
- Undo and Redo functions fixed
- Infinite loop caused by empty brackets in PlayStrings fixed
- Menu bar added with several short-cuts
- Player follow method improved
- Improved documentation
- "style" keyword argument changed to "sample"

1.3.28 v0.2.6 fixes and updates

- OSC Communication is now done through a dedicated SuperCollider Quark

1.3.29 v0.2.3 fixes and updates

- Effects are now implemented using busses on SuperCollider, which uses less CPU
- Effects can be customised and defined
- Sample Player behaviour (i.e. how the string of characters relates to playback) has been altered. Square brackets refer to a single event even though two samples are played.
- SuperCollider is booted on startup with a compiled startup file.

1.3.30 v0.2.2 fixes and updates

- `PDur` added: a pattern that implements Euclidean Rhythms
- Player attributes can be manipulated using the `Player.every` method
- Errors caught and displayed in FoxDot console instead of crashing
- Can set different tempi for Players using the `bpm` keyword
- Sample Player objects can play multiple samples together by grouping them as a PGroup but cannot feature square brackets

1.3.31 v0.2.1 fixes and updates

- Syntax highlighting bugs fixed
- Visual feedback for `shuffle`, `mirror`, and `rotate` methods for `play` SynthDef
- SC3 Plugins disabled by default
- Player Object dictionaries shallow copied before iteration to stop `RunTimeErrors` occurring

1.3.32 v0.2.0 fixes and updates (4/12/16)

- Reorganised project structure. Samples and code are kept separate.
- SuperCollider `OSCFunc.scd` now found in `/osc/` folder
- `setup.py` now included for an easier install
- (in progress) characters can have more than one sample attached to them. These are accessed by supplying a `buf` keyword argument.
- Python lists can be interpreted as FoxDot pattern when attached with a P prefix e.g. `P[1, 2, 3] + [0,2]` will return `P[1,4,3,3,2,5]` not `[1,2,3,0,2]`.

1.3.33 v0.1.9 fixes and updates

- PSparse renamed to PBin
- Loading the icon now works on Linux
- Upheaval of SCLang API
- Player Objects now have visual feedback behaviour via the `bang` method and take Tkinter tag_config keyword arguments.
- Consolas now default font
- Fixed `Pvar` and `linvar` bugs

1.3.34 v0.1.8 fixes and updates

- PSparse pattern type added (all Pattern names can be seen by executing `print(PatternTypes)`
- Major overhaul of Pattern nesting/lacing behaviour. Patterns can now be nested to multiple levels.
- Player object attributes now 'follow' one another and their current values are examined instead of the Pattern value

1.3.35 v0.1.7 fixes and updates

- "Chop" added to default SynthDef behaviour
- GUI icon updated
- Using `var` objects for Player durations no longer crashes
- New Pattern types added
- FoxDot can be run using `python -m FoxDot` if FoxDot is in your PATH

1.3.36 v0.1.6 fixes and updates

- Decimator (a.k.a. bitcrush) added to default SynthDef behaviour
- `SynthDefs` and `BufferManager` can be reloaded
- Removed automatic bootup of sclang as default behaviour
- Added new SynthDefs

1.3.37 v0.1.5 fixes and updates

- Removed RegEx find and replace >> and `$` syntax. FoxDot now uses pure Python code and saved files can be run by themselves.

1.3.38 v0.1.4 fixes and updates

- Save/Open file feature added
- Console can now be toggled
- Reduced CPU usage when the TempoClock queue is empty

- Added a 'grain' attribute to the `sample_player` SynthDef

1.3.39 v0.1.3 fixes and updates

- Key bindings for Linux, Mac, and Windows 10 fixed
- Fixed freeze on keyboard interrupt exit

CHAPTER 2

Indices and tables

- genindex
- modindex
- search