# McMaster Recycling Plant: Optical Sorter Simulator Guide

## Background Information

This guide provides the background and instruction on how to use the Optical Sorter Simulator.

In general, optical sorting units can be used for separating different kinds of waste such as different grades of glass, paper, and plastic from a waste stream, using different sensing techniques. In your project, you are going to design a sorting unit capable of identifying and sorting different kinds of plastics from each other, using infrared (IR) and Raman Spectroscopy techniques.

The waste passing through a conveyer belt is fed into an optical sorter where it is then examined by the sensors. Then the sorted materials are directed to corresponding bins. Typically, this separation is performed by means of series of air jets that blow the specified material out of the mainstream into a separate chute. Figure 1 shows examples of recycling sorting units.

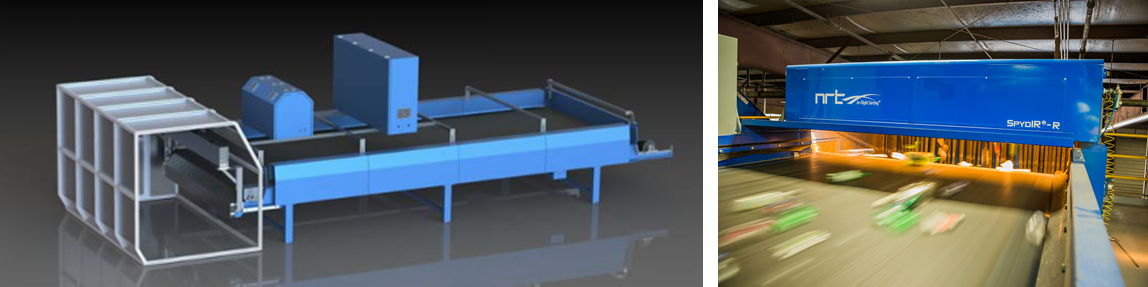


Figure 1: Examples of optical sorting units; Sources: Mayfran, NRT (National Recovery Technologies)

The simulator is a simplified representation of an optical sorter. The simulator is programmed in Python and your team’s task is to create a function that will take in spectroscopy data from a sensor and return the type of plastic. The simulator simply simulates the identification process and does not simulate the separation process. It is assumed that the waste will be separated once the type of plastic is identified.

## Getting Started

1. Strat by downloading and installing Python.
2. You can download the Python official package from <https://www.python.org/>.

For this project, please download Python 3.6+ version.

For installing Python in Windows follow the steps shown the Figures 2 to 5.

For Mac, simply follow the steps below but look for MacOS as the operating system, rather than Windows.

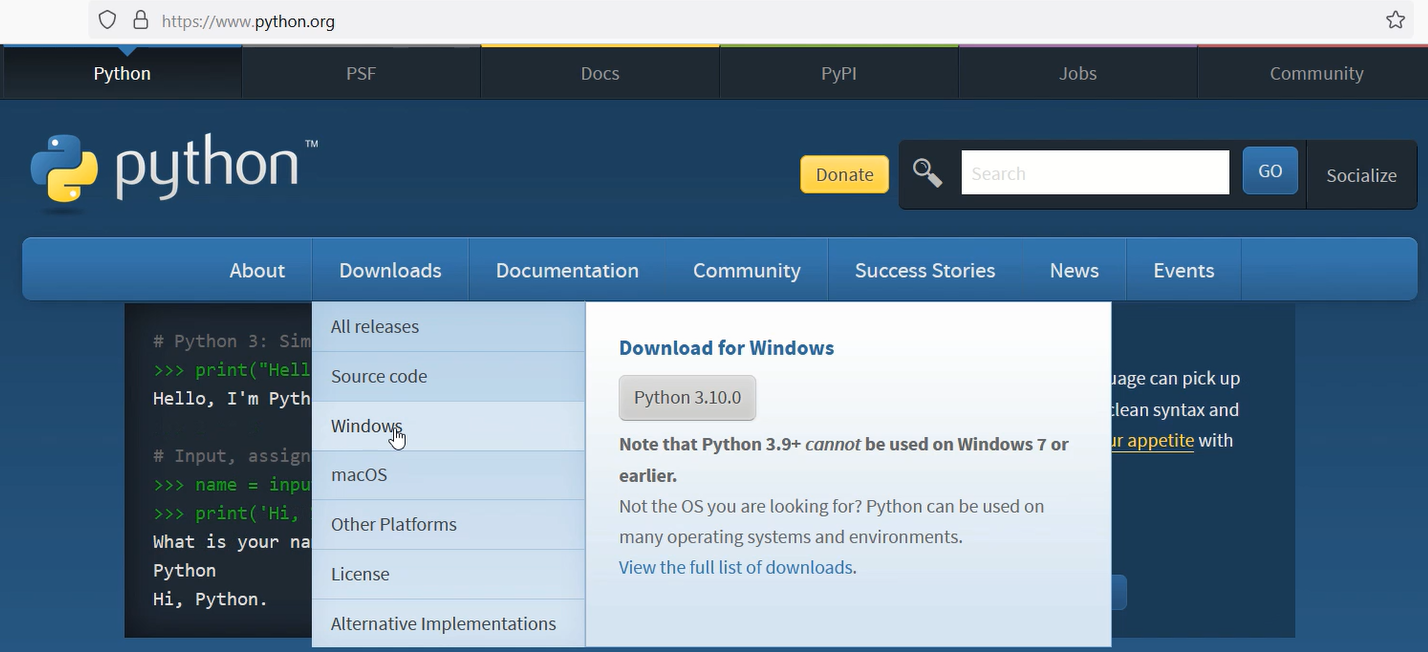


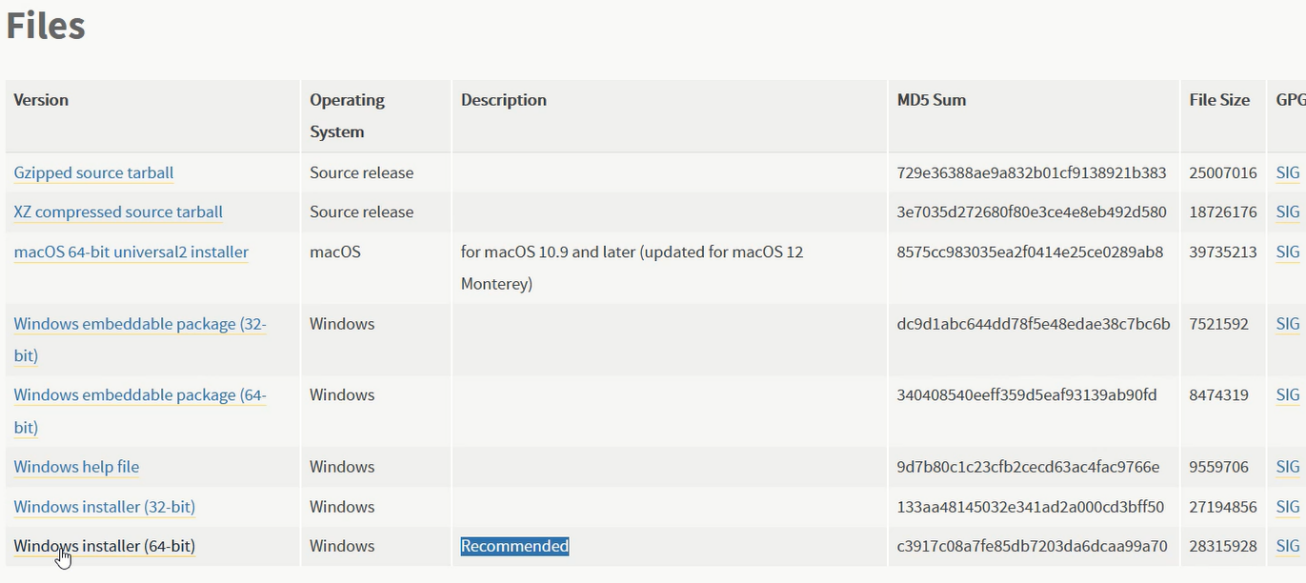
Figure 2: Go to <https://www.python.org/> > Downloads > Windows

1. Choose the latest released version.

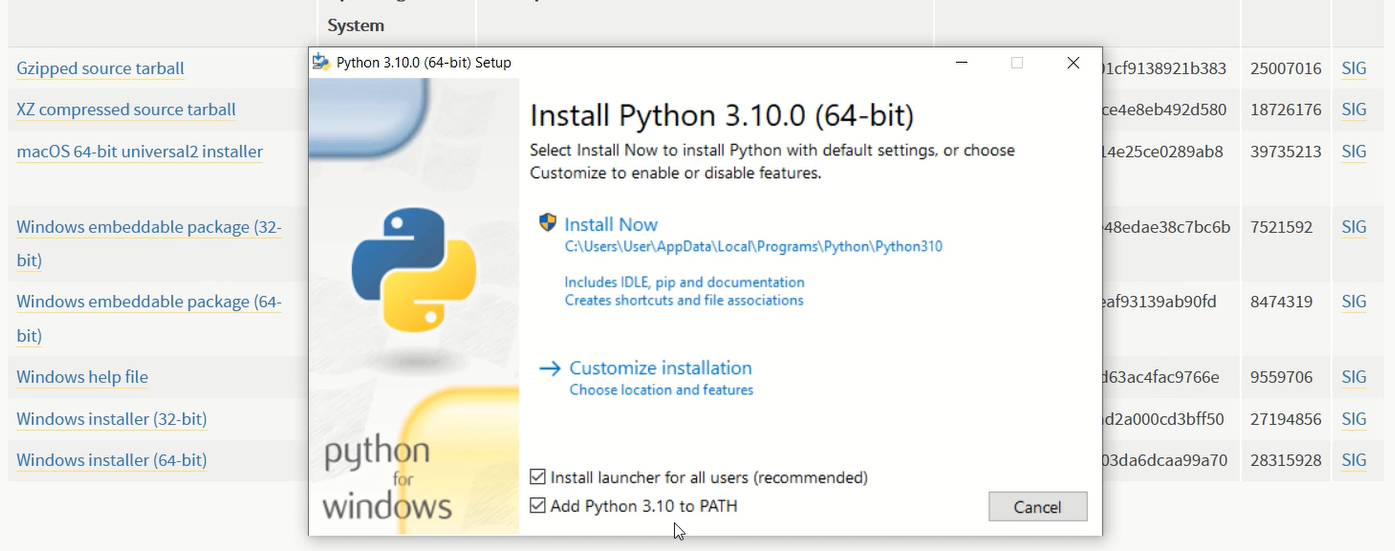


Figure 3: Choose the latest version, here: Python 3.10.2

1. Scroll down and select the recommended file from the list:

Figure 4: Select the recommended file for Windows

1. Download the package and make sure you select the 2 boxes at the bottom and install.

Figure 5: Download the file, hit “Install Now”

1. Now you should have Python installed on your system.
2. You can also download and install Python code editors such as PyCharm, visual studio code or Spyder for easier programming in Python. Setting up these IDEs (Integrated Development Environment) is highly recommended.

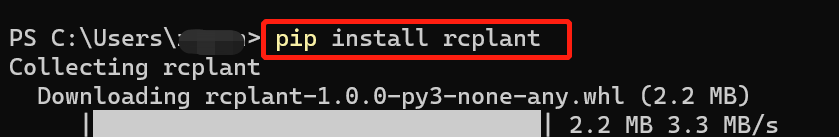
You can download one of these IDEs:

PyCharm from <https://www.jetbrains.com/pycharm/> .

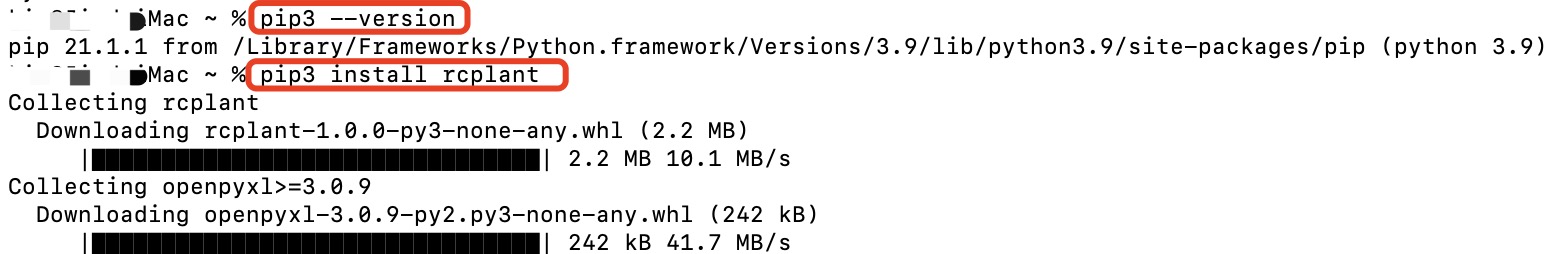
Visual studio code from: <https://code.visualstudio.com/>

Spyder from: <https://www.spyder-ide.org/>

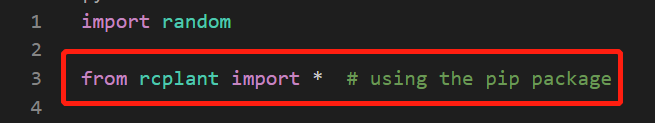
1. You can also download the rcplant package using PIP command or from: [https://pypi.org/project/rcplant/](https://pypi.org/project/rcplant/.)
   1. Windows: Open the Command window, type ‘pip install rcplant‘ to download [.](https://pypi.org/project/rcplant/.) The package is the simulated environment for the recycling plant. You will treat this as a black box and import this package at the beginning of the main.py file (See step 4)



* 1. Mac: Open your Terminal, type ‘pip3 install rcplant’ to download.



1. Download main.py file on Avenue under [4-Project Modules → McMaster Recycling Plant → main.py]. Use one of three code editors mentioned in step 2 to open it. You will see this line of code, that means you are importing the simulated environment – rcplant to your file.



1. The main.py file is the only file you will work on to modify the user\_sorting\_function(), adjust parameters and interact with the environment. The default user\_sorting\_function randomly assigns a type of plastic to the variable *decision* and returns it to the simulation environment.

## Introduction to the Simulator

This simulator simulates different types of plastic containers moving on a conveyor at a specific rate. In this simulation, the containers will be loaded one at a time and each container will have the same x-location (i.e. location along the width of the conveyor). Therefore, your function will only need to identify the type of plastic container and not the location of the container.

You can imagine that there are three distinct zones in the simulation. There is a loading zone, where a container is “loaded” (i.e. populated in the simulation) on the conveyor belt. The container then travels on the conveyor to the sensing zone, where the “sensor” returns spectroscopy data to the user\_sorting\_function. Lastly, the container moves to the sorting zone, where the user\_sorting\_function returns the type of plastic. An example configuration for these zones is depicted in Figure 6. Please note that in the model as soon as the type of the plastic is identified, the containers will be automatically assumed sorted. You are not required to incorporate any actuators in your model for sorting the plastic types at the end of the conveyer belt, like what is shown in Figure 6.

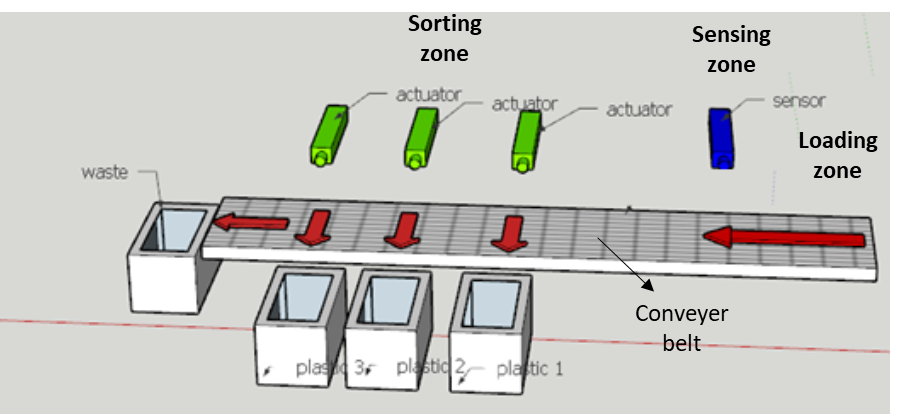


Figure 6: An example configuration of the different zones on a conveyer belt (Original image taken from DaveHakkens.nl)

### Simulation Parameters

In main.py, your team can adjust the following simulation parameters:

**conveyor\_length**: The total length of the conveyor in cm.

**conveyor\_width**: The width of the conveyor in cm. This parameter will not affect your sorting since we are assuming that the containers are passing through one at a time at a fixed location along the width of the conveyor.

The length and width of the conveyor are not critical in the simulation, but you need to consider the physical dimensions of your process units and how it relates to the physical space needed for the recycling plant. The dimensions given are currently in centimeters, but you should consider what a typical size of a conveyor should be.

**conveyor\_speed**: The speed of the conveyor belt is in cm/s. In the simulation, this is associated with the rate of generating containers. To achieve the requirement for the amount of daily recycling waste, the recycling plant needs to have sufficient throughput to process waste. However, the throughput is limited by the sampling frequency of the sensors. If the conveyor moves too fast, there will not be enough time for sensors to get the spectrum of the containers. Thus, you need to consider the trade-off between the conveyor speed and the sampling frequency of the sensor.

**num\_containers**: This is the total amount of container the simulator will run. You can determine the throughput of your design based on num\_containers and the total elasped time.

**sensing\_zone\_location\_1**: Location of the sensor along the length of the conveyor.

**sensors\_samplig\_frequency:** The sampling frequency of the sensor. Available sampling frequency are [1, 2, 5, 10] Hz. Increasing the sampling frequency will increase the signal noise. Decreasing the frequency will reduce your noise and improve your accuracy but at the cost of reducing the throughput. Please note that this is a simple simulation used to illustrate the trade-off between higher throughput and sorting accuracy, typical optical sorters have much higher signal frequencies.

**simulation\_mode:** There are two simulation modes: training and testing. In training mode (simulation\_mode = ‘training’), there is no noise in the sensor signal and the order of the containers will always be the same. In testing mode (simulation\_mode = ‘testing’), the containers will be randomized. Also, noise is added to the sensor signal. The signal noise is proportional to sampling frequency which in turn lowers the resolution of the sensor. The signal to noise ratio is set equal to 50/frequency in the simulation.

### Identification Result for Verification

**simulator.identification\_result.items()**

Identification\_result is a nested dictionary that stores the actual plastic type of each container that was generated by the simulator, and the plastic type identified by the user\_sorting\_function in your code.

The dictionary can be used to verify which containers were mistyped. Note that missed containers (i.e. containers that were not captured by the sensor), will not be included on the list. Only the classified and mistyped containers will be on the list.

**Creating the Sensor**

You can create the sensor using the code

sensors = [

Sensor.create(SpectrumType, sensing\_zone\_location\_1),

]

The spectrum type can either be SpectrumType.FTIR if you want to use the IR spectra or SpectrumType.Raman if you want to use the Raman spectra.

For example, if you are using an IR sensor then your code will be:

sensors = [

Sensor.create(SpectrumType.FTIR, sensing\_zone\_location\_1),

]

When the containers are in the sensing zone, the simulated environment will output the properties of the containers. For areas where there are no containers, the simulated environment will output a blank spectrum (i.e. 0 for all wavelengths). You can choose to either use an IR sensor or a Raman sensor. The only difference is the type of spectral data that the simulation will pass to your user\_sorting\_function().

You will need to interpret the sensors\_output in your user\_sorting\_function() to determine the plastic types of the containers. The result will be passed back through the variable *decision* to the simulated environment to sort the containers.

## Waste Container Properties

There are 9 types of plastics for waste containers in the simulation:

- PET (Polyethylene Terephthalate)

- HDPE (High-Density Polyethylene)

- PVC (Polyvinyl Chloride)

- LDPE (Low-Density Polyethylene)

- PP (Polypropylene)

- PS (Polystyrene)

- Polyester

- PC

- PU

The simulator will randomly generate a container with two properties: the type of plastic and size. The type of plastic will be represented by the associated spectrum and the size will be represented by a length (5 – 15 cm). To simplify the problem, we will assume that the containers do not overlap each other and that the containers will enter the optical sorter one at a time at a fixed location along the width of the conveyor belt (i.e. containers do not move or roll on the belt. Lastly, we define the y-direction as the length of the conveyor belt. We assume that the container will start at y=0 and moves in the y-direction according to the conveyor speed.

## Sensor Output

The sensor will return spectroscopy data at the rate of the sensors\_samplig\_frequency. When a container in the sensing zone (i.e. the y-location of the container is equal to the y-location of the sensor), the sensor will output the spectrum of the container. When no container is present, the sensor will output a blank spectrum. A default FTIR sensor has been created for you in main.py file. You can choose to use either a FTIR or Raman sensors.

## Sorting Function

The sorting mechanism will be based on the user\_sorting\_function developed by your group. The sorting function reads in a nested dictionary containing data from the sensor and returns the type of plastic to the simulator.

The simulator takes the return value from the sorting function and determines whether the container is accurately classified. The container can be classified, mistyped, or missed. Classified containers are the ones that are correctly identified by the sorting function. Mistyped containers are the ones that have been incorrectly identified by the sorting function. Missed containers are the containers that were not captured by the sensor. The total number of classified/sorted, mistyped, and missed containers will be reported by the simulator.

The simulator will also output the list of plastic type returned by the sorting function and the actual plastic type generated by the simulator. You can compare these two lists to determine which plastic containers are being mistyped.

Below is the API documentation detailing the specifications of the simulator.

# API Instructions

These instructions introduce the structure and API for the simulator. You will be asked to write user\_sorting\_function(sensors\_output) as required to complete the whole simulation.

The accuracy of the simulator depends on the performance of user\_sorting\_function(sensors\_output)

## Loading Zone

class Plastic(enum.Enum):  
 HDPE = 'HDPE'  
 LDPE = 'LDPE'  
 PP = 'PP'  
 PS = 'PS'  
 PC = 'PC'  
 PVC = 'PVC'  
 Polyester = 'Polyester'  
 PET = 'PET'  
 PU = 'PU'   
 Blank = 'background'

class Container:  
 def \_\_init\_\_(self, plastic\_type: Plastic, dimension: ContainerDimension, location: ContainerLocation):  
 self.\_material = Material(plastic\_type)

### Containers Generation Description:

The simulator will generate containers with random Plastic types at a rate of X, X is associated with conveyor\_speed. To achieve the requirement of loading rate, you will need to choose an appropriate conveyor\_speed in your design. ContainerDimension is a class with attributes self.\_length, self.\_width, self.\_height, which are assigned randomly in the range of 5 to 15 cm. ContainerLocation is a class with default attributes self.\_x = self.\_y = self.\_z = 0

In our case, the simulation is in 1D (i.e. ignores x and z dimensions). Also, it is assumed that the containers will not overlap each other.

class Conveyor:  
 def \_\_init\_\_(self, speed\_cm\_per\_second: int, dimension: ConveyorDimension):

### Conveyor Description:

Containers are assigned location (x, y) where y-direction is the conveyor\_length of the belt and x-direction is conveyor\_width. We assume container.location.y moves along the belt at a speed of conveyor\_speed while container.location.x does not change with time.

The ratio of missed containers is inversely proportional to conveyor\_speed as the sensor’s performance is limited by its sampling frequency, that is an item will be missed if it moves too fast to be scanned.

num\_containers = 100

## Parameter description:

num\_containers is the total number of containers the simulator will run

## Sensing Zone

sensors = [  
 Sensor.create(SpectrumType.FTIR, sensing\_zone\_location\_1),   
 ]  
  
 class Sensor:  
 def \_\_init\_\_(self, sensor\_type: SpectrumType, location\_cm: int)

### Sensor description:

The sensor is associated with the sensor\_type and location\_cm. As introduced in the project module, sensors have two types -- SpectrumType.FTIR and SpectrumType.Raman. You can choose either of the sensors for this simulation.

When the containers are passing the sensing\_zone\_location, the simulated environment will output the spectra of the containers -- sensor\_output in a nested dictionary. Thus, to access the actual spectra, you need to use the correct indexing syntax. The spectra are stored in Pandas.Series data type.

For areas where there are no containers, a series of 0 that stands for blank spectrum will be output.

sensors\_sampling\_frequency = 1 # Hz  
  
 VALID\_SENSORS\_FREQUENCIES\_HZ = [10, 5, 2, 1] # divisors of SIMULATION\_FREQUENCY\_HZ

### Sampling Frequency:

The sensor have 4 valid sampling frequency = [10, 5, 2, 1]. Higher frequency will decrease the ratio of missed containers when the conveyor speed is a constant. However, in testing mode, a signal noise will be introduced that will increase proportionally to the sampling frequency. Thus, there is a trade-off among conveyor speed, sampling frequency and sorting accuracy.

## Sorting Zone

def user\_sorting\_function(sensors\_output):

### Sorting Function description:

The sensor will scan and output containers’ spectra in the sensing zone, which will be used as an input argument to user\_sorting\_function(). The function will be passed to RPSimulation class. The return value of user\_sorting\_function() will be used to sort containers.

### Input parameters:

* sensor\_output is a nested dictionary where the inside one contains SpectrumType class as key and Spectrum in Pandas.Series structure as value  
  e.g.

class SpectrumType(enum.Enum):  
 FTIR = 'FTIR'  
 Raman = 'Raman'   
  
 sensors\_output = {sensor\_id: {'type': <SpectrumType.FTIR: 'FTIR'>, 'location': 500, 'spectrum': Spectrum of sensed containers in pandas.Series data type}}

Example – how to access the spectra in sensor\_output:

sensors\_output[1]['spectrum']   
 3600 0.001203  
 3598 0.000943  
 3596 0.000712  
 3594 0.000733  
 3592 0.000649  
 ...  
 1258 0.597359  
 1256 0.619357  
 1254 0.641317  
 1252 0.663742  
 1250 0.684752  
 Name: PET, Length: 1176, dtype: float64

The sorting function needs to analyze the value of sensors\_output to determine the type of plastic and return the result into the simulator

### Return value:

* The data type of return value is a dictionary, of which the key is **SpectrumType class** and the value is **Plastic class**

Example – How to return a plastic type in user\_sorting\_function:  
 return {sensor\_id: <Plastic.Polyester: 'Polyester'>}  
 class Plastic(enum.Enum):  
 HDPE = 'HDPE'  
 LDPE = 'LDPE'  
 PP = 'PP'  
 PS = 'PS'  
 PC = 'PC'  
 PVC = 'PVC'  
 Polyester = 'Polyester'  
 PET = 'PET'  
 PU = 'PU'   
 Blank = 'background'

### Simulation Mode

class SimulationMode(enum.Enum):  
 Testing = 'testing'  
 Training = 'training'

In training mode, the simulator runs normally to complete loading, sensing and sorting. The spectra of containers come from the original spectra database (i.e. the Excel file you received from Avenue), which is also stored in the rcplant module you import at the beginning of the code. In testing mode, noise will be introduced to the spectra of sensor\_output to test the adaptability of the sorting function. The signal-to-noise ratio (SNR) is proportional to the sampling frequency, which means higher sampling frequency will incur more noise in the signal.

### Plastic Spectrum

The figure below shows the FTIR spectra for different types of plastic in the dataset. It is for you to have a general sense of what the sensors output. You could also plot the spectra using the value of sensor\_output in Python, or using the Excel files of the FTIR and Raman databases provided for you on Avenue.

