CS1112 Fall 2025 Project 5 due November 14

You must work either on your own or with one partner. If you work with a partner you must first register as a group in CMS and then submit your work as a group. *Adhere to the Code of Academic Integrity.* A group submission requires that both partners are **joint authors** of **all parts** of the project—splitting a project with each partner working on their own part is a violation of academic integrity.

For a group, "you" below refers to "your group." You may discuss background issues and general strategies with others and seek help from the course staff, but the work that you submit must be your own. In particular, you may discuss general ideas with others or look up general programming techniques, but you may not work out the detailed solutions with others, and you may not search for or use tools to generate solutions to specific parts of the assignment. It is not OK for you to see or hear another student's code and it is certainly not OK to copy code from another person or from published/Internet sources, including generative artificial intelligence tools. If you feel that you cannot complete the assignment on you own, seek help from the course staff. Please read the "Academic Integrity" section in the Syllabus for further details and for the rationale of the CS1112 Academic Integrity policy.

Objectives

Completing this project will build your understanding of object-oriented programming. You will also get practice on developing and testing code *incrementally*—one class (or even one method) at a time.

Ground Rule

Since this is an introductory computing course, and in order to give you practice on specific programming concepts, some functions/features are forbidden in assignments. *For the* **required portion** *of this project, use only the concepts and tools discussed in class and in this document.*

Robot Task Allocation

In warehouses, hospitals, and homes, we may have robots that assist us in various ways. Robots as they are today, however, often have physical limitations such that they are capable of performing only specific tasks. Scheduling robots under constraints—physical limitations, limited availability, resource conflicts—is crucial. Some robots can pick up only specific items due to weight lifting limitations or arm arrangements. In this project, you will write code to delegate tasks amongst a team of heterogeneous robots.

You will be playing the role of a manager of a robot-assisted warehouse. You have a team of heterogeneous robots, a set of items you need to pick up, and a time period during which the robots have to pick up as many items as possible. These robots have different skills and abilities, so you must come up with a plan for the robots for the time period in question.

The plan for the team of robots is called an "allocation." A successful allocation assigns robots to pick up specific items, presenting the robots with the paths to take to the items and the windows of time where they will be performing pickup tasks.

We provide the *design* of the classes that you will *implement* in this project. Read carefully about the reasoning behind the design below. You will see *how objects of several classes interact* in this robot task allocation project. You will see that one of the classes that we will use is `Interval`, a class that we have used in lecture. As we have discussed before, a class that is designed well can be used in different projects and situations, including being extended for specific purposes.

Another focus of this project is for you to practice *testing* individual methods and classes, one at a time. We suggest tests throughout the project description, but *you need to generate more tests* beyond our suggestions in order to make sure that the methods you implement are correct. Do not assume that just passing the test cases we have suggested implies that your method is entirely correct. As you have seen in our weekly exercises, *each method/function should be tested with multiple, distinct test cases that are representative of different input scenarios.*

0 Object-Oriented Design

The object-oriented design component of deciding which classes and methods should be made has already been done—your job now is to implement methods according their specifications. A part of this process will involve writing tests as you build new functionality. Our design involves three classes: `Interval`, `Item`, and `Robot`. We then write a function `run_robots()` to read the initial positions and attributes of robots and items as data from a file, instantiate objects of the different classes, and perform task allocation. Class `Interval` is given, and partial code is given in the other classes.

Below is a summary of the classes, showing what attributes and methods are available in each class. Attributes and methods that are "hidden"[1] are shown using an open circle, ○, and those with `public` access are shown using a filled circle, •. The skeleton code given in the zip file `p5files.zip` gives further details on each item. Download the zip *and extract the files from the zip.*

Interval

Attributes:
- • left
- • right

Methods:
- • __init__()
- • get_width()
- • shift()
- • is_in()
- • add()
- • overlap()

Item

Attributes:
- • id_
- • name
- • weight
- • loc
- • arm_requirement
- • duration
- • picked_window

Methods:
- • __init__()
- • valid_pickup()
- • update_pickup_status()
- • draw()

Robot

Attributes:
- ○ _id_
- ○ _max_weight
- ○ _total_time
- ○ _init_loc
- ○ _items_picked

Methods:
- • __init__()
- • get_id()
- • get_items_picked()
- • total_operation_time()
- • latest_resting_loc()
- • draw()
- • travel_steps()
- • pick()
- • get_location()

[1] By convention, attribute and method names that begin with the underscore '(_)' are treated as hidden—to be accessed *within* their class only, not from other classes. However, in Python no attributes or methods are truly "private."

So which class should you work on first? *Answer: the most independent class, the one that doesn't depend on other classes.* We will start with class `Interval`. As you complete the classes, do not change the names of the attributes or methods. You should not need to add any extra methods or attributes.

1 Class `Interval`

Use the `interval.py` file extracted from the Project 5 zip. (We used multiple versions of the class definition during lecture; please be sure to use the version released for Project 5.)

Read the class definition and notice the use of *default values* for some parameters in the initializer. Assuming that all the skeleton files are in your current working directory, experiment with class `Interval` by typing the following statements in the Console:

```
in1= Interval(3,9)          # Instantiate an Interval with endpoints 3 and 9
print(in1)
print(in1.left)             # Should be 3. The attributes are "public", so it's
                            # possible to access the attribute left directly.
in2= Interval()             # Instantiate an Interval with default end points
print(in2)
o= in1.overlap(Interval(5,15))
print(o)                    # o references an Interval with endpoints 5 and 9
print(f"{o.get_width()=}") # Should be 4, the width of the Interval
                            # referenced by o
```

Self-documenting expressions in f-strings for debugging

Take a close look at the last `print` statement and the output that it produced. We know already that an expression inside curly brackets in an f-string gets evaluated. Now you see that the expression *followed by an equal sign* (`=`), all inside curly brackets, produces a string that includes *both the text of the expression and the evaluated value of that expression.* This is super handy for debugging as you can see in the printed output both the result and the code that produced that result.

We started a file `testscript.py` to help you test your classes. `testscript.py` has been populated with the few lines of code presented above (as tests for the `Interval` class), as well as a small number of test cases for other classes. Since class `Interval` is given, you are not actually testing the methods but instead you are practicing how to access the attributes and methods of class `Interval`. For the other classes you will need to add *more* test cases to `testscript.py` in order to fully test the methods you implement. You will submit `testscript.py` as a record of your development process.

If there is anything that you don't understand in this class, ask and figure it out before moving on! You want to make sure that you understand everything here before working on another class that depends on class `Interval`.

2 Class `Item`

An `Item` is some product in your warehouse that needs to be picked up by a robot. Read the partially completed file `item.py`, paying close attention to the class docstring and the docstring of the given initializer in order to learn about the attributes of `Item`.[2]

[2] There is an attribute named `id_` (not `id`). This is because `id` is the name of a Python built-in function, so we do not want to name an attribute the same as a built-in function.

Implement the following methods:

- `valid_pickup()`

  Test your method immediately after implementation. Look at the given code in the section **Chaining up references** below as some examples of tests. As you work on individual methods, be sure to add code to `testscript.py` *to call that method as a test.* Test each method multiple times with different input values representative of various *valid* scenarios.[3] It is tempting to rush through coding without stopping to test thoroughly, but that will burn more time in the end because you will then have to deal with many confounding errors in your buggy atop buggy code when you finally get around to trying to run your program.

- `update_pickup_status()`

- `draw()` Use the `draw_rect()` function we have provided from the `shapes` module. To add text to graphics, use `plt.text()` function. For example

  ```
  s= str(u);  # u is a number
  plt.text(x, y, s, horizontalalignment="center")
  ```

  puts the string `s` at coordinates (x,y) with center-alignment on the current figure. You will use this method later to animate the simulation.

Chaining up references

Consider the following fragment (assuming all necessary `import` is done):

```
# Create an Item with the id 1, the name 'basket', a weight of 2 lbs,
# positioned at (3, 4). It has no arm requirement for pickup and a
# necessary duration for pickup of 3 time steps.
i1= Item(1, 'basket', 2, [3, 4], 0, 3)

print(f"{i1.id_=}")                             # Should be 1
print(f"{i1.loc=}")                             # Should be [3, 4]
print(f"{i1.valid_pickup(4, 2)=}")              # Should be True
print(f"{i1.valid_pickup(1, 0)=}")              # Should be False
print(f"{i1.update_pickup_status(2)=}")         # Schedule i1 for pickup starting
                                                # at time 2
print(f"{i1.picked_window.left=}")              # Should be 2
print(f"{i1.picked_window.right=}")             # Should be 5

# The amount of time it takes for a robot to pick up the item should be
# equal to the item's duration attribute
print(f"{i1.picked_window.get_width()=}")       # Should be 3
```

Notice how references are "chained together" using the dot notation in the last three statements above. In the last statement ...

`i1` references an `Item`.

An `Item` object has the attribute `picked_window`, which references an `Interval`. Therefore,

`i1.picked_window` references an `Interval`.

An `Interval` has an instance method `get_width()`, therefore

`i1.picked_window.get_width()` returns a number (the width).

[3]Where the inputs to the methods are things we'd expect (e.g. the correct type).

Don't forget to test your `draw` method. Here are a couple examples:

```
# At time 3, the item is not yet fully picked up, so it should be drawn if
# method draw is called. The statements below should give a red rectangle in
# figure window 1, centered at (3, 4), with "1" (the id) inside the rectangle.
plt.figure(1); i1.draw(3)

# At time 5, the item is picked up, so it should not be drawn if method draw is
# called. The statements below should show figure window 2 but nothing drawn.
plt.figure(2); i1.draw(5)
```

Test your methods thoroughly—use more tests than shown in the demonstrations above. Make sure you understand everything in this class and fix any bugs before moving on.

3 Class `Robot`

Figure 1: A standard robot: No arms (This is the Anki Vector robot)

The `Robot` has three main states. A waiting state (where it stays motionless at a location), a traveling state (where it's navigating to the location of an item), and a pickup state (where it's in the process of picking up an item).

This standard `Robot` is arm-less. It "picks up" items by coming into contact with `Item`s and then pushing them around as it moves. Note that the `max_weight` attribute is the limit for *picking up* one `Item` and is not a max capacity for holding (storing) `Item`s. As a simplification, we assume that the `Robot` has unlimited storage capacity—it can hold any number of `Item`s onboard as long as each one weighs no more than `max_weight` at pickup. Once an `Item` is picked up, it is held by the `Robot` until the simulation time ends; it cannot be picked up again.

Read the partially completed class `Robot` in the file `robot.py`. A detailed description of its attributes is given in the class docstring; read it carefully. Your job is to implement the initializer and all the methods (except the one with provided code).

3.1 Implement the initializer

Read the specification (docstring) for the class and initializer carefully before writing any code.

Test your code for this initializer and other methods with the provided test case in `testscript.py`, and then add *more* test cases to the script to perform a more thorough check on correctness. We will stop reminding you to generate and use diverse test cases from this point on, but going forward, *please carefully test every method you implement!*

3.2 Implement `get_id()` and `get_items_picked()`

`get_id()` This method and the next "get" method are super short (can be one line of code).

`get_items_picked()` Returns a copy of the robot's `_items_picked`. Careful! The `_items_picked` attribute is a `list` of objects. You need to use the `copy.deepcopy` function to make a true copy. Assuming that the `copy` module has been imported, the syntax is `copy.deepcopy(`*list_you_want_to_copy*`)`

3.3 Implement `total_operation_time()` and `latest_resting_loc()`

The implementations of these two methods should be relatively short and rely heavily on the robot's `_items_picked` attribute. Please read the docstrings in these methods and the hints below carefully.

`total_operation_time()` Return the total operation time of the robot `immediately after` completing its most recent item pickup. How do you find this value? Note that the `Robot` attribute `_items_picked` is a list of all the items picked up by the robot. If that list is empty, then the robot hasn't picked up any item. If the list is not empty, then the last item in that list is the most recent item picked up by the robot. Now recall that an `Item` has an attribute that indicates when it got picked up—look back at the `Item` class or the class summary on Page 2 of this document for that attribute and you will have the answer.

`latest_resting_loc()` If no item has been picked up, the robot's latest resting location is its initial location. Otherwise, the latest resting location is where it was picking up the last item—again, this can be accessed through the `_items_picked` attribute.

3.4 Implement `draw()`

`draw()` Draw a blue circle representing the robot at the specified location. Use the `draw_disk()` function we have provided. We will use this method later for the animation.

3.5 Implement `travel_steps()`

`travel_steps()` This method returns a path that the robot can take from a current location to a destination location. The path is represented as a list of locations, one location for each time step as the robot travels. Therefore the method returns a nested list where each inner list stores the x- and y-coordinates of the robot at a time step. Recall that the `Robot` can move only NESW, one unit distance in one time step.

3.6 Implement `pick()`

`pick()` This method simulates the robot picking up an `Item`; it is the first method we write that has the objects of two classes interact! The method must first determine whether the pickup can happen. As detailed in the step-by-step guide in the starter code, an item can be picked up if:

- The robot's physical properties allow for it.
- There is enough remaining time in the simulation to pick up the item. Here each robot's `_total_time` attribute stores the total simulation time; this attribute is the same for all robots.
- The item is not already scheduled for pickup by another robot.

Method `pick()` has a `do_pick` input parameter, default to `True`, and a `num_arms` parameter, default to 0. These two parameters will *always* take these default values in this project (but will take on different values in the next project).

When `do_pick` evaluates to `True`, as is the case in this project, this method must also:

- Update the `Item`'s attributes to represent the pickup (*Hint*: you wrote a method earlier for updating an `Item`'s pickup status and you should use it).
- Update the `Robot`'s own `_items_picked` attribute.

3.7 The provided `get_location()` method

We have provided the code for this method in the `Robot` class. Do not modify the provided code, but read it to see what it does. Two important things to note:

- You will need to call this method to animate the robots later, in the `main` module.
- This method relies on the correctness of the `total_operation_time()`, `latest_resting_loc()`, and `travel_steps()` methods you implemented. Be sure that you have tested these previous methods fully already!

4 The `run_robots()` function

Let's use the classes that we have so far to attempt an allocation! The main job of the function `run_robots()` is to read data from a file, create robot and item objects, and perform an allocation. Some of the code for reading a data file is given, but you need to add code to complete the job.

Start by taking a look at the data file `room1.txt`, which contains lines of comma-delimited data. The first line gives the simulation parameters (number of time steps, horizontal length of the room, vertical length of the room). Each of the remaining lines contains the data for a `Robot` or an `Item`. A line containing data for a `Robot` has the following format:

Robot, ID, max_weight, [starting_x, starting_y]

A line containing data for an item has the following format:

Item, ID, name, weight, [location_x, location_y], number_of_arms_needed, duration

Notice that the line containing data for a robot does not include its `_total_time`. As mentioned earlier, a robot's `_total_time` is the same as the number of time steps for the simulation.

You should assume that the data in the file is correct: there are no conflicts with IDs, the given locations are all inside the room, and there is no missing data. For simplicity, no two items share the same location, and no two robots share the same location.

4.1 TASK 1: Make lists of `Robots` and `Items`

The given code reads the data file and parses[4] each line. Your job is to modify the code within the file `main.py` marked for **TASK 1** to

[4]To parse a line is to separate it into pieces according to some criterion.

- Create one `Robot` object for each line of robot data and put it into the `robots` list.
- Create one `Item` object for each line of item data and put it into the `items` list.

Given the data in the file `room1.txt`, the resulting `robots` list should have 2 elements, each element storing the reference to a unique `Robot`, and the resulting `items` list should have 5 elements, each element storing the reference to a unique `Item`.

Testing and debugging: use the variable explorer, print statements, and the debugger to catch any bugs at this stage. After debugging, be sure to remove any print statements that you might have added. You can add more data lines to the data file if you wish for further testing.

4.2 TASK 2: Simple allocation

In the file `main.py`, implement the function `simple_allocation()` as specified. The method `pick()` that you implemented in the `Robot` class will be useful here and the thorough testing that you have done earlier will pay off now. Recall that our `Robot` has unlimited storage and can pick up multiple items, one at a time.

4.3 TASK 3: Animate `Items` and `Robots`

Complete the `animate()` function as described in the docstring in order to animate each timestep under your allocation. You have been writing `draw()` methods for our objects along the way to make this part easier. We initialized the drawing of the room (figure window setup) for you and provided the animation loop. You must draw all `Items` and `Robots` in the room at each timestep. Note that:

- The `draw()` method in the `Item` class takes a time step as input.
- The `draw()` method in the `Robot` class needs to take a robot's current location as input. You will need to call the provided `get_location()` method in the `Robot` class.

You can change the "blink time," the argument `b` in the call `plt.pause(b)` for producing the animation. A value of 0.5 to 1.5 seconds is recommended.

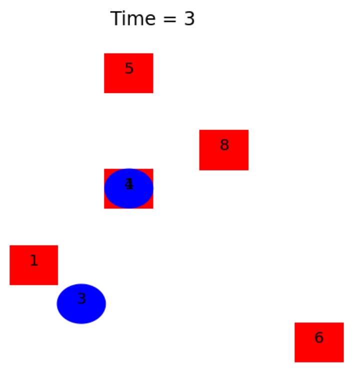

Figure 2: An example of a single timestep of executing an allocation

4.4 TASK 4: Print Out Results of Allocation

Implement the `output_results()` function to print out the results from your task allocation in the format below. Your task allocation result may not be identical to ours.

```
-------------------------------
Robot 1 picked 2 items in 16 timesteps
  rubber duck (ID 4): Assigned at time 2, picked up at time 5
  paperclip (ID 6): Assigned at time 13, picked up at time 16
Robot 3 picked 1 items in 12 timesteps
  hockey puck (ID 8): Assigned at time 10, picked up at time 12
-------------------------------
The robots were not able to pick up:
  apples (ID 1)
  cat (ID 5)
-------------------------------
```

Submit your `testscript.py`, `robot.py`, `item.py`, and `main.py` (after you register your group) on CMS.

Check that you have submitted *your* file and not our skeleton! On CMS, after you upload your files, you should see the option to download your submitted files. Download and open them to check that they are indeed the files that you intended to submit!