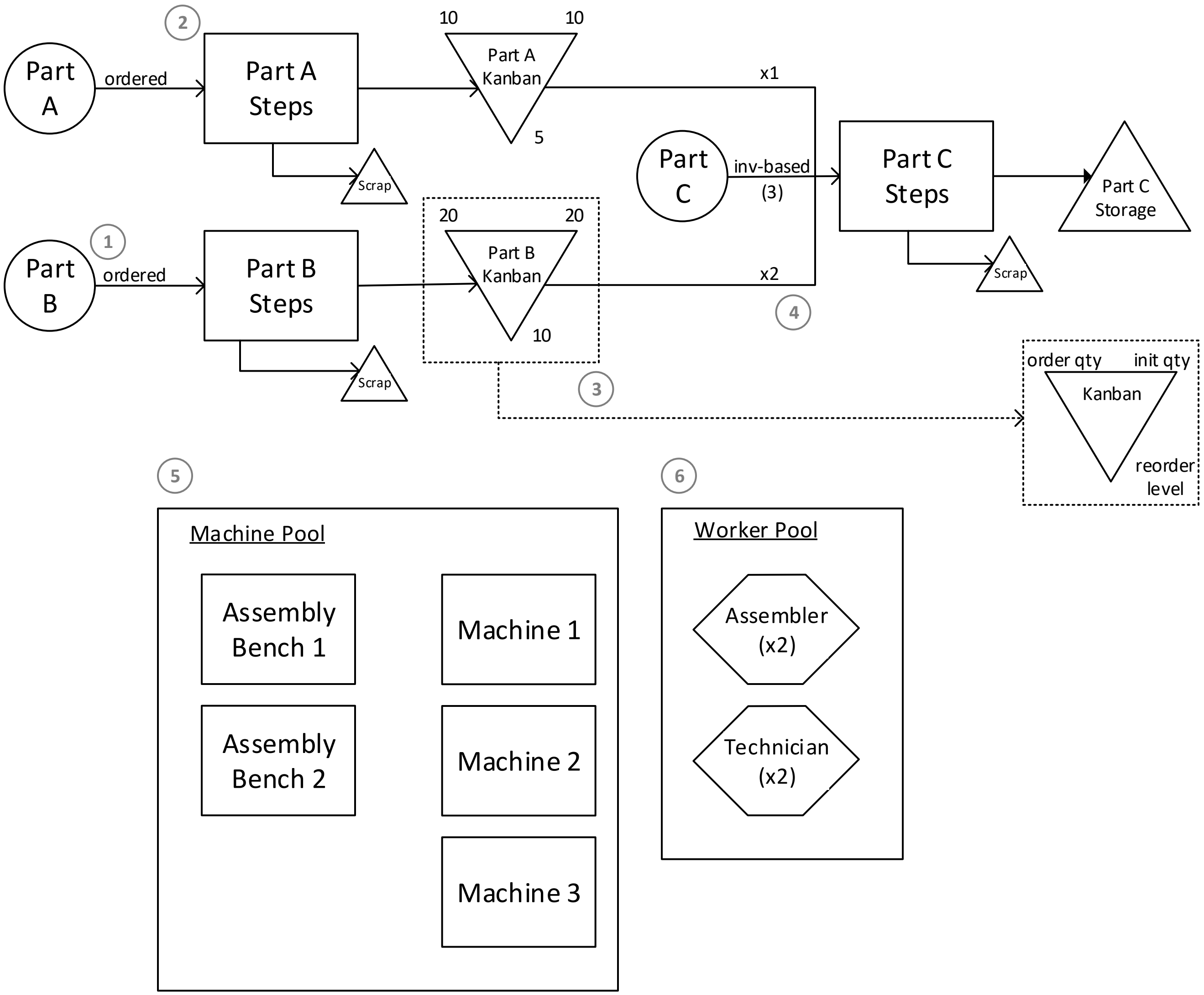
Overall Process Flow: Simple Example



1 Entities ("Parts") can be generated through a variety of methods. They are as follows…

<u>Continuous</u>: Parts are made continuously every 1 time epoch

<u>Periodic</u>: Parts are made every x time epochs, defined by the user

<u>Ordered</u>: A Kanban orders more parts to be made at a certain inventory threshold

<u>Inv-based</u>: Parts are made to maintain x number of parts being processed at all times, defined by the user

2 Entity steps are the detailed set of operations an entity takes before completion or exiting the system. Steps are predefined before entering the system by using a function which determines the steps taken for that particular entity instance. (More detail provided below)

3 The Kanban acts as an ordering system for downstream processes. They can be initialized with a given quantity, "init qty". And then they will order entities at quantity "order qty" when the Kanban contents falls below its "reorder level". Entities must arrived based on an order to use.

4 Upstream processes can have build of materials in which downstream entities can be taken from their storage location and used for the upstream entity. The "bom" variable indicates these dependencies. Here's how Part C would be formatted for the given process:

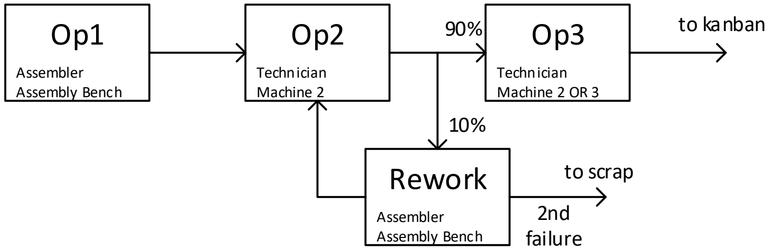
```
{
     "Part A": {
           "location": Part A Kanban,
           "qty": 1
     },
     "Part B": {
           "location": Part B Kanban,
           "qty": 2
     }
}
```

5 Machines should be thought of as physical locations an entity has to pass through in order to move on to the next step in its process.

6 Workers should be thought of as floating requirements that must be present at the physical Machine location in order for an entity to be processed and move to its next step. Workers can have designated shift schedules where they will be unable to fulfill requests at certain times.

<u>NOTE</u>:
Outside of the dotted legend box for Kanban explanation, this would be the expected graphic to be generated if animation were to be added to the simulation model.

7 Part A Steps Flow

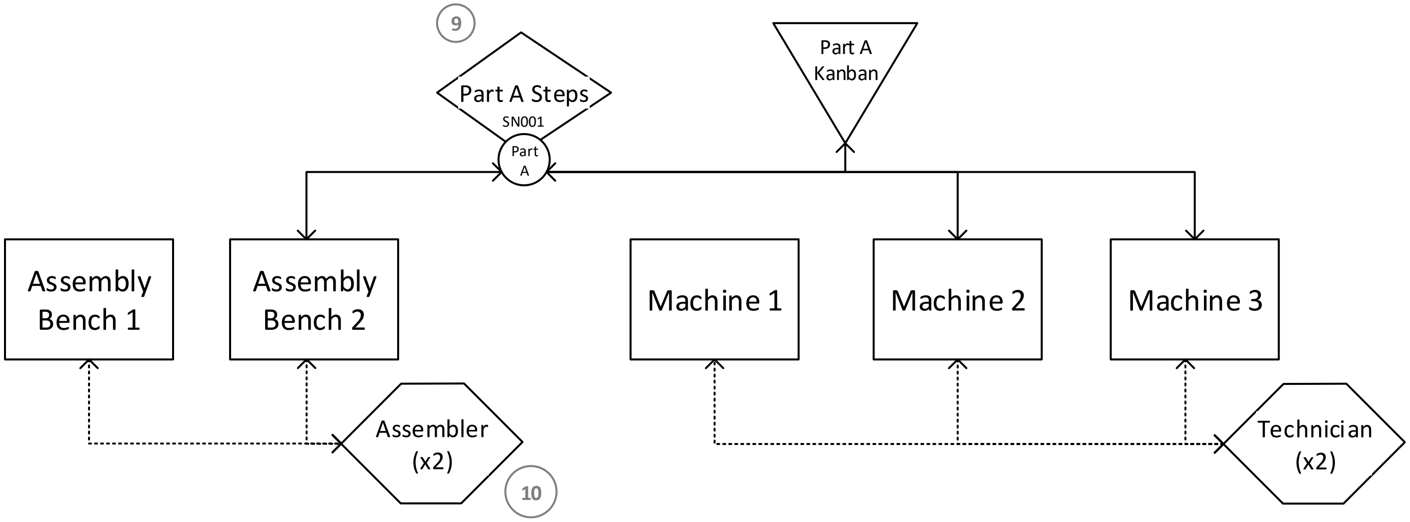


7 Part A Steps Flow depicts the decision tree used in determining the steps that are taken for an entity. A function executes the decision tree upon an entity being created where it determines all the steps to be taken for that entity before entering the system. Each “operation” is a formatted JSON object which indicates what is required for the operation as well as the properties that determine what happens next. JSON objects for Op1 and Op2 are shown on the right.

8 The JSON objects are overwritten in some cases during function execution (i.e. making decisions based on percentages, sampling values from distributions, etc.) to comprise a list on explicit steps the entity will take when it enters the system

```
{
     “Op1”: {
          “location”: Assembly Bench,
          “worker”: Assembler,
          “setup_time”: 0,
          “run_time”: 15,
          “teardown_time”: 0,
          “transit_time”: 1,
          “route_to”: Op2
     },
     “Op2”: {
          “location”: Machine 2,
          “worker”: Technician,
          “manned”: False,
          “setup_time”: 5,
          “run_time”: 10,
          “teardown_time”: 2,
          “transit_time”: 1,
          “yield”: 0.96,
     8    “route_to_pass”: Op3,
          “route_to_fail”: rework
     },
     ...
}
```

Part A Entity Process within salabim_plus



9 Part A steps can be thought of as a list of instructions telling that entity what Machine to go to, which Worker to request once it gets there, and how long it will take to process. The figure shows a more literal sense of what is happening when salabim_plus executes its simulation code.

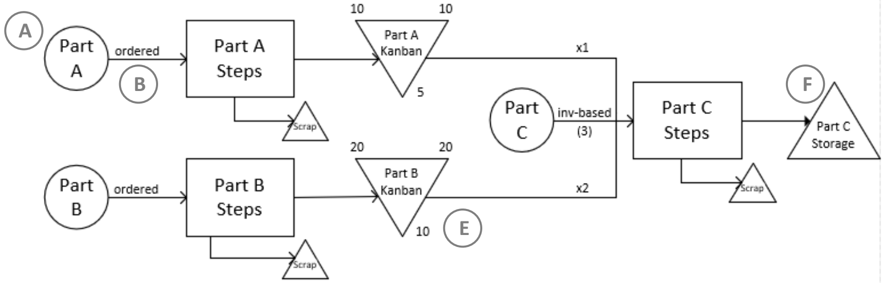
10 Worker availability can also be controlled by the ShiftController which indicates when a Worker is actually working. During unavailable times, any machine using a worker will be interrupted and any requests for that worker will wait in a hold pattern until a worker returns to work and everything resumes. Shifts are formatted in a variety of methods. They are as follows…

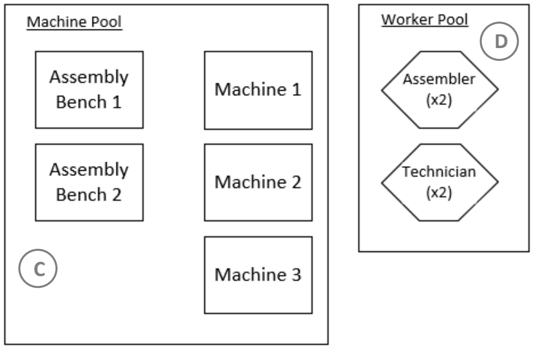
Continuous: A defined shift for one day repeating everyday.

Pattern: A defined list of shifts repeating at the end of the list.

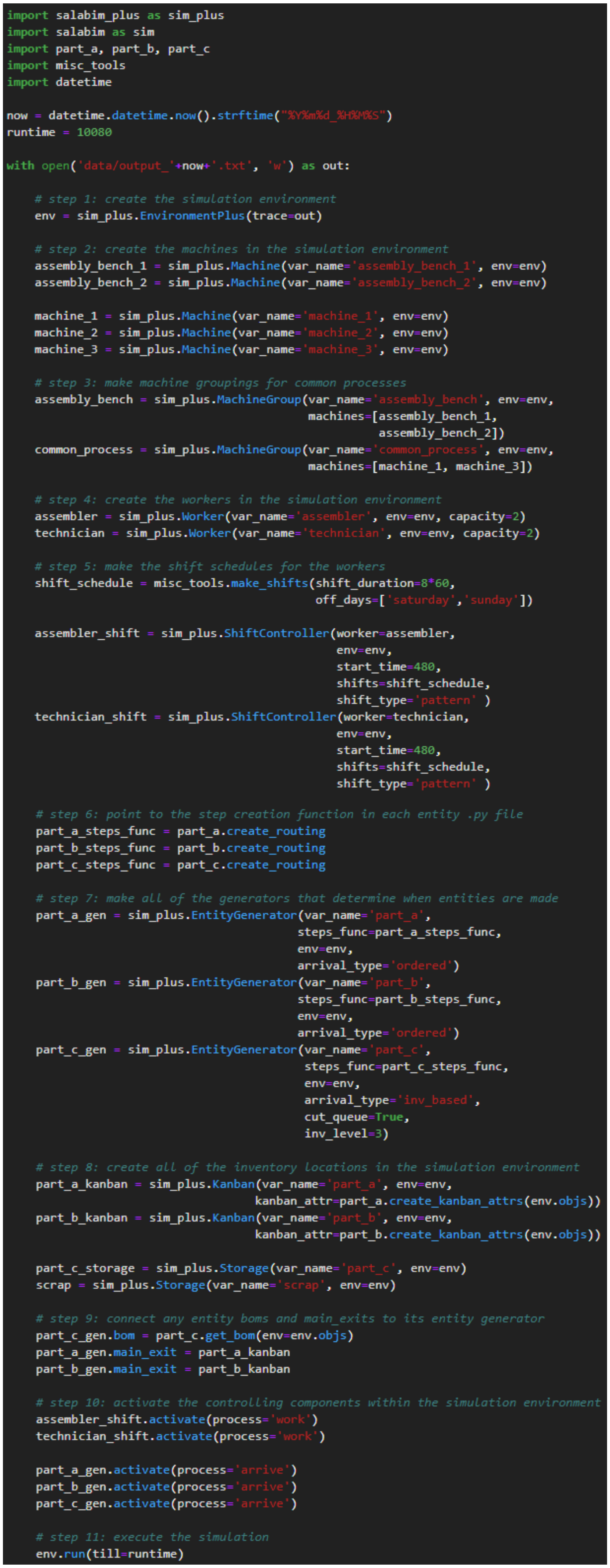
Custom: A defined list of shifts that occur and a worker no longer works after the last shift.

Salabim_Plus Class Description	Inputs
A **Entity()**: A salabim sim.Component that undergoes some process. salabim contents inside class… - state (sim.State) - step_complete (sim.State)	N/A instantiated with EntityGenerator()
EntityTracker(): A salabim sim.Component that keeps track of entities of a certain type. salabim contents inside class… - wip (sim.Queue) - complete (sim.Queue) - wip_count (sim.State) - complete_count (sim.State	N/A instantiated with EntityGenerator()
B **EntityGenerator()**: A salabim sim.Component that creates Entities of a certain type to be processed. salabim_plus contents inside class.. - entity (sim_plus.Entity) - tracker (sim_plus.EntityTracker)	required: var_name, steps_func, env optional: arrival_type, start_at, bom, cut_queue, interval, inv_level
C **Machine()**: A salabim sim.Component that processes Entities at a specific location. salabim contents inside class… - queue (sim.Queue) - in_queue (sim.State) - state (sim.State)	required: var_name, env
MachineGroup(): A salabim sim.Component that groups Machines that perform common processes.	required: var_name, env, machines
D **Worker()**: A salabim sim.Resource that is required to be present at a Machine location to process an Entity. salabim contents inside class… - state (sim.State) - num_working (sim.State)	required: var_name, env, capacity
ShiftController(): A salabim sim.Component that controls when a Worker is available to work.	required: worker, env, start_time, shifts, shift_type
E **Kanban()**: A salabim sim.Component that acts as an inventory based ordering system for Entities of a certain type. salabim contents inside class… - queue (sim.Queue) - count (sim.State) - on_order (sim.State) - total_inv (sim.State)	required: var_name, env, kanban_attr
F **Storage()**: A salabim sim.Component that acts as an inventory location for Entities. salabim contents inside class… - queue (sim.Queue) - count (sim.State)	required: var_name, env





Main .pynb File



```python
import salabim_plus as sim_plus
import salabim as sim
import part_a, part_b, part_c
import misc_tools
import datetime

now = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
runtime = 10080

with open('data/output_'+now+'.txt', 'w') as out:

    # step 1: create the simulation environment
    env = sim_plus.EnvironmentPlus(trace=out)

    # step 2: create the machines in the simulation environment
    assembly_bench_1 = sim_plus.Machine(var_name='assembly_bench_1', env=env)
    assembly_bench_2 = sim_plus.Machine(var_name='assembly_bench_2', env=env)

    machine_1 = sim_plus.Machine(var_name='machine_1', env=env)
    machine_2 = sim_plus.Machine(var_name='machine_2', env=env)
    machine_3 = sim_plus.Machine(var_name='machine_3', env=env)

    # step 3: make machine groupings for common processes
    assembly_bench = sim_plus.MachineGroup(var_name='assembly_bench', env=env,
                                           machines=[assembly_bench_1,
                                                     assembly_bench_2])
    common_process = sim_plus.MachineGroup(var_name='common_process', env=env,
                                           machines=[machine_1, machine_3])

    # step 4: create the workers in the simulation environment
    assembler = sim_plus.Worker(var_name='assembler', env=env, capacity=2)
    technician = sim_plus.Worker(var_name='technician', env=env, capacity=2)

    # step 5: make the shift schedules for the workers
    shift_schedule = misc_tools.make_shifts(shift_duration=8*60,
                                             off_days=['saturday','sunday'])

    assembler_shift = sim_plus.ShiftController(worker=assembler,
                                                env=env,
                                                start_time=480,
                                                shifts=shift_schedule,
                                                shift_type='pattern' )
    technician_shift = sim_plus.ShiftController(worker=technician,
                                                env=env,
                                                start_time=480,
                                                shifts=shift_schedule,
                                                shift_type='pattern' )

    # step 6: point to the step creation function in each entity .py file
    part_a_steps_func = part_a.create_routing
    part_b_steps_func = part_b.create_routing
    part_c_steps_func = part_c.create_routing

    # step 7: make all of the generators that determine when entities are made
    part_a_gen = sim_plus.EntityGenerator(var_name='part_a',
                                          steps_func=part_a_steps_func,
                                          env=env,
                                          arrival_type='ordered')
    part_b_gen = sim_plus.EntityGenerator(var_name='part_b',
                                          steps_func=part_b_steps_func,
                                          env=env,
                                          arrival_type='ordered')
    part_c_gen = sim_plus.EntityGenerator(var_name='part_c',
                                           steps_func=part_c_steps_func,
                                           env=env,
                                           arrival_type='inv_based',
                                           cut_queue=True,
                                           inv_level=3)

    # step 8: create all of the inventory locations in the simulation environment
    part_a_kanban = sim_plus.Kanban(var_name='part_a', env=env,
                                    kanban_attr=part_a.create_kanban_attrs(env.objs))
    part_b_kanban = sim_plus.Kanban(var_name='part_b', env=env,
                                    kanban_attr=part_b.create_kanban_attrs(env.objs))

    part_c_storage = sim_plus.Storage(var_name='part_c', env=env)
    scrap = sim_plus.Storage(var_name='scrap', env=env)

    # step 9: connect any entity boms and main_exits to its entity generator
    part_c_gen.bom = part_c.get_bom(env=env.objs)
    part_a_gen.main_exit = part_a_kanban
    part_b_gen.main_exit = part_b_kanban

    # step 10: activate the controlling components within the simulation environment
    assembler_shift.activate(process='work')
    technician_shift.activate(process='work')

    part_a_gen.activate(process='arrive')
    part_b_gen.activate(process='arrive')
    part_c_gen.activate(process='arrive')

    # step 11: execute the simulation
    env.run(till=runtime)
```

0 Import Relevant Python Files/Packages

- Simulation has a folder structure where other python files are imported into the main simulation file.

1 Create Simulation Environment

- The same as salabim's base environment but keeps track of all salabim components that will be comprised of the simulation. Write trace file out to an output.txt file.

2 Create Machines in Simulation

3 Make Machine Groups that share common processes

4 Create Workers in Simulation

5 Make Shift Schedules for Workers

- Use the misc_tools.make_shifts function tool to format shifts specific to each worker.

6 Point to entity specific function that makes entity steps

- Entity .py file explained further below

7 Make Entity Generators

8 Create Inventory Locations in Simulation

- create_kanban_attrs function explained further in Entity .py file

9 Add Entity Build of Materials and Entity Kanban Exits, if applicable

- get_bom function explained further in Entity .py file

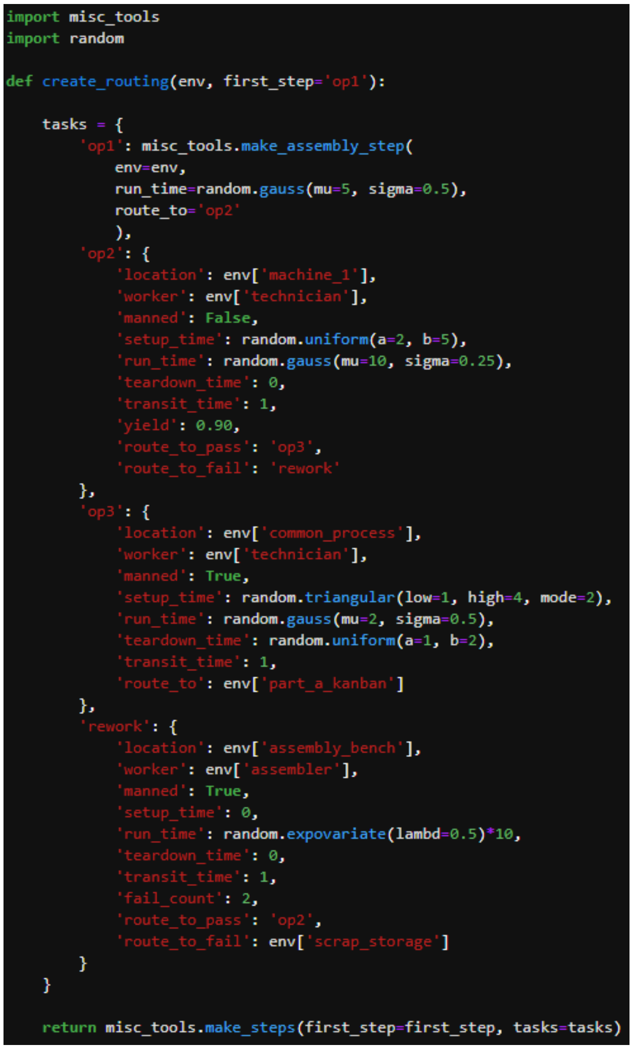
10 Activate all Controlling Components

- Shift Controllers and Entity Generators

11 Run Simulation

Entity Specific .py Files

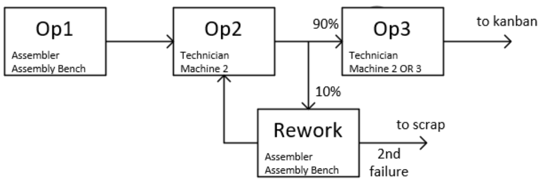
Building Tasks per Entity: part_a



```
import misc_tools
import random

def create_routing(env, first_step='op1'):

    tasks = {
        'op1': misc_tools.make_assembly_step(
            env=env,
            run_time=random.gauss(mu=5, sigma=0.5),
            route_to='op2'
            ),
        'op2': {
            'location': env['machine_1'],
            'worker': env['technician'],
            'manned': False,
            'setup_time': random.uniform(a=2, b=5),
            'run_time': random.gauss(mu=10, sigma=0.25),
            'teardown_time': 0,
            'transit_time': 1,
            'yield': 0.90,
            'route_to_pass': 'op3',
            'route_to_fail': 'rework'
        },
        'op3': {
            'location': env['common_process'],
            'worker': env['technician'],
            'manned': True,
            'setup_time': random.triangular(low=1, high=4, mode=2),
            'run_time': random.gauss(mu=2, sigma=0.5),
            'teardown_time': random.uniform(a=1, b=2),
            'transit_time': 1,
            'route_to': env['part_a_kanban']
        },
        'rework': {
            'location': env['assembly_bench'],
            'worker': env['assembler'],
            'manned': True,
            'setup_time': 0,
            'run_time': random.expovariate(lambd=0.5)*10,
            'teardown_time': 0,
            'transit_time': 1,
            'fail_count': 2,
            'route_to_pass': 'op2',
            'route_to_fail': env['scrap_storage']
        }
    }

    return misc_tools.make_steps(first_step=first_step, tasks=tasks)
```

0 Misc_tools.py provides a series of helper functions that reduce repetitive dictionary formatting as well as conduct commonized functions like make_steps().

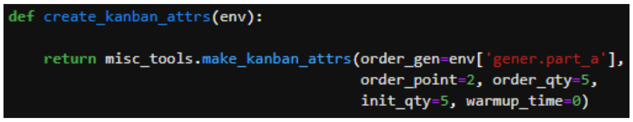
1 Randomness can be introduced by using the random python package on time related fields.



2 Simple tasks with feed forward routings must contain the following key, value pairs:
“location”, “worker”, “manned”, “setup_time”,

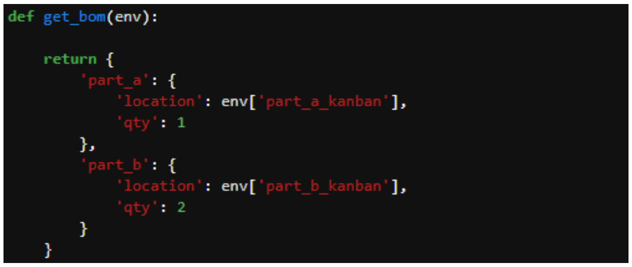
3 Tasks with decision based routings must have “route_to_pass” and “route_to_fail” key value pairs as well as “yield” or “fail_count” key value pairs depending on the nature of the decision.

Formatting Entity Kanban: part_a



```
def create_kanban_attrs(env):

    return misc_tools.make_kanban_attrs(order_gen=env['gener.part_a'],
                                        order_point=2, order_qty=5,
                                        init_qty=5, warmup_time=0)
```

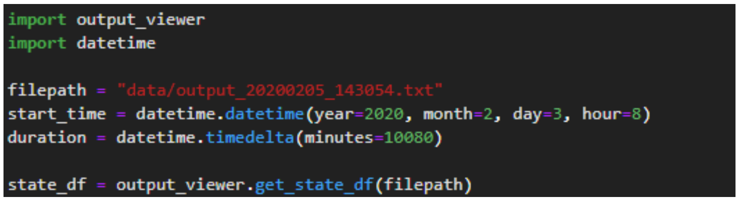
Formatting Build of Material: part_c



```
def get_bom(env):

    return {
        'part_a': {
            'location': env['part_a_kanban'],
            'qty': 1
        },
        'part_b': {
            'location': env['part_b_kanban'],
            'qty': 2
        }
    }
```

Viewing Simulation Outputs

0



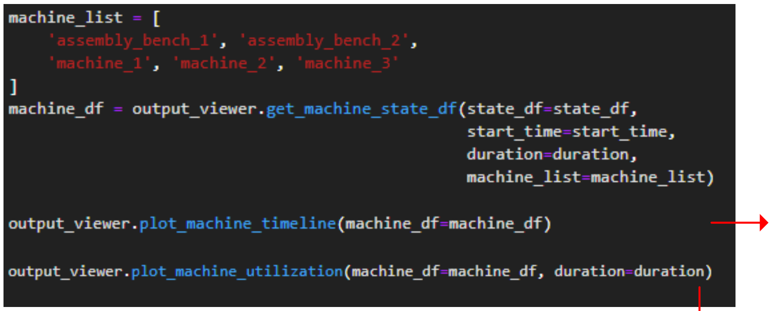
```
import output_viewer
import datetime

filepath = "data/output_20200205_143054.txt"
start_time = datetime.datetime(year=2020, month=2, day=3, hour=8)
duration = datetime.timedelta(minutes=10080)

state_df = output_viewer.get_state_df(filepath)
```

0 output_viewer.py provides a series of plotting functions that allow you to view simulation results. Other steps required:

- Map filename to output.txt from data folder
- Assume a start datetime
- Create duration variable that matches the sim til variable
- Make state change dataframe with helper function

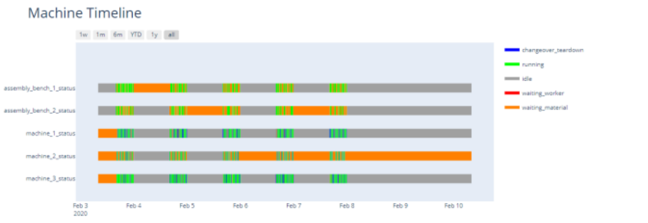
Machine Plots

1

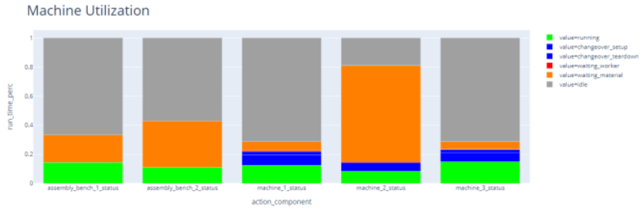


```
machine_list = [
    'assembly_bench_1', 'assembly_bench_2',
    'machine_1', 'machine_2', 'machine_3'
]
machine_df = output_viewer.get_machine_state_df(state_df=state_df,
                                                start_time=start_time,
                                                duration=duration,
                                                machine_list=machine_list)

output_viewer.plot_machine_timeline(machine_df=machine_df)

output_viewer.plot_machine_utilization(machine_df=machine_df, duration=duration)
```

1 Machine list should match machine names in main.ipynb, create dataframe of machine state changes with helper function.



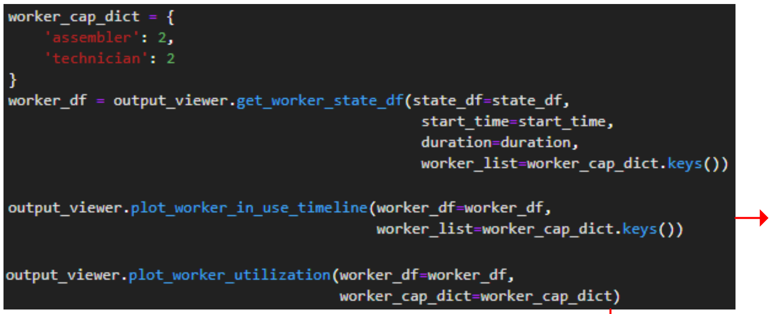
Machine Timeline



Machine Utilization

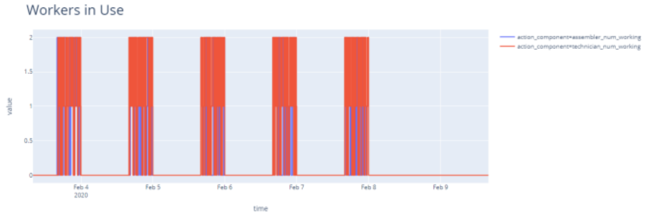
Worker Plots

2

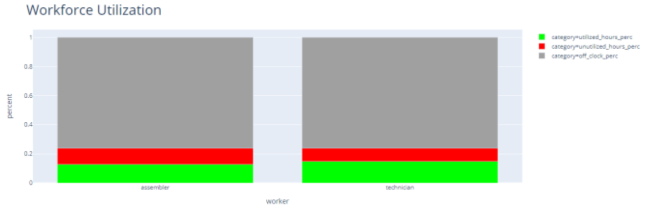


```
worker_cap_dict = {
    'assembler': 2,
    'technician': 2
}
worker_df = output_viewer.get_worker_state_df(state_df=state_df,
                                              start_time=start_time,
                                              duration=duration,
                                              worker_list=worker_cap_dict.keys())

output_viewer.plot_worker_in_use_timeline(worker_df=worker_df,
                                          worker_list=worker_cap_dict.keys())

output_viewer.plot_worker_utilization(worker_df=worker_df,
                                      worker_cap_dict=worker_cap_dict)
```

2 Worker dictionary should match worker names as keys and their capacities as values in main.ipynb, create dataframe of worker state changes with helper function.



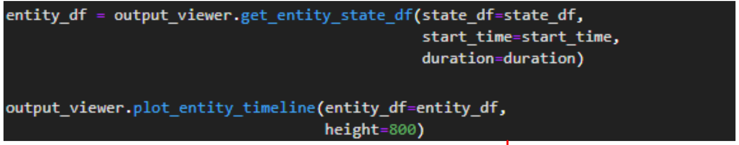
Workers in Use



Workforce Utilization

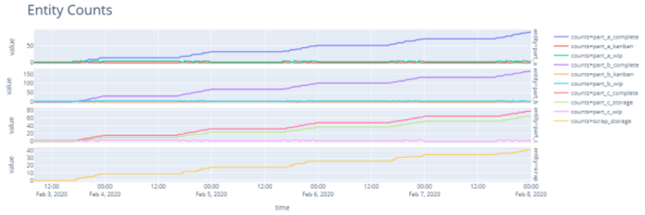
Entity Plots

3



```
entity_df = output_viewer.get_entity_state_df(state_df=state_df,
                                              start_time=start_time,
                                              duration=duration)

output_viewer.plot_entity_timeline(entity_df=entity_df,
                                   height=800)
```

3 Create dataframe of entity count state changes with helper function.



Entity Counts