Practical\_Python\_AI\_Projects\_c01

Introduction

**1.1 What Is This Book About?**

Artificial intelligence is a wide field covering diverse techniques,

objectives, and measures of success. One branch is concerned with finding

provably optimal solutions to some well-defined problems.

This book is an introduction to the art and science of implementing

mathematical models of optimization problems.

An optimization problem is almost any problem that is, or can be,

formulated as a question starting with “What is the best … ?” For instance,

• What is the best route to get from home to work?

• What is the best way to produce cars to maximize profit?

• What is the best way to carry groceries home: paper or plastic?

• Which is the best school for my kid?

• Which is the best fuel to use in rocket boosters?

• What is the best placement of transistors on a chip?

• What is the best NBA schedule?

These questions are rather vague and can be interpreted in a multitude

of ways. Consider the first: by “best” do we mean fastest, shortest, most

pleasant to ride, least bumpy, or least fuel-guzzling? Besides, the question

is incomplete. Are we walking, riding, driving, or snowboarding? Are we

alone or accompanied by a screaming toddler?

To help us formulate solutions to optimization problems, optimizers 1

have established a frame into which we mould the questions; it’s called a

model. The most crucial aspect of a model is that it has an objective and it

has constraints. Roughly, the objective is what we want and the constraints

are the obstacles in our way. If we can reformulate the question to clearly

identify both the objective and the constraints, we are closer to a model.

Let’s consider in more detail the “best route” problem but with an eye

to clarify objective and constraints. We could formulate it as

Given a map of the city, my home address, and the

address of the daycare of my two-year-old son, what

is the best route to take on my bike to bring him to

daycare as fast as possible?

The goal is to find among all the solutions that satisfy the requirements

(that is, paths following either streets or bike lanes, also known as the

constraints) one path that minimizes the time it takes to get there (the

objective).

Objectives are always quantities we want to maximize or minimize

(time, distance, money, surface area, etc.), although you will see examples

where we want to maximize something and minimize something else;

this is easily accommodated. Sometimes there are no objectives. We say

1I use the term “optimizers” to name the mathematicians, theoreticians, and

practitioners, who, since the nineteen-fifties, have worked in the fields of linear

programming (LP) and integer programming (IP). There are others who could

make valid claims to the moniker, chiefly among them researchers in constraint

programming, but my focus will be mostly in LP and IP models, hence my

restricted definition.

that the problem is one of feasibility (i.e. we are looking for any solution

satisfying the requirements). From the point of view of the modeler, the

difference is minimal. Especially since, in most practical cases, a feasibility

model is usually a first step. After noticing a solution, one usually wants

to optimize something and the model is modified to include an objective

function.

1.2 Features of the Text

As this text is an introduction, I do not expect the reader to be already

well versed in the art of modeling. I will start at the beginning, assuming

only that the reader understands the definition of a variable (both in

the mathematical sense and in the programming sense), an equation,

an inequality, and a function. I will also assume that the reader knows

some programming language, preferably Python, although knowing any

other imperative language is enough to be able to read the Python code

displayed in the text.

Note that the code in this book is an essential component. To get the

full value, the reader must, slowly and attentively, read the code. This book

is not a text of recipes described from a birds-eye view, using mathematical

notation, with all the nitty-gritty details “left as an exercise for the reader.”

This is implemented, functional, tested, optimization code that the reader

can use and moreover is encouraged to modify to fully understand. The

mathematics in the book has been reviewed by mathematicians, like any

mathematical paper. But the code has been subjected to a much more

stringent set of reviewers with names Intel, AMD, Motorola, and IBM.2

2My doctoral advisor used to say “There are error-free mathematical papers.” But

we only have found an existence proof of that theorem. I will not claim that the

code is error-free, but I am certain that it has fewer errors than any mathematical

paper I ever wrote.

The book is the fruit of decades of consulting and of years teaching

both an introductory modeling class (MOR242 Intro to Operation

Research Models) and a graduate class (APM568 Mathematical Modeling

in Industry) at Oakland University. I start at the undergraduate level and

proceed up to the graduate level in terms of modeling itself, without

delving much into the attendant theory.

• Every model is expressed in Python using Google

OR-Tools3 and can be executed as stated. In fact, the

code presented in the book is automatically extracted,

executed, and the output inserted into the text without

manual intervention; even the graphs are produced

automatically (thanks to Emacs4 and org-mode5).

• My intention is to help the reader become a

proficient modeler, not a theoretician. Therefore,

little of the fascinating mathematical theory related

to optimization is covered. It is nevertheless used

profitably to create simple yet efficient models.

• The associated web site provides all the code presented

in the book along with a random generator for many of

the problems and variations. The author uses this as a

personalized homework generator. It can also be used

as a self-guided learning tool.

https://github.com/sgkruk/Apress-AI

3https://github.com/google/or-tools

4The one and only editor: http://emacs.org

5<http://orgmode.org/>

1.2.1 Running the Models

There is danger in describing in too much detail installations instructions

because software tends to change more often than this text will change.

For instance, when I started with Google’s OR-Tools, it was hosted on the

Google Code repository; now it is on GitHub. Nevertheless, here are a few

pointers. All the code presented here has been tested with

• Python 3 (currently 3.7), although the models will work on Python 2

• OR-Tools 6.6

The page https://developers.google.com/optimization offers

installation instructions for most operating systems. The fastest and most

painless way is

pip install --upgrade ortools

Once OR-Tools are installed, the software of this text can be

downloaded most easily by cloning the GitHub repo at

git clone https://github.com/sgkruk/Apress-AI.git

where the reader will find a Makefile testing almost all the models detailed

in the text. The reader only has to issue a make to test that the installation

was completed successfully.

The code of each section of the book is separated into two parts: a

model proper, shown in the text, and a main driver to illustrate how to call

the model with some data. For instance, the chapter corresponding to the

set cover has a file named set\_cover.py with the model and a file named

test\_set\_cover.py which will create a random instance, run the model

on it, and display the result. Armed with these examples, the reader should

be able to modify to suit his needs. It is important to understand that the

mainline is in test\_set\_cover.py and that file needs to be executed.

1.2.2 A Note on Notation

Throughout the book, I will describe algebraic models. These models can

be represented in a number of ways. I will use two. I will sketch each model

using common mathematical notation typeset with TEX in math mode. I

will then express the complete, detailed model in executable Python code.

The reader should have no problem seeing the equivalence between the

formulations. Table 1-1 illustrates some of the equivalencies.

1.3 Getting Our Feet Wet: Amphibian Coexistence

The simplest problems are similar to those first encountered in high

school: the dreaded word problems. They are algebraic in nature; that

is, they can be formulated and sometimes solved using the simple tools

of elementary linear algebra. Let’s consider here one such problem

to illustrate the approach to modeling and define some fundamental

concepts.

A zoo biologist will place three species of amphibians (a toad, a

salamander, and a caecilian) in an aquarium where they will feed on

three different small preys: worms, crickets, and flies. Each day 1,500

worms, 3,000 crickets, and 5,500 flies will be placed in the aquarium.

Each amphibian consumes a certain number of preys per day. Table 1-2

summarizes the relevant data.

The biologist wants to know how many amphibians, up to 1,000 of

each species, can coexist in the aquarium assuming that food is the only

relevant constraint.

How to we model this problem? All optimization and feasibility

problems in this book are modeled using a three-step approach. We

will expand on this approach as we encounter problems on increasing

complexity, but the fundamental three steps remain the cornerstone of a

good model.

1. Identify the question to answer. This identification

should take the form of a precise sentence involving

either counting or valuating one or more objects. In

this case, how many amphibians each species can

coexist in the aquarium? Notice that “How many

amphibians?” would not be precise enough because

we are not interested in the total count, but rather

in the count of each species. Formulating a precise

question is often the hardest part.

Once we have this precise question, we assign a

variable to each of the objects to count. We will

use x0 , x1 , and x2 . These are traditionally known as

decision variables. The expression is a misnomer

in our first example but reflects the origins of

optimization problems in logistics where the

decision variables were indeed representative of

quantities under the control of the modeler and

mapped to planning decisions.

2. Identify all requirements and translate them

into constraints. The constraints, as you will see

throughout the book, can take on a multitude of

forms. In this simple problem, they are algebraic,

linear inequalities. It is often best to write down

each requirement in a precise sentence before

translating it into a constraint. For the coexistence

case, the requirements, in words, are

• All amphibians combined consume 1,500 worms.

• All amphibians combined consume 3,000 crickets.

• All amphibians combined consume 5,000 flies.

Note that a statement starting with “The amount

of …” may not be precise enough. In our simple case,

there are no specified units but there could be. For

instance, the amount consumed could be stated

in grams while the availability is in kilograms. This

happens often and is the cause of many a model

going awry.

Yet, even with our seemingly precise statements,

there is an ambiguity left to consider. It is one

of the main contributions of a good modeler

to highlight ambiguity and clarify problem

statements. Here, do we mean that the amphibians

will consume exactly the amounts stated, or that

they will consume at most the amounts stated? 6 We

will assume that “at most” is the proper form of the

requirement, both because it is more interesting

and, in a sense, subsumes the “equal” question.

We will then translate these requirements into

algebraic constraints based on our decision

variables.

Let’s consider worms. The toads eat two per day.

The salamanders and caecilians each eat one. Since

we decided on x0 toads, x1 salamanders, and x2

caecilians, the total number of worms consumed

will be bounded by the following inequality:

2 15000 1 2x x x+ + £ (1.1)

Had we decided that “equal to” was the proper

constraint, we would replace the inequality by an

equality.

6This seemingly trivial change from “exactly equal” to “at most” represents more

than 2,000 years of mathematical development in solution techniques. We have

known how to solve the “equal” form since ancient Babylonians (though it is

known today as “Gaussian elimination”) and we teach it in high school, but we

only discovered how to solve the “at most” form in the twentieth century.

Consider now crickets. Toads consume one per day

while salamanders consume three and caecilians

consume two. They will collectively consume x0 +

3x1 + 2x2 and we obtain the constraint

x x x0 1 23 2 3000+ + £ (1.2)

The constraint on flies is obtained similarly to

produce

x x x0 1 22 3 5000+ + £ (1.3)

3. Identify the objective to optimize. The objective

is, in the case of an optimization problem, what

we want to maximize (or minimize). In the case

of a feasibility problem, there is no objective, but

in practice, most feasibility problems are really

optimization problems that have been incompletely

formulated.

Since the problem is stated as “How many

amphibians of each species can coexist?”, a possible,

even likely, reading is that we want the maximum

number of amphibians. (The minimum number is

zero and is an example of the uninteresting trivial

solution.) In terms of our decision variables, we

want to maximize the sum and obtain

max x x x0 1 2+ + (1.4

At this point we have a model! Not the model, but a model: a simple,

clear, and precise algebraic model that has a solution, one that answers our

original question.

Since we are not mere theoreticians uninterested in practical

applications, our next step is to solve the model. As we will do for every

model in this book, we need to translate the mathematical expressions

above ((1.1)-(1.4)) into a form digestible by one of the many solvers

available.

Over the years, optimizers have developed a number of specialized

modeling languages and solvers. Here is a short list of the better-known

ones:

• Modeling languages

• AMPL (www.ampl.com)

• GAMS (www.gams.com)

• GMPL (http://en.wikibooks.org/wiki/GLPK/GMPL (MathProg))

• Minizinc (www.minizinc.org/)

• OPL (www-01.ibm.com/software/info/ilog/)

• ZIMPL (http://zimpl.zib.de/)

• Solvers

• CBC (www.coin-or.org/)

• CLP (www.coin-or.org/Clp/)

• CPLEX (www-01.ibm.com/software/info/ilog/)

• ECLiPSe (http://eclipseclp.org/)

• Gecode ([www.gecode.org/](http://www.gecode.org/)

• GLOP (https://developers.google.com/

optimization/lp/glop)

• GLPK (www.gnu.org/software/glpk/)

• Gurobi (www.gurobi.com/)

• SCIP (<http://scip.zib.de/>)

We should maintain a distinction between modeling languages,

formal constructions with specific vocabulary and grammars, and solvers,

software packages that can read in models expressed in certain languages

and write out the solutions, although in some cases this distinction is

blurry.

As a modeler, one creates a model (in language X) which is then fed

to a solver (solver Y). This can happen because solver Y knows how to

parse language X or because there is a translator between language X and

another language, say Z, which the solver understands. This, over the

years, has been the cause of much irritation (“What? You mean that I have

to rewrite my model to use your solver?”).

To make matters worse, these languages and solvers are not

equivalent. Each has its strengths and weaknesses, its areas of

specialization. After years of writing models in all the languages above and

then some, my preference today is to eschew specialized languages and to

use a general-purpose programming language, for instance Python, along

with a library interfacing with multiple solvers. Throughout this book I will

use Google’s Operations Research Tools (OR-Tools), a very well-structured

and easy-to-use library.

The OR-Tools library is comprehensive. It offers the best interface I

have ever used to access multiple linear and integer solvers (MPSolver). It

also has special-purpose code for network flow problems as well as a very

effective constraint programming library. In this text, I will display only a

very small fraction of this cornucopia of optimization tools.

One of the many advantages of using a general purpose language like

Python is that we can do the modeling part as well as the insertion of the

models into a larger application, maybe a web or a phone app. We can

also easily present the solutions in a clear format. We have all the power

of a complete language at our disposal. True, the specialized modeling

languages sometimes allow more concise model expression. But, in my

experience, they all, at one point or another, hit a wall, forcing the modeler

to write kludgy glue to connect a model to the rest of the application.

Moreover, writing OR-Tools models in Python can be such a joy.7 The

whole coexistence model is shown at Listing 1-1.

Listing 1-1. Amphibian Coexistence Model

1 from ortools.linear\_solver import pywraplp

2 def solve\_coexistence():

3 t = 'Amphibian coexistence'

4 s = pywraplp.Solver(t,pywraplp.Solver.GLOP\_LINEAR\_

PROGRAMMING)

5 x = [s.NumVar(0, 1000,'x[%i]' % i) for i in range(3)]

6 pop = s.NumVar(0,3000,'pop')

7 s.Add(2\*x[0] + x[1] + x[2] <= 1500)

8 s.Add(x[0] + 3\*x[1] + 2\*x[2] <= 3000)

9 s.Add(x[0] + 2\*x[1] + 3\*x[2] <= 4000)

10 s.Add(pop == x[0] + x[1] + x[2])

11 s.Maximize(pop)

12 s.Solve()

13 return pop.SolutionValue(),[e.SolutionValue() for e in x]

7Writing in Common Lisp would be even better. Alas, there is no Lisp binding for

OR-Tools yet.

Let’s deconstruct the code. Line 1 loads the Python wrapper of the

linear programming subset of OR-Tools. Every model we write will start

this way. Line 4 names and creates a linear programming solver (hereafter

named s) using Google’s own8 GLOP. The OR-Tools library has interfaces

to a number of solvers. Switching to a different solver, say GNU’s9 GLPK or

Coin-or 10 CLP is a simple matter or modifying this line.

On line 5, we create a one-dimensional array x of three decision

variables that can take on values between 0 and 1000. The lower bound

is a physical constraint since we cannot have a negative number of

amphibians. The upper bound is part of the problem statement as the

biologist will not put more than 1,000 of each species in the test tube. It

is possible to state ranges as any contiguous subsets of (−∞, +∞), but,

as a general rule of thumb, restricting the range as much as possible

during variable declaration tends to help solvers run efficiently. The third

parameter of the call to NumVar is used as the name to print if and when

this variable is displayed, for instance, in debugging a model. We will have

little use for this feature as we prefer to write bug-free models.

The constraints on lines 7 to 9 are direct translations of the

mathematical expressions (1.1)-(1.3). The order of the terms is irrelevant.

In contrast to some restrictive modeling languages, we could have written

line 7 as

1500>=x[0]+x[2]+x[1]

or

x[0]+x[1]+x[2]-1500<=0

or any other equivalent algebraic expression.

8https://developers.google.com/optimization/lp/glop

9www.gnu.org/software/glpk/

10https://projects.coin-or.org/Clp

At line, 6, we declare an auxiliary variable, pop. Though there is no

such distinction in the modeling language, this is not a decision variable

but rather a helpful device to model the problem. We use this auxiliary

on line 10 where we add an equation that does not constrain the model

in any way. It simply defines the auxiliary variable pop to be the sum of

our decision variables. This allows us to express the objective easily and,

possibly, to help display the solution.

The objective function is on line 11, a translation of (1.4). The function

choices are, unsurprisingly, either s.Maximize or s.Minimize with, for

parameter, a linear expression in terms of the variables declared previously.

We used

s.Maximize(pop)

We could have written

s.Maximize(x[0]+x[1]+x[2])

We then call on the solver at line 12 to do its job. This is where all the

computational work gets done, work that I will not describe. The interested

reader can search for “simplex method” and “interior-point methods” to

learn about the fascinating theory11 behind the solution methods of linear

optimization models. To understand the simplex method, one needs only

high school algebra. To understand interior-point methods requires a

somewhat more mathematical background.

For some models, solvers may complete their work in a fraction of a

second; for others, it may take hours. Moreover, not all solvers will have the

same runtime behavior. Model A may run faster than model B on solver

X while it may be exactly the reverse on solver Y. One more advantage of

using the OR-Tools library is that we can try out another solver by changing

one line.

11See, for example, Alexander Schrijver, Theory of Linear and Integer Programming

(Hoboken, NJ: Wiley, 1998).

We should, if this code were meant for production and the problem

nontrivial, check the return value to ensure that the solver found an

optimal solution. It may have aborted because of a model error, or because

it ran out of time or memory, or for some other reason. But for this simple

first example, we will forgo good engineering practice in the name of

simplicity of exposition.

We return, on line 13, both the optimal objective function value held

in variable pop and the optimal values of the decision variables (not all the

associated object attributes carried by those variables).

On more complex models, we may post-process the decision variables

to return something simpler and more meaningful to the caller. You will

see a good example of this when we solve the shortest path problem in

Chapter 4, Section 4.4. The general approach I encourage is to create

models that can be used without any knowledge of the internals of OR-Tools.

The modeler is responsible for the creation of the model, but once the

model is created and validated, it should leave the hands of its creator for

those of the domain expert who originally formulated the problem.

When the diligent reader executes Listing 1-2, she will observe a result

similar to Table 1-3.

Listing 1-2. How to Execute the Coexistence Model

1 from \_\_future\_\_ import print\_function

2 from coexistence import solve\_coexistence

3 pop,x=solve\_coexistence()

4 T=[['Specie', 'Count']]

5 for i in range(3):

6 T.append([['Toads','Salamanders','Caecilians'][i], x[i]])

7 T.append(['Total', pop])

8 for e in T:

9 print (e[0],e[1])

Notice that you can look at the solution of Table 1-3 and see that it does

indeed satisfy the constraints. By substituting the solution into (1.1)-(1.3),

we obtain

2(100.0) + 300.0 + 1000.0 = 1500 ≤ 1500,

100.0 + 3(300.0) + 2(1000.0) = 3000 ≤ 3000,

100.0 + 2(300.0) + 3(1000.0) = 3700 < 5000.

Notice that the first two inequalities are satisfied with equality. In the

jargon of optimization, such inequalities are tight or active. The last one is

said to be slack or inactive. In a certain sense, we could delete it from the

problem and nothing would change. (The reader can try this and other

modifications. The code is available in the additional material under the

name coexistence.py).

In summary, the steps to construct and run a model are the following

and are shown in Figure 1-1:

• Formulate the question precisely.

• Define the decision variables by identifying what is required to answer the question.

• Possibly define auxiliary variables to help simplify the

statements of constraints or of the objective function. They can

also help in the analysis and the presentation of the solution.

• Translate each constraint into an algebraic equality or

inequality involving directly the decision variables or

indirectly through the auxiliary variables.

• Construct the objective function as some quantity that

should be minimized or maximized.

• Run the model using an appropriate solver.

• Display the solution in an appropriate manner.

• Validate the results. Does the solution correctly satisfy

the constraints? Is the solution meaningful and

implementable? If so, declare that you are done; if not,

consider the necessary modifications to the model.

The rest of this book will construct models of increasing complexity,

illustrating and expanding the points above.