PuLP: A Linear Programming Toolkit for Python

Stuart Mitchell*,
Stuart Mitchell Consulting,
Michael O'Sullivan,
Iain Dunning
Department of Engineering Science, The University of Auckland, Auckland, New Zealand

September 5, 2011

Abstract

This paper introduces the PuLP library, an open source package that allows mathematical programs to be described in the Python computer programming language. PuLP is a high-level modelling library that leverages the power of the Python language and allows the user to create programs using expressions that are natural to the Python language, avoiding special syntax and keywords wherever possible.

1 Introduction

PuLP is a library for the Python scripting language that enables users to describe mathematical programs. Python is a well-established and supported high level programming language with an emphasis on rapid development, clarity of code and syntax, and a simple object model. PuLP works entirely within the syntax and natural idioms of the Python language by providing Python objects that represent optimization problems and decision variables, and allowing constraints to be expressed in a way that is very similar to the original mathematical expression. To keep the syntax as simple and intuitive as possible, PuLP has focused on supporting linear and mixed-integer models. PuLP can easily be deployed on any system that has a Python interpreter, as it has no dependencies on any other software packages. It supports a wide range of both commercial and open-source solvers, and can be easily extended to support additional solvers. Finally, it is

* Corresponding author.
E-mail address: stu@stuartmitchell.com (S. Mitchell)

available under a permissive open-source license that encourages and facilitates the use of PuLP inside other projects that need linear optimisation capabilities.

2 Design and Features of PuLP

Several factors were considered in the design of PuLP and in the selection of Python as the language to use.

2.1 Free, Open Source, Portable

It was desirable that PuLP be usable anywhere, whether it was as a straightforward modelling and experimentation tool, or as part of a larger industrial application. This required that PuLP be affordable, easily licensed, and adaptable to different hardware and software environments. Python itself more than meets these requirements: it has a permissive open-source license and has implementations available at no cost for a wide variety of platforms, both conventional and exotic. PuLP builds on these strengths by also being free and licensed under the very permissive MIT License[11]. It is written in pure Python code, creating no new dependencies that may inhibit distribution or implementation.

2.2 Interfacing with Solvers

Many mixed-integer linear programming (MILP) solvers are available, both commerical (e.g. CPLEX[1], Gurobi[2]) and open-source (e.g. CBC[6]). PuLP takes a modular approach to solvers by handling the conversion of Python-PuLP expressions into "raw" numbers (i.e. sparse matrix and vector representations of the model) internally, and then exposing this data to a solver interface class. As the interface to many solvers is similar, or can be handled by writing the model to the standard L̇P or Ṁ PS file formats, base generic solver classes are included with PuLP in addition to specific interfaces to the currently popular solvers. These generic solver classes can then be extended by users or the developers of new solvers with minimal effort.

2.3 Syntax, Simplicity, Style

A formalised style of writing Python code[13], referred to as "Pythonic" code, has developed over the past 20 years of Python development. This style is well established and focuses on readability and maintainability of code over "clever" manipulations that are more terse but are considered harmful to the maintainability of software projects. PuLP builds on this style by using the natural idioms of Python programming wherever possible. It does this by having very few special functions or "keywords", to avoid polluting the namespace of the language. Instead it provides two main objects (for a problem and for a variable) and then uses Python's control structures and arithmetic operators (see section 3). In contrast to Pyomo (section 4), another Python-based modelling language, PuLP does

not allow users to create purely abstract models. While in a theoretical sense this restricts the user, we believe that abstract model construction is not needed for a large number of approaches in dynamic, flexible modern languages like Python. These languages do not distinguish between data or functions until the code is run, allowing users to still construct complex models in a pseudo-abstract fashion. This is demonstrated in the Wedding Planner example (§3.3), where a Python function is included in the objective function.

2.4 Standard Library and Packages

One of the strengths of the Python language is the extensive standard library that is available to every program that uses the Python interpreter. The standard library [4] includes hundreds of modules that allow the programmer to, for example:

- read data files and databases;
- fetch information from the Internet;
- manipulate numbers and dates;
- create graphical user interfaces.

In addition to the Python standard library there are over 10,000 third-party packages on The Python Package Index[3]. Many packages related to operations research can be found here, including PuLP [10], Coopr [7], Dippy [12], and Yaposib [5], which are all projects in the COIN-OR repository.

3 Examples: PuLP in Action

In this section we demonstrate how PuLP can be used to model two different problems. The first, the Capacitated Facility Location problem, demonstrates enough of PuLP to allow any MILP to be described. The second, the Wedding Planner problem, extends this by showing some more advanced features and expressions that describe the model more concisely.

3.1 Python Syntax

To aid in the understanding of the examples, it is helpful to introduce some of the relevant language features of Python.

- **Whitespace**: Python uses indentation (with spaces or tabs) to indicate subsections of code.
- **Variable declaration**: Variables do have specific types (e.g. string, number, object), but it is not necessary to pre-declare the variable types - the Python interpreter will determine the type from the first use of the variable.

- **Dictionaries and Lists**: These are two common data structures in Python. Lists are a simple container of items, much like arrays in many languages. They can change size at any time, can contain items of different types, and are one dimensional. Dictionaries are another storage structure, where each item is associated with a "key". This is sometimes called a map or associative array in other languages. The key maybe be almost anything, as long as it is unique. For a more detailed look at the strengths and capabilities of these two structures, consult the Python documentation [14].

```
myList = ['apple', 'orange', 'banana']
myDict = {'apple':'red, 'orange':'orange', 'banana':'yellow}
print myList[0]       % Displays "apple"
print myDict['apple'] % Displays "red"
```

- **List comprehension**: These are "functional programming" devices used in Python to dynamically generate lists, and are very useful for generating linear expressions like finding the sum of a set. Many examples are provided in the code below, but the general concept is to create a new list in place by filtering, manipulating or combininb other lists. For example, a list comprehension that provides the even numbers between 1 and 9 is implemented with the Python code:

```
even = [i for i in [1,2,3,4,5,6,7,8,9] if i%2 == 0]
```

  where we have use the modulo division operator % as our filter.

3.2 Capacitated Facility Location

The Capacitated Facility Location problem determines where, amongst m locations, to place facilities and assigns the production of n products to these facilities in a way that (in this variant) minimises the wasted capacity of facilities. Each product $j = 1, \ldots, n$ has a production requirement r_j and each facility has the same capacity C. Extensions of this problem arise often in problems including network design and rostering.

The MILP formulation of this problem is as follows:

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{m} w_i \\
\text{s.t.} \quad & \sum_{i=1}^{m} x_{ij} = 1, j = 1, \ldots, n \qquad \text{(each product produced)} \\
& \sum_{j=1}^{n} r_j x_{ij} + w_i = C y_i, i = 1, \ldots, m \quad \text{(capacity at location } i\text{)} \\
& x_{ij} \in \{0,1\}, w_i \geq 0, y_i \in \{0,1\}, i = 1, \ldots, m, j = 1, \ldots, n
\end{aligned}
$$

where the decision variables are:

$$x_{ij} = \begin{cases} 1 & \text{if product } j \text{ is produced at location } i \\ 0 & \text{otherwise} \end{cases}$$

$$y_i = \begin{cases} 1 & \text{if a facility is located at location } i \\ 0 & \text{otherwise} \end{cases}$$

$$w_i = \text{“wasted” capacity at location } i$$

To start using PuLP, we need to tell Python to use the library:

```
1 from coinor.pulp import *
```

Next, we can prepare the data needed for a specific instance of this problem. Because PuLP creates concrete models, unlike other modelling languages such as AMPL and Pyomo, we need to do this before defining the model. We will use fixed values defined in the script in this simple example, but the data could easily be stored in a file, on the Internet, or even read from a sensor or other device.

```
3  REQUIRE = {
4      1 : 7,
5      2 : 5,
6      3 : 3,
7      4 : 2,
8      5 : 2
9  }
10 PRODUCTS  = [1, 2, 3, 4, 5]
11 LOCATIONS = [1, 2, 3, 4, 5]
12 CAPACITY = 8
```

An object is then created that will represent our specific instance of the model. This object, an `LpProblem`, has some optional parameters. Here we set the problem name and give the objective sense, minimise.

```
14 prob = LpProblem("FacilityLocation", LpMinimize)
```

All our decision variables in the mathematical program above are indexed, and this can be expressed naturally and concisely with the `LpVariable` object's `dict` functionality. The y_i/`use_vars` and w_i/`waste_vars` are both similar: we provide the name of this variable, the list of indices (`LOCATIONS`, which we defined previously), the lower and upper bounds, and the variable types (if a variable type is not provided, `LpContinuous` is assumed).

```
16 use_vars = LpVariable.dicts("UseLocation", LOCATIONS, 0, 1, LpBinary)
17 waste_vars  = LpVariable.dicts("Waste", LOCATIONS, 0, CAPACITY)
```

The x_{ij}/`assign_vars` are defined in the same way, except for the slightly more complicated definition of the indices. Here we use Python's powerful “list comprehensions” to dynamically create our list of indices, which is a one-to-one match with the indicies used in the formulation. In English, the sub-statement says “create an entry in the list of the form (i,j) for every combination of location and product”.

```
assign_vars = LpVariable.dicts("AtLocation",
              [(i, j) for i in LOCATIONS
                      for j in PRODUCTS],
              0, 1, LpBinary)
```

Now we have a problem and some variables, we can define the objective and constraints. The model is built up by literally adding expressions. In Python and many other programming languages, the operator in `a +=b` is shorthand for "assign the value of $a+b$ to a". Here `prob+=expression` adds the `expression` to the `LpProblem` as an objective function or constraint. Objectives and constraints can be expressed exactly as they are in the mathematical description, but we can employ some of the features to make the model more concise. The objective is simply the sum of the waste variables. PuLP distinguishes the objective from the constraints by observing that there is no comparison operator used in the expression.

```
prob += lpSum(waste_vars[i] for i in LOCATIONS)
```

The constraints are added in a similar way to the objective. Note the use of Python's `for` loop and its direct parallel in the mathematical formulation.

```
for j in PRODUCTS:
    prob += lpSum(assign_vars[(i, j)] for i in LOCATIONS) == 1

for i in LOCATIONS:
    prob += lpSum(assign_vars[(i, j)] * REQUIRE[j] for j in PRODUCTS) \
                        + waste_vars[i] == CAPACITY * use_vars[i]
```

Finally we use the default solver, CBC, to solve the problem. The solution is displayed to the screen by using the `varValue` property of the variables, taking into account any floating-point imprecision in the answers.

```
prob.solve()

TOL = 0.00001
for i in LOCATIONS:
    if use_vars[i].varValue > TOL:
        print "Location ", i, " produces ", \
              [j for j in PRODUCTS
               if assign_vars[(i, j)].varValue > TOL]
```

For the data set used in this example, the optimal solution is as follows:

```
Location 2 produces [3, 5]
Location 3 produces [2, 4]
Location 4 produces [1]
```

3.3 Wedding Planner Problem - a Set Partioning Problem

The Wedding Planner problem is as follows: given a list of wedding attendees, a wedding planner must come up with a seating plan to minimise the unhappiness of all of the guests. The unhappiness of guest is defined as their maximum

unhappiness at being seated with each of the other guests at their table, i.e., it is a pairwise function. The unhappiness of a table is the maximum unhappiness of all the guests at the table. All guests must be seated and there is a limited number of seats at each table.

This is a set partitioning problem, as the set of guests G must be partitioned into multiple subsets, with each member of a given subset seated at the same table. The cardinality of the subsets is determined by the number of seats at a table and the unhappiness of a table can be determined by the contents of a subset. The MILP formulation is:

$$\begin{aligned} \min \quad & \sum_{t \in T} h_t x_t && \text{(total unhappiness of the tables)} \\ & \sum_{t \in T} x_j && \leq M_T \\ & \sum_{t \in T} a_{g,t} x_t && = 1, g \in G \end{aligned}$$

where:

$$x_t = \begin{cases} 1 & \text{if we use set partition/table } t \\ 0 & \text{otherwise} \end{cases}$$

$$a_{g,t} = \begin{cases} 1 & \text{if guest } g \text{ is in table set } t \\ 0 & \text{otherwise} \end{cases}$$

In this implementation of the problem we enumerate each possible table. This approach does become intractable for large numbers of items without using column generation [9], but is sufficient for the purposes of this example. Our "people" will be represented by the single alphabetical letters, and the relative unhappiness between any two people is a function of the number of letters between the two in the English alphabet.

Each possible partition will have a objective function coefficient associated with it. The `happiness` function calculates this cost, which demonstrates an advantage of using a general-purpose language like Python for model definition: all the power and expressiveness of Python is available for implementing a model.

```
14 def happiness(table):
15     if len(table) > 1:
16         result = max(ord(guest_b) - ord(guest_a)
17                      for guest_a in table
18                      for guest_b in table
19                      if ord(guest_a) < ord(guest_b))
20     else:
21         result = 0
22     return result
```

Note that this idea can be extended to allowing multiple methods to calculate the cost or even allowing the cost-generating function to be an "argument" to the Python function that builds the model.

We use PuLP's enumeration function `pulp.allcombinations` to generate a list of all possible table seatings using a list comprehension. Binary variables that will be one if a particular table layout is used in the solution, or zero otherwise, are created using the same `LpVariable.dict` functionality used in the previous example.

```
27      possible_tables = [tuple(c) for c in
28                         allcombinations(guests, max_table_size)]
29      x = LpVariable.dicts('table', possible_tables,
30                                   lowBound = 0, upBound = 1,
31                                   cat = pulp.LpInteger)
```

As before, we create a problem object and add the objective. Note the use of the function we created previously directly in the objective – PuLP does not care that a function is used in the objective as the function is not dependent on the variables – it returns a constant value for each table.

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

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

A constraint is needed to limit the maximum tables that can be selected.

```
    seating_model += lpSum([x[table] for table in possible_tables]) \
                     <= max_tables, "Maximum_number_of_tables"
```

The remaining constraint states that each guest sits at exactly one table. The ability of Python's `for` loops to iterate through each member of a list directly, instead of using a numerical index, leads to code that is highly expressive. Translating back from the code, this implementation of the constraint states that for each guest, the total (`sum`) of table arrangements/subsets that this guest features in must equal one.

```
    for guest in guests:
        seating_model += lpSum([x[table] for table in possible_tables
                                if guest in table]
                               ) == 1, "Must_seat_%s"%guest
```

For an example problem with thirteen guests represented by the letters A to M, the output for the optimal solution is as follows:

```
The choosen tables are out of a total of 1092:
('M',) 0
('A', 'B', 'C', 'D') 3
('E', 'F', 'G', 'H') 3
('I', 'J', 'K', 'L') 3
```

4 Differences between PuLP and Pyomo

The existence of two modelling languages that use Python invites a comparison. The key differences of philosophy between Pyomo [8] and PuLP are that PuLP's models are not abstract, and that PuLP can not express non-linear models. The decision to not support non-linear optimisation models directly is intrinsic to the design of PuLP, allowing the syntax to remain simple and to keep the size of the library small by allowing the code to assume linearity. The necessity for abstract modelling is judged to be low in practice, especially given the vision of PuLP being embeddable in greater systems, where the data is wedded to the model and any separation between them is arbitrary. Given that all abstract models must become concrete to be solved, the interpreted nature of Python blurs the line between them as the uninterpreted model code can be viewed as the model abstraction, and the model that exists at run-time being concrete.

Pyomo takes a different approach to extra, non-model-building functionality, choosing to include support for loading and saving data inside the library. It is

also coupled to the larger Coopr project, which provides even more functionality. PuLP chooses the more traditional path, favoured particularly by the open-source community, of doing one thing and doing it well, relying on other libraries to provide extra functionality. Given Python's rich ecosystem of packages, we feel PuLP loses nothing from not including data loading functionality and instead gains by having less dependencies and a smaller code-base.

The syntax of the two languages are similar, but one interesting difference exists in the modelling of constraints. Consider the following PuLP example, which forces the binary assignment variables of products to zero if the corresponding location variable is zero:

```
for i in PRODUCTS:
    for j in LOCATIONS:
       prob += assign[i,j] <= use_location[j]
```

The `for` loops are very similar to the mathematical formulation, which could be expressed as:

$$x_{ij} \leq y_j, \text{ for } i = 1, 2, ..., n, j = 1, 2, ..., m \tag{1}$$

This constraint would be modelled in Pyomo by the following code:

```
def AssignRule(PRODUCTS, LOCATIONS, model):
    return model.assign[i,j] <= model.use_location[j]
model.secondConstr = Constraint(PRODUCTS, LOCATIONS, rule=AssignRule)
```

The Pyomo code does not have as clear a mapping from formulation to code and is more verbose. It uses the ability of Python to declare functions anywhere and to store a reference to a function in a variable, which is useful in some situations but increases the complexity and required familiarity with Python. It does have the benefit that Pyomo now knows that this set of constraints are grouped, but many solvers do not support using this information to improve solutions, or can not gain much benefit from it. Thus this advantage will typically be wasted in practice.

5 Conclusion: the Case for PuLP

In the above sections we have discussed how PuLP was designed, shown its strengths, demonstrated its use, and contrasted it with its main "competitor". Here we review some of the key points for why PuLP is a useful addition to the world of modelling languages:

5.1 Licensing

The licensing of PuLP under a permissive open-source license allows PuLP to be used anywhere and in any application, whether it is as an Operations Research teaching tool that can students can download and use for free, or integrated transparently into a commercial application without any licensing issues to deal with. It allows advanced users to modify and improve PuLP for their own purposes,

and ensures that any issues with the code can be addressed immediately by a developer or by the wider community.

5.2 Interoperability

The ability for PuLP to call a number of free and non-free solvers is important as an Operations Research 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 which may dramatically reduce the solution time of the model. PuLP allows the free interchange of solvers with little effort – changing a parameter for the `LpProblem.solve` function is usually sufficient.

5.3 Syntax

The mathematical description of a problem is often the most powerful and cleanest description possible, and any modelling language seeks to capture as much of that as possible. We believe PuLP provides a close code-to-math mapping that enables users to rapidly translate a mathematical model. It has great code readability that makes it easy to maintain and update models at a latter date. This is achieved by using "Pythonic" idioms and leveraging the full power of the underlying language. All PuLP models are concrete and model parameters can be provided and manipulated in any way the programmer feels is most suitable for the application. The interpreted nature of Python allows parameter data to be stored as lists, dictionaries or custom classes, as long as these expressions are linear after the code is run.

In conclusion, PuLP is a Python-based modelling tool that has a unique combination of expressiveness, deployability and power that sets it apart from other available high-level modelling languages.

For the complete source code used in the examples, more case studies and installation instructions visit the PuLP documentation hosted at `http://www.coin-or.org/PuLP/`.

References

[1] CPLEX. `http://www.ilog.com/products/cplex/`.

[2] Gurobi. `http://www.gurobi.com/`.

[3] Python Package Index. `http://pypi.python.org/pypi`.

[4] Python Standard Library. `http://docs.python.org/library/`.

[5] C. Duquesne. YAPOSIB. `http://code.google.com/p/yaposib/`.

[6] J. Forrest. Cbc. `http://www.coin-or.org/`.

[7] William Hart. Coopr. https://projects.coin-or.org/Coopr.

[8] William Hart. Python optimization modeling objects (Pyomo). In *Proc INFORMS Computing Society Conference*, 2009.

[9] M. E. Lübbecke and J. Desrosiers. Selected topics in column generation. *Operations Research*, 53(6):1007–1023, 2005.

[10] S. A. Mitchell and J.S. Roy. PuLP. http://www.coin-or.org/PuLP/.

[11] Open Source Iniatitive. MIT license. http://www.opensource.org/licenses/mit-license.php.

[12] M. O'Sullivan. Dippy. https://projects.coin-or.org/CoinBazaar/wiki/Projects/Dippy.

[13] Guido van Rossum. PEP 8 – Style Guide for Python Code. http://www.python.org/dev/peps/pep-0008/.

[14] Guido van Rossum. *Python Reference Manual*. CWI Report, CS-R9525 edition, May 1995.