IBM ILOG CPLEX PYTHON API

Version 20.1.0

The Python API for CPLEX supports full functionality of CPLEX which supports the creation and solution of models for a myriad of optimization disciplines (including non-linear methods). The scope of this documentation covers the installation steps as well as formulations of basic linear programming (LP), integer programming (IP) and mixed integer programming (MIP) models. For a comprehensive guide, please refer to the IBM CPLEX user's manual.

Table of Contents

Download/Installation Instructions

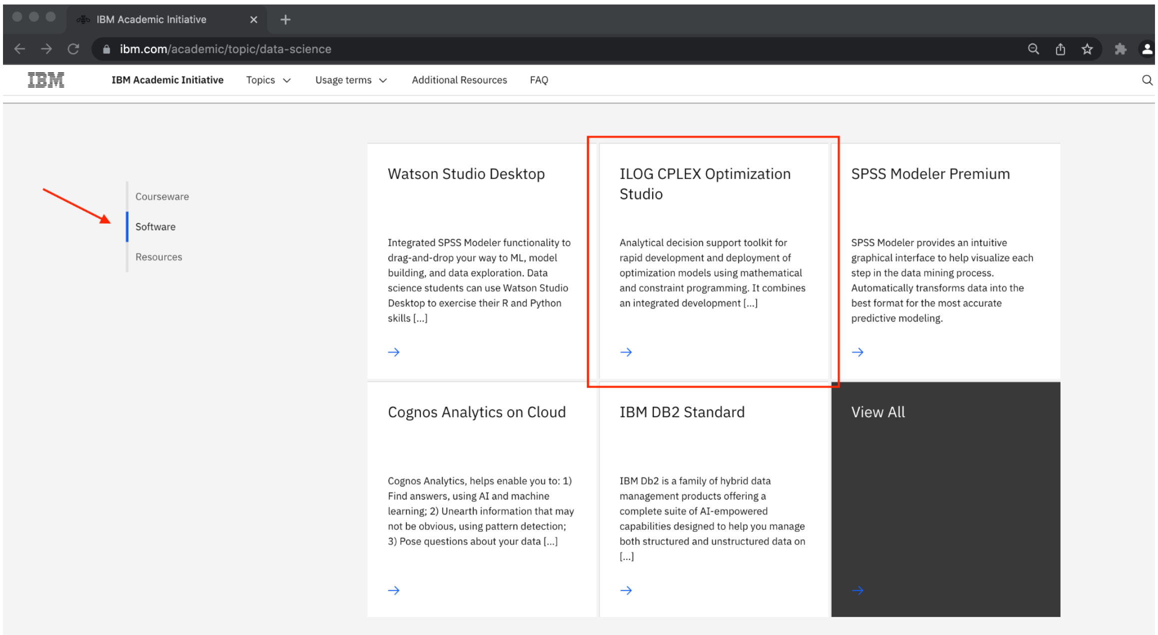
Community License

For the free community module, we could directly install the “cplex” package using any package management system for python. One method is to use “Package Installer for Python” or pip from your computer’s terminal.

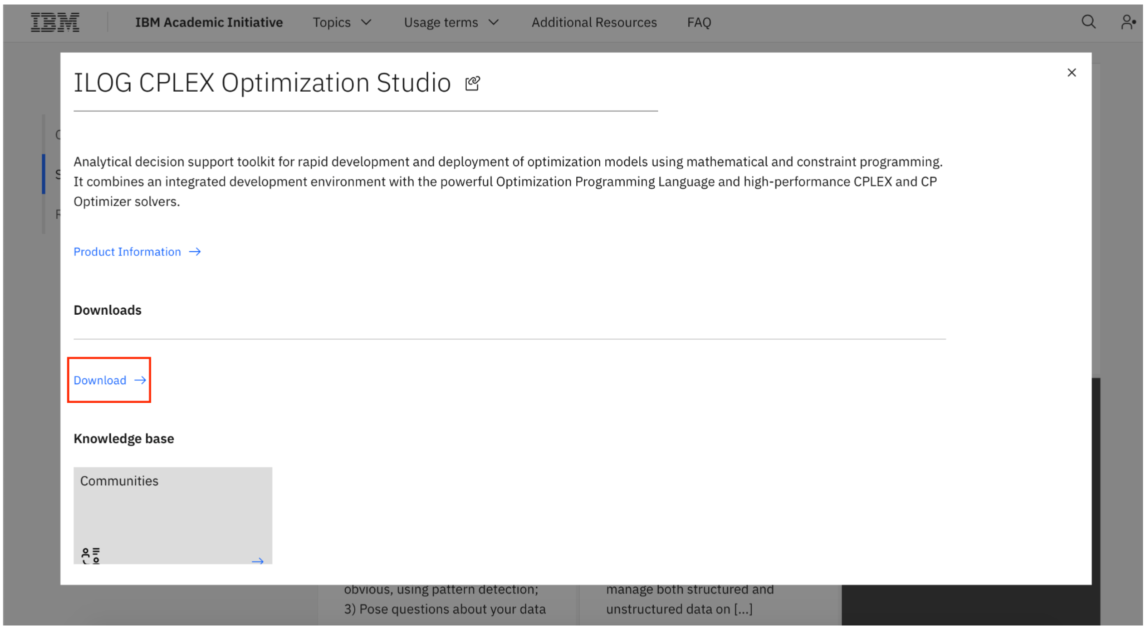
The community version has full functionality for LP and MIP models, however, there are limitations on problem size. The number of decision variables and constraints can not exceed 1000 in this version.

Academic License through IBM Academic Initiative

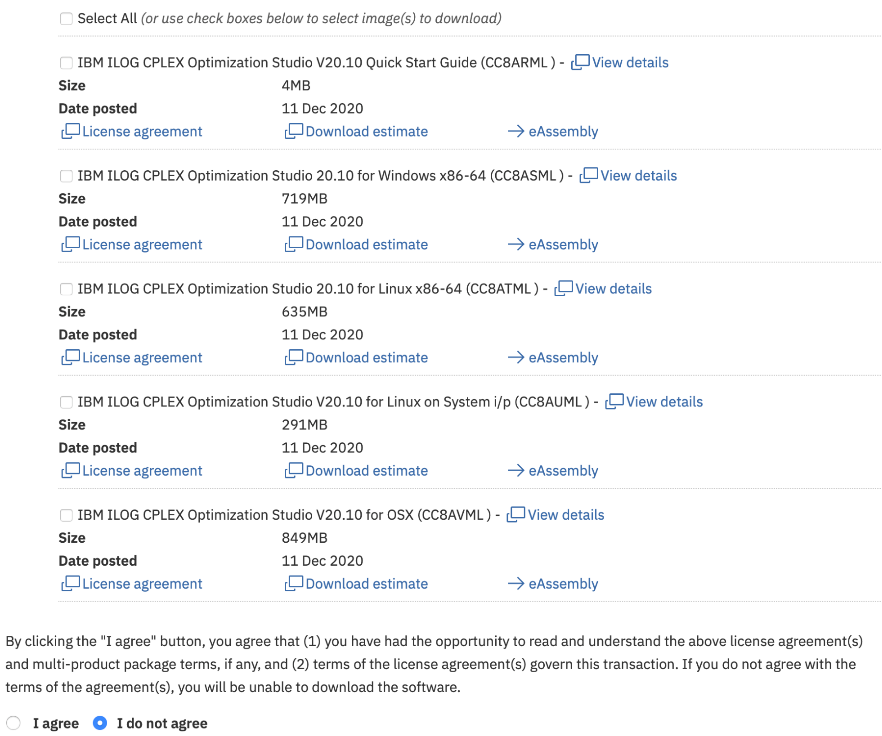
IBM offers students and members of academia an unlimited version of their CPLEX optimization software as a part of their initiative. It could be found on the data science topic page, under the software tab. To be able to access the software, users must register for the initiative using their institutional email address.



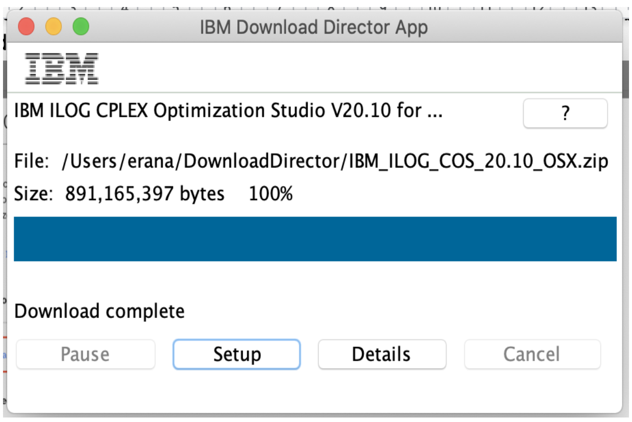
After registering and logging in, if you click the blue arrow in the indicated red box, a pop-up will open. Click on download.

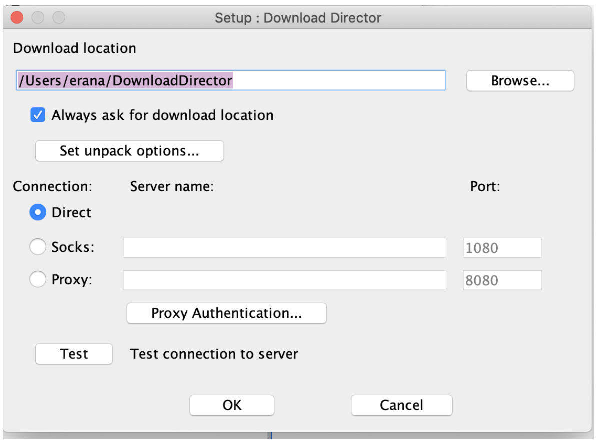


You will be directed to the results of a part number search on a new tab. For validation purposes, we note that the part number for the latest version of IBM ILOG CPLEX is CJ8GXML.



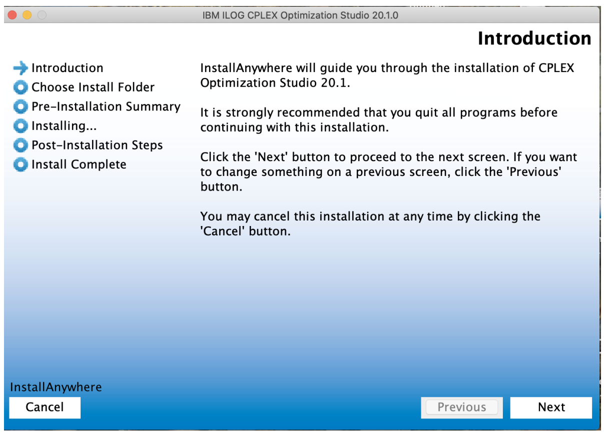
Choose the appropriate version for your operating system and click on I agree. If you are using the IBM download director, note that you must have the appropriate version of java installed. Designate a location for the download. After the download is complete, the compressed file *IBM_ILOG_COS_20.10_*<operating system name>*.zip* should be visible under the selected path.





You could also directly use the HTTP download option which is relatively easier. In that case, you should again see the compressed file *IBM_ILOG_COS_20.10_*<operating system name>*.zip* under your downloads folder.

We click on the file to decompress and run. An installer window will open and prompt the user to select a language. For reference, this is what it should look similar to. Note that MacOS users who use dark mode display settings might experience a bug here. We recommend switching to light mode during installation and reverting back to the original preference after the installation is complete.



Keep clicking on next on the installer window. It will first ask you to accept the terms of use agreement and then choose a location for the application folder to be installed. In the Post-installation steps, you will receive information on the Python API. Copy and enter the given command on your computer's terminal to install the CPLEX engines into your python environment. After this step, simply the command "import cplex" can be used to access CPLEX functionality in your code!

Starting Modeling: How to create a model instance?

Under this section, some useful methods for model formulation could be found. Again let it be stated that you may refer to the IBM documentations linked for a comprehensive list.
We start by importing the cplex module we have installed.
To first create a problem instance we use ".Cplex()" from cplex.

```
import cplex
# Create a modeling problem instance
model = cplex.Cplex()
```

Now the Cplex class object "*model*" has been created. We will use interface and category methods to specify our Cplex object instance:

- Methods for adding,modifying, and calling model specific data such as objective function coefficients, decision variables and bounds on said decision variables could be found under the Cplex.variables interface,
- Objective sense is set through Cplex.Objective interface,
- We add and modify the linear constraints using Cplex.linear_constraints,
- We will solve the model by calling .solve() on the Cplex object and access the outputs through the Cplex.solution interface.

Defining Decision Variables

The Cplex API can differentiate variables using their unique string names. Hence each name defined corresponds to a distinct decision variable. Consequently we can refer to certain decision variables using either their string name or list index.

variables.add(obj, lb, ub, names, types, columns):

This method adds decision variables to the problem with six given arguments. We will mostly make use of the following five :

obj: A list representing objective coefficients (list of floats). Ordered based on the list of decision variables.

lb: A list representing the lower bounds of variables (list of floats). Ordered based on the list of decision variables.

ub: A list representing the upper bounds of variables (list of floats). Ordered based on the list of decision variables.

- Use `cplex.infinity` for ∞

names: A list of decision variable names (list of strings) *types:* A list of types or a string of characters representing the types of variables. Ordered based on the list of decision variables.

- Use `"C"` or `model.variables.type.continuous` for continuous variables
- `"I"` or `model.variables.type.integer` for integer variables
- `"B"` or `model.variables.type.binary` for binary variables

```
# The coefficients of the objective function
objective = [5.0, 2.0, -1.0]

# The lower bounds... Note that the default values are 0.
lower_bounds = [0.0, 0.0, 0.0]

# The upper bounds... The default values are infinity.
upper_bounds = [100, 1000, cplex.infinity]

# The names of our variables
variable_names = ['X1', 'X2', 'X3']

variable_types = [model.variables.type.continuous, model.variables.type.integer,
model.variables.type.binary]
model.variables.add(obj = objective,
                     lb = lower_bounds,
                     ub = upper_bounds,
                     names = variable_names,
                     types= variable_types)# or equivalently types = "CIB")
```

Though not all arguments have to be specified at once, if more than one argument is given this method works properly only when they are of the same length. The reasoning behind this rule could be intuitively explained: we should have as many lower bounds, upper bounds, objective coefficients, types and columns as there are variables being added.
If more flexibility is desired about control over properties of decision variables, individual property setter/getter methods are defined:

variables.set_lower_bounds(i,lb)/variables.get_lower_bounds():

Takes:

- The name or index of the decision variable being bounded *i*
- The lower bound *lb*

We can also give a list of tuples to set more than one variable bound at once, with each tuple containing a variable-bound pair.

```
model.variables.set_lower_bounds([("X1", 0.0 ),(1, 0.0)])

#index 1 in the second tuple represents "X2",recall our list of variable names

model.variables.set_lower_bounds("X3",0)
model.variables.get_lower_bounds() # will return [0.0, 0.0, 0.0] for this problem
```

variables.set_upper_bounds(i,ub)/variables.get_upper_bounds():

Very similar to the method defined above, we specify variable *i* and an upper bound *ub* pairs:

```
model.variables.set_upper_bounds([("X1", 100 ),(1, 1000)])

model.variables.set_upper_bounds("X3",cplex.infinity)

model.variables.get_upper_bounds() # returns [100.0, 1000.0, 1e+20]
```

variables.set_types(i, type)/get_types():

Takes:

- The name or index of the decision variable being bounded *i*
- The type of the decision variable *type*

We can also give a list of tuples to set more than one variable bound at once, with each tuple containing a variable-type pair. For detailed type parameter descriptions please refer to the section explaining the method Cplex.variables.add().

```
model.variables.set_types("X1",model.variables.type.continuous)
model.variables.set_types([("X2",model.variables.type.integer),("X3",model.variable
s.type.binary)])

model.variables.get_types() # will return ['C', 'I', 'B']
```

variables.delete(i):

We can delete a decision variable by specifying either their name or index i.
We can also delete multiple decision variables by giving a list of variable names or indices.

```
model.variables.delete(0) #deletes X1

model.variables.delete([0,"X3"]) #deletes X2 and X3, since X1 was deleted above
```

The statement below deletes all variables.

```
model.variables.delete() #deletes all variables
```

Note that once the variables are deleted, we must also check the constraints and remove the deleted variables. Otherwise the code will raise an error.

Objective Sense and Name

ObjectiveInterface class is composed of set/ get methods related to various properties of the objective function

ObjectiveInterface.set_sense(sense)/ ObjectiveInterface.get_sense():

Set the type of the objective through selecting one of the lines written below. This way, we inform the model on our improving direction:

```
# In this model, we want to specify the improving direction:
model.objective.set_sense(model.objective.sense.maximize)#for max type objective
model.objective.set_sense(model.objective.sense.minimize)#for min type objective
```

After the model sense is set, we can query it using get_sense(). The function will return -1 for a maximization problem and 1 for a minimization type.

```
model.objective.get_sense()
```

ObjectiveInterface.set_name(name)/ObjectiveInterface.get_name():

For tractability we may prefer to label the objective function. Assume we are minimizing total cost hence the name of my objective function is "Total Cost". The setter-getter methods are as follows:

```
model.objective.set_name("Total Cost")#takes name string as parameter
model.objective.get_name()#returns the name given to the obj
```

Adding Constraints

linear_constraints.add(lin_expr, senses, rhs, range_values, names):

Similar to the previous add function discussed, if more than one argument is given, all must be of the same length. Although the function can take 5 parameters, we will mostly make use of four:

lin_expr: An argument specifying the decision variables and their corresponding coefficients for the constraint being added (sparse pair or a list-of-lists, first the decision variables then their coefficients).

> **Note:** A decision variable can only have one coefficient associated with it in a single constraint. Duplicate entries of the same variable in the same lin_expr is

senses: A concatenated string of single character strings or a list of single character strings representing the sense of each constraint. Ordered based on the list of constraints.

- Use `"E"` for equality,
- `"L"` for less than or equal to,
- `"G"` for greater than or equal to constraints.

rhs: A list representing the right hand sides(list of floats). Ordered based on the list of constraints.

names: A list of names given to the constraints. Ordered based on the list of constraints.

For sake of completeness let us state the purpose of *range_values* : They interact with the RHS to create an artificial left-hand-side. By identifying the range value, we "squeeze" the linear expression specified in the constraint. We could achieve the same effect by adding multiple constraints containing the same linear expression.

```
# Say we have the following constraints:
# "2*X1 + X2 - X3 <= 70" and "5*X1 + 4*X2 + 4*X3 <= 130"
# Then the constraints could be represented as:
# [  2   1  -1 ]   [ X1 ] <= [  70 ] --> first constraint
# [  5   4   4 ]   [ X2 ] <= [ 130 ] --> second constraint
#                  [ X3 ]

# We give names to the constraints to be able to query them more easily later on
constraint_names = ['C1', 'C2']

first_constraint = [['X1', 'X2', 'X3'], [2.0, 1.0, -1.0]] #C1 --> first constraint
second_constraint = [[0, 1, 2], [5.0, 4.0, 4.0]] #C2 --> second constraint
                                               # We can also refer to variables
                                               #using their list indices

constraints = [ first_constraint, second_constraint ] #list of constraints

rhs = [70.0, 130.0] #list of corresponding RHS values

constraint_senses = [ "L", "L" ] #list of constraint senses (types)
                                #We can also use the single string "LL"

# All arguments are created: Add the constraints
model.linear_constraints.add(lin_expr = constraints,
                             senses = constraint_senses,
                             rhs = rhs,
                             names = constraint_names)
```

linear_constraints.set_rhs(i,value)/ linear_constraints.get_rhs():

Takes:

- The index of the constraint *i*
- The name of the constraint to be given *name*

We could also give a list of pairs as an input to control several RHS values at once.

```
model.linear_constraints.set_rhs(0,70.0) # Constraint 1 has the RHS 70.0
model.linear_constraints.set_rhs('C2',130.0) # Constraint 2 has the RHS 130.0
model.linear_constraints.set_rhs([(0,70.0), ('C2',130.0)]) #Same functionality
                                                          #as the lines above
```

linear_constraints.set_linear components(i,lin):

Takes:

- The name or index of the constraint *i*
- The pair of decision variable(s) and corresponding coefficient(s) *lin*

```
model.linear_constraints.set_linear_components('C1',[['X1'],[2.0]])
model.linear_constraints.set_linear_components('C1',[['X2','X3'],[1.0,-1.0]])
# "2*X1 + X2 - X3" is specified as the linear component of C1
```

linear_constraints.set_names(i,name)/ linear_constraints.get_names():

Takes:

- The index or old name of the constraint *i*
- The new name of the constraint to be given *name*

```
model.linear_constraints.set_names('C1',"first") #Name 'C1' is now 'first'

model.linear_constraints.set_names([("first","firstFirst"),(1,"second")])

#Name 'first' (C2) is overwritten as 'firstFirst', 'C2' as 'second'.

#Multiple renames could be done at the same time.
```

linear_constraints.get_rows(i)

Takes the name or index of the constraint *i*
Returns the constraint in the form of a list of SparsePairs

```
model.linear_constraints.get_rows()
#Will return --> [SparsePair(ind = [0, 1, 2], val = [2.0, 1.0,
-1.0]),SparsePair(ind = [0, 1, 2], val = [5.0, 4.0, 4.0]]
```

Solving the Model

We simply use the command Cplex.solve() on the model to obtain a solution:

```
# We call the solver on the model
model.solve()
```

Checking the Solution

The solution interface contains many useful methods to explicitly view the properties of the solution. We can display the solution method used, elapsed time, slack variables on each constraint at the solution found, optimality gap and many more… Here are some of the basic methods to display the solution itself:

solution.get_objective_value:

Returns the value of the objective function associated with the solution found. We need to call print() on the method to display the objective value.

solution.get_values:

Returns the values of decision variables found in an ordered list, based on the order of the said variables. We need to call print() on the method to display the values.

solution.write(filename):

Writes the solution found to a file name specified.

```
# Print the solution
print(model.solution.get_objective_value())
print(model.solution.get_values())
model.solution.write('solution.txt') #creates a new file and stores the attributes
                                   #of the solution in a written format
```

Examples:

LP

Consider the following linear programming problem:

$$
\begin{aligned}
\max\ & 2x_1 + x_2 + 6x_3 - 4x_4 \\
\text{s.t.}\ & x_1 + 2x_2 + 4x_3 - x_4 \leq 6 \\
& 2x_1 + 3x_2 - x_3 + x_4 \leq 12 \\
& x_1 + x_3 + x_4 \leq 6 \\
& x_i \geq 0 \ i = 1,2,3,4
\end{aligned}
$$

Here's an example code that can find the optimal solution and a sample output:

```
import cplex

model=cplex.Cplex() #creating the model object

objective= [2.0,1.0,6.0,-4.0] #coefficients of the objective function

lower_bounds = [0.0,0.0,0.0,0.0] #no need to specify lines 7,8, and 9 as they
                                 #are at default values
upper_bounds = [cplex.infinity,cplex.infinity,cplex.infinity,cplex.infinity]
variable_names = ['X1','X2','X3','X4']
variable_types = ['C','C','C','C'] #'C' for continuous!
#These were not passed to the add() function as specifying any
#of the types for decision variables makes the problem into a MIP
#which is not needed for this simple LP...

model.variables.add(obj=objective,
                   lb=lower_bounds,
                   ub= upper_bounds,
                   names= variable_names)

#It is a maximization problem
model.objective.set_sense(model.objective.sense.maximize)



#Now we get ready to add the constraints
constraint_names = ['first','second','third']

first_constraint = [['X1','X2','X3','X4'], [1.0, 2.0, 4.0, -1.0]]
second_constraint = [['X1','X2','X3','X4'], [2.0, 3.0, -1.0, 1.0]]
```

```
third_constraint = [['X1','X3','X4'], [1.0, 1.0, 1.0]]
constraints = [first_constraint, second_constraint, third_constraint]

rhs = [6.0,12.0,6.0]

constraint_senses = ['L','L','L']

model.linear_constraints.add(lin_expr= constraints,
                              senses= constraint_senses,
                              rhs= rhs,
                              names= constraint_names)

model.solve()
print("Obj Value:",model.solution.get_objective_value())
print("Values of Decision Variables:",model.solution.get_values())
model.solution.write('LP_solution.txt') #creates a new file and stores the attr.
                                 #of the solution in a written format
```

Once we run the code and receive the output, we see that the optimal value is 12 and the decision variables take the values $X_1 = 6$, $X_2 = 0$, $X_3 = 0$,$X_4 = 0$ at optimality :

```
(base) 192:Desktop erana$ /opt/anaconda3/bin/python /Users/erana/Desktop/LP_Example.py
Version identifier: 20.1.0.0 | 2020-11-10 | 9bedb6d68
CPXPARAM_Read_DataCheck                          1
Tried aggregator 1 time.
No LP presolve or aggregator reductions.
Presolve time = 0.00 sec. (0.00 ticks)

Iteration log . . .
Iteration:     1   Dual infeasibility =            0.000000
Obj Value: 12.0
Values of Decision Variables: [6.0, 0.0, 0.0, 0.0]
```

IP

Consider the following integer programming problem:

$$\begin{aligned} \min\ & 9x_1 + 13x_2 + 10x_3 + 8x_4 + 8x_5 \\ \text{s.t.}\ & 6x_1 + 3x_2 + 2x_3 + 4x_4 + 7x_5 \geq 40 \\ & x_1 \leq 1, x_2 \geq 1, x_3 \geq 2, x_4 \geq 1, x_5 \leq 3 \\ & x_1, x_2, x_3, x_4, x_5 \geq 0 \\ & x_1, x_2, x_3, x_4, x_5 \text{ integer} \end{aligned}$$

Here's an example code that can find the optimal solution and a sample output:

```
import cplex

model=cplex.Cplex() #creating the model object

objective= [9.0,13.0,10.0,8.0,8.0] #coefficients of the objective function

#For this problem, there are 5 constraints which are restricting a single decision
# variable
# When defining the lower and upper bounds, we will exploit this problem specific
# property for practicality!

lower_bounds = [0.0,1.0,2.0,1.0,0.0] #Constraints 3,4,5

upper_bounds = [1,cplex.infinity,cplex.infinity,cplex.infinity,3] #Constraints 2,6

variable_names = ['X1','X2','X3','X4','X5']
variable_types = ['I','I','I','I','I'] #For this problem, since we have integer
                           #decision variables, we do need to specify the types...

model.variables.add(obj=objective,
                   lb=lower_bounds,
                   ub= upper_bounds,
                   names= variable_names,
                   types= variable_types)

#It is a minimization problem
model.objective.set_sense(model.objective.sense.minimize)


#Now we get ready to add the constraints
constraint_names = ['first']

first_constraint = [['X1','X2','X3','X4','X5'], [6.0, 3.0, 2.0, 4.0,7.0]]
```

```
constraints = [first_constraint]

rhs = [40.0]

constraint_senses = ['G']

model.linear_constraints.add(lin_expr= constraints,
                               senses= constraint_senses,
                               rhs= rhs,
                               names= constraint_names)

model.solve()
print("Obj Value:",model.solution.get_objective_value())
print("Values of Decision Variables:",model.solution.get_values())
model.solution.write('IP_solution.txt') #creates a new file and stores the
attributes
                                  #of the solution in a written format
```

Once we run the code and receive the output, we see that the optimal value is 81 and the decision variables take the values $X_1 = 0$, $X_2 = 1$, $X_3 = 2$, $X_4 = 3$, $X_5 = 3$ at optimality.

As it could be seen, the solve procedures for an IP model is more complicated:

```
(base) 192:Desktop erana$ /opt/anaconda3/bin/python /Users/erana/Desktop/IP_Example.py
Version identifier: 20.1.0.0 | 2020-11-10 | 9bedb6d68
CPXPARAM_Read_DataCheck                          1
Found incumbent of value 171.000000 after 0.00 sec. (0.00 ticks)
Tried aggregator 1 time.
MIP Presolve added 1 rows and 1 columns.
Reduced MIP has 2 rows, 6 columns, and 10 nonzeros.
Reduced MIP has 1 binaries, 5 generals, 0 SOSs, and 0 indicators.
Presolve time = 0.00 sec. (0.00 ticks)
Probing time = 0.00 sec. (0.00 ticks)
Tried aggregator 1 time.
MIP Presolve eliminated 1 rows and 1 columns.
MIP Presolve added 1 rows and 1 columns.
Reduced MIP has 2 rows, 6 columns, and 10 nonzeros.
Reduced MIP has 1 binaries, 5 generals, 0 SOSs, and 0 indicators.
Presolve time = 0.00 sec. (0.01 ticks)
Probing time = 0.00 sec. (0.00 ticks)
MIP emphasis: balance optimality and feasibility.
MIP search method: dynamic search.
Parallel mode: deterministic, using up to 8 threads.
Root relaxation solution time = 0.00 sec. (0.00 ticks)

        Nodes                                         Cuts/
   Node  Left     Objective  IInf  Best Integer    Best Bound    ItCnt     Gap

*     0+    0                          171.0000       41.0000           76.02%
      0     0       78.0000     1      171.0000       78.0000        0   54.39%
*     0+    0                           82.0000       78.0000            4.88%
*     0     0      integral     0       81.0000    MIRcuts: 1        2    0.00%
      0     0        cutoff             81.0000       81.0000        2    0.00%
Elapsed time = 0.01 sec. (0.04 ticks, tree = 0.01 MB, solutions = 3)

Mixed integer rounding cuts applied:  1

Root node processing (before b&c):
  Real time             =    0.01 sec. (0.04 ticks)
Parallel b&c, 8 threads:
  Real time             =    0.00 sec. (0.00 ticks)
  Sync time (average)   =    0.00 sec.
  Wait time (average)   =    0.00 sec.
                          ------------
Total (root+branch&cut) =    0.01 sec. (0.04 ticks)
Obj Value: 81.0
Values of Decision Variables: [0.0, 1.0, 2.0, 3.0, 3.0]
(base) 192:Desktop erana$
```

References:

"IBM ILOG CPLEX Optimization Studio Cplex User's Manual." http://home.eng.iastate.edu/, 2015. http://home.eng.iastate.edu/~jdm/ee458/CPLEX-UsersManual2015.pdf.

"IBM(R) ILOG CPLEX Python API Reference Manual." Help. Accessed December 3, 2021. https://www.ibm.com/docs/en/icos/20.1.0?topic=cplex-python-reference-manual.