**‘DMSG (Dark Matter Signal Generation)’ Package Overview:**

The DMSG package is essentially a fancy collection of storage and data generation classes/codes to be used for generating photon spectra from a pre-defined Dark Matter model. Currently the code only contains data generation code for DDM spectra, specifically those found defined by the constraints given in Boddy and Al. The data generation as such follows heavily the methodology of their Lines and Boxes paper. I should note that the data generation I implemented for the DDM decay/annihilation signals is photon by photon as opposed to a more statistical method such as the one found in [1]. Mock data to be organized must be in the form of a list of photon energies determined by the dark matter model in question.

This does not limit the capability of the code to produce other spectra however because only the data generation is model specific. The main parts of the software are the organizational pieces designed to store spectral counts, which is model independent so long as you pass in a list of photon energies coming from the model to be analyzed. This section is organized according to the “normal” use of the DMSG package. That is, we cover the overall picture first then the largest container Generation after which we work our way into the DDM Generation algorithms, the Bin objects and then we finally put it together to plot a photon spectrum for DDM. Additionally the Start file is covered as well. By the end of this you should be familiar with the basic flow of the code.



I’ll describe the storage technology of DMSG first. Two classes handle ALL of the event organization and storage. These are the Generation class and the Bin class. Generation objects, Bin objects and photon energies work together like Russian doll sets. Each Generation object houses a list of Bin objects and each Bin object houses the number of photon’s inside of it. (Note however the Bin object does not store the list of photon energies explicitly.) How the Bin object decides the number photons contained in it and whether those photons are background or DM signals is explained later in the Bin section.



**Generation Objects**

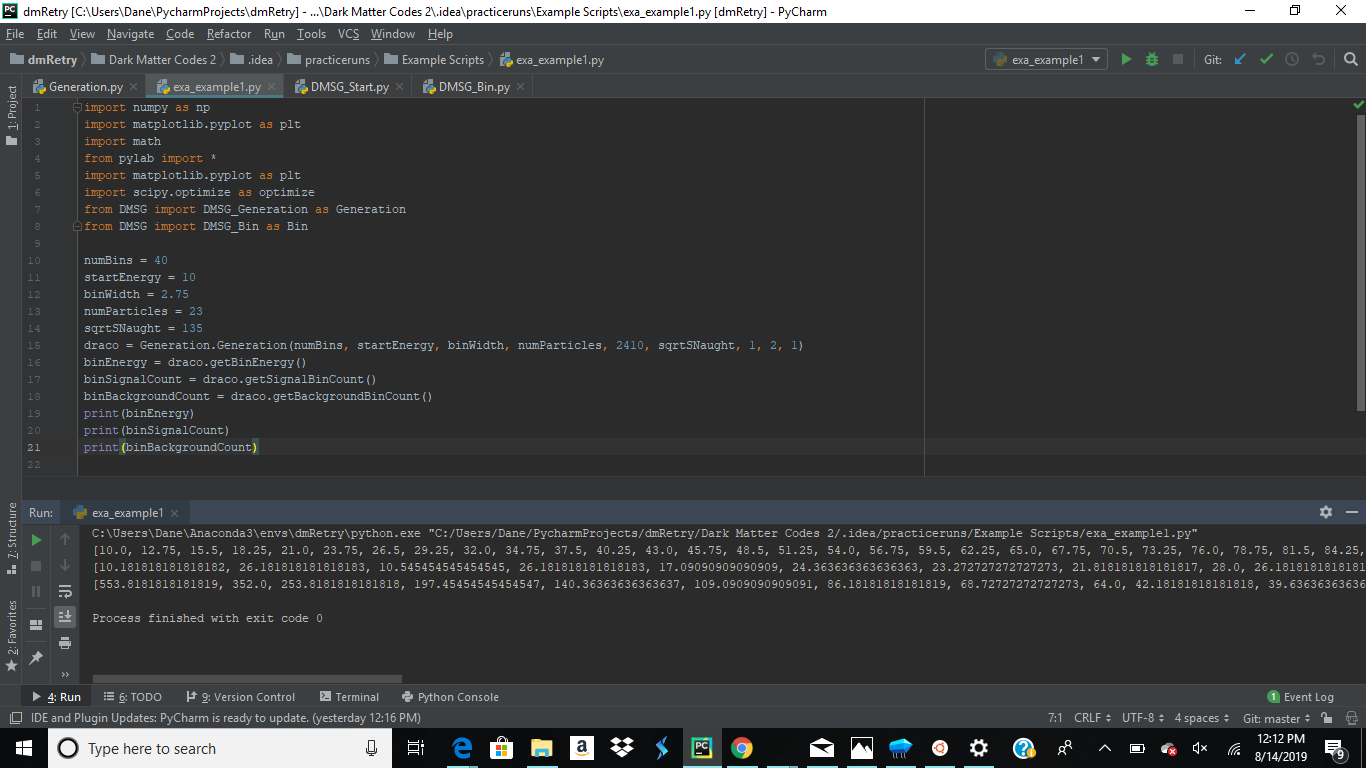
Class Generation(self,numBins,startEnergy,binWidth,totalNum,

totalSignals,sqrtSNaught,delta,deltaSSquare,eta)

Generation objects as explained above handle the storage and access of a list of Bins. This chapter will cover the main points of how Generation is used and how it can be extended to accommodate new dark matter models/ backgrounds.

Creating a new Generation object is the first step in the program. As soon as one is instantiated a list of Bins is automatically created and stored inside of Generation. The rules for how you instantiate those Bins are called from the Generation class constructor. Typically, the background counts are determined right as the bins themselves are created while the signal counts must be generated differently (more on this later).

User access to the bin energies, bin counts (background or decay/annihilation signal) and other information is also done through an instantiated Generation object. In summary the Generation class is a way to store and access bin data. Example1 in the Example folder demonstrates Generation object creation and using its class method *getBackgroundBinCount* to return a list of bin energies, signal counts and background event counts:

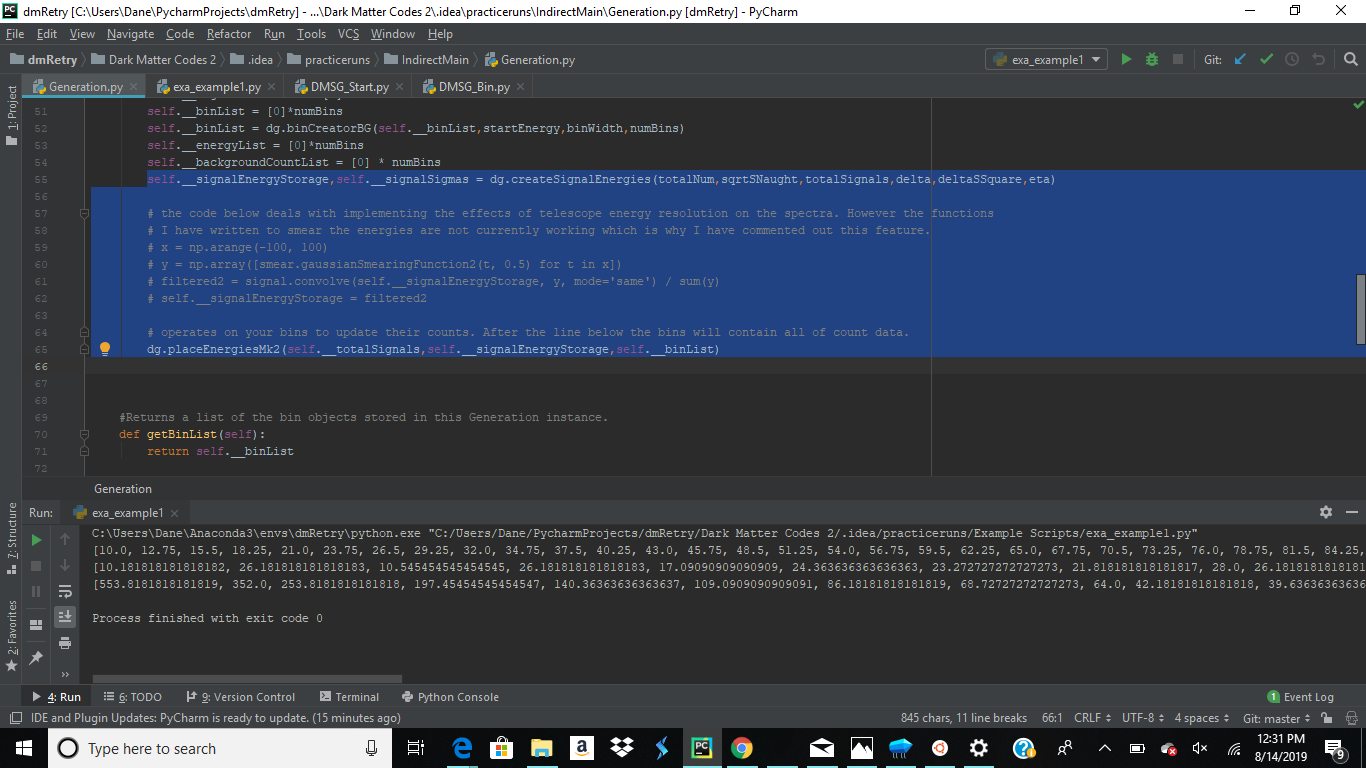


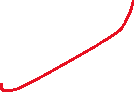
**New Dark Matter Models**

Now let’s move on to the nitty gritty details of Generation objects. My original design was to use an “abstract” class called Generation from which model-specific Generation objects could inherit from. That being said, I was not able to complete this feature in time and the Generation class in the code right now is specific to the analysis in the Line’s and Boxes paper. If you want to “plug in” a different dark matter model to generate photons I suggest you do the following:

* Create code which generates the requisite amount of photons for discovery.
* Make sure that code implements the photon creation by creating a list of photon energies ordered from least to greatest.
* Have a method that places the photon energies in the proper bins of the bin list in the Generation object. Note that I assume your method takes in the Generation object’s bin list as an argument and works directly on the same copy (the pythonic way). Doing all of this allows you to make use of the object method *getSignalBinCount.*

Inside the Generation class constructor we would place a call to a method capable of the things outlined above. An example of this is shown here in blue (ignore the commented out section), where createSignalEnergies() and placeEnergiesMk2() are the methods handling photon creation and placement respectively.



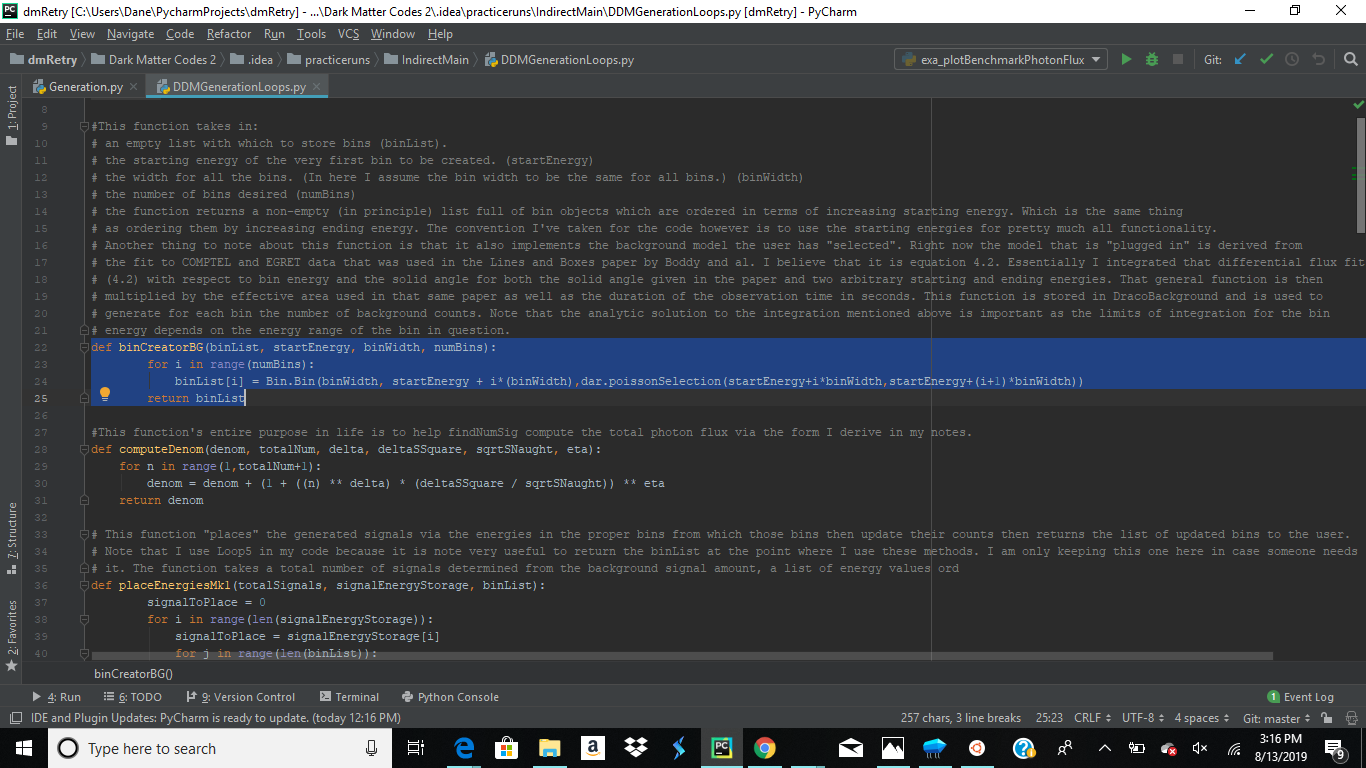


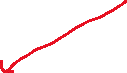
**New Background Models**

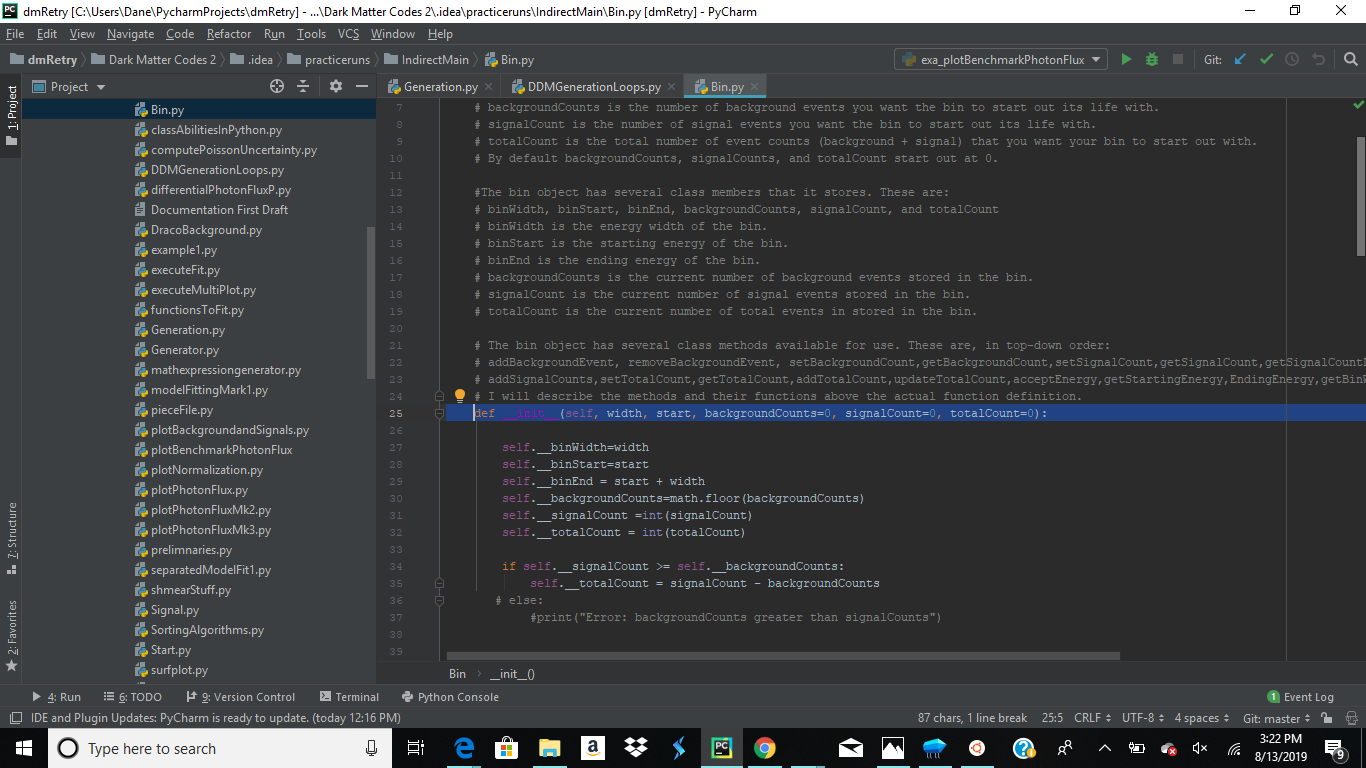
Plugging in a new background model can be different from plugging in a new dark matter model depending on if you have access to an equation describing a background. The Generation code currently implements the background of Draco via equation 4.2 in Lines and Boxes. I did not generate a list of photon energies in the usual way because equation 4.2 when integrated depends on the starting and ending energies of a specific bin. In other words 4.2 is meant to be used bin by bin to generate expected counts as opposed to being used to generate things photon by photon.

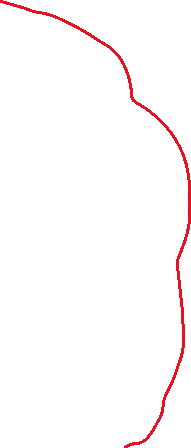


A similar problem occurs if you have a different background with an equation whose predictions are bin specific. In order to account for this, Bin objects have the option to be instantiated with a background/signal count instead of having the default counts of 0. This is shown here:









More details on how the flexibility of Bin objects are covered in the next chapter.

**Class/Object Variables and Methods**

**Class Variables:**

\_\_numBins = 0  
\_\_binWidth = 2  
\_\_backgroundCountList = []  
\_\_signalCountList = []  
  
\_\_energyList = []  
\_\_signalEnergyStorage = []

**Object Variables:**

* \_\_binWidth
* \_\_endEnergyList
* \_\_totalSignals
* \_\_signalCountList
* \_\_binList
* \_\_energyList
* \_\_backgroundCountList
* \_\_signalEnergyStorage
* \_\_signalSigmas

**Methods:**

* getBinList(self):
* getBackgroundBinCount(self):
* getSignalBinCount(self):
* getBinEnergy(self):
* getEndingEnergy(self):
* getUncertaintyList(self):
* getSignalEnergyStorage(self):

**DDM Generation Loops/ Alternative Algorithms**

DDM Generation Loops is a file containing all the data generation and storage algorithms needed to plug into Generation. If the user wants a different model to be analyzed a file like this containing methods for data generation, bin list manipulation and signal storage into a bin list must be written. I will not go into the details of the file very much other than listing the methods it contains.

**Methods:**

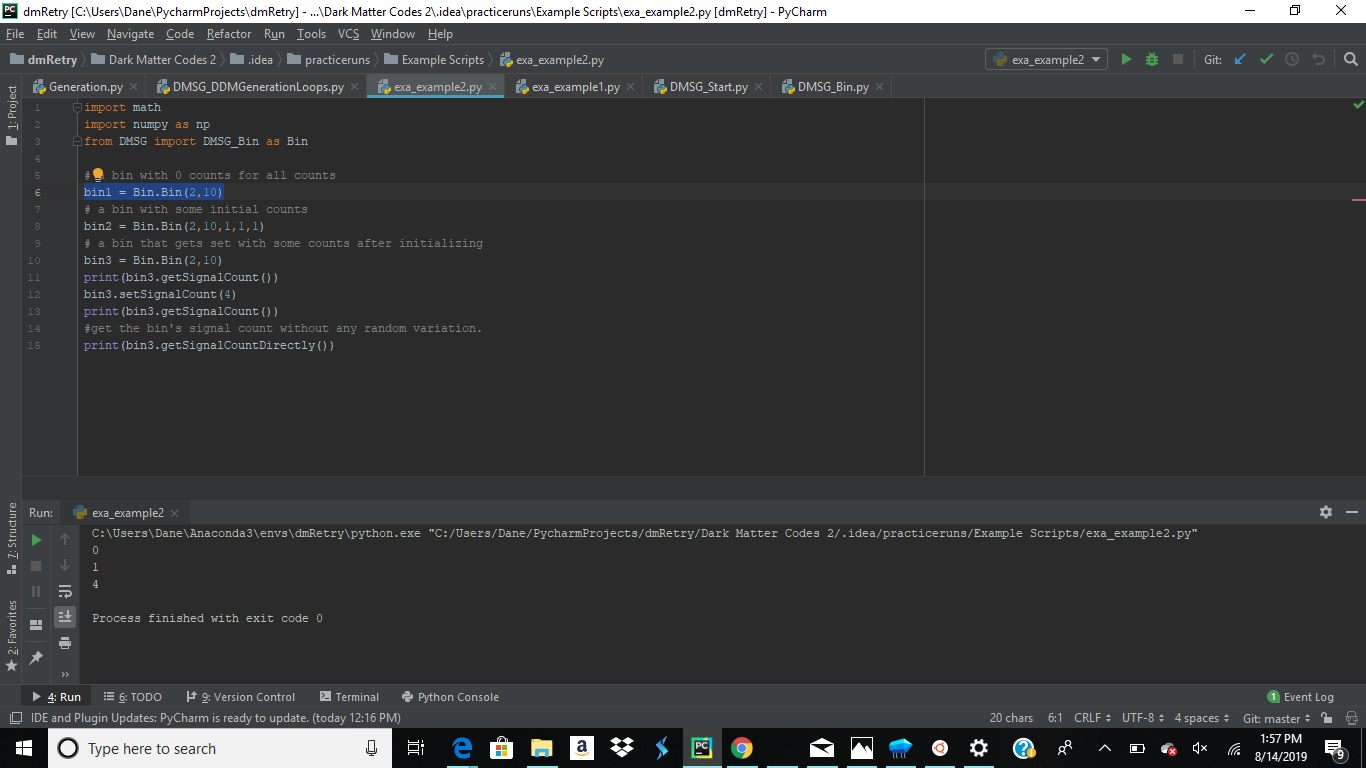
* binCreatorBG()
* computeDenom()
* placeEnergiesMk1()
* findNumSig()
* createStorageDictionary()
* fillOutArrays()
* createSignalEnergies()
* placeEnergiesMk2()

**Bin Objects**

Class Bin(self, width, start, backgroundCounts=0, signalCount=0, totalCount=0)

Bin Objects are the workhorse of the DMSG package. They keep track of the background,signal, and total counts within it as well as the range of energies that it can store. Almost all of the functionality required to accept a signal is contained within it. Bin objects work with DDMGenerationLoop’s method *placeEnergiesMk2* to place signalsvia its two class methods: *acceptEnergy*, *addSignalCounts.* Bin objects also have additional features that add flexibility to their usage.

Instantiating a basic Bin object is very straightforward and is demonstrated in the first piece of example2 in the examples folder:



After instantiation the bin object has the ability to set,get and increment (the word add is used in the method name) event counts. A few other ways of using Bin Objects are demonstrated in example2 if the reader is curious.

**Class Variables:**

None (I ran out of time and forgot to copy all of the object variables into class variables).

**Object Variables:**

* \_\_binWidth
* \_\_binStart
* \_\_binEnd
* \_\_backgroundCounts
* \_\_signalCount
* \_\_totalCount

**Methods:**

addBackgroundEvent

removeBackgroundEvent

setBackgroundCount

getBackgroundCount

setSignalCount

getSignalCount

getSignalCountDirectly

addSignalCounts

setTotalCount

getTotalCount

addTotalCount

updateTotalCount

acceptEnergy

getStartingEnergy

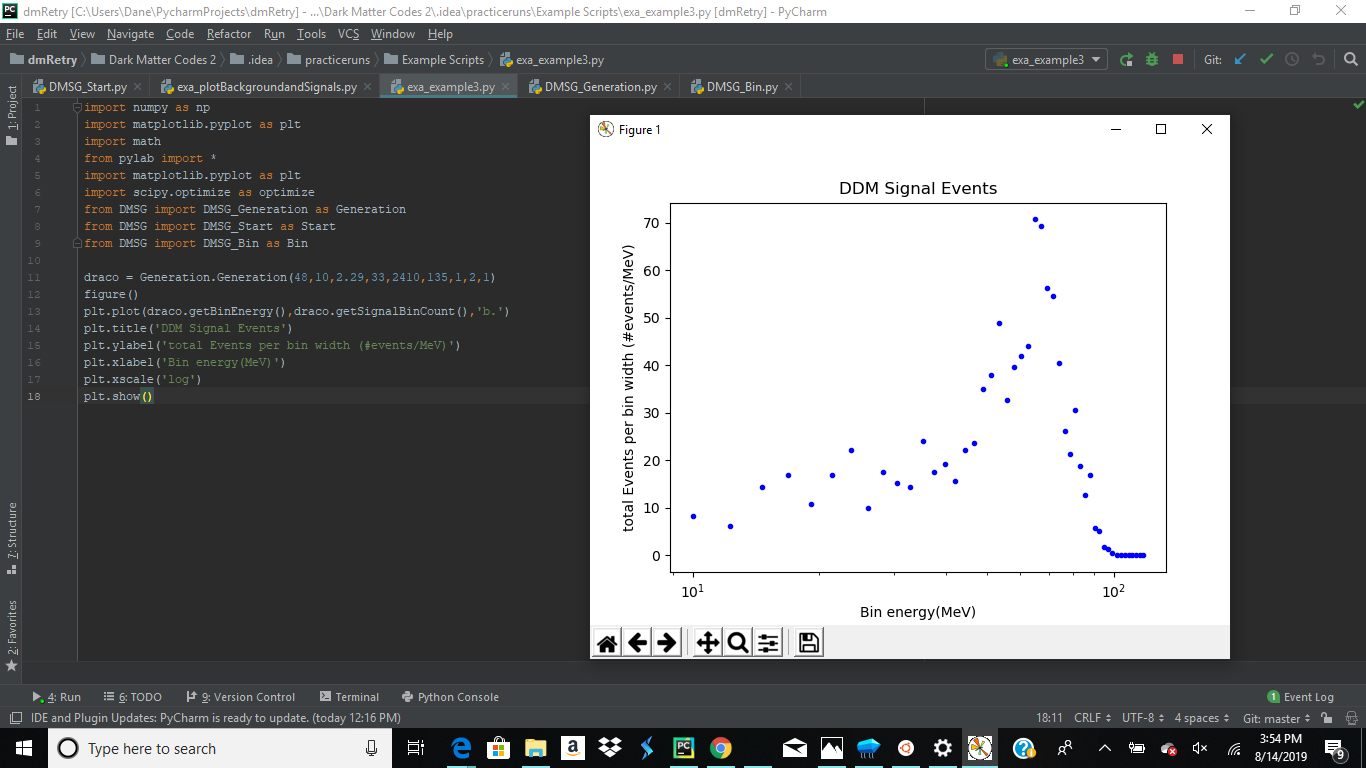
getEndingEnergy

getBinWidth

smearSignal

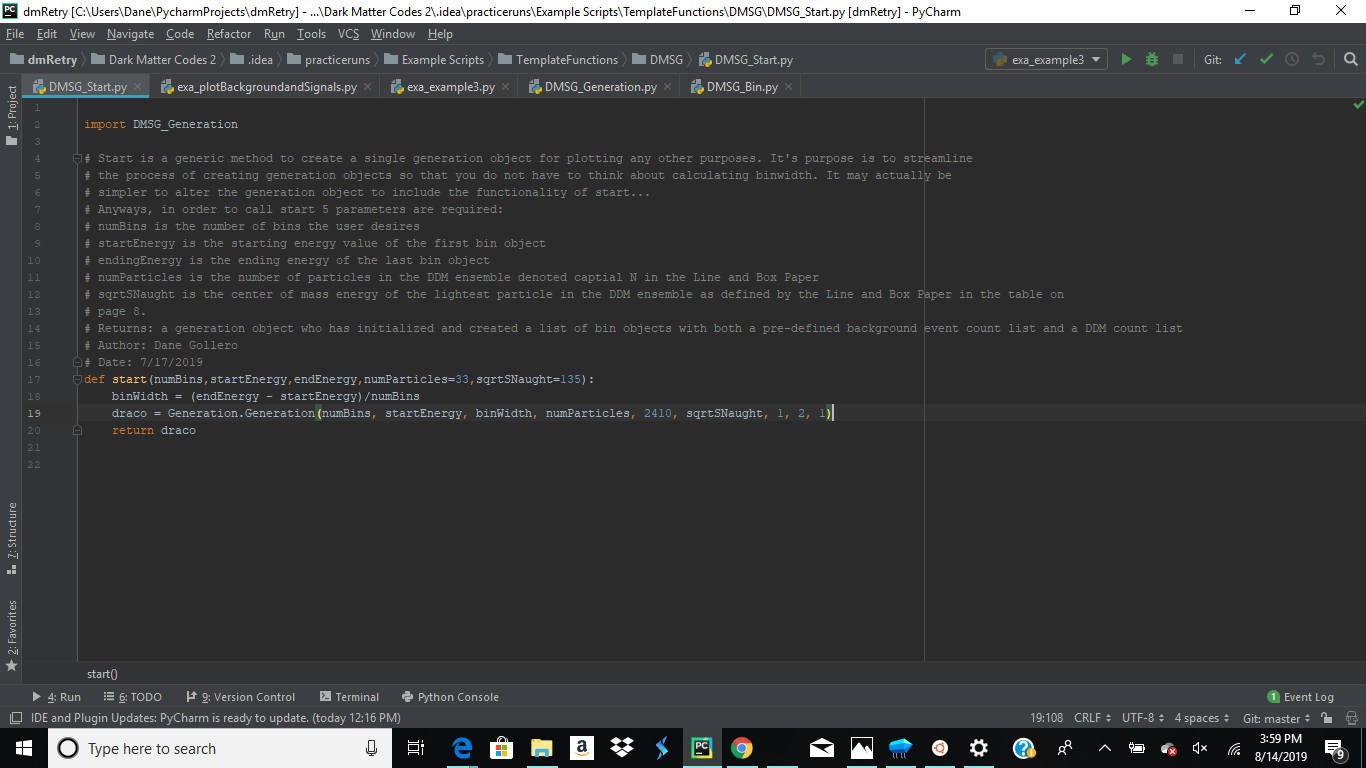
**Putting it Together**

Suppose we want to analyze benchmark B of the Lines and Boxes paper. We would create an appropriate Generation object with the correct arguments for B and then operate on it to get out our plotting elements. Example 3 shows this:



**Start**

Start is a “wrapper” for instantiating a Generation object. Generation takes in 9 arguments while Start takes in only 5. Start simplifies the process further by calculating binWidth for you and assuming the value of the parameters delta, deltaSSquare, eta and totalSignals.



**Dependencies**

DMSG and its associated files depend on the following packages:

* Numpy
* Scipy
* Lmfit
* mpmath

**References**

1. “Perspective of monochromatic gamma-ray line detection with the High Energy cosmic-Radiation Detection (HERD) facility onboard China’s Space Station” Huan, Xiaoyuan et al. (2016)

arxiv: 1509.02672