sphinx Documentation

Release 0.1

anita kean

November 19, 2009

CONTENTS

Sphinx was released in early 2008 as a tool to process the python documentation

- It renders text with relatively little markup into html/pdf
- `LaTeX`'s math-mode is available.
- It renders code with language-specific highlighting (defaults to python)
- provides most of the features of html and latex markup:
  - linking
  - referencing (sections, arbitrary locations)
  - indexing (terms, objects)
  - navigation (sections, optionally numbered)
  - inclusion (images, tables, text, code, objects)
  - exclusion (tag-based, e.g. for html/pdf)
  - math-mode (e.g. $e^{i\theta} = \cos(\theta) + i\sin(\theta)$)
  - citation
  - footnotes
- By default, has a `Show Source` link in its Table-of-Contents panel where you can *view un-rendered text*.
- It is being actively developed.
- It has an active Usenet newsgroup sphinx-dev.
- It is used by many projects now - you can learn from these.

The code depends on

- `docutils` (restructured text parser),
- `jinja2` (templating tool) and
- `pygments` (highlighter).
- **latex** (fonts, mathematical formatting) - with the texlive package you'll need `latex-recommended`, `latex-extra` and `fonts-recommended`.
  - see the sphinx extension docs for how to use ReportLab's `rst2pdf` instead.

The markup extends restructured text of `docutils` - adding more directives and roles, and linking across documents. As with ReST, text is white-space sensitive.

It has a set of extensions which includes:

autodoc extract docstrings from documented code

intersphinx link documentation to other python documentation

doctest run docstrings through doctest

linkcheck check all hyperlinks in document tree

CHAPTER

ONE

TABLE OF CONTENTS

1.1 General Sphinx usage

With a few exceptions, `sphinx` markup is a superset of docutils restructured text.

In your sphinx installation, the doc directory contains sphinx-generated html as well as the source rst files. You can regenerate html or pdf with the *sphinx-build* command.

This documentation contains an introduction to restructured text markup, but you will probably need to consult some more detailed `docutils` ReSt documentation (e.g. `docutils` ReSt quickref)

Some initial pointers:

Note:

- tab-indentations in re-structured text are three spaces
- text enclosed in single back-ticks is *interpreted*
- text enclosed in double back-ticks is simply emphasized

1.1.1 A brief tour

getting started

A sphinx installation provides you with two command-line tools:

sphinx-quickstart initiate a sphinx project:

sphinx-build build the documentation (to html or pdf) from rst

sphinx-quickstart

At the command-line, type:

```
$ sphinx-quickstart
```

for an interactive program which queries setup options and writes out a file *conf.py* in the root directory of your project. Edit this file by hand afterwards to change configuration as you prefer.

I suggest accepting default locations.

Filenames referenced throughout the documentation are either relative to the current file (e.g. `subdirectory/filename`, `../../filename`), or relative to this root directory (`/root/subdirectory/../filename`)

You can elect to have sphinx install a `Makefile` (linux) or `make.bat` (MS) which provides common invocations of **sphinx-build**.

Extensions are offered in this setup - you can elect to include these now, or do so later [1] :

- autodoc - to pull in docstrings from code
- intersphinx - to link with other python documentation
- pngmath or jsmath - to enable latex math-mode in html

sphinx-build

```
$ man sphinx_build
```

shows **sphinx-build** *usage*.

```
$ sphinx-build  -b html -d build/doctrees source build/html

$ sphinx-build  -b latex -d build/doctrees source build/latex

$ sphinx-build  -b doctest -d build/doctrees source
```

or use the convenience of **make** and the *Makefile* written by **sphinx-quickstart**:

```
$ make help                      (to see all possibilities)
$ make html                      (to render docs as html)
Build finished. The HTML pages are in build/html.

$ browse build/html/index.html   (to view the html-docs)


$ make latex                     (to render docs as tex)
$ cd build/latex
$ make all-pdf                   (to process tex file to pdf)
```

for the pdf, or

```
$ make doctest
```

to run doctest on the docstrings of your documented code.

In addition, you'll see from the **make help** that you can also run:

```
$ make linkcheck
```

to have sphinx check all the links in your documents.

[1] by editing the `extensions` variable in the file `conf.py`

Footnote

root-directory location of the file `conf.py` from the **sphinx-quickstart** output.

some markup features

What follows is a mixture of restructured text and sphinx markup to illustrate some useful features. Refer to the docutils and sphinx documentation for full resources.

We start out by mentioning a couple of ReST markups which sphinx uses. Both these are in paragraph-mode - they occur as a blank line followed by a line at the same indentation as the previous non-blank line, starting with two periods. They are

comment text following is ignored till the occurence of the next blank line:

```
..  commented text here
```

link label: this sets up a link to this location which is able to be referenced by the `:ref:` mode below:

```
.. _link_label_here:
```

The label to reference is the text `link_label_here`.

The form of comments and link labels is similar to but different from the whole family of directives. They follow.

directives /paragraph mode): - a blank line followed by a line at the same indentation as the previous non-blank line, beginning with:

```
.. directive_name_here::
```

(Each is followed by *two* colons.)

There are many directive names, including: `code-block` `image` `include` `index` `literalinclude` `math` `note` `only` `seealso` `table` `warning`

Each of these performs an operation on the argument that follows. There may be modifying arguments on subsequent indented lines (before any blank lines) which take the form:

```
:modification: modification_argument
```

See for example the form of equation-numbered *math-equations* and *csv-tables*.

roles (inline mode): - of the form:

```
:role_name_here:`text to be interpreted here`
```

Role names include:

math inline math-mode,

`obj`, `meth`, `class`, ... will reference any documented python objects see the source of *Pulp example* for an illustration

ref will reference any labelled location (of the form `..  _link_label_here:`)

literal blocks

Sphinx interprets anything following the form `::` `<newline>` as pre-formatted text:

```
Text in a literal block::

   The indented paragraph following this is highlighted and presented as
   is.
```

results in :

Text in a literal block:

```
The indented paragraph following this is highlighted and presented as
is.
```

(Notice one of the colons has been removed.)

An isolated *::* functions similarly - both colons removed in output.

code

Code-blocks are highlighted when they follow a `..` `code-block::` directive, where a trailing argument can specify the language.

The following python code is thus highlighted simply by appearing in an indented region preceded by `::`:

```
#!/usr/bin/env python
# -*- encoding: utf-8 -*-
"""Fibonacci sequences using generators

This program is part of "Dive Into Python", a free Python book for
experienced programmers.  Visit http://diveintopython.org/ for the
latest version.
"""


def fibonacci(max):
    a, b = 0, 1
    while a < max:
        yield a
        a, b = b, a+b

for n in fibonacci(1000):
    print n,
```

See Also:

literal includes for including code from files, numbering it and including code selectively.

Using the `..` `code-block::` `<language here>` directive, code will be appropriately presented. So:

```
.. code-block:: sh

   echo $PATH
   for i in $(seq 1 10);
      do
```

```
        echo $$( 2*i+3 ))
    done;
```

will be highlighted as shell-code:

```
echo $PATH
for i in $(seq 1 10);
   do
      echo $(( 2*i+3 ))
   done;
```

inclusion

Text in files may be included into the current text either in-line, or pre-formatted.

in-line inclusion The `..  include::` directive causes in-line inclusion:

```
.. include:: ../../src/some_file.txt
```

pre-formatted inclusion to include the file pre-formatted (e.g. quotes or code):

```
.. literalinclude:: /src/fib.py
```

This produces:

```
#!/usr/bin/env python
# -*- encoding: utf-8 -*-
"""Fibonacci sequences using generators

This program is part of "Dive Into Python", a free Python book for
experienced programmers.  Visit http://diveintopython.org/ for the
latest version.
"""

def fibonacci(max):
    a, b = 0, 1
    while a < max:
        yield a
        a, b = b, a+b

for n in fibonacci(1000):
    print n,
```

(the root directory which contains the file `conf.py` also contains `src`, so the file is named absolutely - relative to that directory)

To see that file with line-numbers, add the *:linenos:* role:

```
.. literalinclude:: ../../src/fib.py
   :linenos:
```

which gives:

```
1  #!/usr/bin/env python
2  # -*- encoding: utf-8 -*-
3  """Fibonacci sequences using generators
4
5  This program is part of "Dive Into Python", a free Python book for
6  experienced programmers.  Visit http://diveintopython.org/ for the
7  latest version.
8  """
9
10 def fibonacci(max):
11     a, b = 0, 1
12     while a < max:
13         yield a
14         a, b = b, a+b
15
16 for n in fibonacci(1000):
17     print n,
```

To extract just lines 16, 17 of this file, add the `:lines:` role:

```
.. literalinclude:: ../../src/fib.py
   :lines: 16-17
```

to get

```
for n in fibonacci(1000):
    print n,
```

We can also include specific functions from python code with the `:pyobject:` role:

```
.. literalinclude:: /src/fib.py
   :pyobject: fibonacci
```

```
def fibonacci(max):
    a, b = 0, 1
    while a < max:
        yield a
        a, b = b, a+b
```

links

www-links Sphinx renders text of the form `http://www...` as a web-link without it requiring any markup. Alternate text appears for the link when of the form (note the trailing underscore - rst-format):

```
`Alternate text <http://www.google.com>`_
```

cross-references Sphinx extends restructured text in linking between documents. Links to other parts of the document tree are of two forms:

1. section links, whose label precedes the section title

```
.. _label_here:

section title
-------------
```

This creates a hyperlink target (label) at this section title.

We then link to the section title in this or other documents with text of the form

```
:ref:`<label_here>`
```

A label different from the section title may be inserted too, with the form:

```
:ref:`alternate title <label_here>`
```

A link to the *literal blocks* section is created in this way.

2. links to arbitrary text:

   Any text other than a section title (or figure caption) can be referenced similarly, except that it *must* have an explicit title:

   ```
   :ref:`label name here <label_here>`
   ```

indexing

Section titles are not automatically indexed. Preceding any line with an `index directive` of the form:

```
.. index:: item1, item2, item3, ...
```

inserts index entries for these items, which link back to this position.

An `index directive` of the form:

```
.. index:: pair: itemA; indexB
```

inserts two entries of the form:

itemA itemB

and

itemB itemA

tables

Tables are entered literally:

```
=====  ===== ============
  x    y     x implies y
=====  ===== ============
True   True  True
True   False False
False  True  True
False  False True
=====  ===== ============
```

yield

x	y	x implies y
True	True	True
True	False	False
False	True	True
False	False	True

A simple list-entry table is generated with:

```
.. list-table:: Caption
   :header-rows: 1

   *  - Left
      - Right
   *  - 1
      - 2
```

which produces :

Table 1.1: Caption

Left	Right
1	2

images

The directive is:

```
.. image:: filename.type
```

Given that the pdflatex-builder will prefer a pdf and the html-builder an image-format, using the form:

```
.. image:: filename.*
```

allows builders to select their preferred format, if available.

As with other inclusions, path name is relative to the current directory or to the *root directory*:

```
.. image:: path/to/filename.type
   or
.. image:: /path/to/filename.type
```

See the page *Pulp case study* for an example.

math (latex)

inline As with other markup, there is a directive and a mode:

to insert $c = \pm\sqrt{a^2 + b^2}$ in your code, just prepend it with the :math: mode tag:

```
to insert :math:`c= \pm \sqrt{a^2+b^2}` in your code, ...
```

paragraph-mode The quadratic formula gives in general, two solutions to the equation $ax^2 + bx + c = 0$:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{1.1}$$

in your code, just prepend it with the `..math::` directive:

```
.. math::  x = \frac{-b\pm \sqrt{b^2-4ac}}{2a}
   :label:  quadratic
```

The `:label:` option creates a label and allows us to refer to the equation elsewhere (see (1.1)).

See Also:

The *case study in pulp*

footnotes

Footnote labels are established with references enclosed in square brackets and trailing underscore::

```
place a footnote here [1]_ please.
```

The footnote label is either a number or a `#` (hash). Occurrences in the text of `[#]_` or `[3]_` link to the set of footnotes found in following incantation of

```
.. rubric:: Footnotes
.. [#] first footnote here
.. [3] third footnote
```

citations

These are of the same form as footnotes, but are any text except hashes and numbers. They can be referenced from any file in the document tree. See the sphinx documentation for more details (or the source of this page)[Ref]_

1.2 Documenting Python

In this set of pages, the `autodoc`, `autosphinx` and `pngmath` sphinx extensions have been enabled, so that docstrings from code are pulled in to restructured-text pages, and with the `:show-inheritance:` switch, documentation of classes that current classes inherit from is also accessible. A regular **sphinx-build** incantation builds in this linked documentation.

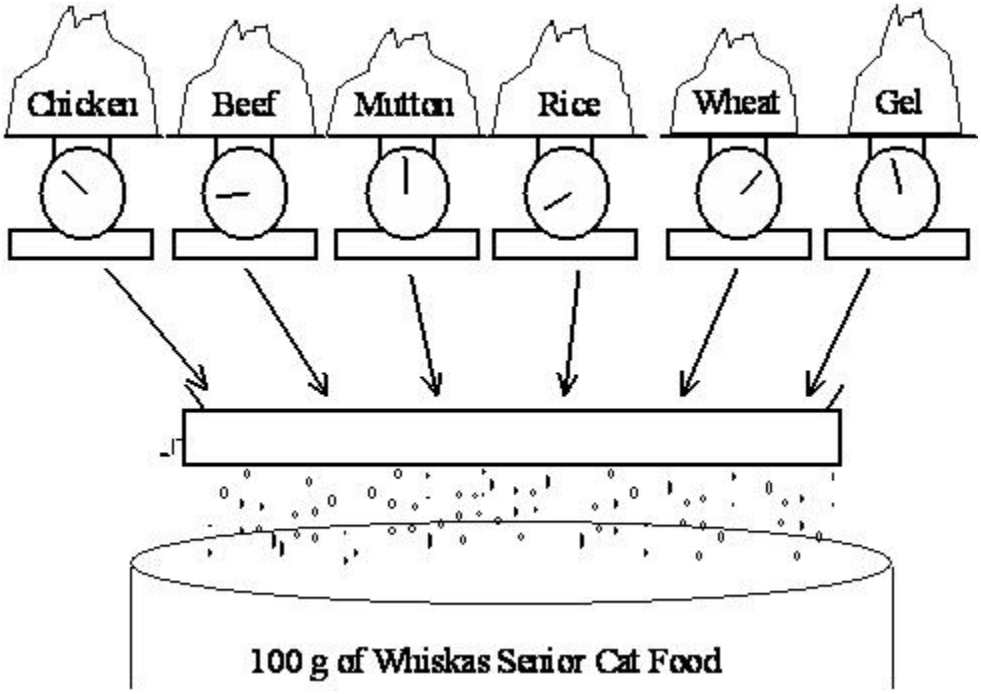
Follow the links to python objects to see their docstrings. Using the *current* version of sphinx (0.6.3), only class attributes are documented, but instance attribute documentation has recently been added to the development code (version 1.0.0).

1.2.1 Referencing python code

A draft-article which links to python code, and allows access to code documentation.

Pulp Case Study

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.

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:

$$1.000x_1 + 1.000x_2 = 100.0$$
$$0.100x_1 + 0.200x_2 \geq 8.0$$
$$0.080x_1 + 0.100x_2 \geq 6.0$$
$$0.001x_1 + 0.005x_2 \leq 2.0$$
$$0.002x_1 + 0.005x_2 \leq 0.4$$

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-block for this example is found in invalid url

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-block:

```
# 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` function. 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)
```

```
# The 2 variables Beef and Chicken are created with a lower limit of zero
x1 = LpVariable("ChickenPercent",0)
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 >= 8.0, "ProteinRequirement"
prob += 0.080 * x1 + 0.100 * x2 >= 6.0, "FatRequirement"
prob += 0.001 * x1 + 0.005 * x2 <= 2.0, "FibreRequirement"
prob += 0.002 * x1 + 0.005 * x2 <= 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-block runs successfully, you can open this .lp file with Notepad 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 a .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.status` (`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.

```
for variable in prob.variables():
    print variable.name, "=", variable.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.

The segmented .py file shown above can be obtained in full. Opening the file normally will run it, which will execute the commands and exit in less than 1 second. You will need to open this file with IDLE or !PyDev to run it and view the output properly.

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. In STATS 255, the decisions were the amount in g in each cat food, but it is more convenient to use percentages. 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 $0.013 x_1 + $0.008 x_2 + $0.010 x_3 + $0.002 x_4 + $0.005 x_5 + $0.001 x_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:

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

4. Identify the Data We know the total weight of the can. We also know the cost of the ingredients, the nutritional analysis requirements and the contribution (per gram) of each ingredient in terms of the nutritional analysis. This is enough to formulate the necessary mathematical programme, but we will reconsider our data after solving the mathematical programme and performing some post-optimal analysis.

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-block for this example is found in the downloadable file.

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 fibre 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
```

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]) >= 8.0 "ProteinRequireme
prob += lpSum([fatPercent[i] * ingredient_vars[i] for i in Ingredients]) >= 6.0 "FatRequirement"
prob += lpSum([fibrePercent[i] * ingredient_vars[i] for i in Ingredients]) <= 2.0, "FibreRequirement"
prob += lpSum([saltPercent[i] * ingredient_vars[i] for i in Ingredients <= 0.4, "Salt Requirement"
```

Following this, the *writeLP* line etc follow exactly the same as in the simplified example. To see the entire file, it is available here

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

Post-Optimal Analysis

For The Whiskas Cat Food Problem we will not perform any post-optimal analysis. However, as you will remember from STATS255, a Sensitivity Analysis and a Parametric Analysis can be performed.

1. Validation To validate our solution for The Whiskas Cat Food Problem, we can do a quick check that our solution makes sense. First, the percentages should add up to 100%. If not, there is something wrong. Next, we can do a quick check of the constraints to ensure none of them are violated.

Note: For large models you won't always be able to check the solution by hand.

The final validation is to write up a management summary for your manager and/or client and see if they think our solution is a valid one. If they identify some (or all) of the solution that is not valid, then you should discuss with them the reasons why it is invalid and start a "feedback" loop in your optimisation process.

Presentation of Solution and Analysis

Note: The solution for The Whiskas Cat Food Problem is a simple one to summarise. Here is an example management summary (with the numbers removed):

The Whiskas Cat Food Problem Stuart Mitchell, 2007

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 stated nutritional analysis requirements are: _____ The cost for each ingredient is: _____

To minimise the cost per 100g can of Whiskas cat food, Whiskas should use _____ g of Chicken, _____ g of Beef,

The cost per can is _____

Warning: IMPORTANT When presenting your solution you must be careful about the number of decimal places you use. You should not use a greater accuracy than your data allows.

Implementation and Ongoing Monitoring

The first step towards Implementation is a good management summary. You may also want to discuss any issues that may arise from the solution you have found with Uncle Ben's (for example, if your solution can be implemented on their production line).

Ongoing Monitoring may take the form of:

- Updating your data files and resolving as the data changes (changing costs, nutritional requirements, etc);
- Resolving our model for new products (e.g., 200g cans);
- Looking for possible savings (e.g., analysing the cost of the ingredients to see if any price changes will affect the ingredient mix).

1.2.2 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

1.2.3 pulp.pulp

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

Bases: object

An LP Problem

Three important attributes of the problem are:

objective
The objective of the problem.

constraints
An ordered dictionary of constraints(of type LpConstraint) 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.

Returns status of solution (pulp.constants.LpStatus)

Side Effects:

- The attributes of the problem object are changed in solver.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.

Inputs:

- filename – the name of the file to be created.

Side Effects:

- The file is created.

__init__ (*name='NoName', sense=1*)
Creates an LP Problem This function creates a new LP Problem with the specified associated parameters :param name: The name of the problem used in the output .lp file :param sense: (optional) The LP problem objective: `pulp.constants.LpMinimize(default)` or `pulp.constants:LpMaximise` :returns: An LP Problem

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

class LpVariable (*name, lowBound=None, upBound=None, cat='Continuous', e=None*)
Bases: `pulp.pulp.LpElement`

A LP variable

__init__ (*name, [lowBound = None, upBound = None, cat = LpContinuous, e = None]*)
The constructor for this class is :

__init__(name [, lowBound = None, upBound = None, cat = LpContinuous, e = None]):()

Creates an LP variable.

Parameters

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

Returns An LP variable

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

A linear combination of LP variables

An LpAffine Expression can be initalised with the following e = None: an empty Expression e = dict: gives an expression with the values being the coefficents of the keys e = list: equivalent to dict.items() e = LpElement: an expression of length 1 with the coefficent 1 e = other: the constant is intialised as e

__init__ (*e = None, constant = 0, name = None*)

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

An LP constraint

__init__(*self, e = None, sense = LpConstraintEQ, name = None, rhs = None*)

Parameter *sense* – one of {pulp.constants.LpConstraintEQ, Lpulp.constants.pConstraintGE, pulp.constants.LpConstraintLE`}

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

Bases: pulp.pulp.LpProblem

Contains the subproblem generated by converting an fixed constraint

$$\sum_i a_i x_i = b$$

into an elastic constraint

Parameters

- *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]

__init__(*self, e = None, sense = LpConstraintEQ, name = None, rhs = None*)

lpSum(*vector*)

Calculate the sum of a list of linear expressions

Inputs:

- vector – A list of linear expressions

1.2.4 pulp.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 windows only)

actualSolve(*lp*)

Solve a well formulated lp problem

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 '...'

class COIN_CMD (*path=None, keepFiles=0, mip=1, msg=1, cuts=1, presolve=1, dual=1, strong=5, options=, []*)
Bases: pulp.solvers.LpSolver_CMD

The COIN CLP/CBC LP solver

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

readsol_CLP (*filename, lp, vs*)
Read a CLP solution file

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

solve_CLP (*lp*)
Solve a problem using CLP

CPLEX
alias of CPLEX_CMD

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, options=, [], *args, **kwargs*)
Bases: pulp.solvers.LpSolver

The CPLEX LP/MIP solver PHANTOM Something went wrong!!!!

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

available ()
True if the solver is available

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, options=, [], *args, **kwargs*)
Bases: pulp.solvers.LpSolver

The GUROBI LP/MIP solver PHANTOM Something went wrong!!!!

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

available()
True if the solver is available

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 (*file=None*)
reads the configuration file to initialise the module

1.2.5 pulp.odict

A dict that keeps keys in insertion order

class OrderedDict (*init_val=(), strict=False*)
Bases: dict

A class of dictionary that keeps the insertion order of keys.

All appropriate methods return keys, items, or values in an ordered way.

All normal dictionary methods are available. Update and comparison is restricted to other OrderedDict objects.

Various sequence methods are available, including the ability to explicitly mutate the key ordering.

__contains__ tests:

```
>>> d = OrderedDict(((1, 3),))
>>> 1 in d
1
>>> 4 in d
0
```

__getitem__ tests:

```
>>> OrderedDict(((1, 3), (3, 2), (2, 1)))[2]
1
>>> OrderedDict(((1, 3), (3, 2), (2, 1)))[4]
Traceback (most recent call last):
KeyError: 4
```

__len__ tests:

```
>>> len(OrderedDict())
0
>>> len(OrderedDict(((1, 3), (3, 2), (2, 1))))
3
```

get tests:

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.get(1)
3
>>> d.get(4) is None
1
>>> d.get(4, 5)
5
>>> d
OrderedDict([(1, 3), (3, 2), (2, 1)])
```

has_key tests:

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.has_key(1)
1
>>> d.has_key(4)
0
```

clear()

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.clear()
>>> d
OrderedDict([])
```

copy()

```
>>> OrderedDict(((1, 3), (3, 2), (2, 1))).copy()
OrderedDict([(1, 3), (3, 2), (2, 1)])
```

index(*key*)

Return the position of the specified key in the OrderedDict.

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.index(3)
1
>>> d.index(4)
Traceback (most recent call last):
ValueError: list.index(x): x not in list
```

insert(*index, key, value*)

Takes `index`, `key`, and `value` as arguments.

Sets `key` to `value`, so that `key` is at position `index` in the OrderedDict.

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.insert(0, 4, 0)
>>> d
OrderedDict([(4, 0), (1, 3), (3, 2), (2, 1)])
>>> d.insert(0, 2, 1)
```

```
>>> d
OrderedDict([(2, 1), (4, 0), (1, 3), (3, 2)])
>>> d.insert(8, 8, 1)
>>> d
OrderedDict([(2, 1), (4, 0), (1, 3), (3, 2), (8, 1)])
```

items()
items returns a list of tuples representing all the `(key, value)` pairs in the dictionary.

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.items()
[(1, 3), (3, 2), (2, 1)]
>>> d.clear()
>>> d.items()
[]
```

iteritems()

```
>>> ii = OrderedDict(((1, 3), (3, 2), (2, 1))).iteritems()
>>> ii.next()
(1, 3)
>>> ii.next()
(3, 2)
>>> ii.next()
(2, 1)
>>> ii.next()
Traceback (most recent call last):
StopIteration
```

iterkeys()

```
>>> ii = OrderedDict(((1, 3), (3, 2), (2, 1))).iterkeys()
>>> ii.next()
1
>>> ii.next()
3
>>> ii.next()
2
>>> ii.next()
Traceback (most recent call last):
StopIteration
```

itervalues()

```
>>> iv = OrderedDict(((1, 3), (3, 2), (2, 1))).itervalues()
>>> iv.next()
3
>>> iv.next()
2
>>> iv.next()
1
>>> iv.next()
Traceback (most recent call last):
StopIteration
```

keys()
Return a list of keys in the `OrderedDict`.

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.keys()
[1, 3, 2]
```

pop(*key, *args*)
No dict.pop in Python 2.2, gotta reimplement it

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.pop(3)
2
>>> d
OrderedDict([(1, 3), (2, 1)])
>>> d.pop(4)
Traceback (most recent call last):
KeyError: 4
>>> d.pop(4, 0)
0
>>> d.pop(4, 0, 1)
Traceback (most recent call last):
TypeError: pop expected at most 2 arguments, got 3
```

popitem(*i=-1*)
Delete and return an item specified by index, not a random one as in dict. The index is -1 by default (the last item).

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.popitem()
(2, 1)
>>> d
OrderedDict([(1, 3), (3, 2)])
>>> d.popitem(0)
(1, 3)
>>> OrderedDict().popitem()
Traceback (most recent call last):
KeyError: 'popitem(): dictionary is empty'
>>> d.popitem(2)
Traceback (most recent call last):
IndexError: popitem(): index 2 not valid
```

rename(*old_key, new_key*)
Rename the key for a given value, without modifying sequence order.

For the case where new_key already exists this raise an exception, since if new_key exists, it is ambiguous as to what happens to the associated values, and the position of new_key in the sequence.

```
>>> od = OrderedDict()
>>> od['a'] = 1
>>> od['b'] = 2
>>> od.items()
[('a', 1), ('b', 2)]
>>> od.rename('b', 'c')
>>> od.items()
[('a', 1), ('c', 2)]
>>> od.rename('c', 'a')
Traceback (most recent call last):
ValueError: New key already exists: 'a'
```

```
>>> od.rename('d', 'b')
Traceback (most recent call last):
KeyError: 'd'
```

reverse()
Reverse the order of the OrderedDict.

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.reverse()
>>> d
OrderedDict([(2, 1), (3, 2), (1, 3)])
```

setdefault(*key, defval=None*)

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.setdefault(1)
3
>>> d.setdefault(4) is None
True
>>> d
OrderedDict([(1, 3), (3, 2), (2, 1), (4, None)])
>>> d.setdefault(5, 0)
0
>>> d
OrderedDict([(1, 3), (3, 2), (2, 1), (4, None), (5, 0)])
```

setitems(*items*)
This method allows you to set the items in the dict.

It takes a list of tuples - of the same sort returned by the `items` method.

```
>>> d = OrderedDict()
>>> d.setitems(((3, 1), (2, 3), (1, 2)))
>>> d
OrderedDict([(3, 1), (2, 3), (1, 2)])
```

setkeys(*keys*)
`setkeys` all ows you to pass in a new list of keys which will replace the current set. This must contain the same set of keys, but need not be in the same order.

If you pass in new keys that don't match, a `KeyError` will be raised.

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.keys()
[1, 3, 2]
>>> d.setkeys((1, 2, 3))
>>> d
OrderedDict([(1, 3), (2, 1), (3, 2)])
>>> d.setkeys(['a', 'b', 'c'])
Traceback (most recent call last):
KeyError: 'Keylist is not the same as current keylist.'
```

setvalues(*values*)
You can pass in a list of values, which will replace the current list. The value list must be the same len as the OrderedDict.

(Or a `ValueError` is raised.)

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.setvalues((1, 2, 3))
>>> d
OrderedDict([(1, 1), (3, 2), (2, 3)])
>>> d.setvalues([6])
Traceback (most recent call last):
ValueError: Value list is not the same length as the OrderedDict.
```

sort(**args, **kwargs*)

Sort the key order in the OrderedDict.

This method takes the same arguments as the `list.sort` method on your version of Python.

```
>>> d = OrderedDict(((4, 1), (2, 2), (3, 3), (1, 4)))
>>> d.sort()
>>> d
OrderedDict([(1, 4), (2, 2), (3, 3), (4, 1)])
```

update(*from_od*)

Update from another OrderedDict or sequence of (key, value) pairs

```
>>> d = OrderedDict(((1, 0), (0, 1)))
>>> d.update(OrderedDict(((1, 3), (3, 2), (2, 1))))
>>> d
OrderedDict([(1, 3), (0, 1), (3, 2), (2, 1)])
>>> d.update({4: 4})
Traceback (most recent call last):
TypeError: undefined order, cannot get items from dict
>>> d.update((4, 4))
Traceback (most recent call last):
TypeError: cannot convert dictionary update sequence element "4" to a 2-item sequence
```

values(*values=None*)

Return a list of all the values in the OrderedDict.

Optionally you can pass in a list of values, which will replace the current list. The value list must be the same len as the OrderedDict.

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.values()
[3, 2, 1]
```

class SequenceOrderedDict(*init_val=(), strict=True*)

Bases: `pulp.odict.OrderedDict`

Experimental version of OrderedDict that has a custom object for `keys`, `values`, and `items`.

These are callable sequence objects that work as methods, or can be manipulated directly as sequences.

Test for `keys`, `items` and `values`.

```
>>> d = SequenceOrderedDict(((1, 2), (2, 3), (3, 4)))
>>> d
SequenceOrderedDict([(1, 2), (2, 3), (3, 4)])
>>> d.keys
[1, 2, 3]
>>> d.keys()
[1, 2, 3]
```

```
>>> d.setkeys((3, 2, 1))
>>> d
SequenceOrderedDict([(3, 4), (2, 3), (1, 2)])
>>> d.setkeys((1, 2, 3))
>>> d.keys[0]
1
>>> d.keys[:]
[1, 2, 3]
>>> d.keys[-1]
3
>>> d.keys[-2]
2
>>> d.keys[0:2] = [2, 1]
>>> d
SequenceOrderedDict([(2, 3), (1, 2), (3, 4)])
>>> d.keys.reverse()
>>> d.keys
[3, 1, 2]
>>> d.keys = [1, 2, 3]
>>> d
SequenceOrderedDict([(1, 2), (2, 3), (3, 4)])
>>> d.keys = [3, 1, 2]
>>> d
SequenceOrderedDict([(3, 4), (1, 2), (2, 3)])
>>> a = SequenceOrderedDict()
>>> b = SequenceOrderedDict()
>>> a.keys == b.keys
1
>>> a['a'] = 3
>>> a.keys == b.keys
0
>>> b['a'] = 3
>>> a.keys == b.keys
1
>>> b['b'] = 3
>>> a.keys == b.keys
0
>>> a.keys > b.keys
0
>>> a.keys < b.keys
1
>>> 'a' in a.keys
1
>>> len(b.keys)
2
>>> 'c' in d.keys
0
>>> 1 in d.keys
1
>>> [v for v in d.keys]
[3, 1, 2]
>>> d.keys.sort()
>>> d.keys
[1, 2, 3]
>>> d = SequenceOrderedDict(((1, 2), (2, 3), (3, 4)), strict=True)
>>> d.keys[::-1] = [1, 2, 3]
>>> d
SequenceOrderedDict([(3, 4), (2, 3), (1, 2)])
```

```
>>> d.keys[:2]
[3, 2]
>>> d.keys[:2] = [1, 3]
Traceback (most recent call last):
KeyError: 'Keylist is not the same as current keylist.'


>>> d = SequenceOrderedDict(((1, 2), (2, 3), (3, 4)))
>>> d
SequenceOrderedDict([(1, 2), (2, 3), (3, 4)])
>>> d.values
[2, 3, 4]
>>> d.values()
[2, 3, 4]
>>> d.setvalues((4, 3, 2))
>>> d
SequenceOrderedDict([(1, 4), (2, 3), (3, 2)])
>>> d.values[::-1]
[2, 3, 4]
>>> d.values[0]
4
>>> d.values[-2]
3
>>> del d.values[0]
Traceback (most recent call last):
TypeError: Can't delete items from values
>>> d.values[::2] = [2, 4]
>>> d
SequenceOrderedDict([(1, 2), (2, 3), (3, 4)])
>>> 7 in d.values
0
>>> len(d.values)
3
>>> [val for val in d.values]
[2, 3, 4]
>>> d.values[-1] = 2
>>> d.values.count(2)
2
>>> d.values.index(2)
0
>>> d.values[-1] = 7
>>> d.values
[2, 3, 7]
>>> d.values.reverse()
>>> d.values
[7, 3, 2]
>>> d.values.sort()
>>> d.values
[2, 3, 7]
>>> d.values.append('anything')
Traceback (most recent call last):
TypeError: Can't append items to values
>>> d.values = (1, 2, 3)
>>> d
SequenceOrderedDict([(1, 1), (2, 2), (3, 3)])


>>> d = SequenceOrderedDict(((1, 2), (2, 3), (3, 4)))
>>> d
```

```
SequenceOrderedDict([(1, 2), (2, 3), (3, 4)])
>>> d.items()
[(1, 2), (2, 3), (3, 4)]
>>> d.setitems([(3, 4), (2 ,3), (1, 2)])
>>> d
SequenceOrderedDict([(3, 4), (2, 3), (1, 2)])
>>> d.items[0]
(3, 4)
>>> d.items[:-1]
[(3, 4), (2, 3)]
>>> d.items[1] = (6, 3)
>>> d.items
[(3, 4), (6, 3), (1, 2)]
>>> d.items[1:2] = [(9, 9)]
>>> d
SequenceOrderedDict([(3, 4), (9, 9), (1, 2)])
>>> del d.items[1:2]
>>> d
SequenceOrderedDict([(3, 4), (1, 2)])
>>> (3, 4) in d.items
1
>>> (4, 3) in d.items
0
>>> len(d.items)
2
>>> [v for v in d.items]
[(3, 4), (1, 2)]
>>> d.items.count((3, 4))
1
>>> d.items.index((1, 2))
1
>>> d.items.index((2, 1))
Traceback (most recent call last):
ValueError: list.index(x): x not in list
>>> d.items.reverse()
>>> d.items
[(1, 2), (3, 4)]
>>> d.items.reverse()
>>> d.items.sort()
>>> d.items
[(1, 2), (3, 4)]
>>> d.items.append((5, 6))
>>> d.items
[(1, 2), (3, 4), (5, 6)]
>>> d.items.insert(0, (0, 0))
>>> d.items
[(0, 0), (1, 2), (3, 4), (5, 6)]
>>> d.items.insert(-1, (7, 8))
>>> d.items
[(0, 0), (1, 2), (3, 4), (7, 8), (5, 6)]
>>> d.items.pop()
(5, 6)
>>> d.items
[(0, 0), (1, 2), (3, 4), (7, 8)]
>>> d.items.remove((1, 2))
>>> d.items
[(0, 0), (3, 4), (7, 8)]
>>> d.items.extend([(1, 2), (5, 6)])
```

```
>>> d.items
[(0, 0), (3, 4), (7, 8), (1, 2), (5, 6)]
```

1.3 Indices and tables

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

1.4 Glossary

LP Linear Programming

MIP Mixed Integer Programming

CHAPTER

TWO

INDICES AND TABLES

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

BIBLIOGRAPHY

[Ref] In the doc directory of your sphinx installation.

MODULE INDEX

P

INDEX

Symbols

A

C

D

E

F

G

I

K