GenerIter

Release 0.0.1

generiter@gmx.com

Feb 15, 2021

CONTENTS:

CHAPTER

ONE

PREFACE

1.1 A Few Words From One Creator Of The GenerIter

By Thomas Park, 10th Feb 2021

It is now time to deploy the GenerIter, and I wanted to share some thoughts with folks.

My name is Thomas Park– the GenerIter was originally a console Python application called "RoboDJ". I created the application to help me with composing music– I was tired of some of the hard labor of trimming and preparing samples, only to combine them in all of the familiar ways.

I went to Python to help, which is a language I learned in a Launchcode code camp in 2018.

While creating the original code, I made some breakthroughs– perhaps the most exciting was when the code could literally create and output thousands of tracks per hour. As neat as that was, I was aware that I had stepped on some folk's shoes.

There are those who disavow all of Modern Art as a waste. There are many who do not love technology, and shun its use. And there are some who feel that music made largely by a processor must be bad and harmful to this world.

Sadly, the weekend of this release, my Mother appears to be dying, with little hope of recovery, due to a number of causes. She did not love hospitals, and in her 70s, did not have even a checkup for her last seven years. As a result, when they finally had her in to treat her for Covid, they found symptom after symptom of other issues– too many, alas, for a favorable diagnosis.

We live in a changing world, and when we need help, I feel that we should ask for it. I know that my Mother did not approve of my recent coding and musical developments. I did not approve of her lack of care. There was no lack of love, only understanding.

The GenerIter, so well-developed by Jeremy Pavier, already does more than I expected it to, and better. I cannot thank him enough for working so many hours at no charge to help achieve this dream. I would say it has also become his.

This tool goes with sadness, too, out to my mother, a lovely woman who will never understand what it does, will never approve of it, or anything like it. Sad but true that the world moves on without us, regardless of our feelings. I choose to embrace the future, and try to make it better.

And I hope that this Module comes to enhance the lives of many, easing a process that can be unnecessarily complex, and opening the doors to new sounds and new combinations.

Thank you for your time, Thomas Park Co-creator of the GenerIter

CHAPTER

TWO

INTRODUCTION

2.1 Overview

The **GenerIter** package is a software development kit (SDK) with which Python coders can manipulate sampled sounds programmatically to create new generative and iterative compositions.

The SDK is designed in such a way that it does all the heavy lifting of organising sample library contents according to whatever criteria makes sense to the composer, and then applying generative algorithms to those libraries using randomised or directed selection techniques.

The objective of the SDK is to give the composer some powerful assistance without constraining or dictating the creative aspects of their projects. To that end, you will find the SDK fairly lightweight in its implementation, but very configurable and agnostic about workflows and processes.

One of the key features of the package is that, although supplied with a set of example processors and algorithms for the novice user to start getting quick results, there is a simple facility that allows the composer-programmer to extend, replace, organise and control whatever algorithmic processes they wish to apply to their sample library external to the example processors we provide.

CHAPTER

THREE

INSTALLATION

3.1 Installing on macOS

GenerIter requires Python 3.6+ but older versions of macOS only had Python 2.7. First determine if you're running 3.x or 2.x: Open up the command line via the Terminal application which is located at Applications -> Utilities -> Terminal.

In the terminal, at the *$* prompt, enter the following command, and look at the output:

```
$ python --version
Python 2.7.17
```

If this is version 2.x, continue, if it's 3.x, skip down to 'Installing via pip'

3.1.1 Installing Python 3.x

To update macOS's version of Python, we'll use the tool [Homebrew](brew.sh), to do this, cut and paste the following command into your terminal, and press 'Enter'

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/
↪install.sh)"
```

This will install Apple's Xcode commandline tools and other required software, it could take up to 5 minutes to complete. When this is done, initialize and test that *homebrew* is installed and working:

```
brew doctor
```

If that's successful, use *homebrew* to install Python 3.9:

```
brew install --build-from-source python@3.9
```

This command will take longer than the last one, as it's going to build Python from source.

3.1.2 Installing via pip

Now we'll use *pip*, a package manger for Python, to install generiter

```
pip3 install generiter
```

3.2 Installing on Debian-derived Linux

Generiter requires Python 3.6+ which most recent version of Linux will have. First determine if you're running 3.x or 2.x: Open up terminal application in Linux, and prompt, enter the following command, and look at the output:

```
$ python --version
Python 2.7.17
```

If you see a version of Python starting with a 2, such as Python 2.7.10, then try the same command using python3 instead of python, it's possible both will be installed.

If it's not and you only have version 2.x, continue, if it's 3.x, skip down to 'Installing generiter via pip'

GNU Debian Linux and Ubuntu Linux

Install Python 3.x as defined at [Install Python](https://installpython3.com/linux/), by using the deadsnakes PPA:

```
sudo add-apt-repository ppa:deadsnakes/ppa
sudo apt-get update
sudo apt install python3.8
```

Now we'll use *pip*, a package manger for Python, to install generiter

```
pip3 install generiter
```

3.3 Installing on Arch Linux

Install Python 3.x via *pacman*:

```
pacman -S python3
```

Now we'll use *pip*, a package manger for Python, to install generiter

```
pip3 install generiter
```

3.4 Installing on Windows

Install Python 3.x as defined at [Install Python](https://installpython3.com/windows/)

Open Powershell, and run the following command:

```
pip3 install generiter
```

3.5 Advanced Installation Notes

The **GenerIter** package uses pydub, a dependency which should be satisfied during the package installation.

However, **pydub** has a partial dependency on FFmpeg

At the moment we only use **WAV** format audio throughout, so the package will work without this dependency and only generate a benign warning. Ultimately we wish to be able to transparently convert into and out of other formats (e.g. **mp3**, **FLAC**, etc.). In order to use that functionality, you will need to install FFmpeg on your machine later.

The source for the GenerIter package can also be found on GitHub.

CHAPTER

FOUR

TUTORIALS

Important Note :

This module currently depends on the **pydub** module for manipulating audio segments in memory. The nature of the **pydub** implementation is such that it can hog memory, particularly if your algorithm doesn't clear redundant references to "old" immutable audio segments. This means that there may be occasions where you can be throwing "out of memory" exceptions.

This is not a **GenerIter** defect, but a **pydub** design flaw.

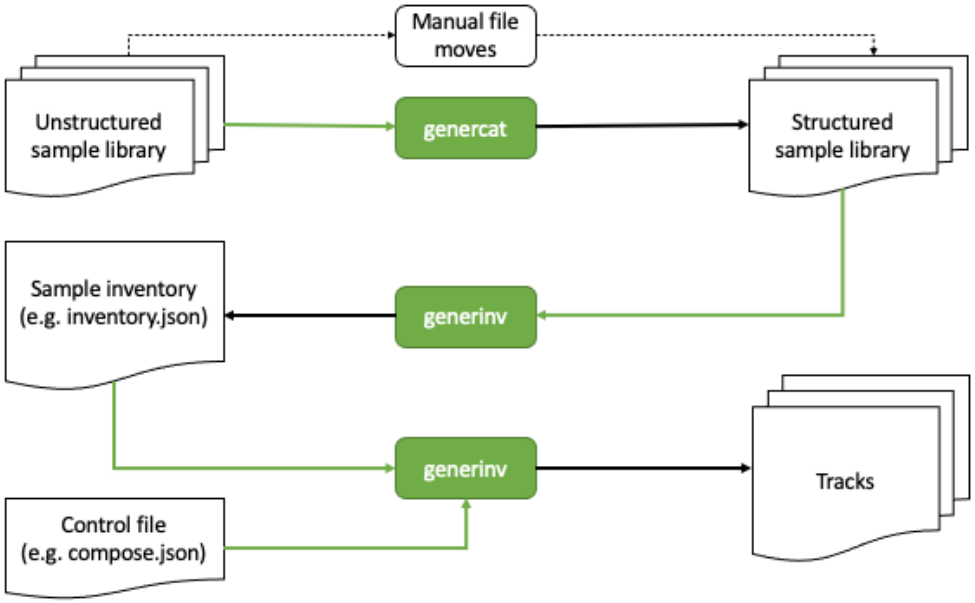
As a **GenerIter** user, it should be possible to ameliorate this problem by using smaller samples, sample sets and reducing the configuration workload. Future enhancements to **GenerIter** include a plan to replace **pydub** with something that manages memory better and minimises the likelihood of hitting this problem.

4.1 Basic Workflow

This text uses MacOS/Linux-style pathnames generally. The code has been written such that it should also support DOS-style paths in a Windows-based command shell. Key differences will be highlighted when they occur.

To illustrate this workflow with a worked example, I am going to create a very small sample library under */tmp/samples* using the WAV samples provided Archive.org. Note that the "GenerIter_Demo_Track" files should be moved into a different directory outside the */tmp/samples* tree as they are not used here.

The core **GenerIter** process is illustrated in the diagram:



The basic workflow breaks down into 4 phases:

1. Categorising the sample library
2. Creating the sample inventory
3. Creating the composer control file
4. Generating music

4.1.1 Categorising the sample library

This first phase assumes the you starts with an unstructured set of samples in youre directory.

To make the most out of the inventory classification system, it helps if the samples are organised in subdirectories named after the categories the composer finds useful. These category names can be clompletely arbitrary, but for this illustration I am going to use musical voice/instrument category specifiers like “Drums”, “Bass”, “Synths”, etc.

The good news is that these categories can be specified at any level in your sample tree and don’t require the “Drums” samples, for example to all be in a single directory.

The genercat utility is a helper command that allows the composer to allocate sets of sample files into named subdirectories, where the names of those directories will ultimately correspond to the categories in the inventory.

To illustrate this workflow with a worked example, I am going to create a very small sample library under */tmp/samples* using the WAV samples I downloaded from Archive.org:

```
cd /tmp/samples
.
├── Bass_1.wav
├── Bass_2.wav
├── Bass_3.wav
├── Beats_1.wav
├── Beats_2.wav
```

(continues on next page)

(continued from previous page)

```
├── Beats_3.wav
├── Drone_1.wav
├── Drone_2.wav
├── Drone_3.wav
├── Guitar_1.wav
├── Guitar_2.wav
├── Guitar_3.wav
├── Organ_1.wav
├── Organ_2.wav
├── Organ_3.wav
├── Pepper_1.wav
├── Pepper_2.wav
├── Pepper_3.wav
├── Saxophone_1.wav
├── Saxophone_2.wav
└── Saxophone_3.wav
```

This is then categorised, thus:

```
genercat -C Bass
genercat -C Beats
genercat -C Drone
genercat -C Guitar
genercat -C Pepper
genercat -C Organ
genercat -C Saxophone
```

resulting in a tree structure that looks like:

```
.
├── Bass
│   ├── Bass_1.wav
│   ├── Bass_2.wav
│   └── Bass_3.wav
├── Beats
│   ├── Beats_1.wav
│   ├── Beats_2.wav
│   └── Beats_3.wav
├── Drone
│   ├── Drone_1.wav
│   ├── Drone_2.wav
│   └── Drone_3.wav
├── Guitar
│   ├── Guitar_1.wav
│   ├── Guitar_2.wav
│   └── Guitar_3.wav
├── Organ
│   ├── Organ_1.wav
│   ├── Organ_2.wav
│   └── Organ_3.wav
├── Pepper
│   ├── Pepper_1.wav
│   ├── Pepper_2.wav
│   └── Pepper_3.wav
└── Saxophone
    ├── Saxophone_1.wav
    ├── Saxophone_2.wav
```

(continues on next page)

(continued from previous page)

```
    └── Saxophone_3.wav
```

4.1.2 Creating the sample inventory

Once you are satisfied that your sample libraries are organised in the way you want, it is time to create the inventory file from which you will be selecting samploes with your algorithms.

To do this you use the generinv utility in the directory in which you intend to run the **generiter** command eventually.

Top Tip - It's a good idea to keep your inventory and composer files away from your sample library, just to avoid cluttering the library up with spurious files.

For this simple demonstration, assuming your sample tree is rooted in */tmp/samples* the basic commands would look something like:

```
cd $HOME
generinv -I /tmp/samples -o inventory
```

This creates an *inventory.json* file that will look like:

```
{
    "Bass":{
        "/tmp/samples/Bass/Bass_1.wav":true,
        "/tmp/samples/Bass/Bass_2.wav":true,
        "/tmp/samples/Bass/Bass_3.wav":true
    },
    "Beats":{
        "/tmp/samples/Beats/Beats_1.wav":true,
        "/tmp/samples/Beats/Beats_2.wav":true,
        "/tmp/samples/Beats/Beats_3.wav":true
    },
    "Drone":{
        "/tmp/samples/Drone/Drone_1.wav":true,
        "/tmp/samples/Drone/Drone_2.wav":true,
        "/tmp/samples/Drone/Drone_3.wav":true
    },
    "Guitar":{
        "/tmp/samples/Guitar/Guitar_1.wav":true,
        "/tmp/samples/Guitar/Guitar_2.wav":true,
        "/tmp/samples/Guitar/Guitar_3.wav":true
    },
    "Organ":{
        "/tmp/samples/Organ/Organ_1.wav":true,
        "/tmp/samples/Organ/Organ_2.wav":true,
        "/tmp/samples/Organ/Organ_3.wav":true
    },
    "Pepper":{
        "/tmp/samples/Pepper/Pepper_1.wav":true,
        "/tmp/samples/Pepper/Pepper_2.wav":true,
        "/tmp/samples/Pepper/Pepper_3.wav":true
    },
    "Saxophone":{
        "/tmp/samples/Saxophone/Saxophone_1.wav":true,
        "/tmp/samples/Saxophone/Saxophone_2.wav":true,
        "/tmp/samples/Saxophone/Saxophone_3.wav":true
```

(continues on next page)

(continued from previous page)

```
    }
}
```

The Windows equivalent will look something like:

```
{
    "Bass":{
        "C:\\Users\\mysti\\Desktop\\Sample_Sounds\\Bass\\Bass_1.wav":true,
        "C:\\Users\\mysti\\Desktop\\Sample_Sounds\\Bass\\Bass_2.wav":true,
        "C:\\Users\\mysti\\Desktop\\Sample_Sounds\\Bass\\Bass_3.wav":true
    },
    "Beats":{
        "C:\\Users\\mysti\\Desktop\\Sample_Sounds\\Beats\\Beats_1.wav":true,
        "C:\\Users\\mysti\\Desktop\\Sample_Sounds\\Beats\\Beats_2.wav":true,
        "C:\\Users\\mysti\\Desktop\\Sample_Sounds\\Beats\\Beats_3.wav":true
    },
    "Drone":{
        "C:\\Users\\mysti\\Desktop\\Sample_Sounds\\Drone\\Drone_1.wav":true,
        "C:\\Users\\mysti\\Desktop\\Sample_Sounds\\Drone\\Drone_2.wav":true,
        "C:\\Users\\mysti\\Desktop\\Sample_Sounds\\Drone\\Drone_3.wav":true
    },
    "Guitar":{
        "C:\\Users\\mysti\\Desktop\\Sample_Sounds\\Guitar\\Guitar_1.wav":true,
        "C:\\Users\\mysti\\Desktop\\Sample_Sounds\\Guitar\\Guitar_2.wav":true,
        "C:\\Users\\mysti\\Desktop\\Sample_Sounds\\Guitar\\Guitar_3.wav":true
    },
    "Organ":{
        "C:\\Users\\mysti\\Desktop\\Sample_Sounds\\Organ\\Organ_1.wav":true,
        "C:\\Users\\mysti\\Desktop\\Sample_Sounds\\Organ\\Organ_2.wav":true,
        "C:\\Users\\mysti\\Desktop\\Sample_Sounds\\Organ\\Organ_3.wav":true
    },
    "Pepper":{
        "C:\\Users\\mysti\\Desktop\\Sample_Sounds\\Pepper\\Pepper_1.wav":true,
        "C:\\Users\\mysti\\Desktop\\Sample_Sounds\\Pepper\\Pepper_2.wav":true,
        "C:\\Users\\mysti\\Desktop\\Sample_Sounds\\Pepper\\Pepper_3.wav":true
    },
    "Saxophone":{
        "C:\\Users\\mysti\\Desktop\\Sample_Sounds\\Saxophone\\Saxophone_1.wav":true,
        "C:\\Users\\mysti\\Desktop\\Sample_Sounds\\Saxophone\\Saxophone_2.wav":true,
        "C:\\Users\\mysti\\Desktop\\Sample_Sounds\\Saxophone\\Saxophone_3.wav":true
    }
}
```

All the paths are absolute, which makes the inventory file fully movable to anywhere in your filesystem.

4.1.3 Creating the composer control file

The first composer control file is going to be very simple. So, fire up the text editor of your choice and create a file called *compose.json* with the following content:

```
{
    "Basic" : {
        "beatsbassdrone" : {
            "tracks" : 20,
            "repeats" : 6
```

(continues on next page)

(continued from previous page)

```
        }
    },
    "Globals" : {
        "destination" : "<your path here>"
    }
}
```

The **Globals** parameters are those that are literally global to the generiter instance when it runs. In this simplified form, you will need to supply it with the path of the directory into which your generated tracks will be created.

For Linux/MacOS users this will look like:

```
"Globals" : {
    "destination" : "/my/output/path"
}
```

Whereas Windows users will need to escape the backslashes in the DOS form:

```
"Globals" : {
    "destination" : "c:\\my\\output\\path"
}
```

The top part of the composition file translates into

- "Basic" - use the *Basic* processor as defined in the generiter.processor library
- "beatsbassdrone" - use the *beatsbassdrone* method of the *Basic* processor (in pure Python terms this equates to calling *generiter.processor.Basic.beatsbassdrone()* through configuration)
- "tracks" - create 20 tracks in this run
- "repeats" - use up to 6 internal loop repeats for each track (controls the length of each output track for this method)

4.1.4 Generating music

Finally, we get to generate some tracks, and for this we use generiter thus:

```
generiter -L inventory.json -C compose.json
```

You can then watch as your tracks are generated into a time- and date-stamped subdirectory of your target directory. This allows you to do multiple runs without accidentally overwriting any earlier works.

Your compositions will come out looking like:

```
20210117130701
└── Basic
    ├── Basic_beatsbassdrone_00.wav
    ├── Basic_beatsbassdrone_01.wav
    ├── Basic_beatsbassdrone_02.wav
    ├── Basic_beatsbassdrone_03.wav
    ├── Basic_beatsbassdrone_04.wav
    ├── Basic_beatsbassdrone_05.wav
    ├── Basic_beatsbassdrone_06.wav
    ├── Basic_beatsbassdrone_07.wav
    ├── Basic_beatsbassdrone_08.wav
    ├── Basic_beatsbassdrone_09.wav
```

(continues on next page)

(continued from previous page)

```
    ├── Basic_beatsbassdrone_10.wav
    ├── Basic_beatsbassdrone_11.wav
    ├── Basic_beatsbassdrone_12.wav
    ├── Basic_beatsbassdrone_13.wav
    ├── Basic_beatsbassdrone_14.wav
    ├── Basic_beatsbassdrone_15.wav
    ├── Basic_beatsbassdrone_16.wav
    ├── Basic_beatsbassdrone_17.wav
    ├── Basic_beatsbassdrone_18.wav
    └── Basic_beatsbassdrone_19.wav
```

As you will see, the naming of the files and the organisation of them follows the specification of the *compose.json* file, making them easy to navigate and understand. As your compositions become bigger and more complex, this will also allow you observe/extract interesting intermediate forms.

4.1.5 The Next Iteration

That's all well and good, but it's only using 3 of the voices in your inventory. Let's explore a slightly more flexible algorithm **voices3**, which allows you to arbitrarily assign three difference voices from your inventory and then use those in exactly the same way.

Edit your *compose.json* to look like:

```
{
    "Basic" : {
    "beatsbassdrone" : {
        "tracks" : 20,
        "repeats" : 6
     },
     "voices3" : {
         "tracks" : 20,
         "repeats" : 6,
         "voices" : [ "Beats",
                      "Bass",
                      "Guitar" ]
     }
 },
     "Globals" : {
         "destination" : "<your path here>"
     }
 }
```

As you can see, all that's happened is that a new *voices3* method of *Basic* is being invoked and that method can be configured here to use an arbitrary set of 3 of the available voices; although in this example I have only replaced the *Drone* voice with *Guitar*.

Running **exactly the same** *generiter* command

```
generiter -L inventory.json -C compose.json
```

yields output arranged thus:

```
20210117132943/
└── Basic
    ├── Basic_beatsbassdrone_00.wav
```

(continues on next page)

(continued from previous page)

```
    ├── Basic_beatsbassdrone_01.wav
    ├── Basic_beatsbassdrone_02.wav
    ├── Basic_beatsbassdrone_03.wav
    ├── Basic_beatsbassdrone_04.wav
    ├── Basic_beatsbassdrone_05.wav
    ├── Basic_beatsbassdrone_06.wav
    ├── Basic_beatsbassdrone_07.wav
    ├── Basic_beatsbassdrone_08.wav
    ├── Basic_beatsbassdrone_09.wav
    ├── Basic_beatsbassdrone_10.wav
    ├── Basic_beatsbassdrone_11.wav
    ├── Basic_beatsbassdrone_12.wav
    ├── Basic_beatsbassdrone_13.wav
    ├── Basic_beatsbassdrone_14.wav
    ├── Basic_beatsbassdrone_15.wav
    ├── Basic_beatsbassdrone_16.wav
    ├── Basic_beatsbassdrone_17.wav
    ├── Basic_beatsbassdrone_18.wav
    ├── Basic_beatsbassdrone_19.wav
    ├── Basic_voices3_00.wav
    ├── Basic_voices3_01.wav
    ├── Basic_voices3_02.wav
    ├── Basic_voices3_03.wav
    ├── Basic_voices3_04.wav
    ├── Basic_voices3_05.wav
    ├── Basic_voices3_06.wav
    ├── Basic_voices3_07.wav
    ├── Basic_voices3_08.wav
    ├── Basic_voices3_09.wav
    ├── Basic_voices3_10.wav
    ├── Basic_voices3_11.wav
    ├── Basic_voices3_12.wav
    ├── Basic_voices3_13.wav
    ├── Basic_voices3_14.wav
    ├── Basic_voices3_15.wav
    ├── Basic_voices3_16.wav
    ├── Basic_voices3_17.wav
    ├── Basic_voices3_18.wav
    └── Basic_voices3_19.wav
```

So, we have not only created a set of *voices3*-derived compositions, we have also created a new set of *beatsbassdrone*-derived compositions. These are not copies of the previous set, but a completely new set using the same algorithm, but with different random selections and decisions.

At this point, when you listen to all the outputs, you may start to hear a certain *same-y* quality to some of the outputs. It should be clear that, even with these very basic algorithms, the diversity and variation in your compositions is going to depend very much on the breadth and size of the sample sets in your libraries.

4.1.6 Layering and sequencing

Having done some variations on the **Basic** algorithms (and there are more to play with when you check the code documentation), you might think that it's time to add some complexity to the music you are creating.

GenerIter obviously provides facilities for generating different layers as separate outputs. However, it would be good if there was an algorithmic way of combinging these outputs into newer, richer compositions. So that feature is also built into the system.

Here's an example composition file that illustrates how this is done.

```
{
    "Basic" : {
        "voices3" : {
            "tracks" : 20,
            "repeats" : 3,
            "voices" : [
                "Beats",
                "Bass",
                "Drone"
            ]
        }
    },
    "Solo" : {
        "generic" : {
            "tracks" : 20,
            "voice" : "Guitar"
        }
    },
    "Mix" : {
        "multitrack" : {
            "tracks" : 20,
            "voices" : {
                "Basic" : 12,
                "Solo" : -6,
                "Solo" : -6,
                "Solo" : -6
            }
        }
    },
    "Globals" : {
        "destination" : "<your path here>",
        "sequence" : [
            "Basic",
            "Solo",
            "Mix"
        ]
    }
}
```

The first section for the **Basic** processor should look familiar.

Two new processors are invoked:

- **Solo** : a draft generic mechanism for using a single voice and generating solo or lead lines.
- **Mix** : a simple multitrack mixer for creating a combined output and setting relative gain levels for the disparate voices in the mix.

To understand the change to the **Globals** section, a bit of understanding of the software structure is required and a

short lesson in some features of Python data structures.

When you created the *inventory.json* file in the earlier tutorial, what actually happened was that an internal Python data structure was converted into a JSON string representation and then written to your disc. This is done because, as well as being easily read by humans, it is also easily read by Python. This means that the entire data structure can be recreated in memory, with all of its earlier properties, by reading and parsing the file. This a very easy operation in Python.

So, when you set the **-L** option on the **generiter** command, that's what happens; an object of class *Selector* is created and used throughout the process lifetime.

One of the features of this design is that as the generative process continues and writes out compositions, each of those compositions is also registered in the *Selector* object in memory. This is done precisely so that later iterations can use the outputs from earlier iterations.

However, this throws up a problem in the way Python Python deals with the different built-in data structures in use. When you express a list, the ordering is embedded in the definition of that list: *[0, 1 ,2, 3, 4 . . . .]* which means that if the software iterates over the list, the contents will be accessed in the index order in a predictable fashion. The same is not so for a dictionary: *{ "a" : 0, "b" : 1, "c" : 2, . . . }*. The order in which entries are iterated is not guaranteed to be in any useful order, either order of insertion or sorted.

This means that if the **Mix** algorithm depends on the existence of previously-generated **Basic** and **Solo** compositions, it is necessary to tell the overall process that it needs to process the algorithms in the correct order such that when the **Mix** algorithm wants to select material, the material exists for it to be able to do so. This is achieved using the **"sequence"** field in the **"Globals"** setting. Implementated as a list, this order is guaranteed.

The **Mix** configuration also illustrates the application of output balancing. Each of the chosen voices has a mute vale expressed in dB. For this example all the tracks are unmuted. This might result in some clipping in the output, depending on the source material. This is where you can literally implement the mix levels to get the balance you want.

Beta Testers Note: This is a change of config format for the **Mix.multitrack** module from that with which you were originally testing. The *voices* are now represented as a mapping dictionary between the voice and a level rather than as a simple list of voices.

4.2 Advanced Grooving

For this tutorial, it will be assumed that the user has compiled their own inventory of samples. The voicings given here are purely illustrative and should be adapted to fit the user's own library organisation.

This tutorial will use two new algorithms *Basic.groove* and *Solo.multivoice_serial_ordered*. It also reuses the *Mix.multitrack* algorithm to integrate each component part back into a whole.

4.2.1 Creating the composer control file

The new composer control file is more complex, but follows a layering pattern with which you should now be familiar:

```
{
    "Basic" : {
        "groove" : {
            "tracks" : 50,
            "cycle" : 4,
            "voices" : [
                "Bass",
                "Beat",
```

(continues on next page)

(continued from previous page)

```
                "Percussion",
                "Pad"
            ]
        }
    },
    "Solo" : {
        "multivoice_serial_ordered" : {
            "tracks" : 50,
            "voices" : [
                "Synth",
                "Synth",
                "Piano",
                "Vox",
                "Synth",
                "Synth"
            ]
        }
    },
    "Mix" : {
        "multitrack" : {
            "tracks" : 30,
            "voices" : {
                "Basic" : 12,
                "Solo" : -3
            }
        }
    },
    "Globals" : {
        "destination" : "<your path here>",
        "tsize" : "s",
        "sequence" : [
            "Basic",
            "Solo",
            "Mix"
        ]
    }
}
```

4.2.2 Global variables

The **Globals** section has a new parameter which allows you to roughly control the overall length of the tracks in your compositions. These are specified in t-shirt sizes and are currently defined in seconds thus:

```
TSHIRT = {
    "s" : 180,
    "m" : 300,
    "l" : 480,
    "xl" : 780,
    "xxl" : 1260,
    "xxxl" : 2640
 }
```

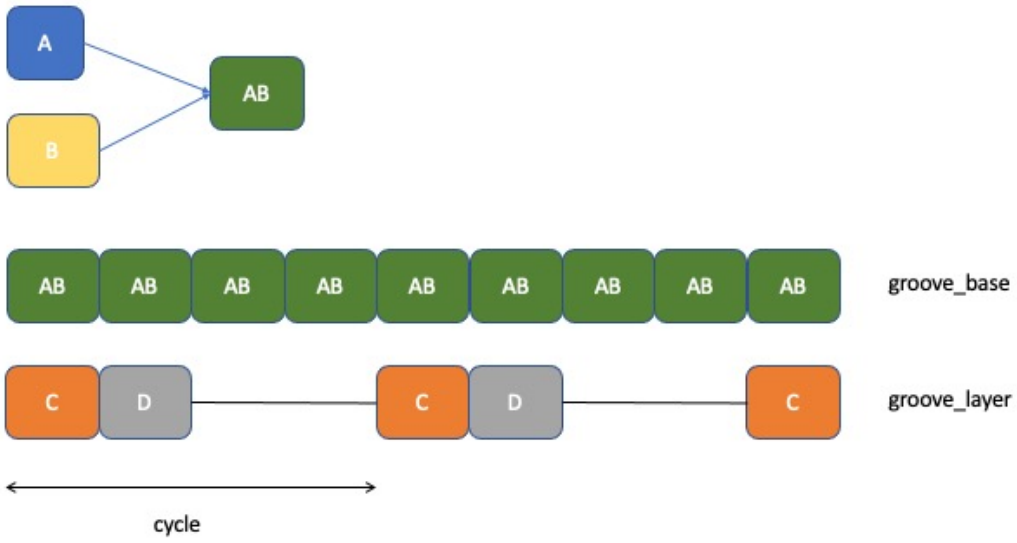
So a *tsize* of "s" represents a soft threshold of 3 minutes +/- 10% (calculated at runtime). This is a "soft" threshold because the final length of any track will be determined by the width of the segments used and the number of repeats that can be concatenated before the overall track length exceeds the limit. It is a very approximate threshold and

merely represents a promise to keep the track within an order of magnitude. Hence the use of a t-shirt sizing metaphor for approximation.

As before, the *sequence* list specifies the order in which to execute the layers, so that outputs from early layers are available to be selected into the final micx layers.

4.2.3 Basic.groove

The *Basic.groove* algorithm is used to form a rhythmic backbone to the composition. The configuration here will look very similar to the *Basic.voices3* used in the earlier tutorial, but the algorithm is more sophisticated and general purpose.



The number of voices can be 3 or more. For this example I am using 4, which we can think of as **A**, **B**, **C** and **D** corresponding to their declaration position in the configuration list.

Segments **A** and **B** are randomly selected from their inventory categories. These are then length-aligned. That is, whichever is the shorter of the two is padded with exactly the right amount of silence to make the frame lengths of both segments equal. These are then overlayed to form a composite rhythmic unit referred to as **AB**.

The basic rhythmic beat is created by replicating a sequence of these **AB** segments end-to-end until the track size limit is surpassed. This is the *groove_base*.

The *cycle* parameter specifies the number of these **AB** components tahe represent a higher level repetition unit called the *cycle* (obviously).

Subsequent voices are also length-aligned (or curtailed) to match the size of the **AB** unit. A new segment, *groove_layer* is created, into which the later voices **C**, **D**, etc. are injected in a cyclic sequence, aligned precisely to the *groove_base* cycles and beats.

Finally, the *groove_base* is overlayed with the *groove_layer* to create a composite whole track which is output as an audio file.

4.2.4 Solo.multivoice_serial_ordered

This algorithm, as its name should indicate, is just one possible approach amongst a whole forest of potential variants. As such it is provided merely as an exemplar and starting point for the user's own explorations into algorithmic creativity.

Each of the voices in the configuration list is randomly selected from their inventory categories. At this point we have a fixed selection of solo voices for the rest of the track composition.

For each voice in sequence, the selected segment is padded with silence both front and back in a randomised non-uniform manner. The resulting segment is appended to the solo track and the sequence cycled until the track length exceeds its soft limit. This will be of the same order of magnitude as the limit used in the *Basic.groove* algorithm, but not identical or coupled in any way.

The solos are then output as audio files, existing independently.

Clearly, for an "s" sized track, setting 24 voices in the configuration is likely to result in a lot of redundant voicings that just never appear in the final track. Conversely, building an "xxl" solo out of just one or two voices will get very repetitive and uninteresting over time.

4.2.5 Mix.multitrack

This performs the same function it did in the earlier basic workflow tutorial, combining randomly selected tracks from the generated *Basic* and *Solo* populations and generating audio output files of the results. The level setting values are in dB, applied to the corresponding voice: the *Basic* voice is boosted by 12 dB, the *Solo* is muted by 3 dB. These values represent a reasonable median starting point, but they will undoubtedly need adjusting for your own sample library.

4.3 Roll Your Own Algorithm

(For this tutorial, it is assumed that the user has a familiarity with Python programming terms and idioms.)

One of the design objectives of this module is to provide a framework into which users can extend and customise the algorithmic capabilities of the GenerIter command without touching the supplied code.

This is achieved through the use of the *GenerIter.Process* as an abstract base class from which new processors can be derived, encompassing the use of the Selector and Config classes and drivine the *generiter* app purely through data configuration.

For this tutorial a dummy *Process*-derived class will be implemented to illustrate the mechanism. For real generative audio algorithmic development, the user is encouraged to examine the existing algorithms and processors, and read the **pydub** API documentation, to understand and use the example implementations as references for further development. Feel free to cut and paste implementation fragments if they are useful.

4.3.1 Create your library module

In this example we want to create a new algorithm. Let's call it **local01**.

To do this, the algorithm needs to be attached to a **Process**-derived object, which we'll call **Myprocess**.

In order to access the object module at runtime, we need to put it in a library module. We'll call this **mylib**.

Pythonistas will be able to do this easily enough from the description above, but even then they are likely to make errors. To simplify the process and minimise basic system errors,the **GenerIter** module comes with its own code generator for local algorithm development, called **generalg**.

To achieve the structures described above:

```
generalg -L mylib -M myprocess -A local01
```

will ensure that a directory named **testlib** is created.

Within **mylib**, the app ensures that the **__init__.py** and **Myprocess.py** files exist.

Note : Only the library name is case-sensitive. All other parameters will be pushed into lower case and the module name will be capitalized.

The **Myprocess.py** will contain:

```
from GenerIter.process import Process

class Myprocess(Process):

    def __init__(self):
        super().__init__()


    def local01(self):
        print("Executing local01")
        # Your code goes here
```

4.3.2 Testing your module

This is easy to test.

Create a simple **test.json** file:

```
{
    "Myprocess" : {
        "local01" : {
        }
    }
}
```

In the following example, the name of the inventory file is irrelevant - although it has to be there to satisfy the constraints of the application, no selection processing is specified so no output files will get generated.

To test the new library module:

```
generiter -L someinventoryfile.json -C test.json -P mylib
```

This should produce the following output:

```
mylib
Process()
Myprocess.local01()
Executing local01
```

Congratulations!!! your new library and modules work.

It's not very interesting at the moment, but you have created a new algorithm. All you need to do now is fill in the missing code.

4.3.3 Further expansions

The **generalg** utility is designed so that you can add new algorithms to existing processors:

```
generalg -L mylib -M myprocess -A local02
```

Thus:

```
from GenerIter.process import Process

class Myprocess(Process):

    def __init__(self):
        super().__init__()


    def local01(self):
        print("Executing local01")
        # Your code goes here


    def local02(self):
        print("Executing local02")
        # Your code goes here
```

Add new processor modules to your library:

```
generalg -L mylib -M anotherprocess -A algorithm
```

Or create a completely new library:

```
generalg -L newlib -M anotherprocess -A algorithm
```

However it is NOT sufficiently smart to prevent you adding a repeat copy of the same algorithm name to the same processor module:

```
generalg -L newlib -M anotherprocess -A algorithm
generalg -L newlib -M anotherprocess -A algorithm
```

Will create:

```
from GenerIter.process import Process

class Anotherprocess(Process):

    def __init__(self):
        super().__init__()


    def algorithm(self):
        print("Executing algorithm")
        # Your code goes here


    def algorithm(self):
        print("Executing algorithm")
        # Your code goes here
```

Which will throw a fatal Python exception when you try to use it.

CHAPTER

FIVE

CONSOLE APPS

5.1 genercat

```
usage: genercat [-h] -C C [-D D]

optional arguments:
    -h, --help  show this help message and exit
    -C C        Category search pattern
    -D D        Destination category name
```

The **genercat** console application is a helper utility for managing sample libraries. It can be used when you have a large, flat directory full of sample files and wish to apply some structure on the collection. This is particularly useful when you wish to create different voices or subcategories in you inventory (see below). You can use the app to assign sets of samples into a named subdirectory.

So, for example, if you wished to assign all your **Piano**-related content into a **Piano** subdirectory:

```
genercat -C Piano
```

will create a *Piano* subdirectory (unless it already exists) in the current directory, and then move all the sample files whose name contains the string "Piano" into that subdirectory.

A more flexible use case is where you may want to organise different sample file types under a common category. One example would be a directory that contains a range of samples relating to different percussive sounds which you wanted to group together:

```
genercat -D Percussion -C Drum
genercat -D Percussion -C Hi-Hat
genercat -D Percussion -C Snare
genercat -D Percussion -C Topper
genercat -D Percussion -C Riser
```

This takes all the files that match to patterns defined by the **-C** specifier and puts them in a common destination subdirectory as defined by the **-D** specifier.

The rule is that if only **-C** is specified, the destination subdirectory mirrors that selection. Otherwise, the files are moved into the subdirectory specified by **-D**.

This means that:

```
genercat -C Drum
genercat -D Drum -C Drum
```

are functionally identical.

5.2 generinv

```
usage: generinv [-h] [-I I] [-L L] -o O

optional arguments:
    -h, --help  show this help message and exit
    -I I        Source for inclusion in searches
    -L L        Source for inclusion in loads
    -o O        Output file root name
```

5.3 generiter

```
usage: generiter [-h] [-I I] [-L L] -C C

optional arguments:
    -h, --help  show this help message and exit
    -I I        Source for inclusion in searches
    -L L        Source selection file for inclusion in loads
    -C C        Configuration file for algorithms
```

5.4 generalg

```
usage: generalg [-h] -A A -M M -L L

optional arguments:
   -h, --help  show this help message and exit
   -A A        Algorithm name
   -M M        Module name
   -L L        Library name
```

This is a helper utility that generates code stubs when you want to create a local module with hooks onto which new algorithms can be developed.

Thus:

```
generalg -L testlib -M testmod -A testalg
```

will ensure that a directory named **testlib** is created.

Within **testlib**, the app ensures that the **__init__.py** and **Testmod.py** files exist.

Note : Only the library name is case-sensitive. All other parameters will be pushed into lower case and the module name will be capitalized.

The **Testmod.py** will contain:

```
from GenerIter.process import Process

class Testmod(Process):

    def __init__(self):
        super().__init__()
```

(continues on next page)

(continued from previous page)

```
    def testalg(self):
        print("Executing testalg")
        # Your code goes here
```

CHAPTER

SIX

REFERENCE

6.1 GenerIter

6.1.1 GenerIter package

Subpackages

GenerIter.app package

Submodules

GenerIter.app.algorithm module

App that is used to generate code stubs for local algorithmic development.

```
usage: generalg [-h] -A A -M M -L L

optional arguments:
   -h, --help  show this help message and exit
   -A A        Algorithm name
   -M M        Module name
   -L L        Library name
```

Thus:

```
generalg -L testlib -M testmod -A testalg
```

will ensure that a directory named **testlib** is created.

Within **testlib**, the app ensures that the **__init__.py** and **Testmod.py** files exist.

The **Testmod.py** will contain:

```
from GenerIter.process import Process

class Testmod(Process):

    def __init__(self):
        super().__init__()


    def testalg(self):
```

(continues on next page)

(continued from previous page)

```
        print("Executing testalg")
        # Your code goes here
```

class GenerIter.app.algorithm.**Algorithm**

Bases: *GenerIter.app.clibase.CLIBase*

alg_template = '\n def {ALG}(self):\n print("Executing {ALG}")\n # Your code goes here

build_class()

build_lib()

build_method()

klass_template = 'from GenerIter.process import Process\n\nclass {KLASS}(Process):\n\n

parseArguments()

There is no implementation of this function for the abstract base class.

Raises *GenerIter.excepts.GINotImplementedErr* –

process()

There is no implementation of this function for the abstract base class.

Raises *GenerIter.excepts.GINotImplementedErr* –

GenerIter.app.categorise module

App that is used to subcategorise a sample set according to a string search parameter.

The app will iterate through all the files in the current directory looking for files containing the category search pattern substring (the **-C** argument).

If the category search pattern is found in a filename, the file is moved into a subdirectory, which will be created if necessary. The name of the subdirectory will either correspond to the category search string or is specified by the **-D** argument.

Thus:

```
genercat -C Drums
```

and

```
genercat -C Drums -D Drums
```

are functionally identical, whereas

```
genercat -C Drums - D Percussion
```

puts the same files into a different subdirectory.

class GenerIter.app.categorise.**Categorise**

Bases: *GenerIter.app.clibase.CLIBase*

parseArguments()

Parses the CLI arguments for the app.

```
usage: genercat [-h] -C C [-D D]

optional arguments:
-h, --help  show this help message and exit
-C C        Category search pattern
-D D        Destination category name
```

process()
The execution function for the app.

The basic flow:

1. A destination subdirectory is created, unless it already exists.
2. The app then examines all the files in the current directory.
3. If the specfied category search substring is found in the filename, that file is moved into the subdirectory.

GenerIter.app.clep_algorithm module

GenerIter.app.clep_algorithm.**main**()
The Command Line Entry Point for the packaged GenerIter.app.inventory app.

GenerIter.app.clep_categorise module

GenerIter.app.clep_categorise.**main**()
The Command Line Entry Point for the packaged GenerIter.app.categorise app.

GenerIter.app.clep_generator module

GenerIter.app.clep_generator.**main**()
The Command Line Entry Point for the packaged GenerIter.app.generator app.

GenerIter.app.clep_inventory module

GenerIter.app.clep_inventory.**main**()
The Command Line Entry Point for the packaged GenerIter.app.inventory app.

GenerIter.app.clibase module

Abstract base class for Command Line Interface apps in the GenerIter ecosystem

Copyright 2020 Thomas Jackson Park & Jeremy Pavier

class GenerIter.app.clibase.**CLIBase**
Bases: object

parseArguments()
There is no implementation of this function for the abstract base class.

Raises *GenerIter.excepts.GINotImplementedErr* –

process()
There is no implementation of this function for the abstract base class.

Raises *GenerIter.excepts.GINotImplementedErr* –

GenerIter.app.generator module

App to generate music.

Copyright 2020 Thomas Jackson Park & Jeremy Pavier

class GenerIter.app.generator.**Generator**
Bases: *GenerIter.app.clibase.CLIBase*

loadConfiguration()

loadSelections()

parseArguments()
There is no implementation of this function for the abstract base class.

Raises *GenerIter.excepts.GINotImplementedErr* –

process()
There is no implementation of this function for the abstract base class.

Raises *GenerIter.excepts.GINotImplementedErr* –

GenerIter.app.inventory module

App to catalogue and select source files into a configuration.

Copyright 2020 Thomas Jackson Park & Jeremy Pavier

class GenerIter.app.inventory.**Inventory**
Bases: *GenerIter.app.clibase.CLIBase*

parseArguments()
There is no implementation of this function for the abstract base class.

Raises *GenerIter.excepts.GINotImplementedErr* –

process()
There is no implementation of this function for the abstract base class.

Raises *GenerIter.excepts.GINotImplementedErr* –

Module contents

GenerIter.processor package

Submodules

GenerIter.processor.Basic module

Generator class for some basic Process-based algorithms for rhythmic generation.

class GenerIter.processor.Basic.**Basic**

Bases: *GenerIter.process.Process*

beatsbassdrone()

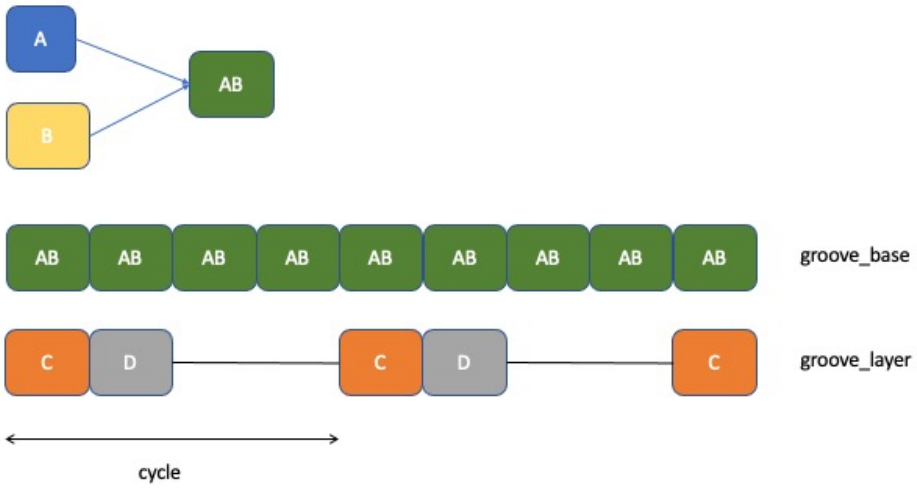
This is a developmental prototype variant of the algorithm01 in the original DJProcessor code.

As the name suggests, it expects to find categories in the sample inventory mapped into the *Beats*, *Bass* and *Drone* voices. Otherise this algorithm will fail.

> **Warning:** This is example code used during early development, and should really only be used for reference and study.

groove()

The *Basic.groove* algorithm is used to form a rhythmic backbone to a composition. This algorithm attempts to make sensible alignment choices for the samples in use. It is considered a *production* algorithm.



The number of voices can be 3 or more. In this example we are using 4, which we can think of as **A**, **B**, **C** and **D** corresponding to their declaration position in the configuration list.

Segments **A** and **B** are randomly selected from their inventory categories. These are then length-aligned. That is, whichever is the shorter of the two is padded with exactly the right amount of silence to make the frame lengths of both segments equal. These are then overlayed to form a composite rhythmic unit referred to as **AB**.

The basic rhythmic beat is created by replicating a sequence of these **AB** segments end-to-end until the track size limit is surpassed. This is the *groove_base*.

The *cycle* parameter specifies the number of these **AB** components tahe represent a higher level repetition unit called the *cycle* (obviously).

Subsequent voices are also length-aligned (or curtailed) to match the size of the **AB** unit. A new segment, *groove_layer* is created, into which the later voices **C**, **D**, etc. are injected in a cyclic sequence, aligned precisely to the *groove_base* cycles and beats.

Finally, the *groove_base* is overlayed with the *groove_layer* to create a composite whole track which is output as an audio file.

Parameters

- **tracks** (*int*) – number of times the process is to run.
- **voices** (*list*) – list of voice categories to select and use in the track
- **cycle** (*int*) – length of a repeat cycle in terms of groove_base *beats*

Example:

```
{
    "Basic" : {
        "groove" : {
            "tracks" : 50,
            "cycle" : 4,
            "voices" : [
                "Bass",
                "Beat",
                "Percussion",
                "Pad"
            ]
        }
    }
}
```

Raises *GenerIter.excepts.GIParameterErr* –

voices()

This developmental prototype algorithm uses the same algorithm as voices3, but allows the user to designate the 3 or more voices to be used in the compose file.

> **Warning:** This is example code used during early development, and should really only be used for reference and study.

voices3()

This developmental prototype algorithm uses the same algorithm as beatsbassdrone, but allows the user to designate the 3 voices to be used in the compose file.

The number of voices allowed is hard-coded to 3 only.

> **Warning:** This is example code used during early development, and should really only be used for reference and study.

voices_shifted()

This algorithm uses the same algorithm as voices, but implements a sequential shift as each new voice is incorporated.

> **Warning:** This is example code used during early development, and should really only be used for reference and study.

GenerIter.processor.Mix module

Generator class for some Process-based mixing algorithms.

Copyright 2020 Thomas Jackson Park & Jeremy Pavier

class GenerIter.processor.Mix.**Mix**
Bases: *GenerIter.process.Process*

multitrack()

GenerIter.processor.Solo module

Generator class for some Process-based soloing algorithms.

Copyright 2020 Thomas Jackson Park & Jeremy Pavier

class GenerIter.processor.Solo.**Solo**
Bases: *GenerIter.process.Process*

SOLOMAP = [{'upper': 10000, 'variation': (800, 2400), 'fades': 100, 'back_silence':

generic()

multivoice_serial_ordered()

Module contents

Submodules

GenerIter.config module

Class to configure generator algorithms.

Copyright 2020 Thomas Jackson Park & Jeremy Pavier

class GenerIter.config.**Config**(*confpath=None*)
Bases: object

load(*inpath*)

subcats()

GenerIter.excepts module

Domain-specific Exception classes for GenerIter

Copyright 2020 Thomas Jackson Park & Jeremy Pavier

exception GenerIter.excepts.**GIErr**
Bases: Exception

Base class for domain-specific exceptions.

exception GenerIter.excepts.**GINotImplementedErr**
Bases: *GenerIter.excepts.GIErr*

Class for propogating not implemented errors.

exception GenerIter.excepts.**GIParameterErr**
Bases: *GenerIter.excepts.GIErr*

Class for propogating parameter errors.

exception GenerIter.excepts.**GIResourceErr**
Bases: *GenerIter.excepts.GIErr*

Class for propogating resource errors.

exception GenerIter.excepts.**GIValidationErr**
Bases: *GenerIter.excepts.GIErr*

Class for propogating validation errors.

GenerIter.factory module

Class that constructs the correct processor object.

Copyright 2020 Thomas Jackson Park & Jeremy Pavier

class GenerIter.factory.**ProcessorFactory**(*vname*, *pname*, *fname*, *procmodule*)
Bases: object

configure(*invent*, *config*, *dest*, *form*, *size*)

property klass

property method

process()

setMethod(*fname*)

GenerIter.process module

Abstract base class for all Process-based generator algorithms.

Copyright 2020 Thomas Jackson Park & Jeremy Pavier

class GenerIter.process.**Process**(*prefix=None*)
Bases: object

This is the abstract base class from which all other processors are derived.

As such it implements the core interface as well as several important generic helper services which can simplify the derived algorithm implementations.

SUPPORTED_FORMATS = ['wav']

TSHIRT = {'l': 480, 'm': 300, 's': 180, 'xl': 780, 'xxl': 1260, 'xxxl': 2640}

bracket(*segment*, *frontmult=1.0*, *backmult=1.0*)

configure(*inventory*, *configuration*, *destination*, *forrmat*, *tsize*)

deamplify(*segment*, *limits*)

declick(*segment*, *value=10*)
This is a helper function with which a fade/rise can be applied to each end of an AudioSegment to reduce the potential for 'clicking' when they are connected end-to-end.

Can also be used, with longer fade times, if large track fades are required.

Parameters **value** (*int*) – number of milliseconds across which the segment will be faded from full gain to zero (typically 10 seems to work well).

Returns AudioSegment

default()

getsegment(*sample*, *limits*, *fade*)

getsegmentm(*sample*, *muted*, *fade*)

intwidth(*value*)

padtolength(*segment*, *length*, *fader*, *front=False*)

supported(*value*)

threshold()

write(*algorithm*, *counter*, *source*)

GenerIter.selector module

Class to catalogue and select source files.

Copyright 2020 Thomas Jackson Park & Jeremy Pavier

class GenerIter.selector.**Selector**(*searchpath=None*, *loadpath=None*)
Bases: object

Represents the inventory of samples available for algorithmic processing.

Listing 1: Example constructor usage

```
from GenerIter.selector import Selector
# Selector
selector = Selector(searchpath=pathstring,loadpath=loadfile)
```

insert(*source*, *include=True*)
Attempt to insert a source into the Selector configuration

load(*lpath=None*)
Load a previously-saved Selector configuration.

This method is uniquely additive in that it can be run repeatedly and any repeats are silently overwritten.

Parameters **lpath** (*str*) – path to the loadable JSON file containing a saved Selector state.

Listing 2: Example usage

```
from GenerIter.selector import Selector
# Empty Selector
selector = Selector()
# Load a previously-saved inventory file
selector.load(lpath=pathstring)
```

search(*spath=None*)
Walk a directory tree to add to the Selector configuration.

This method walks the specified directory tree and adds any discovered WAV files to its inventory. This method is uniquely additive in that it can be run repeatedly across the different or the same trees. Uniqueness is enforced during this process, so any repeats are silently overwritten. Any files encountered that do not match the criteria for a WAV file are ignored.

Parameters **spath** (*str*) – path to the root of the searchble directory tree.

Listing 3: Example usage

```
from GenerIter.selector import Selector
# Empty Selector
selector = Selector()
# Search a directory tree
selector.search(spath=pathstring)
```

selectRandom(*key*)

Method for getting a random selection from within a sub-category of the Selector.

This method will attempt to randomly choose an entry for the specified sub-category in the Selector.

It will fail if there are no `true` enabled entries in the sub-category or if the randomised function repeatedly fails to find a `true` enabled entry because they are too sparse.

The number of attempts is limited by the size of the sub-category array.

Parameters **key** (*str*) – the name of a sub-category within the Selector's structure.

Raises **RDJParameterErr** – if unable to select a return value.

subcats()

Get the list of top-level sub-categories in the Selector.

Returns [] (str)

Listing 4: Example usage

```
# Enumerate the sub-categories
cats = selector.subcats()
for cat in cats:
    # Get the sub-category
    category = selector[cat]
```

GenerIter.source module

Classes to represent references to GenerIter source files.

Copyright 2020 Thomas Jackson Park & Jeremy Pavier

class GenerIter.source.**FlacSource**(*dpath=None*, *dexist=False*)

Bases: *GenerIter.source.Source*

Derived class specialised for FLAC source files.

This is a placeholder for future developments.

class GenerIter.source.**Mp3Source**(*dpath=None*, *dexist=False*)

Bases: *GenerIter.source.Source*

Derived class specialised for MP3 source files.

This is a placeholder for future developments.

class GenerIter.source.**Source**(*path=None*, *exist=False*)

Bases: object

Generic file source representation.

Parameters

- **path** (*string*) – string representation of the absolute or relative path to the target file.
- **exist** (*boolean*) – flag setting to test for the file's existence when the reference object is instantiated.

Raises ***GIResourceErr*** – if `exist == True` and the file does not exist.

property dname
Returns the name of the file's directory.

exists()
Test if the file referred to exists.

Returns boolean

property ext
Returns the filename extension of the file

property fname
Returns the basename of the file.

isValidExtension(*ref=None*)
Tests if the filename extension matches an expected value.

Indicates whether the Source or derived object carries the same file extension, either as upper- or lower-case forms.

Parameters **ref** (*str*) – the extension value to compare against.

Returns boolean

property path
Accessor to the value of full path value

property root
Returns the root name of the file.

class GenerIter.source.**WavSource**(*dpath=None*, *dexist=False*)
Bases: *GenerIter.source.Source*

Derived class specialised for WAV source files.

WAV is the only currently supported format.

GenerIter.util module

Useful unencapsulated functions to be reused across the domain.

Copyright 2020 Thomas Jackson Park & Jeremy Pavier

GenerIter.util.**debug**(*astring*)
Only print if in DEBUG mode.

Parameters **astring** (*str*) – any valid Python string

GenerIter.util.**debug_except**(*inst*)
Only print exception diagnostics if in DEBUG mode.

Parameters **inst** – any valid exception instance

GenerIter.util.**jStr**(*struct*)
Default JSON output human-readable string format.

The output string is formatted for ease of reading, with an indent value of 4 chars, using standard separators and all fields sorted by name at their appropriate level.

Parameters **struct** – arbitary Python iterable data structure i.e. list or dict.

Returns string

GenerIter.util.**jsonSerial**(*obj*)
JSON serializer for objects not serializable by default json code.

Currently supports datetime objects.

Parameters **obj** (*any type*) – arbitary Python object or type.

Returns string

Raises **TypeError** – if the object is not serializable.

GenerIter.util.**localTimestamp**(*some_time=None*, *time_format=None*)
Return a specified UTC time, formatted by the given string.

Parameters

- **some_time** – a datetime object (default None uses datetime.utcnow())
- **time_format** – format to covert a datetime object to string (default None uses %Y%m%d%H%M%S)

Returns A timestamp string

GenerIter.util.**mkdir_p**(*path*)
Replicates mkdir -p functionality.

For a given path, any missing directories are created to ensure the full path exists

Parameters **path** (*str*) – absolute path to the target directory.

Raises **OSError** – if the path already exists as a file, or the target directory cannot be created because of a permissions error.

GenerIter.util.**nextPowerOf2**(*x*)

GenerIter.util.**shCmd**(*cspec*, *trace=False*)
Executes a shell command and returns a string list of the output.

Parameters

- **[]** (*cspec*) – array of command line options and parameters.
- **trace** (*boolean*) – flag to set for text output if required (default : False)

Returns [] (str) if trace == True else None

Raises **CalledProcessError** – if the subprocess call fails

GenerIter.util.**utf8**(*array*)
Preserves byte strings, converts Unicode into UTF-8.

Parameters **array** (*bytearray or str*) – input array of bytes or chars

Returns UTF-8 encoded bytearray

Module contents

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

7.1 Contacts

7.1.1 Language and installation support

For support relating to Python and OS-specific installation issues, please write to Phil Cryer : *phil at philcryer dot com*.

7.1.2 GenerIter support and information

See the GenerIter GitHub repository for source code.

Write to us : *generiter at gmx dot com* with any questions, bugs or suggestions relating to GenerIter.

PYTHON MODULE INDEX

g

INDEX

I

J

K

L

M

N

P

R

S

T

U

V

W