

Computational Decision Making for Regular People *Debugging*

November 11, 2024

Nathan Davis Barrett



Today's Outline



1. Types of Errors
2. Python Traceback
3. Math Modeling (Errors)
4. How to handle "Unbounded" Results
5. How to handle "Infeasible" Results



This image was created with the assistance of DALL-E 3

Caveat: This might not seem like a lot. But the skills we'll cover today are among the VERY MOST IMPORTANT skills you can develop as an independent coder.

Types of Errors



	Syntax Errors	Semantic Errors	Mathematical Errors
Definition	The computer doesn't know how to interpret your code	The computer can interpret your code, but it doesn't do what you want	Your combinations of variables, constraints, and/or objective can't produce a solution
What it will look like	Python will show an error message and immediately stop	Your code will run just fine, but your answers might be weird	When you go to solve your model, you will get an "infeasible" or "unbounded" result
Example	<pre>ModuleNotFoundError: Traceback (most recent call last): <ipython-input-1-b4c97a746086> in <cell line: 1()> ----> 1 import pyomo.environ as pyo 2 3 model = pyo.ConcreteModel() 4 5 model.x = pyo.Var() ModuleNotFoundError: No module named 'pyomo'</pre>	<pre>1 def Add(x,y): 2 return x - y 3 4 print(Add(5,2)) 3</pre>	<pre>SCIP Status : problem is solved [unbounded] Solving Time (sec) : 0.00 Solving Nodes : 0 Primal Bound : -1.0000000000000000e+20 (3 solutions) Dual Bound : -1.0000000000000000e+20 Gap : 0.00 % WARNING: Loading a SolverResults object with a warning status into model.name="unknown"; - termination condition: unbounded - message from solver: unbounded</pre>

Handling Syntax Errors



- ▶ Remember, a syntax error is when Python does not know how to interpret your code.
- ▶ Since it doesn't know what to do, it just stops right then and there.
- ▶ Luckily, Python gives us some useful hints as to where the problem happened and why.
- ▶ Hint #1: The **error type** tells us a little about **what kind of thing happened**.
 - ▶ A **TypeError** indicates that something went wrong with the type of two variables.
 - ▶ A **ZeroDivisionError** indicates that we divided by zero somewhere.
 - ▶ A **ModuleNotFoundError** indicates that Python couldn't find an external package that we're trying to use.
- ▶ Hint #2: The **error message** tells us a little about **what exactly happened**.
 - ▶ **ModuleNotFoundError**: No module named 'pyomo'
 - ▶ Python can't find the 'pyomo' package. Is it installed correctly?
 - ▶ **TypeError**: unsupported operand type(s) for +: 'int' and 'str'
 - ▶ You can't add an integer number to a string (e.g. `5 + "hello"`). Apparently we're doing that somewhere.
- ▶ Hint #3: The **traceback** tells us exactly **where the problem happened**.

Handling Syntax Errors



Error Type

```
[2] 1 print(5 + "X")  
  
-----  
TypeError                                Traceback (most recent call last)  
  <ipython-input-2-cb8ba3496cdd> in <cell line: 1>()  
    ----> 1 print(5 + "X")  
  
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Location

(e.g. file, cell, or line number)

Error Message

The exact line causing
the problem is shown.

Handling Syntax Errors



What about errors that occur within a function?

```
1 def AssembleModel(xMax):
2     model = pyo.ConcreteModel()
3     model.x = pyo.var(bounds=(0,xMax))
4     return model
5
6 def Determine_xMax():
7     return 10
8
9 def MyProblem():
10    xMax = Determine_xMax()
11    model = AssembleModel(xMax)
12    return model
13
14
15 MyProblem()
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-8-8617821daae6> in <cell line: 15>()
    13
    14
--> 15 MyProblem()

----- 2 frames -----
<ipython-input-8-8617821daae6> in MyProblem()
     9 def MyProblem():
    10     xMax = Determine_xMax()
--> 11     model = AssembleModel(xMax)
    12     return model
    13

<ipython-input-8-8617821daae6> in AssembleModel(xMax)
     1 def AssembleModel(xMax):
     2     model = pyo.ConcreteModel()
--> 3     model.x = pyo.var(bounds=(0,xMax))
     4     return model
     5

/usr/local/lib/python3.10/dist-packages/pyomo/common/deprecation.py in __getattr__(name)
    437         elif _mod_getattr is not None:
    438             return _mod_getattr(name)
--> 439         raise AttributeError(
    440             "module '%s' has no attribute '%s'" % (f_globals['__name__'], name)
    441         )

AttributeError: module 'pyomo.environ' has no attribute 'var'
```

Syntax Error Practice



```
1 def FindAll(myStr,searchCharacter):
2     """
3     A function that returns a list of all the indices of a given searchCharacter in a given string.
4     """
5     indices = []
6     for i in range(len(myStr)):
7         if myStr[i] == searchCharacter:
8             indices.add(i)
9     return indices
10
11 def Test():
12     return FindAll("Hello World!","l")
13
14 Test()
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-9-143b3a391355> in <cell line: 14>()
      12     return FindAll("Hello World!","l")
      13
--> 14 Test()
```

```
----- 1 frames -----
<ipython-input-9-143b3a391355> in Test()
      10
      11 def Test():
--> 12     return FindAll("Hello World!","l")
      13
      14 Test()

<ipython-input-9-143b3a391355> in FindAll(myStr, searchCharacter)
       6     for i in range(len(myStr)):
       7         if myStr[i] == searchCharacter:
----> 8             indices.add(i)
       9     return indices
      10
```

```
AttributeError: 'list' object has no attribute 'add'
```

Syntax Error Practice



```
1 model = pyo.ConcreteModel()
2 model.x = pyo.Var(bounds=(0,10))
3 model.y = pyo.Var(bounds=(0,10))
4
5 def ConstrFunc1(model):
6     return model.x < 4
7 model.constr1 = pyo.Constraint(rule=ConstrFunc1)
8
9 def ConstrFunc2(model):
10    return model.x > 5
11 model.constr2 = pyo.Constraint(rule=ConstrFunc2)
12
13 def ConstrFunc3(model):
14    return model.x == 1/2 * model.y
15 model.constr3 = pyo.Constraint(rule=ConstrFunc3)
```

```
ValueError                                Traceback (most recent call last)
~/python-learn-12-526b79ba187d in <cell line: 7>():
      5 def ConstrFunc1(model):
      6     return model.x < 4
----> 7 model.constr1 = pyo.Constraint(rule=ConstrFunc1)
      8
      9 def ConstrFunc2(model):

      * 6 frames *
~/usr/local/lib/python3.10/dist-packages/pyomo/core/base/block.py in _setattr__(self, name, val)
    569     # Pyomo components are added with the add_component method.
    570
--> 571     self.add_component(name, val)
    572     else:
    573         #

~/usr/local/lib/python3.10/dist-packages/pyomo/core/base/block.py in add_component(self, name, val)
    1103
    1104     try:
--> 1105         val.construct(data)
    1106     except:
    1107         err = sys.exc_info()[1]

~/usr/local/lib/python3.10/dist-packages/pyomo/core/base/disable_methods.py in construct(self, data)
    122     self._name = base.__name__
    123     self.__class__ = base
--> 124     return base.construct(self, data)
    125
    126     construct.__doc__ = base.construct.__doc__

~/usr/local/lib/python3.10/dist-packages/pyomo/core/base/constraint.py in construct(self, data)
    668     # Bypass the index validation and create the member directly
    669     for index in self.index_set():
--> 670         self._setitem_when_not_present(index, rule(block, index))
    671
    672     except Exception:
    673         err = sys.exc_info()[1]

~/usr/local/lib/python3.10/dist-packages/pyomo/core/base/indexed_component.py in _setitem_when_not_present(self, index, value)
    1105     try:
    1106         if value is not _NotSpecified:
--> 1107             obj.set_value(value)
    1108     except:
    1109         self._data.pop(index, None)

~/usr/local/lib/python3.10/dist-packages/pyomo/core/base/constraint.py in set_value(self, expr)
    829     if not self._data:
    830         self._data[None] = self
--> 831     return super(ScalarConstraint, self).set_value(expr)
    832
    833     #

~/usr/local/lib/python3.10/dist-packages/pyomo/core/base/constraint.py in set_value(self, expr)
    375     if expr.__class__ in _known_relational_expressions:
    376         if getattr(expr, 'strict', False) in _strict_relational_exprs:
--> 377             raise ValueError(
    378                 "Constraint '%s' encountered a strict "
    379                 "inequality expression ('>' or '<'). All "
```

ValueError: Constraint 'constr1' encountered a strict inequality expression ('>' or '<'). All constraints must be formulated using using '<=', '>=', or '=='.

Handling Semantic Errors



- ▶ Remember, semantic errors are when the computer is able to interpret your code, but it doesn't do what you want.
- ▶ These kinds of errors are much harder to solve since Python can't give any hints as to where or why the problem occurred.
- ▶ We need to be a bit more clever with how we find the cause of the problem.
- ▶ The makers of Python gave us some very powerful tools to help: **The Python Debugger**
 - ▶ *Caveat: The Python Debugger looks and works different depending on which environment you're using (e.g. Google Colab, Spyder, VS Code, etc.)*
 - ▶ To start, we'll go over the basic idea. Then we'll dive into how to use it in each of a few different environments.

The Idea Behind The Python Debugger



- ▶ The debugger is a magic button that **freezes my code at a very specific spot** allowing me to look at the value of any of my variables in an exact moment.
- ▶ Additionally, I can step **line-by-line** through my code to see how these variables change as the code progresses.
- ▶ I can press this "magic button" by placing what is called a **breakpoint** in my code.
- ▶ With that in mind, there are a few different options I can use to navigate my code:
 - ▶ **Step Into:** If my code is frozen on a line that calls a function, I can "step into" that function to see what is going on in that function.
 - ▶ **Step Over:** If my code is frozen on a line that calls a function, I can "step over" that function to skip over the execution of that function and keep going line-by-line through the code I'm viewing right now.
 - ▶ **Step Out:** If my code is frozen within a function and I want to jump to return to the code that called that function, I can "step out" of that function.
 - ▶ **Fast Forward:** If I learned what I want from the point in my code that I'm at right now, I can fast forward to the next break point. If there are none, the code will just fast-forward to the end (or until it hits an error).

At this point, please skip to the section that deals with the coding environment you're using.

Google Colab: Debugger Tool

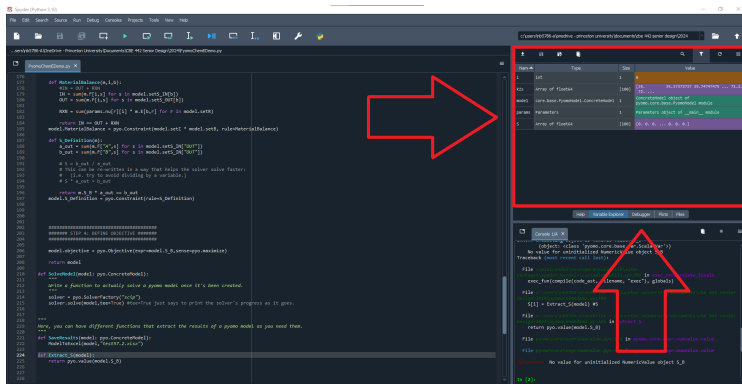


- ▶ Google Colab is great for many reasons. But unfortunately, debugging is not one of them.
- ▶ There is a debugger tool, but it's very clunky.
- ▶ To trigger a break point:
 - ▶ Type the command `"breakpoint()"` on the line of your code you'd like to freeze at. Then run your code as usual.
 - ▶ Once the code reaches the `breakpoint()` command, it will freeze and give you a little text box.
 - ▶ In this text box you can tell the debugger tool what you want to do next.
 - ▶ **Step Into:** Type `"s"` followed by pressing the Enter key on your keyboard.
 - ▶ **Step Over:** Type `"n"` followed by pressing the Enter key on your keyboard.
 - ▶ **Step Out:** Type `"r"` followed by pressing the Enter key on your keyboard.
 - ▶ **Fast Forward:** Type `"c"` followed by pressing the Enter key on your keyboard.
 - ▶ **Show the value of a variable at this moment:** Type the name of that variable followed by pressing the Enter key on your keyboard.

Spyder: Debugger Tool







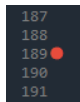
- ▶ Spyder naturally allows you to view the most recent state of any variable defined in the current scope using the **variable explorer**.
 - ▶ Scope is defined as any code that has already been executed in the same function, loop iteration, etc. as the current position of the code.
- ▶ If we can put a breakpoint on any line of our code, Python will freeze there allowing us to view the current state of any variable in that function.



Spyder: Debugger Tool



- ▶ Similar to Spyder's normal "Run File" () and "Run Current Cell" () buttons, Spyder has special "Debug File" () and "Debug Current Cell" () buttons.
- ▶ These buttons put you in "Debug Mode" where Spyder will detect breakpoints.
 - ▶ *NOTE: Spyder will not consider breakpoints if it is not running in Debug Mode. Therefore you must press "Debug File" or "Debug Current Cell" for the breakpoints to be considered.*
- ▶ To place a breakpoint at a certain line of code, simply click just to the right of the line number for that line. A small red dot should appear indicating that a breakpoint has been placed.



- ▶ To remove a breakpoint, simply click in it's red dot and it should disappear.
- ▶ Once a breakpoint is placed, press "Debug File" or "Debug Current Cell". Once the code execution reaches the line on which you've placed a breakpoint, it will freeze.

Spyder: Debugger Tool




- ▶ Once a breakpoint has been placed and the code execution has reached it in Debug Mode, it will freeze.
- ▶ Each of the variables defined in this scope should be visible in the variable explorer.
 - ▶ *NOTE: You may need to un-select the "Filter Variables" option (🔍) in order to make ALL the variables appear. This filter only shows variables that are somewhat easy to visualize in the window.*
- ▶ In order to step through your code, you'll need to switch to the "Debugger" window next to the "Variable Explorer" window.
- ▶ At the top of this window, there will be a collection of buttons.
 - ▶ **Step Over:** ⏮
 - ▶ **Fast Forward:** ⏭
 - ▶ **Step Into:** ⏴
 - ▶ **Step Out:** ⏴
- ▶ As you step through your code, you can switch back to the Variable Explorer window to view the state of any variables.
- ▶ Additionally, as you step through, Spyder should display the current line using an arrow next to the line number.

VS Code: Debugger Tool



- ▶ To place a breakpoint, simply click to the left of the line number for the line you'd like to place a breakpoint. A small red dot should appear.
- ▶ To remove a breakpoint, simply click on the red dot and it should disappear.

```
1  import pyomo.environ as pyo
2
3  model = pyo.ConcreteModel()
4
```

- ▶ Once a breakpoint is placed, the code must be run in "Debug Mode" for VS Code to recognize it.
- ▶ Normally, we'd press the "Play" button (▶) to execute our code.
- ▶ Instead, we need to go to the "Run and Debug" menu on the left-most panel: ▶
- ▶ There are lots of different configurations you could choose when debugging in VS Code.
- ▶ To use the default configuration, press "Run and Debug": 
- ▶ If this is your first time running the debugger on this python file, you will need to click "Python Debugger" followed by "Python File" in the menus that pop up.

VS Code: Debugger Tool



The screenshot shows the VS Code interface with a Python file named `Example.py` open. The file contains the following code:

```
1 import pyomo.environ as pyo
2
3 model = pyo.ConcreteModel()
4
5 model.x = pyo.Var()
6
7 model.obj = pyo.Objective(expr=model.x)
8
9 solver = pyo.SolverFactory("scip")
10 solver.solve(model, tee=True)
```

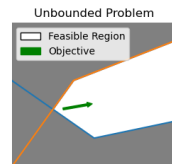
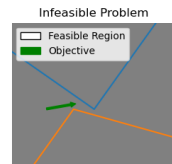
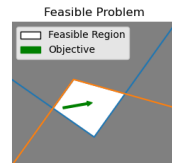
Annotations and labels in the image:

- VARIABLES**: Points to the **VARIABLES** panel on the left, which shows the current state of variables in the `model` namespace.
- BREAKPOINT**: Points to the **DEBUG** icon (a small 'D' in a circle) in the left margin next to line 5.
- CURRENT LINE**: Points to the line number `5` in the left margin, indicating the current execution position.
- TRACEBACK**: Points to the **CALL STACK** panel at the bottom left, which shows the sequence of function calls leading to the current line.
- FAST FORWARD**: Points to the **Run** button (a play icon) in the top right toolbar.
- STEP OVER**: Points to the **Step Over** button (a right arrow with a dot) in the bottom right toolbar.
- STEP INTO**: Points to the **Step Into** button (a right arrow with a dot and a downward arrow) in the bottom right toolbar.
- STEP OUT**: Points to the **Step Out** button (a right arrow with a dot and an upward arrow) in the bottom right toolbar.
- QUIT**: Points to the **Quit** button (a red square) in the bottom right toolbar.

Handling Math Errors



- ▶ First, let's understand what these errors mean.
 - ▶ Infeasible: There are no solutions that can satisfy all of the constraints you have in your model
 - ▶ Unbounded: The solver says you can keep traveling forever in the direction indicated by your objective. Thus your optimal objective function value will be infinity.
- ▶ Unfortunately, **most solvers will not give you any information** about what is causing the problem, nor will they give you any solution if one of these problems is encountered.
- ▶ **NOTE:** Part of why these are hard to solve is that sometimes our model itself is unbounded or infeasible even if we've coded everything correctly.
- ▶ The main goal of these tips and tricks is to **temporarily modify your model to make it feasible and bounded** so that you can get a solution. Then, depending on the solution and/or modifications made, we can **alter the original model to prevent this** infeasibility or unboundedness from happening.

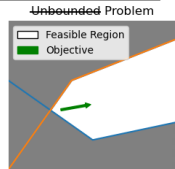


Handling Unbounded Problems



- ▶ Recall that a problem will be unbounded if there are no bounds on the values that certain variables can take on.
- ▶ Thus, **imposing an artificial bound on key variables** can cause an unbounded problem to become bounded.
- ▶ When defining a variable, including a bound on each variable that is **so big that it would be noticeably too big** (but technically not infinity) could solve this problem.
 - ▶ Example: If I'm maximizing revenue, if I say that revenue must be between $-\$1,000,000,000$ and $\$1,000,000,000$ then the problem would no longer be unbounded.
- ▶ One word of caution: Computers have a hard time with REALLY REALLY large numbers and it might crash if you enter a number absurdly big.
- ▶ I suggest thinking of the biggest reasonable quantity that that variable should have, and multiplying by 10.

1. With this new bound, if you re-solve the solver should at least give you a solution.
2. Inspect this solution to see which variables are unreasonably large and why.
3. Add a new constraint and/or bound as needed.



Handling Infeasible Problems



- ▶ Recall that a problem will be infeasible if two or more constraints make it so that no possible solution can satisfy all of them.
- ▶ There are 3 approaches (that I'm familiar with) to address this:
 1. Sequentially remove constraints until the problem is no longer infeasible
 2. If you know a feasible solution, enter it and observe which constraints it violates.
 3. Slightly modify the problem to quantify **just how infeasible** a solution is. Minimize this infeasibility.
- ▶ Next we'll go over each of these approaches along with their pros and cons.

Infeasible: Sequentially Remove Constr.



1. Remove a constraint.
2. Re-optimize.
3. If still infeasible, remove another.
4. If not infeasible, you know that (at least one of) the constraint(s) is causing the issue.
 - ▶ Go look at those constraints. Are they coded right? Do they make sense?
5. Repeat (potentially trying different combinations of constraints to see which combination causes the issue).

EXAMPLE:

C1	C2	C3	Feasible?
✓	✓	✓	✗
✗	✓	✓	✗
✗	✗	✓	✓
✓	✗	✓	✓
✓	✓	✗	✓

The Gist: **Easiest but the least reliable approach.**

- ▶ PROS:
 - ▶ Very easy to do (just comment-out constraints).
 - ▶ You don't need any additional information or re-formulation.
- ▶ CONS:
 - ▶ You technically need to try every possible combination in order to prove where the problem is happening.
 - ▶ Often when you remove a constraint, an infeasible solution can turn into an unbounded solution.

Infeasible: Known Solution



The Gist: **Best approach if you have a known solution (or one is easy to generate).**

1. Insert a known feasible solution into the model.
2. Iterate over all the constraints to determine which ones are violated by this solution.
3. Those constraints must be coded and/or ideologically incorrect.

► PROS:

- Guaranteed to work.

► CONS:

- Requires you to know of and be able to insert a known solution.

- Inserting a known solution, iterating over constraints, and testing feasibility are somewhat nontrivial in Pyomo.
- I've made some tools to automate this process:

```
from PyomoTools.IO import ModelToExcel, LoadModelSolutionFromExcel
from PyomoTools import InfeasibilityReport

#STEP 1: Create an excel sheet with a spot for each of the variables in your model.
ModelToExcel(model,"myKnownSolution.xlsx")

#STEP 2: Insert your known solution into the excel sheet
# Do this in excel

#STEP 3: Load your solution back into the Pyomo model
LoadModelSolutionFromExcel(model,"myKnownSolution.xlsx")

#STEP 4: Identify and print out any violated constraints
report = InfeasibilityReport(model)
report.WriteFile("infeasibilityReport.txt")
```

Infeasible: Minimize Infeasibility



1. Re-formulate the problem using non-negative auxiliary variables to eliminate the possibility of an infeasible solution.
2. Define a new objective to minimize these aux. vars.
3. Optimize the re-formulated model.
4. Any constraints with non-zero aux. vars. are incompatible with other constraints that have non-zero aux. vars.

-
- ▶ Equality Constraint: Add an aux. var. to both sides.
 - ▶ \leq Constraint: Add an aux. var. to the right-hand-side.
 - ▶ \geq Constraint: Add an aux. var. to the left-hand-side.

The Gist: **Think of this as the automated version of approach 1.**

▶ PROS:

- ▶ Is a bit more rigorous/thorough than approach 1.
- ▶ Doesn't require a known solution.

▶ CONS:

- ▶ Requires the computer to solve a separate (potentially very difficult) optimization problem.
- ▶ Can require a lot of work to reformulate. I've automated the process for you:

```
from PyomoTools import FindLeastInfeasibleSolution, InfeasibilityReport
#STEP 1: Find the least infeasible solution.
solver = pyo.SolverFactory("scip")
FindLeastInfeasibleSolution(model, solver)

#STEP 2: Identify and print out any violated constraints
report = InfeasibilityReport(model)
report.WriteFile("infeasibilityReport.txt")
```

Conclusion



- ▶ Debugging can be a long, tedious, frustrating, and unavoidable process.
 - ▶ Even professional software engineers can spend up to 50% of their time debugging.
- ▶ I'd suggest testing your code **frequently AS YOU WRITE IT**.
 - ▶ Don't wait till you're done with the whole problem to try to run your code.
 - ▶ The closer you can narrow down the source of a problem the faster.
- ▶ You won't always have a teacher or classmates to depend on.
 - ▶ One resource you will always have: **The Internet!**
 - ▶ Asking Google (or ChatGPT) about your problem is often very effective.
 - ▶ Easy, free, and effective place to find ChatGPT: <https://copilot.microsoft.com>
- ▶ If you can get good at this, you'll do well as an independent programmer.