The mixed-integer program coding cookbook (using Python and Pyomo)

Andreas Wiese

This document contains a "recipe" for coding a mixed-integer program (MIP) in Python via the modeling framework Pyomo. We first recap what a MIP is (Section 1) and then discuss different MIP solvers (Section 2) and modeling frameworks (Section 3). After that, in Section 4 we see how to code a simple MIP with a concrete set of variables and constraints using Python and Pyomo. Subsequently, in Section 5 we see an example for coding an abstract MIP formulation whose variables and constraints depend on input data which is loaded separately (e.g., from an external source).

The examples in this document are designed such that they contain most elements that appear in arbitrary MIP formulations. Therefore, you may use them as templates when you code your own MIP models.

1 Mixed-integer programs

In this section, we discuss what a mixed-integer program (MIP) is. Feel free to skip this section if you already know this. We start with a concrete simple example and then define MIPs more generally.

1.1 Example for a simple MIP

Imagine that you want to invite some of your friends to a dinner party and you want to cook pizzas and lasagna for them. Of course, you need ingredients. Let's assume you have enough of all ingredients, apart from tomatoes and cheese: you have only 18 tomatoes and 24 pieces of cheese at home. If you prepare one pizza, you need two tomatoes and four pieces of cheese; if you prepare one serving of lasagna, you need three tomatoes and three pieces of cheese.

	one pizza	one serving of lasagna	available
tomatoes	2	3	18
cheese	4	3	24

We assume that each of your invited friends will eat one pizza or one serving of lasagna and that your friends are indifferent between the two dishes. So now you ask yourself: *How many pizzas and how many servings of lasagna shall I prepare so that I can invite as many friends as possible?*

You could answer this question by simply trying all combinations for the number of pizzas and lasagnas; however, we want to model it mathematically. We introduce a variable x_1 for the number of pizzas that you prepare and a variable x_2 for the number of servings of lasagna. Our objective is to prepare as many pizzas and servings of lasagna as possible. Hence, mathematically, we want to maximize $x_1 + x_2$. Therefore, we write

$$\max\ x_1+x_2$$

Also, we need to make sure that we do not try to prepare more pizzas and lasagnas than our ingredients allow us. Therefore, we write the following two inequalities (the first one for the tomatoes and the second

one for the cheese).

$$2x_1+3x_2 \leq 18$$
$$4x_1+3x_2 \leq 24$$

Finally, we want to express that the number of prepared pizzas and lasagnas is not negative and that both are integers (e.g., we do not allow to prepare 1.5 pizzas or 4/3 lasagnas). Therefore, we write

$$x_1,x_2 \geq 0$$
$$x_1,x_2 \in \mathbb{Z}$$

Putting this all together, this yields the following mathematical program.

$$\begin{aligned} \max \quad & x_1 \quad +x_2 \\ \text{s.t.} \quad & 2x_1 +3x_2 \leq 18 \\ & 4x_1 +3x_2 \leq 24 \\ & x_1, \quad x_2 \geq 0 \\ & x_1, \quad x_2 \in \mathbb{Z} \end{aligned}$$

When you solve it, it turns out that an optimal solution is $x_1 = 3$ and $x_2 = 4$, i.e., this solution satisfies all constraints above, and there is no other such solution for which $x_1 + x_2$ is larger.

1.2 General definition of MIPs

In general, a MIP consists of the following:

- **Variables.** In our example, the variables were x_1 and x_2. In general, there can be arbitrarily (finitely) many and they can have arbitrary names. For some variables, we might have the additional requirement that they are only allowed to take integral values (like x_1 and x_2 in our example above). We say that these are *integer variables* and write $x \in \mathbb{Z}$ in our formulation to indicate that a variable x is an integer variable. In a MIP it can happen that some but not variables are integer variables. An important special case of integer variables are *binary variables*. These are integer variables that can take only the values 0 and 1. If a variable x is binary, this is described by $x \in \{0, 1\}$ in the formulation. In fact, this is simply a short form of writing $x \in \mathbb{Z}$ and $x \geq 0$ and $x \leq 1$.

- **Linear constraints.** Assume that our variables are called $x_1, ..., x_n$. A MIP may contain an arbitrary (finite) number of constraints such that each constraint is of the form

  $$a_1x_1 + a_2x_2 + ... + a_nx_n \leq b$$

  or

  $$a_1x_1 + a_2x_2 + ... + a_nx_n \geq b$$

  or

  $$a_1x_1 + a_2x_2 + ... + a_nx_n = b$$

  for values $a_1, ..., a_n \in \mathbb{R}$ and $b \in \mathbb{R}$. Note that it is *not* allowed to introduce a constraint with a strict inequality, i.e., $a_1x_1 + a_2x_2 + ... + a_nx_n < b$ or $a_1x_1 + a_2x_2 + ... + a_nx_n > b$. Also, it is not allowed to multiply variables or divide by them, e.g., $x_1 \cdot x_2 + (x_3)^2 - x_2/x_4 \leq 5$.

- **Linear objective function.** Assume again that our variables are called $x_1, ..., x_n$. A MIP has an objective function of the form

  $$\min c_1x_1 + c_2x_2 + ... + c_nx_n$$

or

$$\max c_1x_1 + c_2x_2 + ... + c_nx_n$$

for values $c_1, ..., c_n \in \mathbb{R}$. Note that the MIP can have only *one* objective function. Also, note that a special case is when $c_1 = c_2 = ... = c_n = 0$ and then the objective is simply min 0 or max 0.

When we solve the MIP, we are looking for a vector $x = (x_1, ..., x_n)$ that satisfies every single constraint; we say that such a vector x is *feasible*. The *objective function value* of a vector x is $c_1x_1 + c_2x_2 + ... + c_nx_n$. When we solve our MIP, we are looking for a feasible vector x with the minimum or maximum objective function value among all feasible solutions (depending on whether our objective is $\min c_1x_1 + c_2x_2 + ...$ or $\max c_1x_1 + c_2x_2 + ...$). Note that if the objective function is min 0 or max 0 then we are only interested in computing *some* feasible vector x, but any feasible vector is equally good.

An important special case of a MIP arises when there are no integer variables. Such a program is called a *linear program (LP)*. Also, in some textbooks and papers in the literature, authors make a distinction between an integer program (IP) and a MIP, where the former is a MIP in which *all* variables are integer. All steps below for coding a MIP in Python can be applied equally for modeling IPs and LPs. Therefore, in the following we will simply talk about solving MIPs, rather than distinguishing between LPs, IPs, and MIPs.

1.3 Outcomes when solving a MIP

When you solve a MIP, there are three possible answers that you may get:

- **Optimal solution.** The solution is a feasible vector x such that there is no other feasible vector x' whose objective function value is strictly better than the objective function value of x. Thus, the vector x is the best solution that you could hope for.
- **Infeasible.** It can happen that there does not even exist a single feasible vector x because our constraints are (intuitively) too strict. In this case, we say that the MIP is *infeasible* which is the correct output when we solve it.
- **Unbounded.** It can happen that there are infinitely many solutions to a MIP such that their objective function values are not bounded by any value. For example, suppose that you want to maximize your objective function and there is a solution with objective function value 1, another solution with value 2, one with value 3, and so on. In this case, the correct answer when solving a MIP is that the MIP is *unbounded*.

When you solve your MIP with a solver, you expect to get one of these three answers. In practice, you may want to give a time limit to your solver or interrupt its computation manually after some time. In this case, it can be that you get a solution x that is feasible but potentially not optimal. Typically, then the solver still gives you some guarantee regarding how close the objective function value of x is to the optimal objective function value, e.g., it states that it is at most 15% worse (but potentially better than that). Also, it may happen that a solver figures out that your MIP is infeasible *or* unbounded, but that it cannot decide which of the two is the correct answer. However, the latter is a rather rare case in practice.

2 MIP solver

You need a MIP solver in order to solve your MIP. Such a solver is an external program that you typically call within your Python program via a library. There are several solvers you can choose from and they come in two categories:

- **Commercial solvers.** Well known commercial solvers are for example Gurobi, CPLEX, COPT, and FICO Xpress. Typically, commercial solvers offer a free academic licence that you can, for example, use in your bachelor/master thesis, PhD thesis, or in a project that is part of your studies. Please

check the webpages of the respective solvers for the conditions of their academic licences. If you want to use these solvers for other purposes, you need to purchase a non-academic licence which can be quite expensive (easily more than 10,000 USD). However, commercial solvers can be *much* faster than non-commercial solvers and, therefore, solve instances that non-commercial solvers cannot solve. If you are in doubt which commercial solver you should use (e.g., for your master thesis), then Gurobi is probably a good solver to start.

- **Non-commercial solvers.** These are solvers that you can typically use for free, even for commercial purposes, and they usually have open-source code. Well known non-commercial solvers are for example SCIP, HiGHS, CBC, and GLPK. In my experience, SCIP and HiGHS tend to be the fastest non-commercial solvers (even though it is hard to make such a general statement since this depends on the instance you want to solve).

Before using your solver, you need to install it and, possibly, request a licence for them. Please check the installation instructions for the solver you want to use.

3 Modeling framework

Apart from a solver, you need a *modeling framework* in order to define the MIP that you want to solve. These frameworks provide you with a Python interface via which you can call the MIP solver, pass your MIP model to it, and receive the computed solution from the solver. There are two categories of frameworks:

- **Solver-independent frameworks.** In such frameworks, you define your MIP and then choose a MIP solver among a variety of possible options (see Section 2). Typically, to change the used MIP solver, you need to change only a few lines of your code. Examples of such frameworks are Pyomo, Pulp, and Python-MIP for Python and JuMP for Julia. Note that Python-MIP supports only the solvers Gurobi and CBC. Solver-independent frameworks are great if you want to try which solver works best for the MIP you want to solve. Typically, these frameworks are open source (please check their licences for details, e.g., for commercial usage).

- **Solver-dependent frameworks.** Such frameworks work only for one specific solver and, therefore, there is a chance that it uses the abilities of this specific solver better than solver-independent frameworks. In particular, they may be faster, allow more options, etc. Typically, these frameworks are provided by the developers of the corresponding solver. An example for such a framework is the Gurobi Python interface that, as the name suggests, works only with Gurobi.

The main difference between these frameworks is the syntax for defining your MIP and possibly their performance. Note that it is not that easy to change from one framework to another since then you need to change your code a lot (even though ChatGPT can probably help you a lot with this). In this document, we will work with the solver-independent framework Pyomo.

Before using your framework, you need to install it. Please check the installation instructions for the framework that you want to use.

4 Solving a simple MIP

In this section, we discuss a simple concrete MIP and we will see how we can solve it via Pyomo. It is designed such that it contains all elements, i.e., types of variables and constraints, that could appear in a MIP. Thus, you should be able to use the code below easily as a template for other MIPs.

Consider the following MIP.

$$
\begin{array}{lrrrrrrrl}
\min & x_1 & +5x_2 & +3x_3 & & -x_5 & & & \\
\text{s.t.} & 2x_1 & +3x_2 & -4x_3 & + & x_4 + x_5 & + y_1 & & \leq 180 \\
& 4x_1 & +3x_2 & & & -4x_4 + x_5 & & -y_2 & \geq -240 \\
& 5x_1 & +2x_2 & & & -5x_4 + x_5 & & & = 120 \\
& x_1 & & & & & & & \geq 0 \\
& & x_2 & & & & & & \leq 0 \\
& & & x_3 & & & & & \geq 0 \\
& & & x_3 & & & & & \in \mathbb{Z} \\
& & & & & x_4 & & & \text{free} \\
& & & & & & x_5 & & \in \{0,1\} \\
& & & & & & & y_1 & \geq 0 \\
& & & & & & & & y_2 \geq 0
\end{array}
$$

You can solve it with the code below.

```
1  import pyomo
2  import pyomo.environ as pyo
3
4  m = pyo.ConcreteModel()
5
6  m.x1 = pyo.Var(domain=pyo.NonNegativeReals)
7  m.x2 = pyo.Var(domain=pyo.NonPositiveReals)
8  m.x3 = pyo.Var(domain=pyo.NonNegativeIntegers)
9  m.x4 = pyo.Var(domain=pyo.Reals)
10 m.x5 = pyo.Var(domain=pyo.Binary)
11 m.y = pyo.Var([1,2], domain=pyo.NonNegativeReals)
12
13 m.constr1 = pyo.Constraint(expr=2*m.x1 + 3*m.x2 -4*m.x3 +m.x4 +m.x5 + m.y[1] <= 180)
14 m.constr2 = pyo.Constraint(expr=4*m.x1 + 3*m.x2 -4*m.x4 +m.x5 - m.y[2] >= -240)
15 m.constr3 = pyo.Constraint(expr=5*m.x1 + 2*m.x2 -5*m.x4 +m.x5 == 120)
16
17 m.obj = pyo.Objective(expr=m.x1 + 5*m.x2 + 3*m.x3 - m.x5, sense = pyo.minimize)
18
19 solver=pyomo.opt.SolverFactory('cbc')
20 results=solver.solve(m)
21
22 print("x1: " + str(m.x1.value))
23 print("x2: " + str(m.x2.value))
24 print("x3: " + str(m.x3.value))
25 print("x4: " + str(m.x4.value))
26 print("x5: " + str(m.x5.value))
27 print("y1: " + str(m.y[1].value))
28 print("y2: " + str(m.y[2].value))
29
30 print("Objective function value of found solution: " + str(m.obj()))
```

Note that we defined all y-variables in one single line (line 11) which was possible since all these variables are non-negative real variables (i.e., we have the constraints $y_1 \geq 0$ and $y_2 \geq 0$). In contrast, we defined

the x-variables one by one since x_1 is a non-negative real variable, x_2 is a non-positive real variable, x_3 is a non-negative integer variable, x_4 is a free (real) variable, and x_5 is a binary variable.

If you run the code above, you get the following output:

```
x1: 0.0
x2: -240.14286
x3: 0.0
x4: -119.85714
x5: 1.0
y1: 0.0
y2: 0.0
Objective function value of found solution: -1201.7143
```

Note that you can access the computed values of the variables with the expressions `m.x1.value`, `m.x2.value`, etc. Also, you can access the objective function value with `m.obj()`.

If in your objective function you want to *maximize* the function $x_1+4x_2+3x_3-x_5$ (instead of minimizing it) then you need to change line 17 to

```
17 m.obj = pyo.Objective(expr=m.x1 + 5*m.x2 + 3*m.x3 - m.x5, sense = pyo.maximize)
```

Similarly, you can define arbitrary minimization or maximization objective functions by changing `expr=...` and `sense=...` accordingly.

In the code above, we solved our MIP via the solver CBC. In line 19, we selected CBC with the statement

```
19 solver=pyomo.opt.SolverFactory('cbc')
```

It is very easy to use a different solver: just replace `'cbc'` by a different term, depending on the solve that you want to use. For example, you can write `'gurobi'`, `'glpk'`, `'scip'`, or `'xpress'` if you want to use Gurobi, GLPK, SCIP or Xpress, respectively.

4.1 Infeasible or unbounded MIPs

Note that the above MIP was feasible and not unbounded. If you solve a MIP that is infeasible or unbounded, then you do not get a solution. In particular, you get an exception if you solve such a MIP and then access the value of a variable, e.g., via `m.x1.value`, or the value of your objective function value via `m.obj()`.

Normally, you want to avoid getting an exception. Therefore, after you solved your MIP you might want to check whether your solver found an optimal solution, whether the MIP was infeasible, or whether the solver found out that the MIP is unbounded. You can check this with `results.solver.termination_condition` as you can see in the code below.

```
1  import pyomo
2  import pyomo.environ as pyo
3  from pyomo.opt import SolverStatus, TerminationCondition
4
5  m = pyo.ConcreteModel()
6
7  m.x1 = pyo.Var(domain=pyo.NonNegativeReals)
8  m.x2 = pyo.Var(domain=pyo.NonPositiveReals)
9  m.x3 = pyo.Var(domain=pyo.NonNegativeIntegers)
10 m.x4 = pyo.Var(domain=pyo.Reals)
11 m.x5 = pyo.Var(domain=pyo.Binary)
```

```
m.y = pyo.Var([1,2], domain=pyo.NonNegativeReals)

m.constr1 = pyo.Constraint(expr=2*m.x1 + 3*m.x2 -4*m.x3 +m.x4 +m.x5 + m.y[1] <= 180)
m.constr2 = pyo.Constraint(expr=4*m.x1 + 3*m.x2 -4*m.x4 +m.x5 - m.y[2] >= -240)
m.constr3 = pyo.Constraint(expr=5*m.x1 + 2*m.x2 -5*m.x4 +m.x5 == 120)

m.obj = pyo.Objective(expr=m.x1 + 5*m.x2 + 3*m.x3 - m.x5, sense = pyo.minimize)

solver=pyomo.opt.SolverFactory('cbc')
results=solver.solve(m)

if (results.solver.termination_condition == TerminationCondition.optimal):
    print("Solution: ")
    print("---------")
    print("x1: " + str(m.x1.value))
    print("x2: " + str(m.x2.value))
    print("x3: " + str(m.x3.value))
    print("x4: " + str(m.x4.value))
    print("x5: " + str(m.x5.value))
    print("y1: " + str(m.y[1].value))
    print("y2: " + str(m.y[2].value))
    print("Objective function value of found solution: " + str(m.obj()))
elif (results.solver.termination_condition == TerminationCondition.infeasible):
    print("MIP is infeasible, no solution exists")
elif (results.solver.termination_condition == TerminationCondition.unbounded):
    print("MIP is unbounded")
```

Thus, with the statement `results=solver.solve(m)` we store certain information about the solving progress in `results` and then check the value of `results.solver.termination_condition`. There are also more termination conditions, e.g., that indicate a missing licence for your solver.

5 Solving an abstract MIP

Our MIP in Section 4 was a MIP that was defined via a concrete set of variables and constraints. In many applications, we define a MIP model in an abstract sense. We specify that we assume certain input data and then define how our MIP looks like, depending on these input data. For example, in a scheduling problem, we might assume that we are given a set of jobs J and a processing time $p_j \geq 0$ for each job $j \in J$ and a set of machines M. Then we might define that our MIP has a constraint for each machine $i \in M$, a variable x_{ij} for each combination of a machine $i \in M$ and a job $j \in J$, etc. Thus, the actual variables and constraints depend on the given input data.

In this section, we discuss one way to implement such an abstract MIP model in Pyomo (there are also other ways to implement this in Pyomo). We do this with an example problem. This problem itself is not very important though, it was chosen to have an example that contains most construction elements that you could find in an arbitrary (abstract) MIP formulation.

In our example problem, the input data consists of two sets J and M that represent a set of jobs and machines, respectively. For each job $j \in J$ we are given a set of machines $M(j) \subseteq M$. Also, for each combination of a job $j \in J$ and a machine $i \in M(j)$ we are given values $p_{ij} \geq 0$ and $c_{ij} \geq 0$. Moreover, we are given parameters $T \geq 0$ and $c^* \geq 0$. The goal is to assign each job $j \in J$ to some machine $i \in M(j)$. On such a machine i, the job j has a processing time of p_{ij} and it incurs a cost c_{ij}. On each machine $i \in M$, the total processing time of the assigned jobs must be at most T. There is also the additional constraint that each machine i can get at most three jobs j for which $c_{ij} > c^*$ or $p_{ij} \geq T/5$. This constraint is actually not

very well motivated; it was only added so that the example contains more complicated syntax constructions. The objective is to minimize the total cost due to assigning the jobs.

The complete MIP model is the following:

$$
\begin{aligned}
\min \quad & \sum_{i \in M} \sum_{j \in J} c_{ij} x_{ij} & & \\
\text{s.t.} \quad & \sum_{i \in M} x_{ij} = 1 & & \forall j \in J \\
& \sum_{j \in J : i \in M(j)} p_{ij} x_{ij} \leq T & & \forall i \in M \\
& \sum_{j \in J : i \in M(j) \wedge (c_{ij} > c^* \vee p_{ij} \geq T/5)} x_{ij} \leq 3 & & \forall i \in M \\
& x_{ij} = 0 & & \forall j \in J \, \forall i \in M(j) : p_{ij} > T \\
& x_{ij} \in \{0, 1\} & & \forall i \in M, j \in J : i \in M(j)
\end{aligned}
$$

Note that "$\wedge$" stands for "and" and "$\vee$" stands for "or". Thus, the term $j \in J : i \in M(j) \wedge (c_{ij} > c^* \vee p_{ij} \geq T/5)$ means "all jobs $j \in J$ for which $i \in M(j)$ is true and additionally $c_{ij} > c^*$ it is true or $p_{ij} \geq T/5$ it is true".

You can implement the MIP above as follows:

```
import pyomo
import pyomo.environ as pyo


jobs = {1, 2, 3, 4, 5}
machines = {"m1", "m2", "m3"}

p = {
    "m1": {
        1: 5,
        3: 2,
        4: 1,
        5: 6
    },
    "m2": {
        2: 2,
        3: 7,
        4: 6
    },
    "m3": {
        2: 5,
        3: 28,
        5: 3
    }
}

c = {
    "m1": {
        1: 3,
        3: 5,
        4: 7,
```

```
        5: 2
    },
    "m2": {
        2: 4,
        3: 1,
        4: 6
    },
    "m3": {
        2: 5,
        3: 9,
        5: 3
    }
}

M = {
    1: {"m1"},
    2: {"m2", "m3"},
    3: {"m1", "m2", "m3"},
    4: {"m1", "m2"},
    5: {"m1", "m3"}
}

T=15

c_star=2

m = pyo.ConcreteModel()

# set x_indices contains all combinations (i,j) such that job j can be assigned on machine i
x_indices = {(i, j) for i in machines for j in jobs if i in M[j]}

# initialize set of variables
m.x = pyo.Var(x_indices, within=pyo.Binary)

# objective: minimize total cost
m.cost = pyo.Objective(expr= sum(c[i][j]*m.x[(i,j)] for j in jobs for i in M[j]), sense=pyo.minimize)

# constraint: each job assigned
m.eachJobAssigned = pyo.Constraint(jobs, rule=lambda m,j: sum(m.x[(i,j)] for i in M[j])==1)

# constraint: for each machine the makespan is at most T
m.makespan = pyo.Constraint(machines, rule=lambda m,i:
    sum(p[i][j]*m.x[(i,j)] for j in jobs if i in M[j])<=T)

# constraint: on each machine i at most three jobs j with c_ij>c_star or p_j >= T/5
m.atmostthree = pyo.Constraint(machines, rule=lambda m,i:
    sum(m.x[(i,j)] for j in jobs if i in M[j] and (c[i][j]> c_star or p[i][j] >= T/5))<=3)

# constraint: set x_ij to 0 if p_ij > T
m.setzero = pyo.Constraint(x_indices, rule=lambda m,i,j:
    m.x[(i,j)] ==0 if p[i][j]>T else pyo.Constraint.Skip)
```

```
83
84  solver = pyomo.opt.SolverFactory('cbc')
85  solver.solve(m)
86
87  print("Objective function value of found solution: " + str(m.cost()))
88
89  for j in jobs:
90      for i in M[j]:
91              if m.x[(i,j)].value == 1:
92                  print("Job " + str(j) + " on machine "+ str(i))
```

As you can see, at the beginning we define the input data via Python dictionaries (lines 5-54) and variables (lines 54-56). Note that here you could also load the input data from external files or databases (which you typically do in practice). In particular, the code after `m = pyo.ConcreteModel()` is completely independent from the actual input data! Therefore, you can use that code for *any* input data for this problem.

Also note that we defined the set `x_indices` to be the set of all pairs `(i,j)` for $i \in M(j)$. In principle, we could as well have introduced an x-variable for *all* pairs of a machine i and a job j and then added a constraint that sets each variable x_{ij} to 0 if $i \notin M(j)$, i.e.,

$$\begin{aligned} \min \quad & \sum_{i \in M} \sum_{j \in J} c_{ij} x_{ij} & \\ \text{s.t.} \quad & \sum_{i \in M} x_{ij} = 1 & \forall j \in J \\ & \vdots & \\ & x_{ij} = 0 & \forall j \in J, i \in M : i \notin M(j) \\ & x_{ij} \in \{0,1\} & \forall i \in M, j \in J \end{aligned}$$

The corresponding code would look as follows

```
model.x = pyo.Var(machines,jobs,within=pyo.Binary)
...
m.setzero2 = pyo.Constraint(machines, jobs, rule=lambda m,i,j:
        m.x[i,j] ==0 if not(i in M[j]) else pyo.Constraint.Skip)
```

Notice that in this version we address the x-variables with `x[i,j]` instead of `x[(i,j)]` as before, since in the definition of the variables we write `pyo.Var(machines,jobs,within=pyo.Binary)` instead of `pyo.Var(x_indices,within=pyo.Binary)`. However, in this way we would have introduced many variables that we set to 0 anyway. You save running time if you do not introduce such variables in first place (in the model above, if a variable is always 0 then it is equivalent to simply omit it). In a similar way, in our model above we could have excluded all variables x_{ij} for which $p_{ij} > T$. We did not do this for didactical reasons since the version above includes an example of the construction `if ... else pyo.Constraint.Skip`).

If you execute the code above, you get the following output:

```
Objective function value of found solution: 16.0
Job 1 on machine m1
Job 2 on machine m2
Job 3 on machine m2
Job 4 on machine m2
Job 5 on machine m1
```

6 Conclusion

I hope that this document helps you solving MIPs. In particular, when you use the codes above as templates, you can probably code and solve many MIPs using Python and Pyomo. If you find any errors in this document or if you have any feedback or suggestions, please feel free to contact me via `andreas.wiese@tum.de`.