
# Putting It Together: Reading Files + Using Data Structures

Now you have seen a lot of the "pieces" for programs: loops, data structures (lists, sets, dictionaries), file IO. One of the next major steps is putting them together to make a big program. In this lesson, we are going to talk about how to approach a larger program, and put these pieces together. At the end of this lesson you are going to write a slightly larger program ("random story"). Next week, you are going to write a much larger program (Monte Carlo poker simulation) which will be spread out across the week.

## Top-Down Design, Bottom-Up Implementation

One challenge in putting together a large program is breaking it down into manageable pieces. If a software project sounds too big to handle, the first thing you should do is think about how to break it down into smaller tasks. The approach we like is "top-down design, bottom-up implementation." What exactly does this mean? Top-down design means that you start with the whole problem, and come up with a high-level algorithm for it. That algorithm will be in terms of tasks that are too big and complex to just write as single functions, but that is OK. You then repeat the process for those sub-tasks until you get to something you can write. At this point you are at the "bottom" of the design—the smallest pieces that will go into it. You start implementation from there, making (and testing!) small pieces and putting them together. As you assemble these pieces from the bottom up, you get a larger and larger working program until finally you have your entire program.

To make this concept more concrete, we are going to consider an example problem: you want to write a program that reads an input file with one line per student listing their courses, and want to produce one file per course with the class roster. For example, you might read in a file like

- Vin Venture: ECE551, ECE550, ECE651
- Elend Venture: ECE551, ECE550, ECE568
- Kaladin Stormblessed: ECE568, ECE581, ECE651
- Frodo Baggins: ECE550, ECE651, ECE581, ECE590
- Hermione Granger: ECE590, ECE551

and then produce one roster per class (ECE550, ECE551, ECE651, ECE568, ECE581, ECE590) with all the students in that class listed—for example, the 551 roster would contain

- Vin Venture
- Elend Venture
- Hermione Granger

This problem is larger than what you all have been doing, but all the pieces that go into it are things you have mastered in the previous lessons. While it might seem daunting to start on this larger problem, we can start with a very high level algorithm (and we'll note the Seven Steps can help you greatly here!). My high level algorithm is:

```
Start by reading the input into a list of Student objects (each of which has their name + list of courses)

Find the list of all courses taken by any student

For each course c in the list of all courses:

    Produce the output file for course c
```

This high-level algorithm breaks our problem down into 4 tasks: create a Student class, read the input, find the list of all courses, and produce the output for a given course. The first of those, creating a Student class, is already at the size of "we can implement it." In fact, what we have described above for it lends itself to implementation with the following class

```
class Student:
    def __init__(self, name):
        self.name = name
        self.courses = []
        pass
    pass
```

Note that there is nothing super-fancy here: we just created a class with a name and a list of courses. The class does not really do much of anything yet, but that is OK. It gives us a starting point to work with, and we can add features to it later. For example, we might quickly realize that we want a way to add classes to the list of courses the student is taking, and get the name and list of courses:

```
class Student:
    def __init__(self, name):
        self.name = name
        self.courses = []
        pass

    def get_name(self):
        return self.name

    def get_courses(self):
        return self.courses

    def add_course(self, course):
        self.courses.append(course)
        pass

    pass
```

Now we should do some testing to make sure we did not make any mistakes here and we have finished our first task!

The remaining three tasks may be a bit more complex, but that is OK. We will come up with an algorithm for them, and end up with more tasks to do. As we continue the example, we might take reading the input as our next task. My algorithm for this might just be:

```
Start with ans = the empty list

For each line in the input file

   Make a student object from the current line's data

   Add the newly created student object to ans

Now ans is your answer
```

This algorithm really only produces one new sub-task (make a Student object from the current line's data) as each other line translates into one or two lines of Python. We'll come back to translating this algorithm to code *after* we do our new sub-task. We could do it now, but we want to test it as soon as we write it—if we do make_student_from_line first, then we can test that, and build this algorithm on well-tested code.

To do make_student_from_line, we need to take an input line like this

- Vin Venture: ECE551, ECE550, ECE651

and produce a Student object with name='Vin Venture' and courses = ['ECE551', 'ECE550', 'ECE651']

This particular task could be small enough to "bottom out" (meaning we don't have anymore sub-tasks from it), or we could end up with a few other sub-tasks. We'll note that which way we go depends on a few factors. One is how you think about the problem. For example, we might think of our algorithm in terms of getting the name and courses from the line—in which case, those result in natural sub-tasks. On the other hand, you might think about this in terms of built-in string operations in Python (such as split() and strip()). Even if you think about this in terms of built-in string operations, there can still be some readability benefits to breaking out smaller tasks—giving things a name (by making them a function) helps the reader understand what the code is doing. For example, we might come up with the following Python code:

```
# Takes a string like "Vin Venture: ECE551,  ECE550, ECE651" and
# produces a Student object with the name being the first field
# (before the colon)
# The list of courses comes from the remainder of the line, separated
# by commas

def make_student_from_line(line):
    split_by_colon=line.split(sep=":", maxsplit=1)
    name=split_by_colon[0]
    ans = Student(name)
    if (len(split_by_colon) > 1):
        courses = split_by_colon[1]

    else:
        courses = []
        pass

    for c in courses.split(","):
        ans.add_course(c.strip())
        pass

    return ans
```

I would say that this code is moderately easy to read, but could be improved. If we pulled out the code to get the courses into its own function and gave it a name (like get_courses_for_student) it could become clearer to the reader. We will also note that if we pulled this out into a function to get the courses as a list, we might want to change that to take the whole list rather than to add one course at a time, and that is OK—in real software development, people change their code as their thoughts on what is needed and how to design it evolve.

Now is a great time to emphasize that we should test this function before proceeding. We do NOT want to wait until we have written the entire program to test. We want to test each piece as we build it—which is why we like "bottom-up" implementation. This make_student_from_line function can be tested all on its own. Once we finish testing it, we can go back to our algorithm to read the input line-by-line:

```
Start with ans = the empty list

For each line in the input file

  Make a student object from the current line's data     # this is the function we just wrote

  Add the newly created student object to ans

Now ans is your answer
```

Now everything can be translated into Python, as we wrote the function to do the complex part (and tested it so we are confident!)

```
def read_input(fname):
    ans=[]   # Start with ans = the empty list

    with open(fname, "r") as f:   # this + the next line are "for each line in the input file, as we have to open it first

        for line in f:
            curr_student = make_student_from_line(line)  # Make a student object from the current line's data
            ans.append(curr_student)  # Add the newly created student object to ans
            pass

        pass

    return ans # Now ans is your answer
```

At this point we once again have something we can test—we can go create some small input files and try read_input on them to make sure it gets the right answer.

Having finished our task to read the input, let us return to the 4 sub-tasks of our highest level algorithm:

1. Create a Student class - done!
2. Read the input - done!
3. Find the list of all courses
4. Produce the output for a given course

We could just go ahead and do task 3, but we are going to take a moment to show you another technique to help with larger problems. Instead of actually doing task 3 now (implementing a function to find all the courses), we are going to make a placeholder implementation called a "stub".

```
def get_all_courses(students):

    return ["ECE551", "ECE651"]
```

This function isn't right (and isn't even right for the sample input we have), but it can serve as a placeholder for us to get a simplified version of our program working. In this case, the simplified version will only produce rosters for ECE551 and ECE550, and then we will come back and implement get_all_courses correctly later. We note that get_all_courses is not a super complicated function, so in real development, we would probably not use a stub here. However, we wanted to show you this technique in case it comes up in other contexts.

We'll also note that in many real contexts, a particular function may have a lot of different features that it needs to support. It is perfectly reasonable (and highly recommended) to build a working end-to-end program for a very small set of the features. Here, an end-to-end program means that it does all the basic parts of what the program needs to do (e.g., reads some input and produces some output in the right formats), but may be missing a lot of what it really needs to do. Once you have that, adding more features into a working system is easier (especially in a team) than building all of something at once. We will also note that a stub can be a good way to handle code that a different team-member will write. If someone else on your team is writing get_all_courses, you can put in a place holder until they finish it, and then swap it out for their real implementation. We note that in a real team setting, you would want to split work up into larger pieces than get_all_courses would be, but we want to show you the idea in the context of a managable example.

After we "skip over" task 3, we could go and do task 4. Here, we need to iterate over the course list (from step 3), and for each course, open the proper file and write the output (iterating over the students to see if they have the current course in their list). All of that fits well into one function and is correspondingly similar in size to what you all have been doing—hopefully if we gave you a few more specifics (like how to name the files, etc.) you could do that task.

Once you have done task 4, you could go back and translate the high level algorithm into code. With all of the sub-pieces ready, this translation is fairly straightforward:

```
def make_rosters(fname):
  # Start by reading the input into a list of Student objects (each of which has their name + list of courses)
  students = read_input(fname)
  # Find the list of all courses taken by any student
  all_courses = get_all_courses(students)
  #For each course c in the list of all courses:
  for c in all_courses:
      #Produce the output file for course c
      write_roster_for_course(students, c)    #subtask 4, not shown
      pass
  pass
```

Once we have put that all together, we could test the whole program with the understanding that it only produces rosters for ECE551 and ECE550. After we are happy with that, we would go back and replace the stub for get_all_courses with a real implementation, then test that function on its own and test the whole program again. We will note that that the task we skipped over—getting all the courses—is a great use for a set (which you learned about before). One of the things with a set is that if you add an item that is already there, nothing happens. This is exactly the behavior we would want here, as we want each course to appear once in the final result. I'll also note that our original algorithm said "the list of all courses". We could build up that answer in a set, then convert it to a list, or we could realize that we do not really need anything that a list has that a set does not, and change our algorithm to work with a set (which in this case is just changing the comment). Having seen this idea in one context, we are now going to have you do a programming assignment on a different problem. This task is a bit larger than what you have been previously working with, so we are allocating a lot of time to it today on the page below.

## Random Story Assignment

Random Story