The Python Papers Monograph, Vol. 1 (2009)
Available online at http://ojs.pythonpapers.org/index.php/tppm

An Introduction to pulp for Python Programmers

Stuart Mitchell
Light Metals Research Centre
University of Auckland
Auckland, New Zealand
s.mitchell@auckland.ac.nz

Pulp-or (referred to as pulp for the rest of this paper) is a linear programming framework in Python. Pulp is licensed under a modified BSD license. The aim of pulp is to allow an Operations Research (OR) practitioner or programmer to express Linear Programming (LP), and Integer Programming (IP) models in python in a way similar to the conventional mathematical notation. Pulp will also solve these problems using a variety of free and non-free LP solvers. Pulp models an LP in a natural and pythonic manner. This paper is aimed at the python programmer who may wish to use pulp in their code. As such this paper contains a short introduction to LP models and their uses.

Keywords: Linear programming; Operations Research; pulp-or.

1 Introduction

Operations Research is known by the catch phrase "The science of better". From the website [1] we find this brief description

> In a nutshell, operations research (O.R.) is the discipline of applying advanced analytical methods to help make better decisions.

The particular area of operations research where pulp is useful is the development and modelling of Linear Programming (LP) and Integer Programming (IP) problems. Mathematically, an LP problem is to find a point in a n-dimensional linearly constrained region that maximises a given linear objective function. IP is an LP where the solution must contain discrete variables which take an integer value at the solution, a common special case of an integer variable is a binary variable which must be either 0 or 1 at the solution.

In general terms, an LP can describe a problem were decisions must be made (for example, the quantity of each ingredient in a can of cat food, detailed in section 2.1). These decisions are constrained by the properties of the problem that they model (the total weight of ingredients must be 100 grams and dietary requirements must be met). The quality of a solution is determined by some sort of cost (the total dollar cost to produce the can) and you seek to minimise or maximise the cost subject to the constraints.

2 A Brief introduction to Linear Programming

In this section we will introduce the reader to the structure of an LP problem and show the syntax used to formulate these models in pulp. The following is a example LP motivated by a real-world case-study. In general, problems of this sort are called 'diet' or 'blending' problems. An extensive set of case studies (including those below) can be found in [2].

2.1 The Whiskas Cat Food Problem (whiskas.py)

Figure 1: A Whiskas cat food label.

Whiskas cat food, shown above, is manufactured by Uncle Ben's. Uncle Ben's want to produce their cat food products as cheaply as possible while ensuring they meet the stated nutritional analysis requirements shown on the cans. Thus they want to vary the quantities of each ingredient used (the main ingredients being chicken, beef, mutton, rice, wheat and gel) while still meeting their nutritional standards.

INGREDIENTS: Selected meat derived from chicken, beef and mutton; rice; wheat bran; gel; all essential vitamins and minerals, including thiamine and taurine. No preservatives.

NUTRITIONAL ANALYSIS:	
Minimum% Crude Protein	8.0
Minimum% Crude Fat	6.0
Maximum% Crude Fibre	2.0
Max % Salt (Naturally Occurring)	0.4

Figure 2: Detail of ingredients and nutritional requirements.

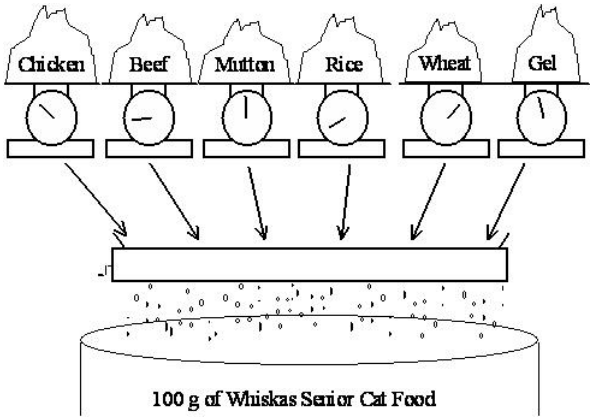

Figure 3: The quantity of each ingredient must be determined.

The costs of the chicken, beef, and mutton are \$0.013, \$0.008 and \$0.010 respectively, while the costs of the rice, wheat and gel are \$0.002, \$0.005 and \$0.001 respectively. (All costs are per gram.) For this exercise we will ignore the vitamin and mineral ingredients. (Any costs for these are likely to be very small anyway.) Each ingredient contributes to the total weight of protein, fat, fibre and salt in the final product which must be 100g (very convenient). The contributions (in grams) per gram of ingredient are given in the table below.

	Protein	Fat	Fibre	Salt
Chicken	0.100	0.080	0.001	0.002
Beef	0.200	0.100	0.005	0.005
Rice	0.000	0.010	0.100	0.008
Wheat bran	0.040	0.010	0.150	0.000

Table 1: Table of ingredient compositions.

A Linear Program consists of three main parts. These are the variable definitions, the objective function and the constraints.

2.1.1 Variable definition

In this problem the variables will be defined as the quantity (in grams) of each ingredient to include in the can. These variables will be continuous and will be able to take any non-negative value. Mathematically, we define set I as the set of ingredients then create an x variable indexed by I

$$\begin{aligned} I &= \{chicken, beef, mutton, rice, wheat, gel\} \\ x_i &\geq 0 \qquad i \in I. \end{aligned}$$

```
4  import pulp

6  #initialise the model
7  whiskas_model = pulp.LpProblem('The Whiskas Problem', pulp.LpMinimize)
8  # make a list of ingredients
9  ingredients = ['chicken', 'beef', 'mutton', 'rice', 'wheat', 'gel']
10 # create a dictionary of pulp variables with keys from ingredients
11 # the default lower bound is -inf
12 x = pulp.LpVariable.dict('x_%s', ingredients, lowBound =0)
```

2.1.2 Objective Function

The objective function is to minimise the cost of production for each can.

$$\min \sum_i c_i x_i.$$

Where:

- c_i is the cost per gram of ingredient i.

```
14 # cost data
15 cost = dict(zip(ingredients, [0.013, 0.008, 0.010, 0.002, 0.005, 0.001]))
16 # create the objective
17 whiskas_model += sum( [cost[i] * x[i] for i in ingredients])
```

2.1.3 Constraints

Constraints serve to control the total mass of the ingredients and to model the protein, fat, salt and fibre requirements.

$$\sum_i x_i = 100 \qquad \text{can mass}$$

$$\sum_i p_i x_i \geq 8.0 \qquad \text{protein}$$

$$\sum_i f_i x_i \geq 6.0 \qquad \text{fat}$$

$$\sum_i b_i x_i \leq 2.0 \qquad \text{fibre}$$

$$\sum_i s_i x_i \leq 0.4 \qquad \text{salt.}$$

Where:

- p_i is the protein per gram of ingredient i;
- f_i is the fat per gram of ingredient i;
- b_i is the fibre per gram of ingredient i;
- s_i is the salt per gram of ingredient i.

```
19 # ingredient parameters
20 protein = dict(zip(ingredients, [0.100, 0.200, 0.150, 0.000, 0.040, 0.000]))
21 fat = dict(zip(ingredients, [0.080, 0.100, 0.110, 0.010, 0.010, 0.000]))
22 fibre = dict(zip(ingredients, [0.001, 0.005, 0.003, 0.100, 0.150, 0.000]))
23 salt = dict(zip(ingredients, [0.002, 0.005, 0.007, 0.002, 0.008, 0.000]))

25 #note these are constraints and not an objective as there is a equality/inequality
26 whiskas_model += sum([protein[i]*x[i] for i in ingredients]) >= 8.0
27 whiskas_model += sum([fat[i]*x[i] for i in ingredients]) >= 6.0
28 whiskas_model += sum([fibre[i]*x[i] for i in ingredients]) <= 2.0
29 whiskas_model += sum([salt[i]*x[i] for i in ingredients]) <= 0.4
```

The pulp model must now be solved with some third party optimisation software. Pulp currently supports solving with coin-or [3], glpk [4], CPLEX [5] and Gurobi [6]. The pulp download includes compiled versions of the coin-or solver for ubuntu and windows computers.

```
31 #problem is then solved with the default solver
32 whiskas_model.solve()

34 #print the result
35 for ingredient in ingredients:
36     print 'The mass of %s is %s grams per can'%(ingredient,
37                                                 x[ingredient].value())
```

3 Other Example LP models

Modelling the diet problem is not the only application of linear programming. Other examples include:

- the transportation problem,
- the set partitioning problem,
- the assignment problem,

- the knapsack problem.

In this section we will give short examples of the first two problems modelled in pulp together with a brief description.

3.1 The Transportation Problem (beerdistribution.py)

A transportation problem involves that shipment of items from a set of sources to a set of sinks (the sources and sinks must be disjoint sets), the problem seeks to minimise the cost of shipping. In this case-study crates of beer must be shipped from two breweries to five bars.

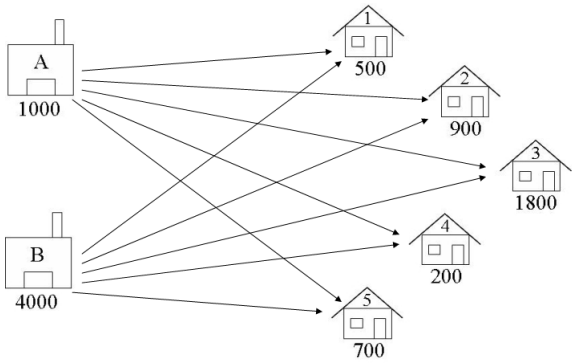

Figure 4: The brewery and bar capacities and possible routes.

The problem is created.

```
37 # Creates the 'prob' variable to contain the problem data
38 prob = pulp.LpProblem("Beer Distribution Problem", pulp.LpMinimize)
```

A list of possible routes for transportation is created:

```
40 # Creates a list of tuples containing all the possible routes for transport
41 routes = [(w,b) for w in warehouses for b in bars]
```

The variables created determine the amount shipped on each route. The variables are defined as `pulp.LpInteger` therefore solutions must not ship fractional numbers of crates.

```
43 # A dictionary called x is created to contain quantity shipped on the routes
44 x = pulp.LpVariable.dicts("route", (warehouses, bars),
45                          lowBound = 0
46                          cat = pulp.LpInteger)
```

The objective is to minimise the total shipping cost of the solution.

```
48 # The objective function is added to 'prob' first
49 prob += sum([x[w][b]*costs[w][b] for (w,b) in routes]), \
50             "Sum_of_Transporting_Costs"
```

These constraints ensure that amount shipped from each brewery is less than the supply available. The names given to the constraints will be preserved when an '.lp' file is created.

```
52 # Supply maximum constraints are added to prob for each supply node (warehouse)
53 for w in warehouses:
54     prob += sum([x[w][b] for b in bars]) <= supply[w], \
55             "Sum_of_Products_out_of_Warehouse_%s"%w
```

These constraints ensure that amount shipped to each bar is greater than the demand of the bar. These could also be equality constraints, depending on how the problem is modelled.

```
57 # Demand minimum constraints are added to prob for each demand node (bar)
58 for b in bars:
59     prob += sum([x[w][b] for w in warehouses]) >= demand[b], \
60     "Sum_of_Products_into_Bar%s"%b
```

3.2 The Set Partitioning Problem (wedding.py)

A set partitioning problem determines how the items in one set (S) can be partitioned into smaller subsets. All items in S must be contained in one and only one partition. Related problems are:

- set packing - all items must be contained in zero or one partitions;
- set covering - all items must be contained in at least one partition.

In this case study a wedding planner must determine guest seating allocations for a wedding. To model this problem the tables are modelled as the partitions and the guests invited to the wedding are modelled as the elements of S. The wedding planner wishes to maximise the total happiness of all of the tables.

A set partitioning problem may be modelled by explicitly enumerating each possible subset. Though this approach does become intractable for large numbers of items (without using column generation [7]) it does have the advantage that the objective function co-efficients for the partitions can be non-linear expressions (like happiness) and still allow this problem to be solved using Linear Programming.

First we use `pulp.allcombinations` to generate a list of all possible table seatings.

```
20 #create list of all possible tables
21 possible_tables = [tuple(c) for c in pulp.allcombinations(guests,
22                                         max_table_size)]
```

Then we create a binary variable that will be 1 if the table will be in the solution, or zero otherwise.

```
24 #create a binary variable to state that a table setting is used
25 x = pulp.LpVariable.dicts('table', possible_tables,
26                            lowBound = 0,
27                            upBound = 1,
28                            cat = pulp.LpInteger)
```

We create the `LpProblem` and then make the objective function. Note that happiness function used in this script would be difficult to model in any other way.

```
30 seating_model = pulp.LpProblem("Wedding Seating Model", pulp.LpMinimize)

32 seating_model += sum([happiness(table) * x[table] for table in possible_tables])
```

We specify the total number of tables allowed in the solution.

```
34 #specify the maximum number of tables
35 seating_model += sum([x[table] for table in possible_tables]) <= max_tables, \
```

This set of constraints defines the set partitioning problem by guaranteeing that a guest is allocated to exactly one table.

```
38 #A guest must seated at one and only one table
39 for guest in guests:
40     seating_model += sum([x[table] for table in possible_tables
41                                 if guest in table]) == 1, "Must_seat_%s"%guest
```

4 Why pulp

Modelling problems using LP and solving them with pulp can be a very useful approach for the python programmer. This approach allows the programmer to focus on the modelling rather than the algorithms that find the solution. The third party LP solvers that pulp interfaces with are all mature pieces of software that can be trusted to produce the correct solution to the model created in pulp. The difference between these solvers, for a user of pulp, lies mainly in the trade off between cost of the solver and its speed to find a solution.

LP and IP are well researched areas of mathematics, therefore once a problem is stated as an LP model it is possible to guarantee some properties of the solutions given. For instance, if the solver delivers an optimal solution it will be impossible for someone to create a solution that is better (as measured by the objective function).

Compared to other LP modelling tools including AMPL [8], GAMS [9], mathprog [4], flopc++ [10], and pyomo [11] and poams; pulp offers a number of advantages. These include its:

- non-restrictive licensing;
- ease of installation;
- clear syntax;
- interoperability with an number of solvers;
- extensive documentation.

4.1 Licensing

The licensing of pulp under a permissive open-source license allows pulp to be used as a medium for teaching operations research as students can download and use pulp for free. The license allows OR practitioners to integrate pulp in commercial applications for their clients without disclosing how the problem was solved. The license also allows advanced users to modify and improve pulp for their own purposes.

4.2 Installation

Pulp is very easy to install. The provision of a setup.py file and registration on pipy, allows the user to use

```
$easy_install pulp-or
```

to download and install pulp on their system. For windows and ubuntu users this binary package also includes the coin-or [3] solver so pulp will be immediately functional. For users on other platforms a compatible solver must be installed for a pulp model to be solved.

4.3 Syntax

Ideally an LP modelling framework will allow a one-to-one translation of symbols from mathematical notation. If the syntax follows the formulation the programmer can ensure that the model written is the model required. The OR practitioner can also quickly read and understand the a model as written pulp, without having much python background. The standard mathematical notation is very concise and therefore code that mimics it will also be clear and concise.

The use of a 'Pythonic' construction of the pulp framework allows the programmer to use it easily in a variety of ways. Pulp can be directly imported into the local name-space (against python style) to allow simple LP models to be written by non-programmers, in fact most of the case studies on the wiki [2] use this style. Pulp also does not force the programmer to use any particular way to store parameter data. Parameter data can be stored as lists, dictionaries or custom classes.

4.4 Interoperability

The ability for pulp to call a number of free and non-free solvers is important as the OR practitioner often wishes to compare the solution of a problem with a variety of solvers. The user may wish to develop simple LP models in a free solver and then provide a solution to a client using a commercial solver. The use of non-free solvers may dramatically reduce the solution time of the model. Pulp allows the free interchange of solvers without much change in the program, only a parameter for the `LpProblem.solve` function is changed.

4.5 Documentation

Pulp is supplied with a user guide with extensive examples (in fact it was converted from a course teaching LP modelling). This user guide and the permissive license were intended to make pulp useful to the largest possible audience.

5 Conclusion

In conclusion pulp nicely bridges the gap between the OR practitioner and the python programmer. This allows the OR practitioner to use python to quickly develop and solve LP and IP models while having access to all of the tools available in the python standard library. The python programmer can now embed LP models in complex programs written in python. So next time you find yourself hacking up an algorithm to solve some problem in your code please consider if it would be appropriate to model this problem as an LP or IP instead. If so download and use pulp.

Appendix - Example Code in full

whiskas.py

```
"""
Example problem file that solves the whiskas blending problem
"""
import pulp

#initialise the model
whiskas_model = pulp.LpProblem('The Whiskas Problem', pulp.LpMinimize)
# make a list of ingredients
ingredients = ['chicken', 'beef', 'mutton', 'rice', 'wheat', 'gel']
# create a dictionary of pulp variables with keys from ingredients
# the default lower bound is -inf
x = pulp.LpVariable.dict('x_%s', ingredients, lowBound =0)

# cost data
cost = dict(zip(ingredients, [0.013, 0.008, 0.010, 0.002, 0.005, 0.001]))
# create the objective
whiskas_model += sum( [cost[i] * x[i] for i in ingredients])

# ingredient parameters
protein = dict(zip(ingredients, [0.100, 0.200, 0.150, 0.000, 0.040, 0.000]))
fat = dict(zip(ingredients, [0.080, 0.100, 0.110, 0.010, 0.010, 0.000]))
fibre = dict(zip(ingredients, [0.001, 0.005, 0.003, 0.100, 0.150, 0.000]))
salt = dict(zip(ingredients, [0.002, 0.005, 0.007, 0.002, 0.008, 0.000]))

#note these are constraints and not an objective as there is a equality/inequality
whiskas_model += sum([protein[i]*x[i] for i in ingredients]) >= 8.0
whiskas_model += sum([fat[i]*x[i] for i in ingredients]) >= 6.0
whiskas_model += sum([fibre[i]*x[i] for i in ingredients]) <= 2.0
whiskas_model += sum([salt[i]*x[i] for i in ingredients]) <= 0.4
```

```
#problem is then solved with the default solver
whiskas_model.solve()

#print the result
for ingredient in ingredients:
    print 'The mass of %s is %s grams per can'%(ingredient,
                                                x[ingredient].value())
```

beerdistribution.py

```
"""
The Beer Distribution Problem for the PuLP Modeller

Authors: Antony Phillips, Dr Stuart Mitchell  2007
"""

# Import PuLP modeler functions
import pulp

# Creates a list of all the supply nodes
warehouses = ["A", "B"]

# Creates a dictionary for the number of units of supply for each supply node
supply = {"A": 1000,
          "B": 4000}

# Creates a list of all demand nodes
bars = ["1", "2", "3", "4", "5"]

# Creates a dictionary for the number of units of demand for each demand node
demand = {"1":500,
          "2":900,
          "3":1800,
          "4":200,
          "5":700,}

# Creates a list of costs of each transportation path
costs = [   #Bars
         #1 2 3 4 5
         [2,4,5,2,1],#A   Warehouses
         [3,1,3,2,3] #B
         ]

# The cost data is made into a dictionary
costs = pulp.makeDict([warehouses, bars], costs,0)

# Creates the 'prob' variable to contain the problem data
prob = pulp.LpProblem("Beer Distribution Problem", pulp.LpMinimize)

# Creates a list of tuples containing all the possible routes for transport
routes = [(w,b) for w in warehouses for b in bars]

# A dictionary called x is created to contain quantity shipped on the routes
x = pulp.LpVariable.dicts("route", (warehouses, bars),
                          lowBound = 0
                          cat = pulp.LpInteger)

# The objective function is added to 'prob' first
prob += sum([x[w][b]*costs[w][b] for (w,b) in routes]), \
            "Sum_of_Transporting_Costs"

# Supply maximum constraints are added to prob for each supply node (warehouse)
for w in warehouses:
    prob += sum([x[w][b] for b in bars]) <= supply[w], \
```

```
            "Sum_of_Products_out_of_Warehouse_%s"%w

# Demand minimum constraints are added to prob for each demand node (bar)
for b in bars:
    prob += sum([x[w][b] for w in warehouses]) >= demand[b], \
    "Sum_of_Products_into_Bar%s"%b

# The problem data is written to an .lp file
prob.writeLP("BeerDistributionProblem.lp")

# The problem is solved using PuLP's choice of Solver
prob.solve()

# The status of the solution is printed to the screen
print "Status:", pulp.LpStatus[prob.status]

# Each of the variables is printed with it's resolved optimum value
for v in prob.variables():
    print v.name, "=", v.varValue

# The optimised objective function value is printed to the screen
print "Total Cost of Transportation = ", prob.objective.value()
```

wedding.py

```
"""
A set partitioning model of a wedding seating problem

Authors: Stuart Mitchell 2009
"""

import pulp

max_tables = 5
max_table_size = 4
guests = 'A B C D E F G I J K L M N O P Q R'.split()

def happiness(table):
    """
    Find the happiness of the table
    - by calculating the maximum distance between the letters
    """
    return abs(ord(table[0]) - ord(table[-1]))

#create list of all possible tables
possible_tables = [tuple(c) for c in pulp.allcombinations(guests,
                                        max_table_size)]

#create a binary variable to state that a table setting is used
x = pulp.LpVariable.dicts('table', possible_tables,
                          lowBound = 0,
                          upBound = 1,
                          cat = pulp.LpInteger)

seating_model = pulp.LpProblem("Wedding Seating Model", pulp.LpMinimize)

seating_model += sum([happiness(table) * x[table] for table in possible_tables])

#specify the maximum number of tables
seating_model += sum([x[table] for table in possible_tables]) <= max_tables, \
                          "Maximum_number_of_tables"

#A guest must seated at one and only one table
for guest in guests:
    seating_model += sum([x[table] for table in possible_tables
```

```
41                                  if guest in table]) == 1, "Must_seat_%s"%guest

43  seating_model.solve()

45  print "The choosen tables are:"
46  for table in possible_tables:
47      if x[table].value() == 1.0:
48          print table
```

References

[1] Operations research: The science of better. [Online] http://www.scienceofbetter.org/

[2] S. A. Mitchell and A. Phillips. Pulp-or wiki. [Online] http://pulp-or.googlecode.com

[3] R. Lougee-Heimer, "The common optimization interface for operations research," *IBM Journal of Research and Development*, vol. 47, no. 1, pp. 57–66, January 2003.

[4] Gnu linear programming kit. [Online] http://www.gnu.org/software/glpk/glpk.html

[5] Cplex website. [Online] http://www.ilog.com/products/cplex/

[6] Gurobi website. [Online] http://www.gurobi.com/

[7] M. E. Lübbecke and J. Desrosiers, "Selected topics in column generation," *OPERATIONS RESEARCH*, vol. 53, no. 6, pp. 1007–1023, November- December 2005.

[8] R. Fourer, D. M. Gay, and B. W. Kernighan, "Ampl: A mathematical programming language," *Management Science*, vol. 36, pp. 519–554, 1990.

[9] A. Brooke, D. Kendrick, and A. Meeraus, "Gams: A user's guide," 1992. [Online] citeseer.ist.psu.edu/brooke92gams.html

[10] T. H. Hultberg, "Flopc++ an algebraic modeling language embedded in c++," in *Operations Research Proceedings 2006*, ser. Operations Research Proceedings, K.-H. Waldmann and U. M. Stocker, Eds., no. 6. Springer Berlin Heidelberg, 2006, pp. 187–190.

[11] W. Hart, "Python optimization modeling objects (pyomo)," in *Proc INFORMS Computing Society Conference*, 2009. [Online] http://www.optimization-online.org/DB_HTML/2008/09/2095.html