pulp Documentation

Release 1.4.6

pulp documentation team

January 27, 2010

CONTENTS

You can begin learning Python and using PuLP by looking at the content below. We recommend that you read The Optimisation Process, Optimisation Concepts, and the Introduction to Python before beginning the case-studies. For instructions for the installation of PuLP see *Installing PuLP at Home*.

The full PuLP function documentation is available, and useful functions will be explained in the case studies. The case studies are in order, so the later case studies will assume you have (at least) read the earlier case studies. However, we will provide links to any relevant information you will need.

CHAPTER

ONE

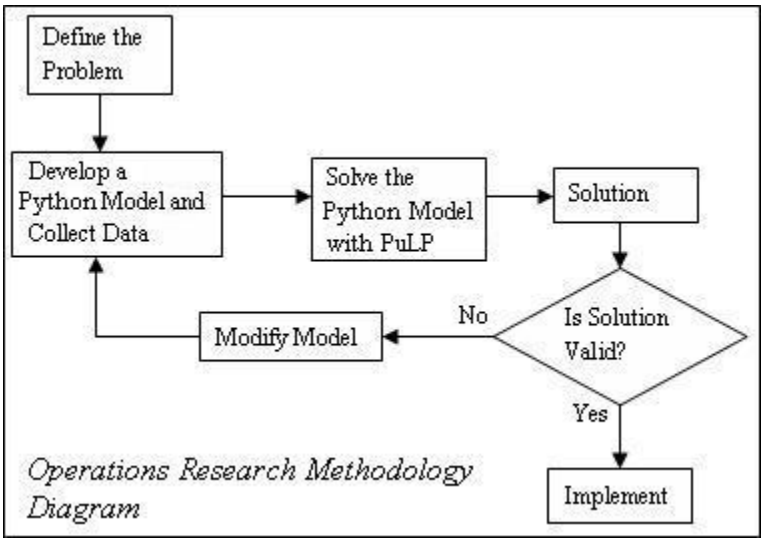
MAIN TOPICS

1.1 The Optimisation Process

Solving an optimisation problem is not a linear process, but the process can be broken down into five general steps:

- Getting the problem description
- Formulating the mathematical program
- Solving the mathematical program
- Performing some post-optimal analysis
- Presenting the solution and analysis

However, there are often "feedback loops" within this process. For example, after formulating and solving an optimisation problem, you will often want to consider the validity of your solution (often consulting with the person who provided the problem description). If your solution is invalid you may need to alter or update your formulation to incorporate your new understanding of the actual problem. This process is shown in the Operations Research Methodology Diagram.



The modeling process starts with a well-defined model description, then uses mathematics to formulate a mathematical program. Next, the modeler enters the mathematical program into some solver software, e.g., Excel and solves the model. Finally, the solution is translated into a decision in terms of the original model description.

Using Python gives you a "shortcut" through the modeling process. By formulating the mathematical program in Python you have already put it into a form that can be used easily by PuLP the modeller to call many solvers, e.g.

CPLEX, COIN, gurobi so you don't need to enter the mathematical program into the solver software. However, you usually don't put any "hard" numbers into your formulation, instead you "populate" your model using data files, so there is some work involved in creating the appropriate data file. The advantage of using data files is that the same model may used many times with different data sets.

1.1.1 The Modeling Process

The modeling process is a "neat and tidy" simplification of the optimisation process. Let's consider the five steps of the optimisation process in more detail:

Getting the Problem Description

The aim of this step is to come up with a formal, rigourous model description. Usually you start an optimisation project with an abstract description of a problem and some data. Often you need to spend some time talking with the person providing the problem (usually known as the client). By talking with the client and considering the data available you can come up with the more rigourous model description you are used to. Sometimes not all the data will be relevant or you will need to ask the client if they can provide some other data. Sometimes the limitations of the available data may change your model description and subsequent formulation significantly.

Formulating the mathematical program

In this step we identify the key quantifiable decisions, restrictions and goals from the problem description, and capture their interdependencies in a mathematical model. We can break the formulation process into 4 key steps:

- Identify the Decision Variables paying particular attention to units (for example: we need to decide how many hours per week each process will run for).
- Formulate the Objective Function using the decision variables, we can construct a minimise or maximise objective function. The objective function typically reflects the total cost, or total profit, for a given value of the decision variables.
- Formulate the Constraints, either logical (for example, we cannot work for a negative number of hours), or explicit to the problem description. Again, the constraints are expressed in terms of the decision variables.
- Identify the Data needed for the objective function and constraints. To solve your mathematical program you will need to have some "hard numbers" as variable bounds and/or variable coefficients in your objective function and/or constraints.

Solving the mathematical program

For relatively simple or well understood problems the mathematical model can often be solved to optimality (i.e., the best possible solution is identified). This is done using algorithms such as the Revised Simplex Method or Interior Point Methods. However, many industrial problems would take too long to solve to optimality using these techniques, and so are solved using heuristic methods which do not guarantee optimality.

Performing some post-optimal analysis

Often there is uncertainty in the problem description (either with the accuracy of the data provided, or with the value(s) of data in the future). In this situation the robustness of our solution can be examined by performing post-optimal analysis. This involves identifying how the optimal solution would change under various changes to the formulation (for example, what would be the effect of a given cost increasing, or a particular machine failing?). This sort of

analysis can also be useful for making tactical or strategic decisions (for example, if we invested in opening another factory, what effect would this have on our revenue?).

Another important consideration in this step (and the next) is the validation of the mathematical program's solution. You should carefully consider what the solution's variable values mean in terms of the original problem description. Make sure they make sense to you and, more importantly, your client (which is why the next step, presenting the solution and analysis is important).

Presenting the solution and analysis

A crucial step in the optimisation process is the presentation of the solution and any post-optimal analysis. The translation from a mathematical program's solution back into a concise and comprehensible summary is as important as the translation from the problem description into the mathematical program. Key observations and decisions generated via optimisation must be presented in an easily understandable way for the client or project stakeholders.

Your presentation is a crucial first step in the implementation of the decisions generated by your mathematical program. If the decisions and their consequences (often determined by the mathematical program constraints) are not presented clearly and intelligently your optimal decision will never be used.

This step is also your chance to suggest other work in the future. This could include:

- Periodic monitoring of the validity of your mathematical program;
- Further analysis of your solution, looking for other benefits for your client;
- Identification of future optimisation opportunities.

1.2 Optimisation Concepts

1.2.1 Linear Programing

The simplest type of mathematical program is a linear program. For your mathematical program to be a linear program you need the following conditions to be true:

- The decision variables must be real variables;
- The objective must be a linear expression;
- The constraints must be linear expressions.

Linear expressions are any expression of the form

$$a_1x_1 + a_2x_2 + a_3x_3 + ...a_nx_n\{<=, =, >=\}b$$

where the a_i and b are known constants and x_i are variables. The process of solving a linear program is called linear programing. Linear programing is done via the Revised Simplex Method (also known as the Primal Simplex Method), the Dual Simplex Method or an Interior Point Method. Some solvers like cplex allow you to specify which method you use, but we won't go into further detail here.

1.2.2 Integer Programing

Integer programs are almost identical to linear programs with one very important exception. Some of the decision variables in integer programs may need to have only integer values. The variables are known as integer variables. Since most integer programs contain a mix of continuous variables and integer variables they are often known as mixed integer programs. While the change from linear programing is a minor one, the effect on the solution process

is enormous. Integer programs can be very difficult problems to solve and there is a lot of current research finding "good" ways to solve integer programs. Integer programs can be solved using the branch-and-bound process.

Note For MIPs of any reasonable size the solution time grows exponentially as the number of integer variables increases.

1.3 Basic Python Coding

In this course you will learn basic programming in Python, but there is also excellent Python Language reference material available on the internet freely. You can download the book Dive Into Python or there are a host of Beginners Guides to Python on the Python Website. Follow the links to either:

- BeginnersGuide/NonProgrammers
- BeginnersGuide/Programmers

depending on your level of current programming knowledge. The code sections below assume a knowledge of fundamental programming principles and mainly focus on Syntax specific to programming in Python.

Note: >>> represents a Python command line prompt.

1.3.1 Loops in Python

for Loop

The general format is:

```
for variable in sequence:
    #some commands
#other commands after for loop
```

Note that the formatting (indents and new lines) governs the end of the for loop, whereas the start of the loop is the colon :.

Observe the loop below, which is similar to loops you will be using in the course. The variable i moves through the string list becoming each string in turn. The top section is the code in a .py file and the bottom section shows the output

```
@#The following code demonstrates a list with strings
ingredientslist = ["Rice","Water","Jelly"]
for i in ingredientslist:
    print i
print "No longer in the loop"
```

gives

```
Rice
Water
Jelly
No longer in the loop
```

while Loop

These are similar to for loops except they continue to loop until the specified condition is no longer true. A while loop is not told to work through any specific sequence.

```
i = 3
while i @textless[]= 15:
    @# some commands
    i = i + 1 @# a command that will eventually end the loop is naturally
    required
@# other commands after while loop
```

For this specific simple while loop, it would have been better to do as a for loop but it demonstrates the syntax. while loops are useful if the number of iterations before the loop needs to end, is unknown.

The if statement

This is performed in much the same way as the loops above. The key identifier is the colon : to start the statements and the end of indentation to end it.

```
if j in testlist:
    # some commands
elif j == 5:
    # some commands
else:
    # some commands
```

Here it is shown that "elif" (else if) and "else" can also be used after an if statement. "else" can in fact be used after both the loops in the same way.

1.3.2 Array types in python

Lists

A list is simply a sequence of variables grouped together. The range function is often used to create lists of integers, with the general format of range(start,stop,step). The default for start is 0 and the default for step is 1.

```
@textgreater[]@textgreater[]@textgreater[] range(3,8)
[3,4,5,6,7]
```

This is a list/sequence. As well as integers, lists can also have strings in them or a mix of integers, floats and strings. They can be created by a loop (as shown in the next section) or by explicit creation (below). Note that the print statement will display a string/variable/list/... to the user

```
@textgreater[]@textgreater[]@textgreater[] a = [5,8,"pt"]
@textgreater[]@textgreater[]@textgreater[] print a
[5,8,'pt']
@textgreater[]@textgreater[]@textgreater[] print a[0]
5
```

Tuples

Tuples are basically the same as lists, but with the important difference that they cannot be modified once they have been created. They are assigned by:

```
@textgreater[]@textgreater[]@textgreater[] x = (4,1,8,"string",[1,0],("j",4,"o"),14)
```

Tuples can have any type of number, strings, lists, other tuples, functions and objects, inside them. Also note that the first element in the tuple is numbered as element "zero". Accessing this data is done by:

```
@textgreater[]@textgreater[]@textgreater[] x[0]
4
@textgreater[]@textgreater[]@textgreater[] x[3]
"string"
```

Dictionaries

A Dictionary is a list of reference keys each with associated data, whereby the order does not affect the operation of the dictionary at all. With dictionaries, the keys are not consecutive integers (unlike lists), and instead could be integers, floats or strings. This will become clear:

```
@textgreater[]@textgreater[]@textgreater[] x = {} @# creates a new empty dictionary - note the curly
@textgreater[]@textgreater[]@textgreater[] x[4] = "programming" @# the string "programming" is added
@textgreater[]@textgreater[]@textgreater[] x["games"] = 12
@textgreater[]@textgreater[]@textgreater[] print x["games"]
12
```

In a dictionary, the reference keys and the stored values can be any type of input. New dictionary elements are added as they are created (with a list, you cannot access or write to a place in the list that exceeds the initially defined list dimensions).

```
costs = {"CHICKEN": 1.3, "BEEF": 0.8, "MUTTON": 12}
print "Cost of Meats"
for i in costs:
    print i
    print costs[i]
costs["LAMB"] = 5
print "Updated Costs of Meats"
for i in costs:
    print i
    print costs[i]
```

gives

```
Cost of Meats
CHICKEN
1.3
MUTTON
12
BEEF
0.8
Updated Costs of Meats
LAMB
5
CHICKEN
```

```
1.3
MUTTON
12
BEEF
0.8
```

In the above example, the dictionary is created using curly brackets and colons to represent the assignment of data to the dictionary keys. The variable i is assigned to each of the keys in turn (in the same way it would be for a list with

```
@textgreater[]@textgreater[]@textgreater[] for i in range(1,10)
```

). Then the dictionary is called with this key, and it returns the data stored under that key name. These types of for loops using dictionaries will be highly relevant in using PuLP to model LPs in this course.

List/Tuple/Dictionary Syntax Note

Note that the creation of a:

- list is done with square brackets [];
- tuple is done with round brackets and a comma (,);
- dictionary is done with parentheses{ }.

After creation however, when accessing elements in the list/tuple/dictionary, the operation is always performed with square brackets (i.e a[3]?). If a was a list or tuple, this would return the fourth element. If a was a dictionary it would return the data stored with a reference key of 3.

List Comprehensions

Python supports List Comprehensions which are a fast and concise way to create lists without using multiple lines. They are easily understandable when simple, and you will be using them in your code for this course.

```
@textgreater[]@textgreater[]@textgreater[] a = [i for i in range(5)]
@textgreater[]@textgreater[]@textgreater[] a
[0, 1, 2, 3, 4]
```

This statement above will create the list [0,1,2,3,4] and assign it to the variable “a”.

```
@textgreater[]@textgreater[]@textgreater[] odds = [i for i in range(25) if i@%2==1]
@textgreater[]@textgreater[]@textgreater[] odds
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23]
```

This statement above uses the if statement and the modulus operator(%) so that only odd numbers are included in the list: [1,3,5,...,19,21,23]. (Note: The modulus operator calculates the remainder from an integer division.)

```
@textgreater[]@textgreater[]@textgreater[] fifths = [i for i in range(25) if i@%5==0]
@textgreater[]@textgreater[]@textgreater[] fifths
[0, 5, 10, 15, 20]
```

This will create a list with every fifth value in it [0,5,10,15,20]. Existing lists can also be used in the creation of new lists below:

```
@textgreater[]@textgreater[]@textgreater[] a = [i for i in range(25) if (i in odds and i not in fift
```

Note that this could also have been done in one step from scratch:

```
@textgreater[]@textgreater[]@textgreater[] a = [i for i in range(25) if (i@%2==1 and i@%5==0)]
```

For a challenge you can try creating

1. a list of prime numbers up to 100, and
2. a list of all "perfect" numbers.

More List Comprehensions Examples

Wikipedia: Perfect Numbers.

1.3.3 Other important language features

Commenting in Python

Commenting at the top of a file is done using """ to start and to end the comment section. Commenting done throughout the code is done using the hash # symbol at the start of the line.

Import Statement

At the top of all your Python coding that you intend to use PuLP to model, you will need the import statement. This statement makes the contents of another module (file of program code) available in the module you are currently writing i.e. functions and values defined in pulp.py that you will be required to call, become usable. In this course you will use:

```
@textgreater[]@textgreater[]@textgreater[] from pulp import *
```

The asterisk represents that you are importing all names from the module of pulp. Calling a function defined in pulp.py now can be done as if they were defined in your own module.

Functions

Functions in Python are defined by: (def is short for define)

```
def name(inputparameter1, inputparameter2, . . .):
    #function body
```

For a real example, note that if inputs are assigned a value in the function definition, that is the default value, and will be used only if no other value is passed in. The order of the input parameters (in the definition) does not matter at all, as long as when the function is called, the positional parameters are entered in the corresponding order. If keywords are used the order of the parameters does not matter at all:

```
def string_appender(head='begin', tail='end', end_message='EOL'):
    result = head + tail + end_message
    return result


@textgreater[]@textgreater[]@textgreater[] string_appender('newbegin', end_message = 'StringOver')
newbeginendStringOver
```

In the above example, the output from the function call is printed. The default value for head is 'begin', but instead the input of 'newbegin' was used. The default value for tail of 'end' was used. And the input value of endmessage was used. Note that end_message must be specified as a key word argument as no value is given for tail

Classes

To demonstrate how classes work in Python, look at the following class structure.

The class name is Pattern, and it contains several class variables which are relevant to any instance of the Pattern class (i.e a Pattern). The functions are

__init__ function which creates an instance of the Pattern class and assigns the attributes of name and lengthsdict using self.

__str__ function defines what to return if the class instance is printed.

trim function acts like any normal function, except as with all class functions, self must be in the input brackets.

```
class Pattern:
    """
    Information on a specific pattern in the SpongeRoll Problem
    """
    cost = 1
    trimValue = 0.04
    totalRollLength = 20
    lenOpts = [5, 7, 9]

def __init__(self,name,lengths = None):
    self.name = name
    self.lengthsdict = dict(zip(self.lenOpts,lengths))

def __str__(self):
    return self.name

def trim(self):
return Pattern.totalRollLength - sum([int(i)*self.lengthsdict[i] for i in self.lengthsdict])
```

This class could be used as follows:

```
@textgreater[]@textgreater[]@textgreater[] Pattern.cost @# The class attributes can be accessed witho
1
@textgreater[]@textgreater[]@textgreater[] a = Pattern("PatternA",[1,0,1])
@textgreater[]@textgreater[]@textgreater[] a.cost @# a is now an instance of the Pattern class and is
1
@textgreater[]@textgreater[]@textgreater[] print a @# This calls the Pattern.__str__() function
"PatternA"
@textgreater[]@textgreater[]@textgreater[] a.trim() @# This calls the Pattern.trim() function. Note t
```

The self in the function definition is an implied input

1.4 Installing PuLP at Home

PuLP is a free open source software written in Python. It is used to describe optimisation problems as mathematical models. PuLP can then call any of numerous external LP solvers (CBC, GLPK, CPLEX, Gurobi etc) to solve this model and then use python commands to manipulate and display the solution.

1.4.1 Installation

Note that to install PuLP you must first have a working python installation as described in installing python.

PuLP requires Python >= 2.5. Though it can be made to work with Python 2.4

The latest version of PuLP can be freely obtained from coin-or. Please note that this version of PuLP has not been tested with operating systems other than Microsoft Windows and Ubuntu Linux.

Easy install and pypi installation

By far the easiest way to install pulp is through the use of EasyInstall and CheeseShop.

- **Install EasyInstall**
  - In windows (please make sure easy_install is on your path):

    ```
    c:\Python26\Scripts\> easy_install -U pulp
    ```

  - In Linux:

    ```
    $ sudo easy_install -U pulp
    ```

- Then follow the instructions below to test your installation

To access the examples and pulp source code use the instructions below to install from source

Windows installation from source

- Install python (installing python)
- Download the PuLP zipfile
- **Extract the zipfile to a suitable location (such as the desktop - the folder will be no longer required after installation)**
  - Open a command prompt by clicking "Run" in the Start Menu, and type 'cmd' in the window and push enter.
  - Navigate to the extracted folder with the setup file in it. [Do this by typing 'cd foldername' at the prompt, where 'cd' stands for current directory and the 'foldername' is the name of the folder to open in the path already listed to the left of the prompt. To return back to a root drive, type 'cd C:']
  - Type 'setup.py install' at the command prompt. This will install all the PuLP functions into Python's site-packages directory.

The PuLP function library is now able to be imported from any python command line. Go to IDLE or PyDev and type

```
@textgreater[]@textgreater[]@textgreater[] from pulp import *
```

to load in the functions. (You need to re-import the functions each time after you close the GUI) PuLP is written in a programming language called Python, and to use PuLP you must write Python code to describe your optimization problem.

Linux Installation

- Extract the PuLP zipfile folder to a suitable location (such as your home directory - the folder will be no longer required after installation)
- Open a command line navigate to the extracted zipfile with the setup file in it. [Do this by typing 'cd foldername' at the prompt]
- Type the following at the command prompt. This will install all the PuLP functions into Python's callable modules.

```
@$ sudo python setup.py install
```

- **install a solver for pulp to use either**
  - use the included 32-bit binaries CoinMP
  - install glpk debain based distributions may use the following

```
@$ sudo apt-get install glpk
```

  - install gurobi (free academic licenses)
  - install cplex (and pay $$)
  - or compile coinMP and pulp from source using buildout and copy the files

```
@$ python bootstrap.py
@$ bin/buildout -c solvers.cfg
@$ cp parts/lib/* src/pulp/
@$ sudo setup.py install
```

1.4.2 Testing your PuLP installation

To test that that you pulp installation is working correctly please type the following into a python interpreter and note that the output should be similar. The output below is what you would expect if you have not installed any other solvers and the CoinMP solver bundled with pulp works.

```
@textgreater[]@textgreater[]@textgreater[] import pulp
@textgreater[]@textgreater[]@textgreater[] pulp.pulpTestAll()
Solver pulp.pulp.COIN_MEM unavailable.
Solver pulp.pulp.COIN_CMD unavailable.
     Testing continuous LP solution
     Testing maximize continuous LP solution
     Testing unbounded continuous LP solution
     Testing MIP solution
     Testing MIP relaxation
     Testing feasibility problem (no objective)
     Testing an infeasible problem
     Testing an integer infeasible problem (Error to be fixed)
     Testing column based modelling
     Testing column based modelling with empty constraints
     Testing dual variables and slacks reporting
     Testing resolve of problem
     Testing Sequential Solves
     Testing fractional constraints
     Testing elastic constraints (no change)
```

```
     Testing elastic constraints (freebound)
     Testing elastic constraints (penalty unchanged)
     Testing elastic constraints (penalty unbounded)
* Solver pulp.pulp.COINMP_DLL passed.
Solver pulp.pulp.GLPK_MEM unavailable.
Solver pulp.pulp.GLPK_CMD unavailable.
Solver pulp.pulp.XPRESS unavailable.
```

CHAPTER

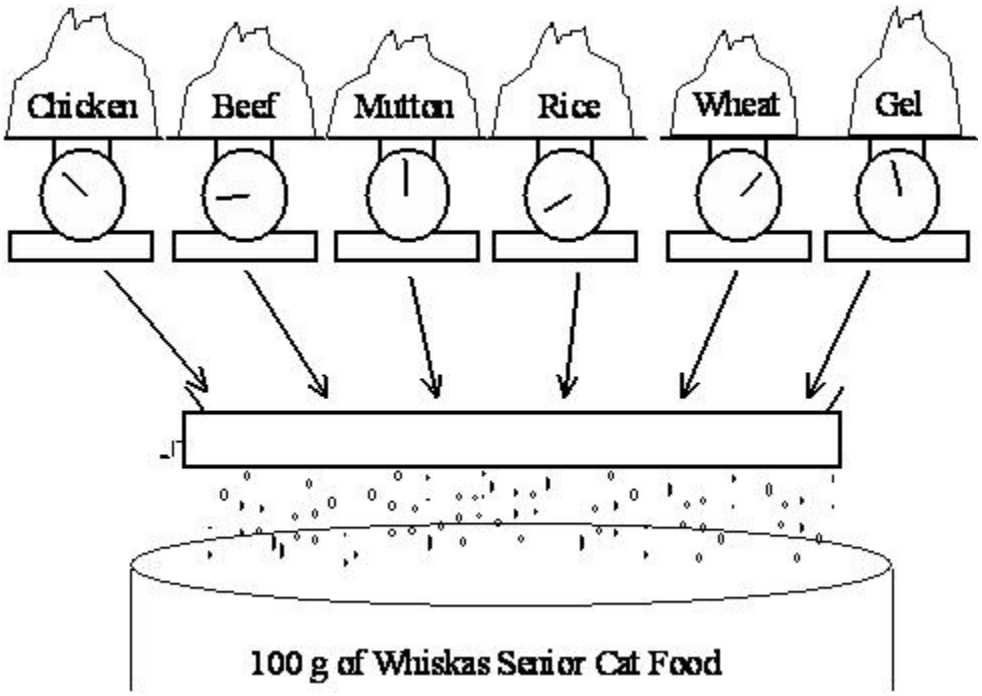
TWO

CASE STUDIES

2.1 A Blending Problem

2.1.1 Problem Description

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.



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. The contributions (in grams) per gram of ingredient are given in the table below.

Stuff	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	0.002
Wheat bran	0.040	0.010	0.150	0.008

Simplified Formulation

First we will consider a simplified problem to build a simple Python model.

Identify the Decision Variables

Assume Whiskas want to make their cat food out of just two ingredients: Chicken and Beef. We will first define our decision variables:

$$x_1 = \text{ percentage of chicken meat in a can of cat food}$$
$$x_2 = \text{ percentage of beef used in a can of cat food}$$

The limitations on these variables (greater than zero) must be noted but for the Python implementation, they are not entered or listed separately or with the other constraints.

Formulate the Objective Function

The objective function becomes:

$$\textbf{min } 0.013x_1 + 0.008x_2$$

The Constraints

The constraints on the variables are that they must sum to 100 and that the nutritional requirements are met:

$$\begin{aligned}
1.000x_1 + 1.000x_2 &= 100.0 \\
0.100x_1 + 0.200x_2 &\geq 8.0 \\
0.080x_1 + 0.100x_2 &\geq 6.0 \\
0.001x_1 + 0.005x_2 &\leq 2.0 \\
0.002x_1 + 0.005x_2 &\leq 0.4
\end{aligned}$$

Solution to Simplified Problem

To obtain the solution to this Linear Program, we can write a short program in Python to call PuLP's modelling functions, which will then call a solver. This will explain step-by-step how to write this Python program. It is suggested that you repeat the exercise yourself. The code for this example is found in WhiskasModel1.py

The start of the your file should then be headed with a short commenting section outlining the purpose of the program. For example:

```
"""
The Simplified Whiskas Model Python Formulation for the PuLP Modeller

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

Then you will import PuLP's functions for use in your code:

```
@# Import PuLP modeler functions
from pulp import *
```

A variable called `prob` (although its name is not important) is created using the `LpProblem` function. It has two parameters, the first being the arbitrary name of this problem (as a string), and the second parameter being either `LpMinimize` or `LpMaximize` depending on the type of LP you are trying to solve:

```
@# Create the 'prob' variable to contain the problem data
prob = LpProblem("The Whiskas Problem",LpMinimize)
```

The problem variables `x1` and `x2` are created using the `LpVariable` class. It has four parameters, the first is the arbitrary name of what this variable represents, the second is the lower bound on this variable, the third is the upper bound, and the fourth is essentially the type of data (discrete or continuous). The options for the fourth parameter are `LpContinuous` or `LpInteger`, with the default as `LpContinuous`. If we were modelling the number of cans to produce, we would need to input `LpInteger` since it is discrete data. The bounds can be entered directly as a number, or `None` to represent no bound (i.e. positive or negative infinity), with `None` as the default. If the first few parameters are entered and the rest are ignored (as shown), they take their default values. However, if you wish to specify the third parameter, but you want the second to be the default value, you will need to specifically set the second parameter as it's default value. i.e you cannot leave a parameter entry blank. e.g:

```
LpVariable("example", None, 100)
```

or:

```
LpVariable("example", upBound = 100)
```

To explicitly create the two variables needed for this problem:

```
@# The 2 variables Beef and Chicken are created with a lower limit of zero
x1=LpVariable("ChickenPercent",0,None,LpInteger)
x2=LpVariable("BeefPercent",0)
```

The variable `prob` now begins collecting problem data with the += operator. The objective function is logically entered first, with an important comma `,` at the end of the statement and a short string explaining what this objective function is:

```
@# The objective function is added to 'prob' first
prob += 0.013*x1 + 0.008*x2, "Total Cost of Ingredients per can"
```

The constraints are now entered (Note: any "non-negative" constraints were already included when defining the variables). This is done with the '+=' operator again, since we are adding more data to the `prob` variable. The constraint is logically entered after this, with a comma at the end of the constraint equation and a brief description of the cause of that constraint:

```
@# The five constraints are entered
prob += x1 + x2 == 100, "PercentagesSum"
prob += 0.100*x1 + 0.200*x2 @textgreater[]= 8.0, "ProteinRequirement"
prob += 0.080*x1 + 0.100*x2 @textgreater[]= 6.0, "FatRequirement"
prob += 0.001*x1 + 0.005*x2 @textless[]= 2.0, "FibreRequirement"
prob += 0.002*x1 + 0.005*x2 @textless[]= 0.4, "SaltRequirement"
```

Now that all the problem data is entered, the `writeLP()` function can be used to copy this information into a .lp file into the directory that your code-block is running from. Once your code runs successfully, you can open this .lp file with a text editor to see what the above steps were doing. You will notice that there is no assignment operator (such as an equals sign) on this line. This is because the function/method called `writeLP()` is being performed to the variable/object `prob` (and the string `"WhiskasModel.lp"` is an additional parameter). The dot `.` between the variable/object and the function/method is important and is seen frequently in Object Oriented software (such as this):

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

The LP is solved using the solver that PuLP chooses. The input brackets after `solve()` are left empty in this case, however they can be used to specify which solver to use (e.g `prob.solve(CPLEX())`):

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

Now the results of the solver call can be displayed as output to us. Firstly, we request the status of the solution, which can be one of "Not Solved", "Infeasible", "Unbounded", "Undefined" or "Optimal". The value of `prob` (`pulp.pulp.LpProblem.status`) is returned as an integer, which must be converted to its significant text meaning using the `LpStatus` dictionary. Since `LpStatus` is a dictionary(`dict`), its input must be in square brackets:

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

The variables and their resolved optimum values can now be printed to the screen.

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

The `for` loop makes `variable` cycle through all the problem variable names (in this case just `ChickenPercent` and `BeefPercent`). Then it prints each variable name, followed by an equals sign, followed by its optimum value. `name` and `varValue` are properties of the object `variable`.

The optimised objective function value is printed to the screen, using the value function. This ensures that the number is in the right format to be displayed. `objective` is an attribute of the object `prob`:

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

Running this file should then produce the output to show that Chicken will make up 33.33%, Beef will make up 66.67% and the Total cost of ingredients per can is 96 cents.

2.1.2 Full Formulation

Now we will formulate the problem fully with all the variables. Whilst it could be implemented into Python with little addition to our method above, we will look at a better way which does not mix the problem data, and the formulation as much. This will make it easier to change any problem data for other tests. We will start the same way by algebraically defining the problem:

1. Identify the Decision Variables For the Whiskas Cat Food Problem the decision variables are the percentages of the different ingredients we include in the can. Since the can is 100g, these percentages also represent the amount in g of each ingredient included. We must formally define our decision variables, being sure to state the units we are using.

$$
\begin{aligned}
x_1 &= \text{percentage of chicken meat in a can of cat food} \\
x_2 &= \text{percentage of beef used in a can of cat food} \\
x_3 &= \text{percentage of mutton used in a can of cat food} \\
x_4 &= \text{percentage of rice used in a can of cat food} \\
x_5 &= \text{percentage of wheat bran used in a can of cat food} \\
x_6 &= \text{percentage of gel used in a can of cat food}
\end{aligned}
$$

   Note that these percentages must be between 0 and 100.

2. Formulate the Objective Function For the Whiskas Cat Food Problem the objective is to minimise the total cost of ingredients per can of cat food. We know the cost per g of each ingredient. We decide the percentage of each ingredient in the can, so we must divide by 100 and multiply by the weight of the can in g. This will give us the weight in g of each ingredient:

$$\min \$0.013x_1 + \$0.008x_2 + \$0.010x_3 + \$0.002x_4 + \$0.005x_5 + \$0.001x_6$$

3. Formulate the Constraints The constraints for the Whiskas Cat Food Problem are that:
   - The sum of the percentages must make up the whole can (= 100%).
   - The stated nutritional analysis requirements are met.

The constraint for the "whole can" is:

$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 = 100$$

To meet the nutritional analysis requirements, we need to have at least 8g of Protein per 100g, 6g of fat, but no more than 2g of fibre and 0.4g of salt. To formulate these constraints we make use of the previous table of contributions from each ingredient. This allows us to formulate the following constraints on the total contributions of protein, fat, fibre and salt from the ingredients:

$$\begin{aligned}
0.100x_1 + 0.200x_2 + 0.150x_3 + 0.000x_4 + 0.040x_5 + 0.0x_60 &\geq 8.0 \\
0.080x_1 + 0.100x_2 + 0.110x_3 + 0.010x_4 + 0.010x_50 + 0.0x_6 &\geq 6.0 \\
0.001x_1 + 0.005x_2 + 0.003x_3 + 0.100x_40 + 0.150x_5 + 0.0x_6 &\leq 2.0 \\
0.002x_1 + 0.005x_2 + 0.007x_30 + 0.002x_4 + 0.008x_5 + 0.0x_6 &\leq 0.4
\end{aligned}$$

Solution to Full Problem

To obtain the solution to this Linear Program, we again write a short program in Python to call PuLP's modelling functions, which will then call a solver. This will explain step-by-step how to write this Python program with it's improvement to the above model. It is suggested that you repeat the exercise yourself. The code for this example is found in the WhiskasModel2.py

As with last time, it is advisable to head your file with commenting on its purpose, and the author name and date. Importing of the PuLP functions is also done in the same way:

```
"""
The Full Whiskas Model Python Formulation for the PuLP Modeller

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

@# Import PuLP modeler functions
from pulp import *
```

Next, before the `prob` variable or type of problem are defined, the key problem data is entered into dictionaries. This includes the list of Ingredients, followed by the cost of each Ingredient, and it's percentage of each of the four nutrients. These values are clearly laid out and could easily be changed by someone with little knowledge of programming. The ingredients are the reference keys, with the numbers as the data.

```
@# Creates a list of the Ingredients
Ingredients = ['CHICKEN', 'BEEF', 'MUTTON', 'RICE', 'WHEAT', 'GEL']

@# A dictionary of the costs of each of the Ingredients is created
costs = {'CHICKEN': 0.013,
         'BEEF': 0.008,
         'MUTTON': 0.010,
         'RICE': 0.002,
         'WHEAT': 0.005,
         'GEL': 0.001}

@# A dictionary of the protein percent in each of the Ingredients is created
proteinPercent = {'CHICKEN': 0.100,
                  'BEEF': 0.200,
                  'MUTTON': 0.150,
                  'RICE': 0.000,
```

```
                  'WHEAT': 0.040,
                  'GEL': 0.000}

## A dictionary of the fat percent in each of the Ingredients is created
fatPercent = {'CHICKEN': 0.080,
              'BEEF': 0.100,
              'MUTTON': 0.110,
              'RICE': 0.010,
              'WHEAT': 0.010,
              'GEL': 0.000}

## A dictionary of the fibre percent in each of the Ingredients is created
fibrePercent = {'CHICKEN': 0.001,
                'BEEF': 0.005,
                'MUTTON': 0.003,
                'RICE': 0.100,
                'WHEAT': 0.150,
                'GEL': 0.000}

## A dictionary of the salt percent in each of the Ingredients is created
saltPercent = {'CHICKEN': 0.002,
               'BEEF': 0.005,
               'MUTTON': 0.007,
               'RICE': 0.002,
               'WHEAT': 0.008,
               'GEL': 0.000}
```

The `prob` variable is created to contain the formulation, and the usual parameters are passed into `LpProblem`.

```
## Create the 'prob' variable to contain the problem data
prob = LpProblem("The Whiskas Problem", LpMinimize)
```

A dictionary called `ingredient_vars` is created which contains the LP variables, with their defined lower bound of zero. The reference keys to the dictionary are the Ingredient names, and the data is `Ingr_IngredientName`. (e.g. MUTTON: Ingr_MUTTON)

```
## A dictionary called 'ingredient_vars' is created to contain the referenced Variables
ingredient_vars = LpVariable.dicts("Ingr",Ingredients,0)
```

Since `costs` and `ingredient_vars` are now dictionaries with the reference keys as the Ingredient names, the data can be simply extracted with a list comprehension as shown. The `lpSum()` function will add the elements of the resulting list. Thus the objective function is simply entered and assigned a name:

```
## The objective function is added to 'prob' first
prob += lpSum([costs[i]*ingredient_vars[i] for i in Ingredients]), "Total Cost of Ingredients per ca
```

Further list comprehensions are used to define the other 5 constraints, which are also each given names describing them.

```
## The five constraints are added to 'prob'
prob += lpSum([ingredient_vars[i] for i in Ingredients]) == 100, "PercentagesSum"
prob += lpSum([proteinPercent[i] * ingredient_vars[i] for i in Ingredients]) @textgreater[]= 8.0, "Pr
prob += lpSum([fatPercent[i] * ingredient_vars[i] for i in Ingredients]) @textgreater[]= 6.0, "FatRe
prob += lpSum([fibrePercent[i] * ingredient_vars[i] for i in Ingredients]) @textless[]= 2.0, "FibreR
prob += lpSum([saltPercent[i] * ingredient_vars[i] for i in Ingredients]) @textless[]= 0.4, "SaltRequ
```

Following this, the *writeLP* line etc follow exactly the same as in the simplified example.

The optimal solution is 60% Beef and 40% Gel leading to a objective Function value of 52 cents per can.

2.2 A Set Partitioning Problem

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) 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 `allcombinations()` to generate a list of all possible table seatings.

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

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

```
@#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)
```

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.

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

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

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

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

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

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

The full file can be found here 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]) @textless[]= 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
                                if guest in table]) == 1, "Must_seat_@%s"@%guest

seating_model.solve()

print "The choosen tables are out of a total of @%s:"@%len(possible_tables)
for table in possible_tables:
    if x[table].value() == 1.0:
        print table
```

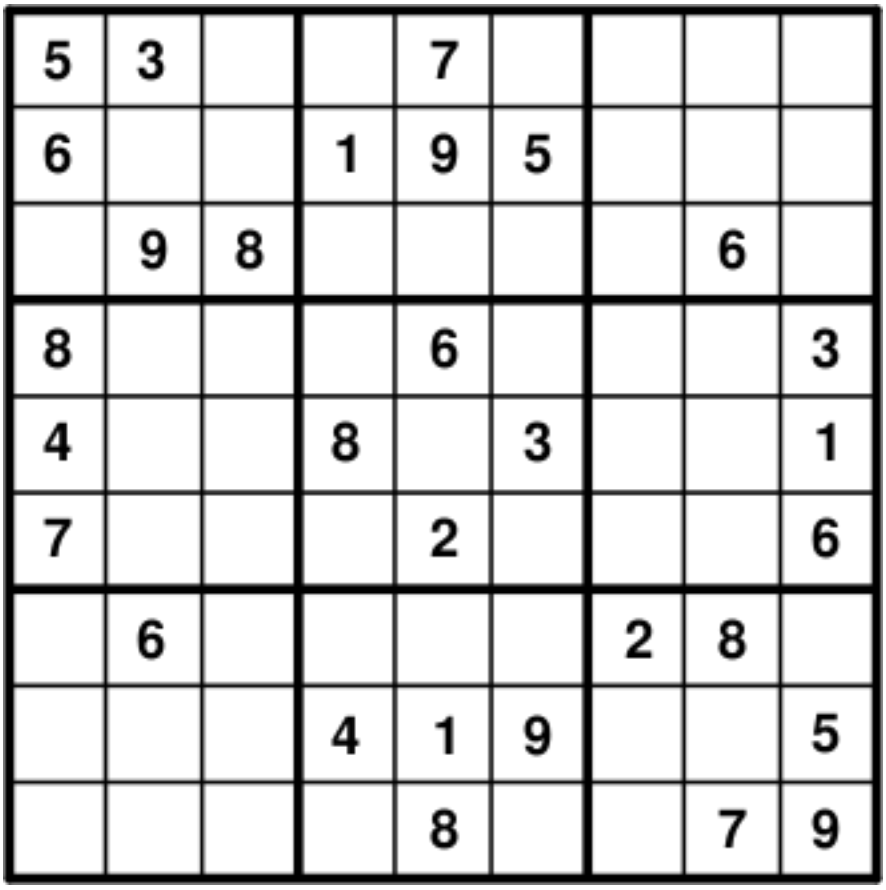
2.3 A Sudoku Problem formulated as an LP

2.3.1 Problem Description

A sudoku problem is a problem where there are is an incomplete 9x9 table of numbers which must be filled according to several rules:

- Within any of the 9 individual 3x3 boxes, each of the numbers 1 to 9 must be found
- Within any column of the 9x9 grid, each of the numbers 1 to 9 must be found
- Within any row of the 9x9 grid, each of the numbers 1 to 9 must be found

On this page we will formulate the below problem from wikipedia to model using PuLP. Once created, our code will need little modification to solve any sudoku problem at all.



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

2.3.2 Formulation

Identify the Decision Variables

In order to formulate this problem as a linear program, we cannot simply create a variable for each of the 81 squares between 1 and 9 representing the value in that square. This is because in linear programming there is no "not equal to" operator and so we cannot use the necessary constraints of no squares within a box/row/column being equal in value to each other. Whilst we can ensure the sum of all the values in a box/row/column equal 45, this will still result in many solutions satisfying the 45 constraint but still with 2 of the same number in the same box/row/column.

Instead, we must create 729 individual binary (0-1) problem variables. These represent 9 problem variables per square for each of 81 squares, where the 9 variables each correspond to the number that might be in that square. The binary nature of the variable says whether the existence of that number in that square is true or false. Therefore, there can clearly be only 1 of the 9 variables for each square as true (1) and the other 8 must be false (0) since only one number can be placed into any square. This will become more clear.

Formulate the Objective Function

Interestingly, with sudoku there is no solution that is better than another solution, since a solution by definition, must satisfy all the constraints. Therefore, we are not really trying to minimise or maximise anything, we are just trying to find the values on our variables that satisfy the constraints. Therefore, whilst either `LpMinimize` or `LpMaximize` must be entered, it is not important which. Similarly, the objective function can be anything, so in this example it is simply zero. That is we are trying to minimize zero, subject to our constraints (meeting the constraints being the important part)

Formulate the Constraints

These are simply the known constraints of a sudoku problem plus the constraints on our own created variables we have used to express the features of the problem:

- The values in the squares in any row must be each of 1 to 9
- The values in the squares in any column must be each of 1 to 9
- The values in the squares in any box must be each of 1 to 9 (a box is one of the 9 non-overlapping 3x3 grids within the overall 9x9 grid)
- There must be only one number within any square (seems logically obvious, but it is important to our formulation to ensure because of our variable choices)
- The starting sudoku numbers must be in those same places in the final solution (this is a constraint since these numbers are not changeable in the actual problem, whereas we can control any other numbers. If none or very few starting numbers were present, the sudoku problem would have a very large number of feasible solutions, instead of just one)

2.3.3 Solution

The introductory commenting and import statement are entered

```
"""
The Sudoku Problem Formulation for the PuLP Modeller

Authors: Antony Phillips, Dr Stuart Mitchell
"""

@# Import PuLP modeler functions
from pulp import *
```

In the unique case of the sudoku problem, the row names, column names and variable option values are all the exact same list of numbers (as strings) from “1” to “9”.

```
@# A list of strings from "1" to "9" is created
Sequence = ["1", "2", "3", "4", "5", "6", "7", "8", "9"]

@# The Vals, Rows and Cols sequences all follow this form
Vals = Sequence
Rows = Sequence
Cols = Sequence
```

A list called *Boxes* is created with 9 elements, each being another list. These 9 lists correspond to each of the 9 boxes, and each of the lists contains tuples as the elements with the row and column indices for each square in that box. Explicitly entering the values in a similar way to the following would have had the same effect (but would have been a waste of time):

```
@# The boxes list is created, with the row and column index of each square in each box
Boxes =[]
for i in range(3):
    for j in range(3):
        Boxes += [[(Rows[3*i+k],Cols[3*j+l]) for k in range(3) for l in range(3)]]
```

Therefore, Boxes[0] will return a list of tuples containing the locations of each of the 9 squares in the first box.

The prob variable is created to contain the problem data. LpMinimize has the same effect as LpMaximise in this case.

```
@# The prob variable is created to contain the problem data
prob = LpProblem("Sudoku Problem",LpMinimize)
```

The 729 problem variables are created since the *(Vals,Rows,Cols)* creates a variable for each combination of value, row and column. An example variable would be: Choice_4_2_9, and it is defined to be a binary variable (able to take only the integers 1 or 0. If Choice_4_2_9 was 1, it would mean the number 4 was present in the square situated in row 2, column 9. (If it was 0, it would mean there was not a 4 there)

```
@# The problem variables are created
choices = LpVariable.dicts("Choice",(Vals,Rows,Cols),0,1,LpInteger)
```

As explained above, the objective function (what we try to change using the problem variables) is simply zero (constant) since we are only concerned with any variable combination that can satisfy the constraints.

```
@# The arbitrary objective function is added
prob += 0, "Arbitrary Objective Function"
```

Since there are 9 variables for each square, it is important to specify that only exactly one of them can take the value of "1" (and the rest are "0"). Therefore, the below code reads: for each of the 81 squares, the sum of all the 9 variables (each representing a value that could be there) relating to that particular square must equal 1.

```
@# A constraint ensuring that only one value can be in each square is created
for r in Rows:
    for c in Cols:
        prob += lpSum([choices[v][r][c] for v in Vals]) == 1, ""
```

These constraints ensure that each number (value) can only occur once in each row, column and box.

```
@# The row, column and box constraints are added for each value
for v in Vals:
    for r in Rows:
        prob += lpSum([choices[v][r][c] for c in Cols]) == 1,""

    for c in Cols:
        prob += lpSum([choices[v][r][c] for r in Rows]) == 1,""

    for b in Boxes:
        prob += lpSum([choices[v][r][c] for (r,c) in b]) == 1,""
```

The starting numbers are entered as constraints i.e a "5" in row "1" column "1" is true.

```
@# The starting numbers are entered as constraints
prob += choices["5"]["1"]["1"] == 1,""
prob += choices["6"]["2"]["1"] == 1,""
prob += choices["8"]["4"]["1"] == 1,""
prob += choices["4"]["5"]["1"] == 1,""
prob += choices["7"]["6"]["1"] == 1,""
prob += choices["3"]["1"]["2"] == 1,""
prob += choices["9"]["3"]["2"] == 1,""
prob += choices["6"]["7"]["2"] == 1,""
prob += choices["8"]["3"]["3"] == 1,""
prob += choices["1"]["2"]["4"] == 1,""
prob += choices["8"]["5"]["4"] == 1,""
prob += choices["4"]["8"]["4"] == 1,""
prob += choices["7"]["1"]["5"] == 1,""
```

```
prob += choices["9"]["2"]["5"] == 1,""
prob += choices["6"]["4"]["5"] == 1,""
prob += choices["2"]["6"]["5"] == 1,""
prob += choices["1"]["8"]["5"] == 1,""
prob += choices["8"]["9"]["5"] == 1,""
prob += choices["5"]["2"]["6"] == 1,""
prob += choices["3"]["5"]["6"] == 1,""
prob += choices["9"]["8"]["6"] == 1,""
prob += choices["2"]["7"]["7"] == 1,""
prob += choices["6"]["3"]["8"] == 1,""
prob += choices["8"]["7"]["8"] == 1,""
prob += choices["7"]["9"]["8"] == 1,""
prob += choices["3"]["4"]["9"] == 1,""
prob += choices["1"]["5"]["9"] == 1,""
prob += choices["6"]["6"]["9"] == 1,""
prob += choices["5"]["8"]["9"] == 1,""
```

The problem is written to an LP file, solved using CPLEX (due to CPLEX's simple output) and the solution status is printed to the screen

```
## The problem data is written to an .lp file
prob.writeLP("Sudoku.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:", LpStatus[prob.status]
```

Instead of printing out all 729 of the binary problem variables and their respective values, it is most meaningful to draw the solution in a text file. The code also puts lines inbetween every third row and column to make the solution easier to read. The sudokuout.txt file is created in the same folder as the .py file.

```
## The solution is written to the sudokuout.txt file
for r in Rows:
    if r == "1" or r == "4" or r == "7":
                    sudokuout.write("+-------+-------+-------+\n")
    for c in Cols:
        for v in Vals:
            if value(choices[v][r][c])==1:

                if c == "1" or c == "4" or c =="7":
                    sudokuout.write("@textbar[] ")

                sudokuout.write(v + " ")

                if c == "9":
                    sudokuout.write("@textbar[]\n")
sudokuout.write("+-------+-------+-------+")
sudokuout.close()
```

A note of the location of the solution is printed to the solution

```
## The location of the solution is give to the user
print "Solution Written to sudokuout.txt"
```

The full file above is given provided Sudoku1.py

The final solution should be the following:

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

2.3.4 Extra for Experts

In the above formulation we did not consider the fact that there may be multiple solutions if the sudoku problem is not well defined.

We can make our code return all the solutions by editing our code as shown after the *prob.writeLP* line. Essentially we are just looping over the solve statement, and each time after a successful solve, adding a constraint that the same solution cannot be used again. When there are no more solutions, our program ends.

```
# A file called sudokuout.txt is created/overwritten for writing to
sudokuout = open('sudokuout.txt','w')

while True:
    prob.solve()
    # The status of the solution is printed to the screen
    print "Status:", LpStatus[prob.status]
    # The solution is printed if it was deemed "optimal" i.e met the constraints
    if LpStatus[prob.status] == "Optimal":
        # The solution is written to the sudokuout.txt file
        for r in Rows:
            if r == "1" or r == "4" or r == "7":
                sudokuout.write("+-------+-------+-------+\n")
            for c in Cols:
                for v in Vals:
```

```
                    if value(choices[v][r][c])==1:
                        if c == "1" or c == "4" or c =="7":
                            sudokuout.write("@textbar[] ")
                        sudokuout.write(v + " ")
                        if c == "9":
                            sudokuout.write("@textbar[]\n")
            sudokuout.write("+-------+-------+-------+\n\n")
            @# The constraint is added that the same solution cannot be returned again
            prob += lpSum([choices[v][r][c] for v in Vals
                                            for r in Rows
                                            for c in Cols
                                            if value(vars[v][r][c])==1]) @textless[]= 80
    @# If a new optimal solution cannot be found, we end the program
    else:
        break
sudokuout.close()

@# The location of the solutions is give to the user
print "Solutions Written to sudokuout.txt"
```

The full file using this is available Sudoku2.py. When using this code for sudoku problems with a large number of solutions, it could take a very long time to solve them all. To create sudoku problems with multiple solutions from unique solution sudoku problem, you can simply delete a starting number constraint. You may find that deleting several constraints will still lead to a single optimal solution but the removal of one particular constraint leads to a sudden dramatic increase in the number of solutions.

CHAPTER

THREE

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*

CHAPTER

FOUR

PULP INTERNAL DOCUMENTATION

4.1 pulp.constants

This file contains the constant definitions for PuLP Note that hopefully these will be changed into something more pythonic

isiterable(*obj*)

LpStatus

Return status from solver:

LpStatus key	string value	numerical value
LpStatusOptimal	"Optimal"	1
LpStatusNotSolved	"Not Solved"	0
LpStatusInfeasible	"Infeasible"	-1
LpStatusUnbounded	"Unbounded"	-2
LpStatusUndefined	"Undefined"	-3

LpStatusOptimal

LpStatusOptimal = 1

LpStatusNotSolved

LpStatusNotSolved = 0

LpStatusInfeasible

LpStatusInfeasible = -1

LpStatusUnbounded

LpStatusUnbounded = -2

LpStatusUndefined

LpStatusUndefined = -3

LpSenses

Dictionary of values for sense:

LpSenses = {LpMaximize:"Maximize", LpMinimize:"Minimize"}

LpMinimize

LpMinimize = 1

LpMaximize

LpMaximize = -1

LpConstraintEQ

LpConstraintEQ = 0

LpConstraintLE
LpConstraintLE = -1

LpConstraintGE
LpConstraintGE = 1

LpConstraintSenses

LpConstraint key	symbolic value	numerical value
LpConstraintEQ	"=="	0
LpConstraintLE	"<="	-1
LpConstraintGE	">="	1

4.2 pulp: Pulp classes

LpProblem LpVariable LpAffineExpression LpConstraint LpConstraint.makeElasticSubProblem FixedElasticSubProblem	

Todo

LpFractionConstraint, FractionElasticSubProblem

4.2.1 The LpProblem Class

class LpProblem(*name='NoName', sense=1*)
Bases: object

An LP Problem

Creates an LP Problem

This function creates a new LP Problem with the specified associated parameters

Parameters

- *name* – name of the problem used in the output .lp file
- *sense* – of the LP problem objective. Either LpMinimize (default) or LpMaximize.

Returns An LP Problem

Three important attributes of the problem are:

objective
The objective of the problem, an LpAffineExpression

constraints
An ordered dictionary of constraints of the problem - indexed by their names.

status
The return `status` of the problem from the solver.

Some of the more important methods:

solve(*solver=None, **kwargs*)
Solve the given Lp problem.

This function changes the problem to make it suitable for solving then calls the solver.actualSolve method to find the solution

Parameter *solver* – Optional: the specific solver to be used, defaults to the default solver.

Side Effects:

- The attributes of the problem object are changed in `actualSolve()` to reflect the Lp solution

roundSolution(*epsInt=1.0000000000000001e-05, eps=9.9999999999999995e-08*)
Rounds the lp variables

Inputs:

- none

Side Effects:

- The lp variables are rounded

setObjective(*obj*)
Sets the input variable as the objective function. Used in Columnwise Modelling

Parameter *obj* – the objective function of type `LpConstraintVar`

Side Effects:

- The objective function is set

writeLP(*filename, writeSOS=1, mip=1*)
Write the given Lp problem to a .lp file.

This function writes the specifications (objective function, constraints, variables) of the defined Lp problem to a file.

Parameter *filename* – the name of the file to be created.

Side Effects:

- The file is created.

4.2.2 Variables and Expressions

class LpElement(*name*)
Base class for LpVariable and LpConstraintVar

---

class LpVariable(*name, lowBound=None, upBound=None, cat='Continuous', e=None*)
This class models an LP Variable with the specified associated parameters

Parameters

- *name* – The name of the variable used in the output .lp file

- *lowbound* – The lower bound on this variable's range. Default is negative infinity
- *upBound* – The upper bound on this variable's range. Default is positive infinity
- *cat* – The category this variable is in, Integer, Binary or Continuous(default)
- *e* – Used for column based modelling: relates to the variable's existence in the objective function and constraints

addVariableToConstraints(*e*)
adds a variable to the constraints indicated by the LpConstraintVars in e

class **dicts**(*name, indexs, lowBound=None, upBound=None, cat=0, indexStart=, []*)
Creates a dictionary of LP variables

This function creates a dictionary of LP Variables with the specified associated parameters.

Parameters

- *name* – The prefix to the name of each LP variable created
- *indexs* – A list of strings of the keys to the dictionary of LP variables, and the main part of the variable name itself
- *lowbound* – The lower bound on these variables' range. Default is negative infinity
- *upBound* – The upper bound on these variables' range. Default is positive infinity
- *cat* – The category these variables are in, Integer or Continuous(default)

Returns A dictionary of LP Variables

setInitialValue(*val*)
sets the initial value of the Variable to val may of may not be supported by the solver

Example:

```
@textgreater[]@textgreater[]@textgreater[] x = LpVariable('x',lowBound = 0, cat='Continuous')
@textgreater[]@textgreater[]@textgreater[] y = LpVariable('y', upBound = 5, cat='Integer')
```

gives $x \in [0, \infty)$, $y \in (-\infty, 5]$, an integer.

---

class LpAffineExpression(*e=None, constant=0, name=None*)
Bases: `pulp.odict.OrderedDict`

A linear combination of `LpVariables`. Can be initialised with the following:

1.e = None: an empty Expression

2.e = dict: gives an expression with the values being the coefficients of the keys (order of terms is undetermined)

3.e = list or generator of 2-tuples: equivalent to dict.items()

4.e = LpElement: an expression of length 1 with the coefficient 1

5.e = other: the constant is initialised as e

Examples:

```
@textgreater[]@textgreater[]@textgreater[] f=LpAffineExpression(LpElement('x'))
@textgreater[]@textgreater[]@textgreater[] f
1*x + 0
@textgreater[]@textgreater[]@textgreater[] d = LpAffineExpression(dict(x_0=1, x_1=-3, x_2=4))
@textgreater[]@textgreater[]@textgreater[] d
1*x_0 + -3*x_1 + 4*x_2 + 0
@textgreater[]@textgreater[]@textgreater[] x_name = ['x_0', 'x_1', 'x_2']
@textgreater[]@textgreater[]@textgreater[] x = [LpVariable(x_name[i], lowBound = 0, upBound = 10
@textgreater[]@textgreater[]@textgreater[] c = LpAffineExpression([ (x[0],1), (x[1],-3), (x[2],4
@textgreater[]@textgreater[]@textgreater[] c
1*x_0 + -3*x_1 + 4*x_2 + 0
@textgreater[]@textgreater[]@textgreater[] c == d
1*x_0 + -3*x_1 + 4*x_2 + -1*x_0 + 3*x_1 + -4*x_2 + 0 = 0
@textgreater[]@textgreater[]@textgreater[] ( c == d) == LpConstraint(0)
@textless[]class 'pulp.pulp.LpConstraint'@textgreater[]
```

In brief, LpAffineExpression([(x[i],a[i]) for i in I]) $= \sum_{i \in I} a_i x_i$ where (note the order):

- `x[i]` is an `LpVariable`
- `a[i]` is a numerical coefficient.

---

lpSum(*vector*)
Calculate the sum of a list of linear expressions

Parameter *vector* – A list of linear expressions

4.2.3 Constraints

class LpConstraint(*e=None, sense=0, name=None, rhs=None*)
Bases: `pulp.pulp.LpAffineExpression`

An LP constraint

Parameters

- *e* – an instance of `LpAffineExpression`
- *sense* – one of `LpConstraintEQ`, `LpConstraintGE`, `LpConstraintLE` (0, 1, -1 respectively)
- *name* – identifying string
- *rhs* – numerical value of constraint target

makeElasticSubProblem(**args, **kwargs*)
Builds an elastic subproblem by adding variables to a hard constraint

uses FixedElasticSubProblem

Elastic Constraints

A constraint $C(x) = c$ (equality may be replaced by $\leq$ or $\geq$) can be elasticized to the form

$$C(x) \in D$$

where D denotes some interval containing the value c.

Define the constraint in two steps:

1. instantiate constraint (subclass of `LpConstraint`) with target c.
2. call its `makeElasticSubProblem()` method which returns an object of type `FixedElasticSubProblem` (subclass of `LpProblem`) - its objective is the minimization of the distance of $C(x)$ from D.

```
constraint = LpConstraint(..., rhs = c)
elasticProblem = constraint.makeElasticSubProblem(
                        penalty = <penalty_value>,
                        proportionFreeBound = <freebound_value>,
                        proportionFreeBoundList = <freebound_list_value>,
                        )
```

where:

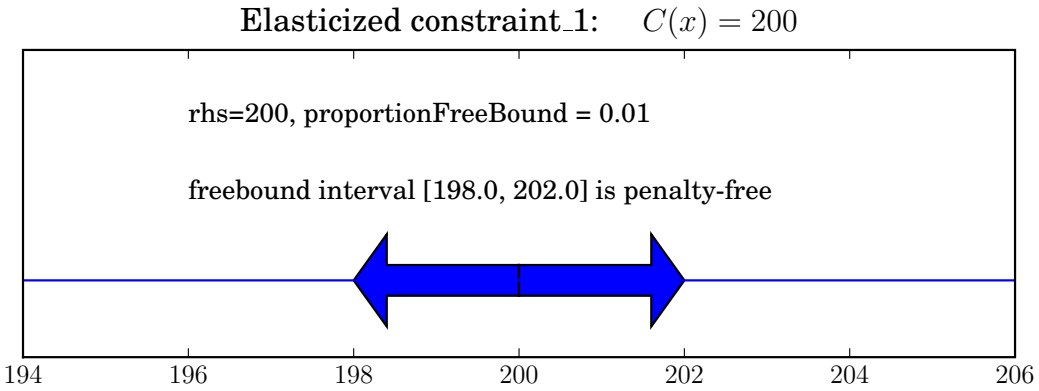
- `<penalty_value>` is a real number
- `<freebound_value>` $a \in [0, 1]$ specifies a symmetric target interval $D = (c(1-a), c(1+a))$ about c
- `<freebound_list_value> = [a,b]`, a list of proportions $a, b \in [0, 1]$ specifying an asymmetric target interval $D = (c(1-a), c(1+b))$ about c

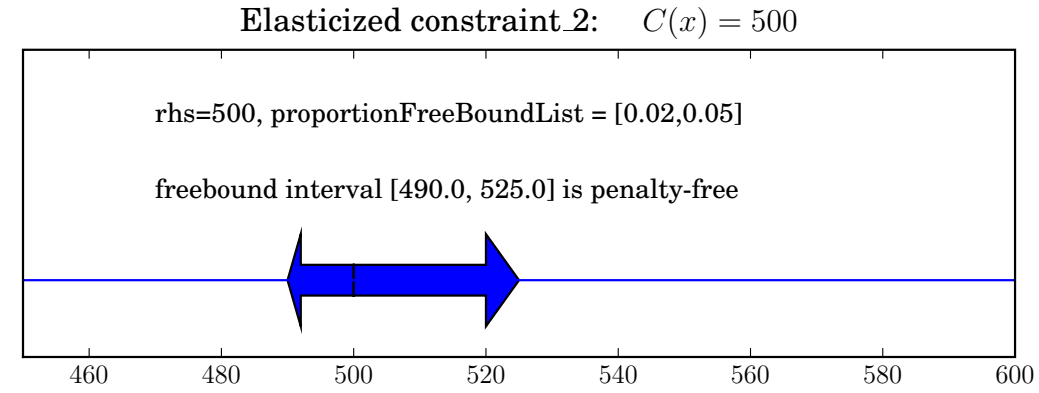
The penalty applies to the constraint at points x where $C(x) \notin D$. The magnitude of `<penalty_value>` can be assessed by examining the final objective function in the `.lp` file written by `LpProblem.writeLP()`.

Example:

```
@textgreater[]@textgreater[]@textgreater[] constraint_1 = LpConstraint('ex_1',sense=1,rhs=200)
@textgreater[]@textgreater[]@textgreater[] elasticProblem_1 = constraint_1.makeElasticSubproblem(pena
@textgreater[]@textgreater[]@textgreater[] constraint_2 = LpConstraint('ex_2',sense=0,rhs=500)
@textgreater[]@textgreater[]@textgreater[] elasticProblem_2 = constraint_2.makeElasticSubproblem(pena
proportionFreeBoundList = [0.02, 0.05])
```

1. constraint_1 has a penalty-free target interval of 1% either side of the rhs value, 200
2. constraint_2 has a penalty-free target interval of - 2% on left and 5% on the right side of the rhs value, 500





Following are the methods of the return-value:

class **FixedElasticSubProblem**(*constraint, penalty=None, proportionFreeBound=None, proportionFreeBoundList=None*)

Bases: `pulp.pulp.LpProblem`

Contains the subproblem generated by converting a fixed constraint $\sum_i a_i x_i = b$ into an elastic constraint.

Parameters

- *constraint* – The LpConstraint that the elastic constraint is based on
- *penalty* – penalty applied for violation (+ve or -ve) of the constraints
- *proportionFreeBound* – the proportional bound (+ve and -ve) on constraint violation that is free from penalty
- *proportionFreeBoundList* – the proportional bound on constraint violation that is free from penalty, expressed as a list where [-ve, +ve]

alterName(*name*)

Alters the name of anonymous parts of the problem

findDifferenceFromRHS()

The amount the actual value varies from the RHS (sense: LHS - RHS)

findLHSValue()

finds the LHS value of the constraint (without the free variable and/or penalty variable) - assumes the

constant is on the rhs

isViolated()
returns true if the penalty variables are non-zero

4.2.4 Combinations and Permutations

combination(*orgset, k=None*)
returns an iterator that lists the combinations of orgset of length k

Parameters

- *orgset* – the list to be iterated
- *k* – the cardinality of the subsets

Returns an iterator of the subsets

example:

```
@textgreater[]@textgreater[]@textgreater[] c = combination([1,2,3,4],2)
@textgreater[]@textgreater[]@textgreater[] for s in c:
...     print s
(1, 2)
(1, 3)
(1, 4)
(2, 3)
(2, 4)
(3, 4)
```

allcombinations(*orgset, k*)
returns all permutations of orgset with up to k items

Parameters

- *orgset* – the list to be iterated
- *k* – the maxcardinality of the subsets

Returns an iterator of the subsets

example:

```
@textgreater[]@textgreater[]@textgreater[] c = allcombinations([1,2,3,4],2)
@textgreater[]@textgreater[]@textgreater[] for s in c:
...     print s
(1,)
(2,)
(3,)
(4,)
(1, 2)
(1, 3)
(1, 4)
(2, 3)
(2, 4)
(3, 4)
```

permutation(*orgset, k=None*)
returns an iterator that lists the permutations of orgset of length k

Parameters

- *orgset* – the list to be iterated
- *k* – the cardinality of the subsets

Returns an iterator of the subsets

example:

```
@textgreater[]@textgreater[]@textgreater[] c = permutation([1,2,3,4],2)
@textgreater[]@textgreater[]@textgreater[] for s in c:
...     print s
(1, 2)
(1, 3)
(1, 4)
(2, 1)
(2, 3)
(2, 4)
(3, 1)
(3, 2)
(3, 4)
(4, 1)
(4, 2)
(4, 3)
```

allpermutations(*orgset, k*)

returns all permutations of orgset with up to k items

Parameters

- *orgset* – the list to be iterated
- *k* – the maxcardinality of the subsets

Returns an iterator of the subsets

example:

```
@textgreater[]@textgreater[]@textgreater[] c = allpermutations([1,2,3,4],2)
@textgreater[]@textgreater[]@textgreater[] for s in c:
...     print s
(1,)
(2,)
(3,)
(4,)
(1, 2)
(1, 3)
(1, 4)
(2, 1)
(2, 3)
(2, 4)
(3, 1)
(3, 2)
(3, 4)
(4, 1)
(4, 2)
(4, 3)
```

value(*x*)

Returns the value of the variable/expression x, or x if it is a number

4.3 pulp.solvers Interface to Solvers

This file contains the solver classes for PuLP Note that the solvers that require a compiled extension may not work in the current version

COIN
alias of COINMP_DLL

class COINMP_DLL(*mip=1, msg=1, cuts=1, presolve=1, dual=1, crash=0, scale=1, rounding=1, integerPresolve=1, strong=5, timeLimit=None, epgap=None*)
Bases: pulp.solvers.LpSolver

The COIN_MP LP MIP solver (via a DLL or linux so)

actualSolve(*lp*)
Solve a well formulated lp problem

class **available**()
True if the solver is available

copy()
Make a copy of self

getSolverVersion()
returns a solver version string

example: >>> COINMP_DLL().getSolverVersion() # doctest: +ELLIPSIS '...'

COINMP_DLL_load_dll(*path*)
function that loads the DLL useful for debugging installation problems

class COIN_CMD(*path=None, keepFiles=0, mip=1, msg=0, cuts=None, presolve=None, dual=None, strong=None, options=, [], fracGap=None, maxSeconds=None, threads=None*)
Bases: pulp.solvers.LpSolver_CMD

The COIN CLP/CBC LP solver now only uses cbc

actualSolve(*lp*)
Solve a well formulated lp problem

available()
True if the solver is available

copy()
Make a copy of self

defaultPath()

readsol_CBC(*filename, lp, vs*)
Read a CBC solution file

solve_CBC(*lp*)
Solve a MIP problem using CBC

CPLEX
alias of CPLEX_DLL

class CPLEX_CMD(*path=None, keepFiles=0, mip=1, msg=1, options=, []*)
Bases: pulp.solvers.LpSolver_CMD

The CPLEX LP solver

actualSolve(*lp*)
Solve a well formulated lp problem

available()
True if the solver is available

defaultPath()

readsol(*filename*)
Read a CPLEX solution file

class CPLEX_DLL(*mip=True, msg=True, timeLimit=None, epgap=None, logfilename=None, emphasizeMemory=False*)
Bases: pulp.solvers.LpSolver

The CPLEX LP/MIP solver (via a Dynamic library DLL - windows or SO - Linux)

This solver wraps the c library api of cplex. It has been tested against cplex 11. For api functions that have not been wrapped in this solver please use the ctypes library interface to the cplex api in CPLEX_DLL.lib

Initializes the CPLEX_DLL solver.

@param mip: if False the solver will solve a MIP as an LP @param msg: displays information from the solver to stdout @param epgap: sets the integer bound gap @param logfilename: sets the filename of the cplex logfile @param emphasizeMemory: makes the solver emphasize Memory over

solution time

actualResolve(*lp*)
looks at which variables have been modified and changes them

actualSolve(*lp*)
Solve a well formulated lp problem

available()
True if the solver is available

callSolver(*isMIP*)
Solves the problem with cplex

changeEpgap(*epgap=0.0001*)
Change cplex solver integer bound gap tolerence

findSolutionValues(*lp, numcols, numrows*)

getSparseCols(*vars, lp, offset=0, defBound=1e+20*)
outputs the variables in var as a sparse matrix, suitable for cplex and Coin

Copyright (c) Stuart Mitchell 2007

grabLicence()
Returns True if a CPLEX licence can be obtained. The licence is kept until releaseLicence() is called.

releaseLicence()
Release a previously obtained CPLEX licence

setMemoryEmphasis(*yesOrNo=False*)
Make cplex try to conserve memory at the expense of performance.

setTimeLimit(*timeLimit=0.0*)
Make cplex limit the time it takes –added CBM 8/28/09

setlogfile(*filename*)
sets the logfile for cplex output

CPLEX_DLL_load_dll(*path*)
function that loads the DLL useful for debugging installation problems

GLPK
alias of GLPK_CMD

class GLPK_CMD(*path=None, keepFiles=0, mip=1, msg=1, options=, []*)
Bases: pulp.solvers.LpSolver_CMD

The GLPK LP solver

actualSolve(*lp*)
Solve a well formulated lp problem

available()
True if the solver is available

defaultPath()

readsol(*filename*)
Read a GLPK solution file

class GUROBI(*mip=True, msg=True, timeLimit=None, epgap=None, **solverParams*)
Bases: pulp.solvers.LpSolver

The Gurobi LP/MIP solver (via its python interface)

The Gurobi variables are available (after a solve) in var.solverVar Constriaints in constraint.solverConstraint and the Model is in prob.solverModel

Initializes the Gurobi solver.

@param mip: if False the solver will solve a MIP as an LP @param msg: displays information from the solver to stdout @param timeLimit: sets the maximum time for solution @param epgap: sets the integer bound gap

actualResolve(*lp, callback=None*)
Solve a well formulated lp problem

uses the old solver and modifies the rhs of the modified constraints

actualSolve(*lp, callback=None*)
Solve a well formulated lp problem

creates a gurobi model, variables and constraints and attaches them to the lp model which it then solves

available()
True if the solver is available

buildSolverModel(*lp*)
Takes the pulp lp model and translates it into a gurobi model

callSolver(*lp, callback=None*)
Solves the problem with gurobi

findSolutionValues(*lp*)

class LpSolver(*mip=True, msg=True, options=, [], *args, **kwargs*)
A generic LP Solver

actualResolve(*lp, **kwargs*)
uses existing problem information and solves the problem If it is not implelemented in the solver just solve again

actualSolve(*lp*)
Solve a well formulated lp problem

available()
True if the solver is available

copy()
Make a copy of self

getCplexStyleArrays(*lp, senseDict={0: 'E', 1: 'G', -1: 'L'}, LpVarCategories={'Integer': 'I', 'Continuous': 'C'}, LpObjSenses={1: 1, -1: -1}, infBound=1e+20*)
returns the arrays suitable to pass to a cdll Cplex or other solvers that are similar

Copyright (c) Stuart Mitchell 2007

solve(*lp*)
Solve the problem lp

class LpSolver_CMD(*path=None, keepFiles=0, mip=1, msg=1, options=, []*)
Bases: pulp.solvers.LpSolver

A generic command line LP Solver

copy()
Make a copy of self

defaultPath()

static **executable**(*command*)
Checks that the solver command is executable, And returns the actual path to it.

static **executableExtension**(*name*)

setTmpDir()
Set the tmpDir attribute to a reasonnable location for a temporary directory

exception PulpSolverError
Bases: exceptions.Exception

Pulp Solver-related exceptions

class XPRESS(*path=None, keepFiles=0, mip=1, msg=1, options=, []*)
Bases: pulp.solvers.LpSolver_CMD

The XPRESS LP solver

actualSolve(*lp*)
Solve a well formulated lp problem

available()
True if the solver is available

defaultPath()

readsol(*filename*)
Read an XPRESS solution file

ctypesArrayFill(*myList, type=<class 'ctypes.c_double'>*)
Creates a c array with ctypes from a python list type is the type of the c array

initialize(*filename*)
reads the configuration file to initialise the module

CHAPTER

FIVE

AUTHORS

The authors of this documentation (the pulp documentation team) include:

- Stuart Mitchell (s.mitchell@auckland.ac.nz)
- Anita Kean
- Andrew Mason
- Michael O'Sullivan
- Antony Phillips

CHAPTER

SIX

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*

MODULE INDEX

P

INDEX