Lab 4: SAT Solver

Table of Contents

1) Preparation

This lab assumes you have Python 3.7 or later installed on your machine.

The following file contains code and other resources as a starting point for this lab: `lab4.zip`

Most of your changes should be made to `lab.py`, which you will submit at the end of this lab. Importantly, you should not add any imports to the file.

This lab is worth a total of 4 points. Your score for the lab is based on:

- answering the questions on this page (0.5 points)
- passing the test cases from `test.py` under the time limit (2 points), and
- a brief "checkoff" conversation with a staff member to share your code, including discussion of the bullets below and a review of your code's clarity/style (1.5 points).

For this lab, you will only receive credit for a test case if it runs to completion within the indicated time limit on the server.

Please also review the collaboration policy before continuing.

The questions on this page (including your code submission) are due at 4pm on Friday, Oct 11.

2) Introduction

From recreational mathematics to standardized tests, one popular problem genre is *logic puzzles*, where some space of possible choices is described using a list of rules. A solution to the puzzle is a choice (often the one unique choice) that obeys the rules. This lab should give you the tools to make short work of any of those puzzles, assuming you have your trusty Python interpreter.

Here's an example of the kind of logic puzzle we have in mind.

The 6.009 staff were pleased to learn that grateful alumni had donated cupcakes for last week's staff meeting. Unfortunately, the cupcakes were gone when the staff showed up for the meeting! Who ate the cupcakes?

1. The suspects are Adam, Karl, Pete, Duane, and Tim the Beaver.
2. Whichever suspect ate any of the cupcakes must have eaten *all* of them.
3. The cupcakes included exactly two of the flavors chocolate, vanilla, and pickles.
4. Adam only eats pickles-flavored cupcakes.
5. Years ago, Pete and Duane made a pact that, whenever either of them eats cupcakes, he must share with the other one.
6. Karl feels strongly about flavor fairness and will only eat cupcakes if he can include at least 3 different flavors.

Let's translate the problem into **Boolean logic**, where we have a set of variables, each of which takes the value True or False. We write the rules as conditions over these variables. Here is each rule as a Python expression over Boolean variables. We include one variable for the guilt of each suspect, plus one variable for each potential flavor of cupcake.

In reading these rules, note that Python `not` binds more tightly than `or`, so that `not p or q` is the same as `(not p) or q`. It's also fine not to follow every last detail, as this rule set is just presented as one example of a general idea!

You may also find that some of our encoding choices don't match what you would come up with, such that our choices lead to longer or less comprehensible rules. We are actually intentionally forcing ourselves to adhere to a restricted format that we will explain shortly, and that will ultimately make the job of *solving* these kinds of problems more straightforward.

```
rule1 = (adam or karl or pete or duane or tim)
# At least one of them must have committed the crime!  Here, one of these
# variables being True represents that person having committed the crime.


rule2 = ((not adam or not karl)
     and (not adam or not pete)
     and (not adam or not duane)
     and (not adam or not tim)
     and (not karl or not pete)
     and (not karl or not duane)
     and (not karl or not tim)
     and (not pete or not duane)
     and (not pete or not tim)
     and (not duane or not tim))
# At most one of the suspects is guilty.  In other words, for any pair of
# suspects, at least one must be NOT guilty (so that we cannot possibly find
# two or more people guilty)
# Together, rule2 and rule1 guarantee that exactly one suspect is guilty.


rule3 = ((not chocolate or not vanilla or not pickles)
     and (chocolate or vanilla)
     and (chocolate or pickles)
```

```
      and (vanilla or pickles))
  # Here is our rule that the cupcakes included exactly two of the flavors.  Put
  # another way: it can't be that all flavors were present, and among any pair of
  # flavors, at least one was present.


  rule4 = ((not adam or pickles)
       and (not adam or not chocolate)
       and (not adam or not vanilla))
  # If Adam is guilty, this will evaluate to True only if only pickles-flavored
  # cupcakes were present.  If Adam is not guilty, this will always evaluate to
  # True.  This is our way of encoding the fact that, if Adam is guilty, only
  # pickles-flavored cupcakes must have been present.


  rule5 = (not pete or duane) and (not duane or pete)
  # If Pete ate cupcakes without sharing with Duane, the first case will fail to
  # hold.  Likewise for Duane eating without sharing.  Since Pete and Duane only
  # eat cupcakes together, this rule excludes the possibility that only one of
  # them ate cupcakes.


  rule6 = ((not karl or chocolate)
       and (not karl or vanilla)
       and (not karl or pickles))
  # If Karl is the culprit and we left out a flavor, the corresponding case here
  # will fail to hold.  So this rule encodes the restriction that Karl can only
  # be guilty if all three types of cupcakes are present.


  satisfied = rule1 and rule2 and rule3 and rule4 and rule5 and rule6
```

The piece of code above is a Python program that will tell us whether a given assignment is consistent with the rules we have laid out. For example, if we had set the following variables (representing the hypothesis that Adam was guilty and that only Pickles-flavored cupcakes were present):

```
adam = True
karl = False
pete = False
duane = False
tim = False

pickles = True
vanilla = False
chocolate = False
```

and then run the code, the `satisfied` variable would be set to `False` (since `rule3` would be `False`), indicating that this assignment did not satisfy the rules we had set out.

While code like the above could be useful in certain situations, it doesn't help us *solve* the problem (it only helps us check a possible solution). In this lab, we'll look at the problem of Boolean Satisfiability: our goal will be, given a description of Boolean variables and constraints on them (like that given above), to find a set of assignments that satisfies all of the given constraints.

2.1) Conjunctive Normal Form

In encoding the puzzle, we followed a very regular structure in our Boolean formulas, one important enough to have a common name: **conjunctive normal form (CNF)**.

In this form, we say that a *literal* is a variable or the `not` of a variable. Then a *clause* is a multi-way `or` of literals, and a CNF formula is a multi-way `and` of clauses.

It's okay if this representation does not feel completely natural. Some people find this form to be "backwards" from the way they would otherwise think about these constraints. But forcing our constraints to be in this form can greatly simplify our problem. We'll try in this writeup to help you with the pieces of the lab involving converting expressions to CNF.

2.1.1) Python representation

When we commit to representing problems in CNF, we can represent:

- a *variable* as a Python string
- a *literal* as a pair (either a list **or** a tuple), containing a variable and a Boolean value (`False` if `not` appears in this literal, `True` otherwise)
- a *clause* as a list of literals
- a *formula* as a list of clauses

For example, our puzzle from above can be encoded as follows, where again it is OK not to read through every last detail. **Note again that literals can either be tuples or lists**.

```
rule1 = [[['adam', True], ['karl', True], ['pete', True],
          ['duane', True], ['tim', True]]]

rule2 = [[('adam', False), ('karl', False)],
         [('adam', False), ('pete', False)],
         [('adam', False), ('duane', False)],
         [('adam', False), ('tim', False)],
         [('karl', False), ('pete', False)],
         [('karl', False), ('duane', False)],
         [('karl', False), ('tim', False)],
         [('pete', False), ('duane', False)],
         [('pete', False), ('tim', False)],
         [('duane', False), ('tim', False)]]


rule3 = [[('chocolate', False), ('vanilla', False), ('pickles', False)],
         [('chocolate', True), ('vanilla', True)],
         [('chocolate', True), ('pickles', True)],
         [('vanilla', True), ('pickles', True)]]

rule4 = [[['adam', False], ['pickles', True]],
         [['adam', False], ['chocolate', False]],
         [['adam', False], ['vanilla', False]]]

rule5 = [[['pete', False], ['duane', True]],
         [['duane', False], ['pete', True]]]

rule6 = [[('karl', False), ('chocolate', True)],
         [('karl', False), ('vanilla', True)],
         [('karl', False), ('pickles', True)]]

rules = rule1 + rule2 + rule3 + rule4 + rule5 + rule6
```

When we have formulated things in this way, the list `rules` contains a formula that encodes all of the constraints we need to satisfy.

2.1.2) Examples

Consider this Boolean formula.

```
c and (a or d) and (not b or a) and (not a or e or not d)
```

Write an equivalent CNF formula (as a Python literal), in the format used in the lab (e.g., `[[('a', True), ('b', False)], [('c', True)]]`).

Submit

You have submitted this assignment 0 times.

This question is due on Friday October 11, 2019 at 04:00:00 PM.

Now, consider this Boolean formula (which is **not** in CNF).

```
(a and b) or (c and not d)
```

This expression looks innocuous, but translating it into CNF is actually a non-trivial exercise! It turns out, though, that this expression does have a representation in CNF as:

```
(a or c) and (a or not d) and (b or c) and (b or not d)
```

or, in our representation, as:

```
[[('a', True), ('c', True)], [('a', True), ('d', False)], [('b', True), ('c', True)], [('b', True), ('d', False)]]
```

Notice that the above expression will be True if `a` and `b` are both True, or if `c` is True and `d` is False.

Now, try your hand at it. Consider the following Boolean Formula (not in CNF):

```
a and (not b or (c and d))
```

Write an equivalent CNF formula (as a Python literal), in the format used in the lab (e.g., `[[('a', True), ('b', False)], [('c', True)]]`).

Submit

You have submitted this assignment 0 times.

This question is due on Friday October 11, 2019 at 04:00:00 PM.

3) SAT Solver

A classic tool that works on Boolean formulas is a **satisfiability solver**, or SAT solver. Given a formula, either the solver finds Boolean variable values that make the formula True, or the solver indicates that no solution exists. In this lab, you will write a

SAT solver that can solve puzzles like ours, as in:

```
>>> print(satisfying_assignment(rules))
{'duane': False, 'pete': False, 'chocolate': False, 'karl': False,
 'adam': False, 'pickles': True, 'tim': True, 'vanilla': True}
```

The return value of `satisfying_assignment` is a dictionary mapping variables to the Boolean values that have been inferred for them (or `None` if no valid mapping exists).

So, we can see that, in our example above, Tim the Beaver is guilty and has a taste for vanilla and pickles!

It turns out that there are other possible answers that have Tim enjoying other flavors, but it also turns out that Tim is the uniquely determined culprit. How do we know? The SAT solver fails to find an assignment when we add an additional rule proclaiming Tim's innocence.

```
>>> print(satisfying_assignment(rules + [[('tim', False)]]))
None
```

3.1) The Naive Approach

There's one very easy way to solve Boolean puzzles: enumerate all possible Boolean assignments to the variables. Evaluate the rules on each assignment, returning the first assignment that works. Unfortunately, this process can take prohibitively long to run! For a concrete illustration of why, consider this Python code to generate all sequences of Booleans of a certain length.

When we have N Boolean variables in our puzzle, the possible assignments can be represented as length-N sequences of Booleans. Note that the example below uses a generator. [1]

```
def all_bools_generator(length):
    if length == 0:
        yield []
    else:
        for v in all_bools_generator(length-1):
            yield [True] + v
            yield [False] + v
def all_bools(length):
    return list(all_bools_generator(length))
```

Here's an example output.

```
>>> all_bools(3)
[[True, True, True], [False, True, True], [True, False, True],
 [False, False, True], [True, True, False], [False, True, False],
 [True, False, False], [False, False, False]]
```

We could get more ambitious and try to generate longer sequences.

```
>>> len(all_bools(3))
8
>>> len(all_bools(4))
16
>>> len(all_bools(5))
32
>>> len(all_bools(6))
```

```
64
>>> len(all_bools(20))
1048576
>>> len(all_bools(25))
# Python runs for long enough that we give up!
```

It's actually quite expensive even to run through all Boolean sequences of nontrivial lengths, let alone to test each sequence against the rules. The reason is that there are 2^N length-N Boolean sequences, and that kind of exponential function grows quite quickly as the length N of our mappings grows.

If we hope to be world logic-puzzle champions, we'll need to be ready for puzzles that lead to hundreds of Boolean variables. Are we out of luck if we want Python to do all the work? Worry not! In this lab, you will implement a SAT solver that uses a much smarter algorithm than *brute-force* enumeration of all assignments, thanks to the magic of backtracking search.

3.2) A Nicer Approach

Instead of enumerating all assignments, we will ask you to implement a more clever approach for Boolean satisfiability. One such approach is outlined below (with some of the details intentionally omitted):

We start by picking an arbitrary variable x from our formula F. We then construct a related formula F_1, which does not involve x but incorporates all the consequences of setting x to be `True`. We then try to solve F_1. If it produces a successful result, we can combine that result with information about x being `True` to produce our answer to the original problem.

If we could not solve F_1, we should try setting x to be `False` instead. If no solution exists in either of the above cases, then the formula F cannot be satisfied.

3.2.1) Updating Expressions

A key operation here is updating a formula to model the effect of a variable assignment. As an example, consider this starting formula.

```
(a or b or not c) and (c or d)
```

If we learn `c = True`, then the formula should be updated as follows.

```
(a or b)
```

We removed `not c` from the first clause, because we now know conclusively that that literal is `False`. Conversely, we can remove the second clause, because with `c` True, it is assured that the clause will be satisfied.

Note a key effect of this operation: *variable `d` has disappeared from the formula, so we no longer need to consider values for `d`.*

In general, this approach often saves us from an exponential enumeration of variable values, because we learn that, in some branches of the search space, some variables are actually irrelevant to the problem.

This pruning will show up in the assignments that your SAT solver returns: **you are allowed (but not required) to omit variables that turn out to be irrelevant**. Please note that in cases where a clause contains literals like so: `[('a', True), ('a', False)]`, the variable `a` is not considered an irrelevant variable because `a` needs to be assigned a value (`True` or `False`) in order for the clause to be valid. The difference here is that ignored irrelevant variables do not have to be assigned any value for the clause to be true.

If we had instead learned that `c` was `False`, we would update the formula instead as follows:

```
d
```

How did we get there? Note that, with `c` being `False`, the first clause is already satisfied. The second clause, though, will only be `True` if `d` is `True`.

3.2.2) Examples

Consider this CNF formula, in the form we use in this lab:

```
[
    [('a', True), ('b', True), ('c', True)],
    [('a', False), ('f', True)],
    [('d', False), ('e', True), ('a', True), ('g', True)],
    [('h', False), ('c', True), ('a', False)],
]
```

Write a CNF formula for the case where `a = True`, without using variable `a`. Write `None` if this assignment falsifies the formula, and write `[]` if this assignment renders the formula True.

Submit

You have submitted this assignment 0 times.

This question is due on Friday October 11, 2019 at 04:00:00 PM.

Write a CNF formula for the case where `a = False`, without using variable `a`. Write `None` if this assignment falsifies the formula, and write `[]` if this assignment renders the formula True.

Submit

You have submitted this assignment 0 times.

This question is due on Friday October 11, 2019 at 04:00:00 PM.

4) Implementation

Implement the function `satisfying_assignment` as described in `lab.py`. Your function should take as input a CNF formula (in the form described throughout this writeup). It should return a dictionary mapping variable names to Boolean values if there exists such an assignment that satisfies the given formula. If no such assignment exists, it should return `None`.

4.1) Doctests

A few small tests are offered for `satisfying_assignment` in the form of doctests, which you can use and extend for some basic checking.

Note that you can run the doctests by running `python3 lab.py` (not `test.py`). The lines at the very bottom of `lab.py` enable this.

Additionally, you can make a new line with the character "\n". For example:

```
>>> print("Hello World\nHello World")
Hello World
Hello World
```

4.2) Optimizations

A couple of further optimizations are likely to be necessary in order to pass all of the test cases quickly enough on the server:

- In the procedure described above, if setting the value of x immediately leads to a contradiction, we can immediately discard that possibility (rather than waiting for a later step in the recursive process to notice the contradiction).
- At the start of any call to your procedure, check if the formula contains any length-one clauses ("unit" clauses). If such a clause `[(x, b)]` exists, then we may set `x` to Boolean value `b`, just as we do in the `True` and `False` cases of the outline above. However, we know that, if this setting leads to failure, there is no need to backtrack and also try `x = not b` (because the unit clause alone tells us exactly what the value of `x` must be)!

  Thus, you can begin your function with a loop that repeatedly finds unit clauses, if any, and propagates their consequences through the formula. Propagating the effects of one unit clause may reveal further unit clauses, whose later propagations may themselves reveal more unit clauses, and so on.

Implementing function `satisfying_assignment` in this way should allow your code to pass the first half of our test cases in time.

You are free to add additional optimizations beyond what we laid out above, or even make broader changes to the algorithm, so long as you avoid "hard coding" for rather specific SAT problems (except for base cases like empty formulas).

5) Scheduling by Reduction

Now that we have a fancy new SAT solver, let's look at applying it to a new problem!

In general, it's possible to write a new implementation of backtracking search for each new problem we encounter, but another strategy is to *reduce* a new problem to one that we already know how to solve well. Boolean satisfiability is a popular target for reductions, because a lot of effort has gone into building fast SAT solvers. In this last part of the lab, you will implement a reduction to SAT from a scheduling problem (and note also that some of the test cases will use your SAT solver to solve sudoku puzzles, which can also be expressed in this form).

In particular, we are interested in the real-life problem of assigning 6.009 students to different sessions for taking a quiz. Each student is available for only some of the sessions, but each session has limited capacity. We want to find a schedule (assignment of students to sessions) that respects all the constraints.

Please implement the function `boolify_scheduling_problem(student_preferences, session_capacities)`, as described both below and in `lab.py`:

- The argument `student_preferences` is a dictionary mapping a student name (string) to a set of session names (strings) when that student is available.
- Argument `session_capacities` is a dictionary mapping each session name to a positive integer for how many students can fit in that session.
- The function returns a **CNF formula encoding the schedule problem**, as we explain next.

Here's an example call:

```
boolify_scheduling_problem({'Alice': {'basement', 'penthouse'},
                            'Bob': {'kitchen'},
                            'Charles': {'basement', 'kitchen'},
                            'Dana': {'kitchen', 'penthouse', 'basement'}},
                           {'basement': 1,
                            'kitchen': 2,
                            'penthouse': 4})
```

In English, Alice is available for the sessions in the basement and penthouse, Bob is available only for the session in the kitchen, etc. The basement can fit 1 student, the kitchen 2 students, and the penthouse 4 students. In this case, one legal schedule would be Alice in the basement, Bob in the kitchen, Charles in the kitchen, and Dana in the penthouse.

Your job is to translate such inputs into CNF formulas, such that your SAT solver can then find a legal schedule (or confirm that none exists).

The CNF formula you output should mention only Boolean variables named like `student_session`, where `student` is the name of a student, and where `session` is the name of a session. The variable `student_session` should be `True` if and only if that student is assigned to that session (for example, the variable `Bob_kitchen` should be `True` if Bob is in the kitchen, and `False` otherwise.

The CNF clauses you include should enforce exactly the following rules (which are discussed in more detail below):

1. Students are assigned to sessions included in their preferences.
2. Each student is assigned to exactly one session.
3. No session has more assigned students than it can fit.

Our requirement for this part of the lab is that your code *should not* solve the optimization problem. Rather, you should implement a translation to CNF formulas (which `satisfying_assignment` can then solve).[2]

During the checkoff, our staff will try to make sure you've followed this rule and not merely, say, implemented a backtracking search directly for the scheduling problem, to output trivial CNF problems that directly encode answers.

5.1) Encoding the Rules

Turning these rules into CNF formulas is a tricky task. We'll try to provide some guidance for thinking about each of these rules in the following sections (though if you get stuck, please don't hesitate to ask, either in person or via Piazza).

Note that each of these rules can be expressed as its own CNF formula, and the AND of these three rules represents the overall formula we need to solve. As such, you may wish to write a helper function to generate the formula for each of these rules, and to use those helper functions in `boolify_scheduling_problem`. **The examples below also make nice test cases (simpler than many we will test your code on)!**

5.1.1) Students Only In Desired Rooms

For each student, we need to guarantee that they are given a room that they selected as one of their preferences. In the example above, for example, we know that Charles must be in the basement or the kitchen, and that Alice must be in the basement or in the penthouse.

In the box below, enter a CNF formula expressing this constraint (students are assigned to rooms in their preferences) for the example data above (with Alice, Bob, Charles, and Dana). Use variable names of the form described above (e.g., `'Bob_kitchen'` represents Bob being in the kitchen).

Submit

You have submitted this assignment 0 times.

This question is due on Friday October 11, 2019 at 04:00:00 PM.

Check Yourself:

How could you generate this formula from the arguments given to `boolify_scheduling_problem`?

5.1.2) Each Student In Exactly One Session

This rule is a little bit trickier, but it may help to separate it into two pieces:

- each student must be in at least one session, **and**
- each student must be in at most one session.

We can generate formulas for each of these conditions, and combine them to construct the overall formula corresponding to this rule.

In fact, the first condition (that each student be assigned to at least one session) is redundant with our first condition (that each student be assigned to a session in their preferences).

So let's turn our attention to making sure that each student is assigned to *at most one session*. This one is a bit trickier, particularly since we need to put things in CNF. One flip of perspective that can be helpful is that, if we need each student to be in at most one session, that means that for any pair of sessions, any given student can be in only one session.

For example, one clause in this expression will say that Bob cannot be in both the kitchen and the basement. That is, we cannot have both `Bob_kitchen` and `Bob_basement` be `True` (or, to phrase it a different way, at least one of them must be `False`).

Note that there is a corresponding clause for every other pair of rooms; and each of these clauses has a corresponding clause for Alice (and the other students). For purposes of this question, include *all* rooms for each student, regardless of their preferences.

In the box below, enter a CNF formula expressing this constraint (students are in at most one session) for the example data above (with Alice, Bob, Charles, and Dana). Use variable names of the form described above (e.g., `'Bob_kitchen'` represents Bob being in the kitchen).

Submit

You have submitted this assignment 0 times.

This question is due on Friday October 11, 2019 at 04:00:00 PM.

Check Yourself:

How could you generate this formula from the arguments given to `boolify_scheduling_problem`?

5.1.3) No Oversubscribed Sections

This last rule is also fairly tricky, and it maybe requires a bit of a shift of perspective to express this constraint in CNF. However, it is similar to the previous rule in some ways.

We can think about this as: if a given room can contain N students, then in every possible group of $N+1$ students, there must be at least one student who is *not* in the given room.

For example, since the kitchen holds 2 people, we would need to consider all possible groups of 3 students and make sure that at least one of them is *not* in that room. For purposes of this question, include *all* groups of students, regardless of their preferences.

What about the penthouse? It has enough room for everyone, so there is no need even to include it in the constraints (it doesn't constrain our decision in any way)!

In the box below, enter a CNF formula expressing this constraint (no oversubscribed sections) for the example data above (with Alice, Bobe, Charles, and Dana). Use variable names of the form described above (e.g., `'Bob_kitchen'` represents Bob being in the kitchen).

Submit

You have submitted this assignment 0 times.

This question is due on Friday October 11, 2019 at 04:00:00 PM.

Check Yourself:

How could you generate this formula from the arguments given to `boolify_scheduling_problem`?

6) UI

As usual, we have provided a browser UI for this lab. When you have implemented both main functions of the lab, run `python3 server.py` for an interface to load scheduling test cases and see the results of scheduling. As usual, visit `http://localhost:8000/` to load the UI. Running a test case involves generating a CNF formula and searching for a satisfying assignment for it, each done by calling one of your functions.

7) Code Submission

Once you have debugged your code locally and are satisfied, upload your `lab.py` below (Please note that if a function only returns `None`, that is considered hardcoding and you will not receive any points for that function even if it passes certain test cases):

Select File No file selected

Submit

You have submitted this assignment 0 times.

This question is due on Friday October 11, 2019 at 04:00:00 PM.

8) Checkoff

Once you are finished with the code, please come to a lab session or office hour and add yourself to the queue asking for a checkoff. **You must be ready to discuss your code in detail before asking for a checkoff.** Since the clarity of your code will be evaluated as part of the checkoff, you may wish to take some time to comment your code, use good variable names, avoid repetitive code (create helper methods), etc.

Be prepared to discuss:

- In English, the general recipe for finding assignments that satisfy a given CNF formula.
- In English, your strategy for updating a formula based on setting an variable to be `True` or `False`.
- Your implementation of `satisfying_assignment`, including any helper functions.
- In English, the general recipe you came up with for translating scheduling problems into CNF formulas (including all three rules).
- Your code for `boolify_scheduling_problem` to implement that recipe.

8.1) Grade

You have not yet received this checkoff. When you have completed this checkoff, you will see a grade here.

Footnotes

[1] Generators are functions that return an object over which we can iterate. Rather than constructing all of the elements in the iterable at once, a generator instead constructs just one element at a time. This is often faster and more space efficient. For example, if we are looking for an element that meets some criteria, we don't have to create every single element--we can stop immediately after having found the first such element. And since we are only making one element at a time, we avoid having to store all of the elements at once, which could cause a MemoryError.

There are two keywords used in generators. The first is `yield`, which on iteration will return a value and pause at that point. When we need the next value on the next iteration the generator will continue ("unpause") from that point. The second is `yield from` which, when called on an iterable (e.g. list, set, another generator) will continually yield each of the elements from it, one at a time.

We will see more on generators later in this course! This is just a preview, and you need not use generators in this lab.

[2] While this particular scheduling problem can be solved using other algorithms such as maxflow, the SAT method is more general and will still work when we have constraints such as that two or more sessions should have equal numbers of students, whereas maxflow would not be directly applicable then. This motivates our reduction to SAT.