SAT Solver

Table of Contents

1) Preparation

This lab assumes you have Python 3.9 or later installed on your machine (3.11+ recommended).

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

Your raw score for this lab will be counted out of 5 points. Your score for the lab is based on:

- a style check (1 points), and
- passing the tests in `test.py` (4 points).

> **Reminder: Academic Integrity**
>
> Please also review the academic integrity policies before continuing. In particular, **note that you are not allowed to use any code other than that which you have written yourself, including code from online sources.**

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 6.101 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 Duane, Jacob, Saman, John, 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. Jacob only eats pickles-flavored cupcakes.
> 5. Years ago, Saman and John made a pact that, whenever either of them eats cupcakes, they must share with the other one.
> 6. Duane 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 = (jacob or duane or saman or john 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 jacob or not duane)
     and (not jacob or not saman)
     and (not jacob or not john)
     and (not jacob or not tim)
     and (not duane or not saman)
     and (not duane or not john)
     and (not duane or not tim)
     and (not saman or not john)
     and (not saman or not tim)
     and (not john 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: we can't have all flavors present; and, additionally, among
# any pair of flavors, at least one was present.


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


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


rule6 = ((not duane or chocolate)
     and (not duane or vanilla)
     and (not duane or pickles))
# If Duane is the culprit and we left out a flavor, the corresponding case here
# will fail to hold.  So this rule encodes the restriction that Duane 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 Jacob was guilty and that only pickles-flavored cupcakes were present):

```
jacob = True
duane = False
saman = False
john = 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.

If we instead try the following bindings, what is the result of `satisfied`?
You can test this by running the code on your own machine!

```
jacob = False
duane = False
saman = False
john = False
tim = True

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

Submit

You have submitted this assignment 0 times.

2.1) Solving the Puzzle

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.

2.2) 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 multiway `or` of literals, and a CNF *formula* is a multiway `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. However, forcing our constraints to be in this form can simplify the problem of implementing our solver, compared to other representations we could choose. We'll try in this writeup to help you with the pieces of the lab involving converting expressions to CNF.

2.2.1) Python Representation

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

- a *variable* as a Python string
- a *literal* as a pair (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.

```
rule1 = [[('jacob', True), ('duane', True), ('saman', True),
          ('john', True), ('tim', True)]]

rule2 = [[('jacob', False), ('duane', False)],
         [('jacob', False), ('saman', False)],
         [('jacob', False), ('john', False)],
         [('jacob', False), ('tim', False)],
         [('duane', False), ('saman', False)],
         [('duane', False), ('john', False)],
         [('duane', False), ('tim', False)],
         [('saman', False), ('john', False)],
         [('saman', False), ('tim', False)],
         [('john', False), ('tim', False)]]


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

rule4 = [[('jacob', False), ('pickles', True)],
         [('jacob', False), ('chocolate', False)],
         [('jacob', False), ('vanilla', False)]]

rule5 = [[('saman', False), ('john', True)],
         [('john', False), ('saman', True)]]

rule6 = [[('duane', False), ('chocolate', True)],
         [('duane', False), ('vanilla', True)],
         [('duane', 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.

2.2.2) 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.

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 nontrivial 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.

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))
{'john': False, 'saman': False, 'chocolate': False, 'duane': False,
 'jacob': 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 straightforward, brute-force way to solve Boolean puzzles: enumerate all possible combination of Boolean assignments for 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 that generates 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.

```
def all_bools(length):
    if length == 0:
        return [[]]
    else:
        out = []
        for v in all_bools(length-1):
            out.append([True] + v)
            out.append([False] + v)
        return out
```

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. This is because 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.

Lots of logic puzzles can 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 this brute-force enumeration of all assignments, thanks to the power 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)
```

In the context of the `satisfying_assignment` function in the lab, this formula would be formatted as:

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

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

`(a or b)` or `[[('a', True), ('b', True)]]` in the lab context.

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 when `c` is `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: your `satisfying_assignment` function does not need to return assignments for these non-essential variables.

If we had instead tried setting `c` to `False`, we would update the formula instead as follows:

`d` or `[[('d', True)]]` in the lab context.

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`.

If we then took the formula containing only `[[('d', True)]]` and set `d` to be `True`, we could update the formula to be a list of how many clauses?

Submit

You have submitted this assignment 0 times.

What does the formula that results from setting `c` to `False` and `d` to `True` imply?

- ○ This formula is satisfied regardless of how the other variables are set.
- ○ This formula cannot be satisfied regardless of how the other variables are set.
- ○ Neither of the above.

Submit

You have submitted this assignment 0 times.

If we instead took the original formula and set both `c` and `d` to be `False`, the formula could be updated to a new formula containing only `[[]]`. What does this imply?

- ○ This formula is satisfied regardless of how the other variables are set.
- ○ This formula cannot be satisfied regardless of how the other variables are set.
- ○ Neither of the above.

Submit

You have submitted this assignment 0 times.

It might be a good idea to implement this process (updating a formula based on a new assignment) as a helper function (which you can test independently), as we will need to perform this operation repeatedly.

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), ('f', True)],
]
```

Starting from the formula above, what formula results when we set `a` to be `True`?

Which of the following is true for this formula?

--

Submit

You have submitted this assignment 0 times.

What formula results if we instead set `a` to be `False`?

Which of the following is true for this formula?

--

Submit

You have submitted this assignment 0 times.

What formula results if we instead set `a` to `True` *and* `f` to `True`?

Which of the following is true for this formula?

--

Submit

You have submitted this assignment 0 times.

What formula results if we instead set `a` to `True` *and* `f` to `False`?

Which of the following is true for this formula?

--

Submit

You have submitted this assignment 0 times.

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`.

5) Optimizations

Suggested Approach

While the details below can make a big difference in terms of efficiency for some of the test cases, each new change below adds complexity to your code. We **strongly** encourage you to write and debug the approach described above first, and only to add the features below once you have something that works.

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 **before making any recursive calls**. Propagating the effects of one unit clause may reveal further unit clauses, whose later propagations may themselves reveal more unit clauses, and so on.

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).

6) Sudoku 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 problem we already saw in the readings: Sudoku puzzles. But we'll actually generalize this idea a bit, expanding the idea to work with boards of different sizes.

6.1) Sudoku Description

Recall from the readings that a solution to a standard Sudoku puzzle represents an arrangment of the digits 1-9 on a grid such that:

- every row contains each digit exactly once
- every column contains each digit exactly once
- each of the 3×3 blocks that make up the grid contains each digit exactly once

Here is an example puzzle:

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

As in the readings, we will represent puzzles like this as 2-d arrays (lists-of-lists) of numbers, where a `0` means that a given cell is empty. In this format, the puzzle above looks like the following:

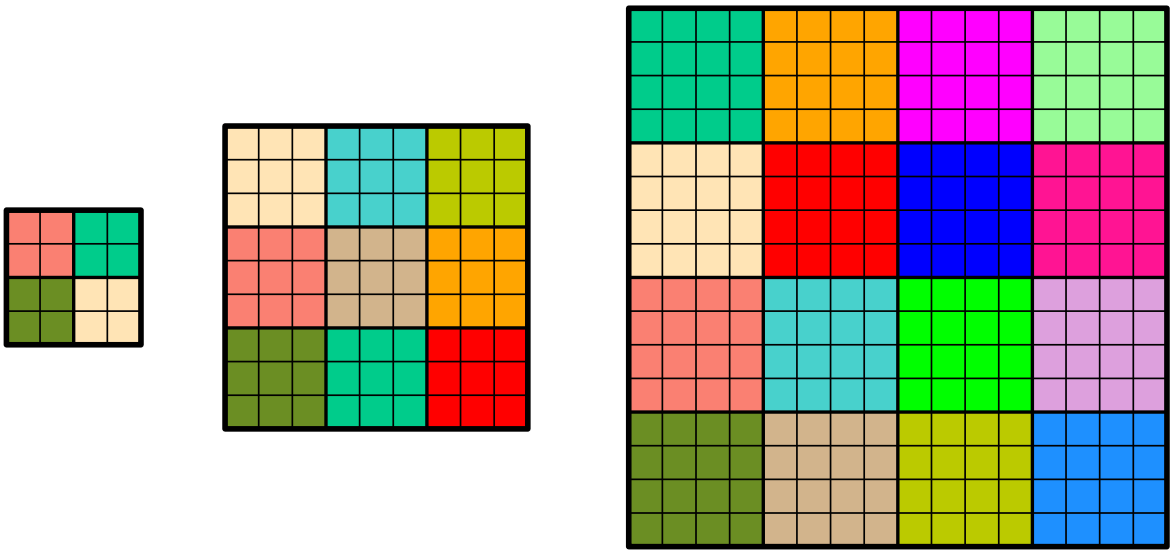
```
[
    [5, 3, 0, 0, 7, 0, 0, 0, 0],
```

```
        [6, 0, 0, 1, 9, 5, 0, 0, 0],
        [0, 9, 8, 0, 0, 0, 0, 6, 0],
        [8, 0, 0, 0, 6, 0, 0, 0, 3],
        [4, 0, 0, 8, 0, 3, 0, 0, 1],
        [7, 0, 0, 0, 2, 0, 0, 0, 6],
        [0, 6, 0, 0, 0, 0, 2, 8, 0],
        [0, 0, 0, 4, 1, 9, 0, 0, 5],
        [0, 0, 0, 0, 8, 0, 0, 7, 9],
    ]
```

We'll expand on this idea a little bit by allowing for boards of size $n \times n$, where n is a perfect square (so 4×4, 9×9, 16×16, 144×144, etc, are all valid board sizes). With this change in mind, we still have similar constraints on the board:

- Each cell must contain one of the numbers between 1 and n, inclusive.
- Each row must contain all of the numbers from 1 to n exactly once.
- Each column must contain all of the numbers from 1 to n exactly once.
- Each subgrid must contain all of the numbers from 1 to n exactly once.

One key difference from the standard 9×9 board is the number of the subgrids, as well as their size and placement. Each subgrid will be $\sqrt{n} \times \sqrt{n}$, and there will be a total of n subgrids. The top-left-most subgrid will always have its upper-left-most corner at row 0, column 0; and subgrids never overlap. The following images show the arrangement of the subgrids in a 4×4, 9×9, and 16×16 game (with each subgrid colored in a distinct color); and the subgrids are arranged in similar patterns on boards of other sizes.



6.2) Reduction

Rather than writing a solver for Sudoku directly like we saw in the readings, in this assignment, we'll take a different approach involving three steps:

- generate a SAT formula based on a given Sudoku board,
- use our SAT solver from above to find a solution to the formula, and
- reinterpret the output from the solver in terms of the original puzzle.

We've already implemented everything we need for the middle one of those three steps, so what remains is to implement the first and third steps. To complete this task, fill in the following functions in your `lab.py` file:

- `sudoku_board_to_sat_formula` should take a board (as above) as input, and it should return a SAT formula (in CNF) for which a satisfying assignment represents a solution to the given puzzle.
- `assignments_to_sudoku_board` should take as input a result from `satisfying_assignment` representing a solution to a Sudoku puzzle and an integer representing the size of the board, and it should return a 2-d array of the form above but with all values filled in. If the given result from `satisfying_assignment` does not represent a valid solution, it should return `None` instead.

Note that we are *not* testing the results of these functions independently, so you are free to set up the associated SAT formula however you prefer. However, this is a tricky task, so here are a few guidelines and hints:

- One way to encode things is to have one variable for the possibility of each value being in each square. So, for example, one variable would represent a "1" being in location $(0, 4)$; and that variable being `True` means that there is, in fact, a "1" at that location. You are free to represent the variable names in any way that is consistent with your SAT solver (i.e., variable names do not necessarily need to be strings).
- We need our representation to be consistent with the starting board. For example, in the original 9×9 example board above, the bottom-right location $(8, 8)$ contains a digit "9"; so our formula should ensure that any satisfying assignment has that property.
- Each cell must contain exactly one digit. This means that a given cell must have one of the digits in it; but it also means that we cannot have multiple digits in the same cell (i.e., for each possible pair of digits, one of them must *not* be in the cell). Your formula will need to include both of these constraints for each cell.
- Within each row, each value must occur exactly once. This means that a given value must occur at least once in the row, but it also means that each value must occur at most once in a row (i.e., for each possible pair of locations in the row, one of the two must *not* have the given value). We then need similar rules for each column and each $\sqrt{n} \times \sqrt{n}$ subgrid.

  While implementing all of the details of our suggested rules may technically not be necessary, it is still a good idea to include all the rules explicitly, because they (coupled with the unit clause optimization above) allow our SAT solver to notice the need to backtrack earlier than we would otherwise notice it.

As you're working, we strongly recommend testing and debugging using small grids (4×4 is a good place to start) since these are easier to look at, and it may even be possible to print out all of the rules for a given board and verify their correctness by inspection, which is *substantially* more difficult for 9×9 or larger boards.

As always, keep an eye out for opportunities to avoid repetitious code (**what helper functions can you write for yourself that would be useful here?**), and, as always, if you need some help, feel free to reach out!

We have also provided a GUI for solving Sudoku puzzles using your code, which you can use by running `server.py` and then navigating to `http://localhost:6101`. You can use this GUI for testing and debugging, or to see your solver in action once your lab is complete! Each of the puzzles from the test cases can be loaded from the relevant file in the `puzzles` directory in the code distribution; and you can load other puzzles into the GUI by making files of that same format for yourself as well!

7) Code Submission

When you have tested your code sufficiently on your own machine, submit your modified `lab.py` using the `6.101-submit` script.

The following command should submit the lab, assuming that the last argument `/path/to/lab.py` is replaced by the location of your `lab.py` file:

```
$ 6.101-submit -a sat /path/to/lab.py
```

Running that script should submit your file to be checked. After submitting your file, information about the checking process can be found below:

When this page was loaded, you had not yet made any submissions.
If you make a submission, results should show up here automatically; or you may click here or reload the page to see updated results.

Submit

You have submitted this assignment 0 times.