

# **Python OR-tools Notes**

Kunlei Lian

2/18/23

# Table of contents

<b>Preface</b>	<b>3</b>
<b>1 Introduction</b>	<b>4</b>
<b>2 Environment Setup</b>	<b>5</b>
2.1 Install Homebrew . . . . .	5
2.2 Install Anaconda . . . . .	5
2.3 Create a Conda Environment . . . . .	6
2.4 Install Google OR-Tools . . . . .	7
<b>3 Linear Programming</b>	<b>9</b>
3.1 Modeling Capabilities . . . . .	9
3.1.1 Solver . . . . .	9
3.1.2 Decision Variables . . . . .	10
3.1.3 Constraints . . . . .	11
3.1.4 Objective . . . . .	11
3.1.5 Objective and Constraint Expressions . . . . .	12
3.1.6 Query the Model . . . . .	13
3.2 Applications . . . . .	14
3.2.1 A Trivial Example . . . . .	14
3.2.2 A Transportation Problem . . . . .	16
3.2.3 A Blending Problem . . . . .	20
<b>4 Integer Programming</b>	<b>21</b>
<b>5 Column Generation</b>	<b>22</b>
<b>6 Summary</b>	<b>23</b>
<b>References</b>	<b>24</b>

# Preface

# 1 Introduction

This book covers the usage of Google OR-Tools to solve optimization problems in Python. There are several major chapters in this book:

In Chapter [2](#), we explain the steps needed to setup OR-Tools in a Python environment.

In Chapter [3](#), we use an example to illustrate the modeling capability of OR-Tools to solve linear programming problems.

In Chapter [4](#), we go through the modeling techniques made available in OR-Tools.

## 2 Environment Setup

In this chapter, we explain the steps needed to set up Python and Google OR-Tools. All the steps below are based on MacBook Air with M1 chip and macOS Ventura 13.1.

### 2.1 Install Homebrew

The first tool we need is Homebrew, ‘the Missing Package Manager for macOS (or Linux)’, and it can be accessed at <https://brew.sh/>. To install Homebrew, just copy the command below and run it in the Terminal.

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

We can then use the `brew --version` command to check the installed version. On my system, it shows the info below.

```
~/ brew --version
Homebrew 3.6.20
Homebrew/homebrew-core (git revision 5f1582e4d55; last commit 2023-02-05)
Homebrew/homebrew-cask (git revision fa3b8a669d; last commit 2023-02-05)
```

### 2.2 Install Anaconda

Since there are several Python versions available for our use and we may end up having multiple Python versions installed on our machine, it is important to use a consistent environment to work on our project in. Anaconda is a package and environment manager for Python and it provides easy-to-use tools to facilitate our data science needs. To install Anaconda, run the below command in the Terminal.

```
~/ brew install anaconda
```

After the installation is done, we can use `conda --version` to verify whether it is available on our machine or not.

```
~/ conda --version
conda 23.1.0
```

## 2.3 Create a Conda Environment

Now we will create a Conda environment named 'ortools'. Execute the below command in the Terminal, which effectively creates the required environment with Python version 3.10.

```
~/ conda create -n ortools python=3.10
Retrieving notices: ...working... done
Collecting package metadata (current_repodata.json): done
Solving environment: done
```

```
## Package Plan ##
```

```
environment location: /opt/homebrew/anaconda3/envs/test
```

```
added / updated specs:
- python=3.10
```

The following packages will be downloaded:

package	build		
----- -----			
setuptools-67.4.0	pyhd8ed1ab_0	567 KB	conda-forge
----- -----			
	Total:	567 KB	

The following NEW packages will be INSTALLED:

bzip2	conda-forge/osx-arm64::bzip2-1.0.8-h3422bc3_4
ca-certificates	conda-forge/osx-arm64::ca-certificates-2022.12.7-h4653dfc_0
libffi	conda-forge/osx-arm64::libffi-3.4.2-h3422bc3_5
libsqlite	conda-forge/osx-arm64::libsqlite-3.40.0-h76d750c_0
libzlib	conda-forge/osx-arm64::libzlib-1.2.13-h03a7124_4
ncurses	conda-forge/osx-arm64::ncurses-6.3-h07bb92c_1
openssl	conda-forge/osx-arm64::openssl-3.0.8-h03a7124_0
pip	conda-forge/noarch::pip-23.0.1-pyhd8ed1ab_0
python	conda-forge/osx-arm64::python-3.10.9-h3ba56d0_0_cpython

```

readline          conda-forge/osx-arm64::readline-8.1.2-h46ed386_0
setuptools        conda-forge/noarch::setuptools-67.4.0-pyhd8ed1ab_0
tk                conda-forge/osx-arm64::tk-8.6.12-he1e0b03_0
tzdata            conda-forge/noarch::tzdata-2022g-h191b570_0
wheel             conda-forge/noarch::wheel-0.38.4-pyhd8ed1ab_0
xz                conda-forge/osx-arm64::xz-5.2.6-h57fd34a_0

```

Proceed ([y]/n)?

Type 'y' to proceed and Conda will create the environment for us. We can use `conda env list` to show all the created environments on our machine:

```

~/ conda env list
# conda environments:
#
base                /opt/homebrew/anaconda3
ortools             /opt/homebrew/anaconda3/envs/ortools

```

Note that we need to manually activate an environment in order to use it: `conda activate ortools`. On my machine, the activated environment `ortools` will appear in the beginning of my prompt.

```

~/ conda activate ortools
(ortools) ~/

```

## 2.4 Install Google OR-Tools

As of this writing, the latest version of Google OR-Tools is 9.5.2237, and we can install it in our newly created environment using the command `pip install ortools==9.5.2237`. We can use `conda list` to verify whether it is available in our environment.

```

(ortools) ~/ conda list
# packages in environment at /opt/homebrew/anaconda3/envs/ortools:
#
# Name                Version                Build    Channel
abs1-py               1.4.0                  pypi_0   pypi
bzip2                 1.0.8                  h3422bc3_4  conda-forge
ca-certificates       2022.12.7              h4653dfc_0  conda-forge
libffi                 3.4.2                  h3422bc3_5  conda-forge

```

libsqlite	3.40.0	h76d750c_0	conda-forge
libzlib	1.2.13	h03a7124_4	conda-forge
ncurses	6.3	h07bb92c_1	conda-forge
numpy	1.24.2	pypi_0	pypi
openssl	3.0.8	h03a7124_0	conda-forge
ortools	9.5.2237	pypi_0	pypi
pip	23.0.1	pyhd8ed1ab_0	conda-forge
protobuf	4.22.0	pypi_0	pypi
python	3.10.9	h3ba56d0_0_cpython	conda-forge
readline	8.1.2	h46ed386_0	conda-forge
setuptools	67.4.0	pyhd8ed1ab_0	conda-forge
tk	8.6.12	he1e0b03_0	conda-forge
tzdata	2022g	h191b570_0	conda-forge
wheel	0.38.4	pyhd8ed1ab_0	conda-forge
xz	5.2.6	h57fd34a_0	conda-forge

Now we have Python and Google OR-Tools ready, we can start our next journey.



## 3 Linear Programming

In this chapter, we first go through the modeling capabilities provided by Google OR-Tools to solve linear programming problems. Then we get our hands dirty by solving some linear programming problems.

### 3.1 Modeling Capabilities

There are three components in a mathematical model, namely, decision variables, constraints and objective, for which we will go over in the following sections.

#### 3.1.1 Solver

In Google OR-Tools, a `Solver` instance must be created first so that variables, constraints and objective can be added to it. The `Solver` class is defined in the `ortools.linear_solver.pywraplp` module and it requires a solver id to instantiate an object. In the code snippet below, the required module is imported first and a `solver` object is created with `GLOP`, Google's own optimization engine for solving linear programming problems. It is good practice to verify whether the desired solver is indeed created successfully or not.

```
from ortools.linear_solver import pywraplp

solver = pywraplp.Solver.CreateSolver("GLOP")

if solver:
    print("solver creation success!")
else:
    print("solver creation failure!")
```

```
solver creation success!
```

### 3.1.2 Decision Variables

The `Solver` class defines a number of ways to create decision variables:

1. `Var(lb, ub, integer, name)`
  2. `NumVar(lb, ub, name)`
  3. `IntVar(lb, ub, name)`
  4. `BoolVar(name)`
- Function `Var()`

The `Var()` method is the most flexible way to define variables, as it can be used to create numerical, integral and boolean variables. In the following code, a numerical variable named 'var1' is created with bound (0.0, 1.0). Note that the parameter `integer` is set to `False` in the call to function `Var()`.

```
var1 = solver.Var(lb=0, ub=1.0, integer=False, name="var1")
```

We could create an integer variable using the same function:

```
var2 = solver.Var(lb=0, ub=1.0, integer=True, name="var2")
```

- Function `NumVar()`

`var1` could be created alternatively using the specialized function `NumVar()`:

```
var1 = solver.NumVar(lb=0, ub=1.0, name="var1")
```

- Function `IntVar()`

Similarly, `var2` could be created alternatively using the specialized function `IntVar()`:

```
var2 = solver.IntVar(lb=0, ub=1.0, name="var2")
```

- Function `BoolVar()`

A boolean variable could be created using the `BoolVar()` function:

```
var3 = solver.BoolVar(name="var3")
```

### 3.1.3 Constraints

Constraints limit the solution space of an optimization problem, and there are two ways to define constraints in Google OR-Tools. In the first approach, we could use the `Add()` function to create a constraint and automatically add it to the model at the same time, as the below code snippet illustrates.

```
cons1 = solver.Add(constraint=var1 + var2 <= 1, name="cons1")  
  
type(cons1)
```

`ortools.linear_solver.pywraplp.Constraint`

Note that the `Add()` function returns an object of the `Constraint` class defined in the `pywraplp` module, as shown in the code output. It is a good practice to retain the reference of the newly created constraint, as we might want to query its information later on.

The second approach works in a slightly different way. It starts with an empty constraint, with potential lower bound and upper bounds provided, and add components of the constraint gradually. The code snippet below shows an example of adding a second constraint to the model. In this approach, we must retain the reference to the constraint, as it is needed to add decision variables to the constraint in following steps.

```
cons2 = solver.Constraint(-solver.infinity(), 10.0, "cons2")  
cons2.SetCoefficient(var1, 2)  
cons2.SetCoefficient(var2, 3)  
cons2.SetCoefficient(var3, 4)  
type(cons2)
```

`ortools.linear_solver.pywraplp.Constraint`

### 3.1.4 Objective

Similar to constraints, there are two ways to define the objective in Google OR-Tools. In the first approach, we directly add an objective to the model by using the `Maximize()` or `Minimize()` function. Below is an example:

```
solver.Minimize(var1 + var2 + var3)
```

Note that the function itself does not return a reference to the newly created objective function, but we could use a dedicated function to retrieve it:

```
obj = solver.Objective()
print(obj)
```

<ortools.linear\_solver.pywraplp.Objective; proxy of <Swig Object of type 'operations\_research'

In the second approach, we build the objective incrementally, just as in the second approach of creating constraints. Specifically, we start with an empty objective function, and gradually add components to it. In the end, we specify the optimization sense - whether we want to maximize or minimize the objective.

```
obj = solver.Objective()
obj.SetCoefficient(var1, 1.0)
obj.SetCoefficient(var2, 1.0)
obj.SetCoefficient(var3, 1.0)
obj.SetMinimization()
print(obj)
```

<ortools.linear\_solver.pywraplp.Objective; proxy of <Swig Object of type 'operations\_research'

### 3.1.5 Objective and Constraint Expressions

When we build constraints or objective functions, sometimes they comprise of complex expressions that we would like to build incrementally, possibly within loops. For example, we might have a mathematical expression of the form  $expr = 2x_1 + 3x_2 + 4x_3 + x_4$ , which could be part of the objective function or any constraints. In this case, we can either use the aforementioned `SetCoefficient()` function to add each element of the expression to the constraint or objective, or we could build an expression first and add it once in the end. The code snippet below shows an example.

```
infinity = solver.Infinity()
x1 = solver.NumVar(0, infinity, name="x1")
x2 = solver.NumVar(0, infinity, name="x2")
x3 = solver.NumVar(0, infinity, name="x3")
x4 = solver.NumVar(0, infinity, name="x4")

expr = []
expr.append(2 * x1)
expr.append(3 * x2)
expr.append(4 * x3)
```

```

expr.append(x4)

constr = solver.Add(solver.Sum(expr) <= 10)
print(constr)

solver.Minimize(solver.Sum(expr))

```

<ortools.linear\_solver.pywraplp.Constraint; proxy of <Swig Object of type 'operations\_research'

Of course, it is not obvious here that the repetitive calls to the `append()` method are any more convenient than the `SetCoefficient()` method. Let's say that we have a slightly more complex expression of the form  $\sum_{0 \leq i < 4} w_i \cdot x_i$ , now we could build the expression using a loop:

```

w = [2, 3, 4, 1]
x = [x1, x2, x3, x4]
expr = []
for i in range(4):
    expr.append(w[i] * x[i])

constr = solver.Add(solver.Sum(expr) <= 10)

```

### 3.1.6 Query the Model

After we build the model, we can query it using some helper functions. For example, to get the total number of constraints, we use the `NumVariables()` function. In a similar fashion, we can retrieve the total number of constraints with the `NumConstraints()` function.

```

num_vars = solver.NumVariables()
print(f"there are a total of {num_vars} variables in the model")

num_constr = solver.NumConstraints()
print(f"there are a total of {num_constr} constraints in the model")

```

```

there are a total of 9 variables in the model
there are a total of 4 constraints in the model

```

## 3.2 Applications

In this section, we use some examples to showcase the modeling capability of Google OR-Tools.

### 3.2.1 A Trivial Example

We now consider an simple linear programming problem with two decision variables  $x$  and  $y$ . The formal mathematical model is defined as below:

$$\max. \quad x + 2y \tag{3.1}$$

$$\text{s.t.} \quad x + y \leq 10 \tag{3.2}$$

$$x \geq 1 \tag{3.3}$$

$$y \geq 1 \tag{3.4}$$

Figure 3.1 shows the three defining constraints represented in blue lines and the feasible space depicted by the orange shaded area. The objective function is indicated by the red dashed lines. It can be seen from the figure that the point in green circle gives the maximal objective value of 19.

Let's now use Google OR-Tools to model and solve this problem. The code snippet below shows the complete program.

```
# import Google OR-Tools library
from ortools.linear_solver import pywraplp

# create a solver
solver = pywraplp.Solver.CreateSolver("GLOP")

# create decision variables
x = solver.NumVar(1.0, solver.Infinity(), "x")
y = solver.NumVar(1.0, solver.Infinity(), "y")

# create constraints
constr = solver.Add(x + y <= 10)

# create objective
solver.Maximize(x + 2 * y)

# solve the problem
```

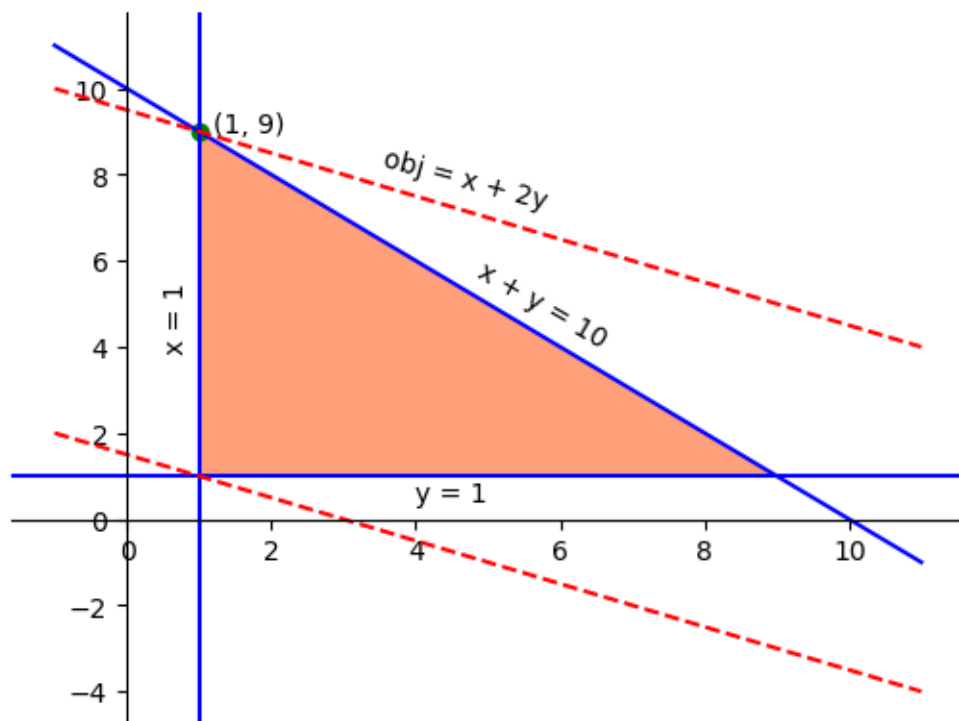


Figure 3.1: A simple LP example

```

status = solver.Solve()

if status == pywraplp.Solver.OPTIMAL or status == pywraplp.Solver.FEASIBLE:
    print(f"obj = {solver.Objective().Value()}")
    print(f"x = {x.solution_value()}, reduced cost = {x.reduced_cost()}")
    print(f"y = {y.solution_value()}, reduced cost = {y.reduced_cost()}")
    print(f"constr dual value = {constr.dual_value()}")

```

```

obj = 19.0
x = 1.0, reduced cost = -1.0
y = 9.0, reduced cost = 0.0
cons dual value = 2.0

```

We can see from the output that the optimal solution is  $x = 1.0$  and  $y = 9.0$ , and the optimal objective is 19.0. This can also be validated from Figure 3.1 that the optimal solution is exactly the green point that sits at the intersection of the three lines  $x = 1$ ,  $x + y = 10$  and  $x + 2y = 19$ .

Figure 3.1 also shows that the point  $(1, 1)$  should give us the minimal value of the objective function. To validate this, we can actually change the optimization sense of the objective function from maximization to minimization using the function `SetOptimizationDirection()`, as shown in the code below:

```

solver.Objective().SetOptimizationDirection(maximize=False)

solver.Solve()

print(f"obj = {solver.Objective().Value()}")
print(f"x = {x.solution_value()}, reduced cost = {x.reduced_cost()}")
print(f"y = {y.solution_value()}, reduced cost = {y.reduced_cost()}")
print(f"constr dual value = {cons.dual_value()}")

```

```

obj = 3.0
x = 1.0, reduced cost = 1.0
y = 1.0, reduced cost = 2.0
cons dual value = 0.0

```

### 3.2.2 A Transportation Problem

The transportation problem involves moving goods from its sources  $\mathcal{S}$  to destinations  $\mathcal{D}$ . Each source  $s \in \mathcal{S}$  has a total amount of goods  $p_s$  it could supply, and each destination  $s \in \mathcal{D}$  has a



certain amount of demands  $m_d$ . There is a transportation cost, denoted by  $c_{sd}$ , to move one unit of goods from a source to a destination. The problem is to find the best set of goods to move from each source to each destination such that all the destination demands are met with the lowest transportation costs.

To model this transportation problem, we define the decision variable  $x_{sd}$  to be the amount of goods moving from source  $s$  to destination  $d$ . Then we could state the problem mathematically as below.

$$\min. \quad \sum_{s \in \mathcal{S}} \sum_{d \in \mathcal{D}} c_{sd} x_{sd} \quad (3.5)$$

$$\text{s.t.} \quad \sum_{d \in \mathcal{D}} x_{sd} = p_s, \quad \forall s \in \mathcal{S} \quad (3.6)$$

$$\sum_{s \in \mathcal{S}} x_{sd} = m_d, \quad \forall d \in \mathcal{D} \quad (3.7)$$

$$x_{sd} \geq 0, \quad \forall s \in \mathcal{S}, d \in \mathcal{D} \quad (3.8)$$

Table 3.1: A transportation problem

	D1	D2	D3	D4	D5	Supply
S1	8	5	13	12	12	58
S2	8	7	18	6	5	55
S3	11	12	5	11	18	64
S4	19	13	5	10	18	71
Demand	44	28	36	52	88	248

```
from ortools.linear_solver import pywraplp

# gather data
num_sources = 4
num_destinations = 5
supplies = [58, 55, 64, 71]
demands = [44, 28, 36, 52, 88]
costs = [[8, 5, 13, 12, 12],
          [8, 7, 18, 6, 5],
          [11, 12, 5, 11, 18],
          [19, 13, 5, 10, 18]]

# create solver
```

```

solver = pywraplp.Solver.CreateSolver("GLOP")

# create decision variables
var_flow = []
for src_idx in range(num_sources):
    vars = [
        solver.NumVar(0, solver.Infinity(),
                       name=f"var_{src_idx}, {dest_idx}")
        for dest_idx in range(num_destinations)
    ]
    var_flow.append(vars)

# create constraints
for src_idx in range(num_sources):
    expr = [var_flow[src_idx][dest_idx]
            for dest_idx in range(num_destinations)]
    solver.Add(solver.Sum(expr) == supplies[src_idx])

for dest_idx in range(num_destinations):
    expr = [var_flow[src_idx][dest_idx]
            for src_idx in range(num_sources)]
    solver.Add(solver.Sum(expr) == demands[dest_idx])

# create objective function
obj_expr = []
for src_idx in range(num_sources):
    for dest_idx in range(num_destinations):
        obj_expr.append(var_flow[src_idx][dest_idx] * costs[src_idx][dest_idx])
solver.Minimize(solver.Sum(obj_expr))

status = solver.Solve()

if status == pywraplp.Solver.OPTIMAL:
    print(f"optimal obj = {solver.Objective().Value()}")

```

optimal obj = 2013.0

```

from ortools.linear_solver import pywraplp

# gather data
num_sources = 4

```

```

num_destinations = 5
supplies = [58, 55, 64, 71]
demands = [44, 28, 36, 52, 88]
costs = [[8, 5, 13, 12, 12],
          [8, 7, 18, 6, 5],
          [11, 12, 5, 11, 18],
          [19, 13, 5, 10, 18]]

# create solver
solver = pywraplp.Solver.CreateSolver("GLOP")

# create decision variables
var_flow = []
for src_idx in range(num_sources):
    vars = [
        solver.Var(0, solver.Infinity(),
                    integer=False,
                    name=f"var_{src_idx}, {dest_idx}")
        for dest_idx in range(num_destinations)
    ]
    var_flow.append(vars)

# create constraints
for src_idx in range(num_sources):
    constr = solver.Constraint(supplies[src_idx],
                               supplies[src_idx])
    for dest_idx in range(num_destinations):
        constr.SetCoefficient(var_flow[src_idx][dest_idx], 1.0)

for dest_idx in range(num_destinations):
    constr = solver.Constraint(demands[dest_idx],
                               demands[dest_idx])
    for src_idx in range(num_sources):
        constr.SetCoefficient(var_flow[src_idx][dest_idx], 1.0)

# create objective function
obj = solver.Objective()
for src_idx in range(num_sources):
    for dest_idx in range(num_destinations):
        obj.SetCoefficient(var_flow[src_idx][dest_idx], costs[src_idx][dest_idx])
obj.SetMinimization()

```

```
status = solver.Solve()

if status == pywraplp.Solver.OPTIMAL:
    print(f"optimal obj = {obj.Value()}")
```

optimal obj = 2013.0

### 3.2.3 A Blending Problem

## 4 Integer Programming

## 5 Column Generation

## 6 Summary

In summary, this book has no content whatsoever.

## References