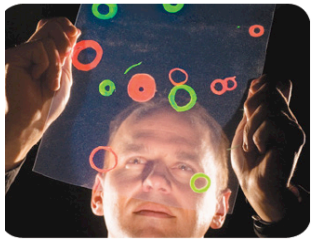
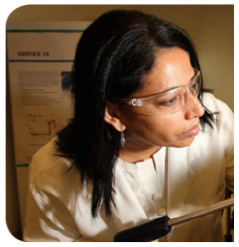




Pyomo Workshop:
Summer 2018

Sandia National Laboratories

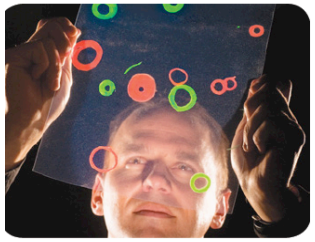
Exceptional service in the national interest

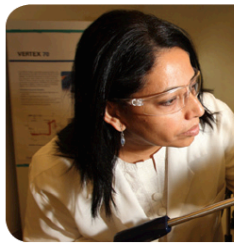
U.S. DEPARTMENT OF ENERGY

National Nuclear Security Administration

CCR Center for Computing Research

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.
SAND2018-6008 PE





Sandia National Laboratories

Exceptional service in the national interest

0. Preliminaries & Installing Pyomo

U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

CCR Center for Computing Research

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Downloading workshop examples

- https://www.pyomo.org/workshop-examples

Use Anaconda3 [all platforms]

- Install Anaconda3
  - https://www.continuum.io/downloads
  - **Install the Python 3.6 version** for your OS
  - Includes several packages for scientific computing already
  - Supports easy installation of pyomo and solvers

- Install pyomo and solvers:
  - Pyomo
  - Glpk (solver for linear and mixed-integer linear problems)
  - IPOPT (solver for continuous nonlinear problems)

Use Anaconda3 : Details [all platforms]

Sandia National Laboratories

1. Download and Install Anaconda3 (https://www.anaconda.com/download/)
   - "Python 3.6 version" for your platform.
   - Large download (approx. 600 MB) including Python and other packages.
   - Tips for the Mac: click on the .pkg file. You might get a message indicating that the package cannot be installed because it was downloaded from the internet. Try clicking again. If this does not work, navigate to the Downloads folder, right click (control-click) on Anaconda3-…x86_64.pkg and choose "Open With…" | "Installer.app".

2. Test Python installation:
   - On Windows, you will access Anaconda's python from the "Anaconda Prompt" application. On the mac, you will use the terminal application. Both of these are command-line interfaces that allow you to run python code. Open "Anaconda Prompt" or a terminal app.
   - Type "python --version", and you should see something similar to:
     Python 3.6.3 :: Anaconda, Inc.

Use Anaconda3 : Details [all platforms]

2. Test Python installation (con't):
   - Type "python". You should see something similar to:
     Python 3.6.3 |Anaconda, Inc.| (default, Oct 6 2017, 12:04:38)
     [GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
     Type "help", "copyright", "credits" or "license" for more information.
     >>>
   - Try writing some python code to test, e.g.,
     >>> print('so…. this is python, eh?')
   - Type "exit()" if everything is working correctly.

3. Install Pyomo:
   - From the prompt, type: "conda install -c conda-forge pyomo".
   - Test your pyomo installation with "pyomo --version". You should see something similar to the following:
     Pyomo 5.3 (CPython 3.6.3 on Darwin 15.6.0)

Use Anaconda3 : Details [all platforms]

Sandia National Laboratories

4. Install glpk:
   - From the prompt, type: “conda install -c conda-forge glpk”
   - Test the glpk solver. Type “glpsol --version”. You should see something like:
     GLPSOL: GLPK LP/MIP Solver, v4.62
     Copyright (C) 2000-2017 Andrew Makhorin, Department for Applied
     …

5. Install IPOPT:
   - From the prompt type: “conda install -c conda-forge ipopt”
   - Test the ipopt solver. Type “ipopt --version”. You should see something like:
     Ipopt 3.12.8 (Darwin x86_64), ASL(20160307)

Install with pip [all platforms]

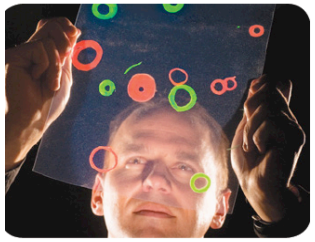
- Install Python
  - Linux & Mac OS/X typically have Python pre-installed
  - Scientific Python distributions have many utilities pre-installed
    - http://www.scipy.org/install.html
- Install Pyomo
  - Install in your system
    - `pip install Pyomo`
  - (or) Install in a user directory
    - `pip install --user Pyomo`
- Install auxiliary software
  - Pyomo has conditional dependencies on various third-party packages
    - `pip install pyomo.extras`
  - Not needed for any of this tutorial
- Installing solvers
  - When using pip, solvers will need to be installed manually (or use the NEOS solver)

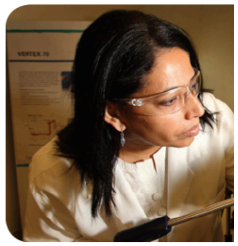
Getting help

- Homepage:
  - www.pyomo.org
- GitHub site:
  - https://github.com/Pyomo
- Other sources
  - StackOverflow: “pyomo” tag
  - pyomo-forum@googlegroups.com
  - pyomo-developers@googlegroups.com

Acknowledgements

- William Hart
- Carl Laird
- John Siirola
- Jean-Paul Watson
- David Woodruff
- Bethany Nicholson
- Gabriel Hackebiel





Sandia National Laboratories

Exceptional service in the national interest

Optional Material

U.S. DEPARTMENT OF ENERGY

National Nuclear Security Administration

CCR Center for Computing Research

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Install with Download Scripts [Windows]

- Install Python
  - Linux & Mac OS/X typically have Python pre-installed
  - Scientific Python distributions have many utilities pre-installed
    - http://www.scipy.org/install.html
- Install pip
  - https://bootstrap.pypa.io/get-pip.py
- Install Pyomo: the `get_pyomo.py` script
  - https://software.sandia.gov/trac/pyomo/downloader
- Install auxiliary software: the `get_pyomo_extras.py` script
  - https://software.sandia.gov/trac/pyomo/downloader

Install Trunk / Offline [All platforms]

- Install Python
  - Linux & Mac OS/X typically have Python pre-installed
  - Scientific Python distributions have many utilities pre-installed
    - http://www.scipy.org/install.htm
- Install pyomo: the `pyomo_install` script
  - https://software.sandia.gov/trac/pyomo/downloader
  - Trunk install
    - `pyomo_install --trunk`
  - Install using a bundled zip file (no network access required)
    - `pyomo_install --zip=pyomo-zipfile.zip`
  - Install into a Virtual Python Environment
    - `pyomo_install --venv=pyomo`
  - *Trunk or zipfile options may be combined with virtual environments*

Install Solvers

Open Source Solvers

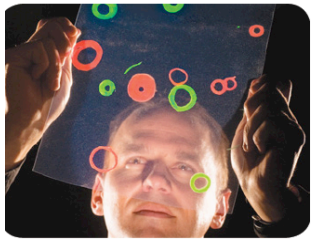
- COIN-OR Binary Distributions (e.g., CBC, IPOPT)
  - http://www.coin-or.org/download/binary/
  - Note: Coin-Binary requires the Coin-HSL Archive (Personal License)
    - http://www.hsl.rl.ac.uk/ipopt/
    - The Personal License permits commercial use ***but not redistribution***
- GLPK
  - http://ftp.gnu.org/gnu/glpk/
- SCIP
  - http://scip.zib.de/#download

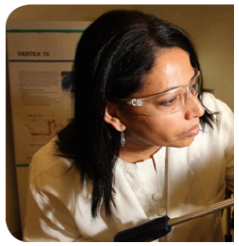
Note: You need to add the solver installation to the PATH environment variable

License

Pyomo released under 3-clause BSD license

- No restrictions on deployment or commercial use





Sandia National Laboratories

Exceptional service in the national interest

1. Overview of Pyomo

U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

CCR Center for Computing Research

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Optimization Modeling

Sandia National Laboratories

Goal:

- Provide a natural syntax to describe mathematical models
- Formulate large models with a concise syntax
- Separate modeling and data declarations
- Enable data import and export in commonly used formats

Impact:

- Robustly model large systems
- Integrated automatic differentiation for complex nonlinear models

Examples:

- AMPL, GAMS, AIMMS, …
- OptimJ, FlopCPP, PuLP, JuMP, …

- **Pyomo: A Python-based optimization modeling package**

CCR Center for Computing Research

PYOMO COIN-OR

Overview

- Three really good questions:
  - Why another Algebraic Modeling Language (AML)?
  - Why Python?
  - Why open-source?
- Pyomo: Team overview and collaborators / users
- Where to find more information…

Pyomo Overview

Sandia National Laboratories

Idea: a Pythonic framework for formulating optimization models

- Keep a natural syntax even though it is Python
- Meet goals of modeling languages (concise, separate model and data)
- And…
- Provide an **open-source** tool for problem solving and research
- Allow for complex workflows through rich programming capabilities already available with Python

Highlights:

- Python provides a clean, intuitive syntax
- Python scripts provide a flexible context for exploring the structure of Pyomo models

```
# simple.py
from pyomo.environ import *

M = ConcreteModel()
M.x1 = Var()
M.x2 = Var(bounds=(-1,1))
M.x3 = Var(bounds=(1,2))
M.o  = Objective(
          expr=M.x1**2 + (M.x2*M.x3)**4 + \
               M.x1*M.x3 + \
               M.x2*sin(M.x1+M.x3) + M.x2)
```

CCR Center for Computing Research

PYOMO COIN-OR

Pyomo Overview

- Equation-oriented models
  - User must provide ***all*** model equations explicitly (e.g., glass-box)
  - Not an optimizer around existing simulators (e.g., not black-box)
- Classes of problems and solvers:
  - Discrete Optimization (MIP): Gurobi, CPLEX, CBC, GLPK
  - Nonlinear Optimization (NLP): IPOPT, KNITRO, … (Any AMPL or GAMS)
  - Mixed-Integer Nonlinear Optimization (MINLP): Baron, COUENNE
- Included packages and capabilities:
  - Pyomo Core: Basic modeling and optimization capabilities
  - Pyomo Blocks: Object-oriented (hierarchical) models
  - Pyomo.DAE: Modeling and optimization of dynamic systems (DAE and PDE)
  - Pyomo.GDP: Generalized disjuctive programming
  - PySP: Optimization under uncertainty
  - Pyomo.Bilevel, Pyomo.MPEC, ...

Pyomo Overview: Two focus areas

- **Flexibility (built on Python)**
  - Ease of Prototyping and new workflows
  - Higher-level interfaces and applications *through Python*
  - Data import & manipulation, plotting, UI development

- **Solution Efficiency**
  - Fast solution of large-scale problems (efficient interfaces with solvers)
  - **HPC Ready**: Parallel execution of many optimization problems and parallel decomposition of large, structured problems
  - Reduced end-to-end solution time

More than *just* mathematical modeling

Sandia National Laboratories

Scripting

- Construct models using native Python data
- Iterative analysis of models leveraging Python functionality
- Data analysis and visualization of optimization results

Model transformations (a.k.a. reformulations)

- Automate generation of one model from another
- Leverage Pyomo's object model to apply transformations sequentially
- E.g.: DAE -> discretized model, GDP -> Big M

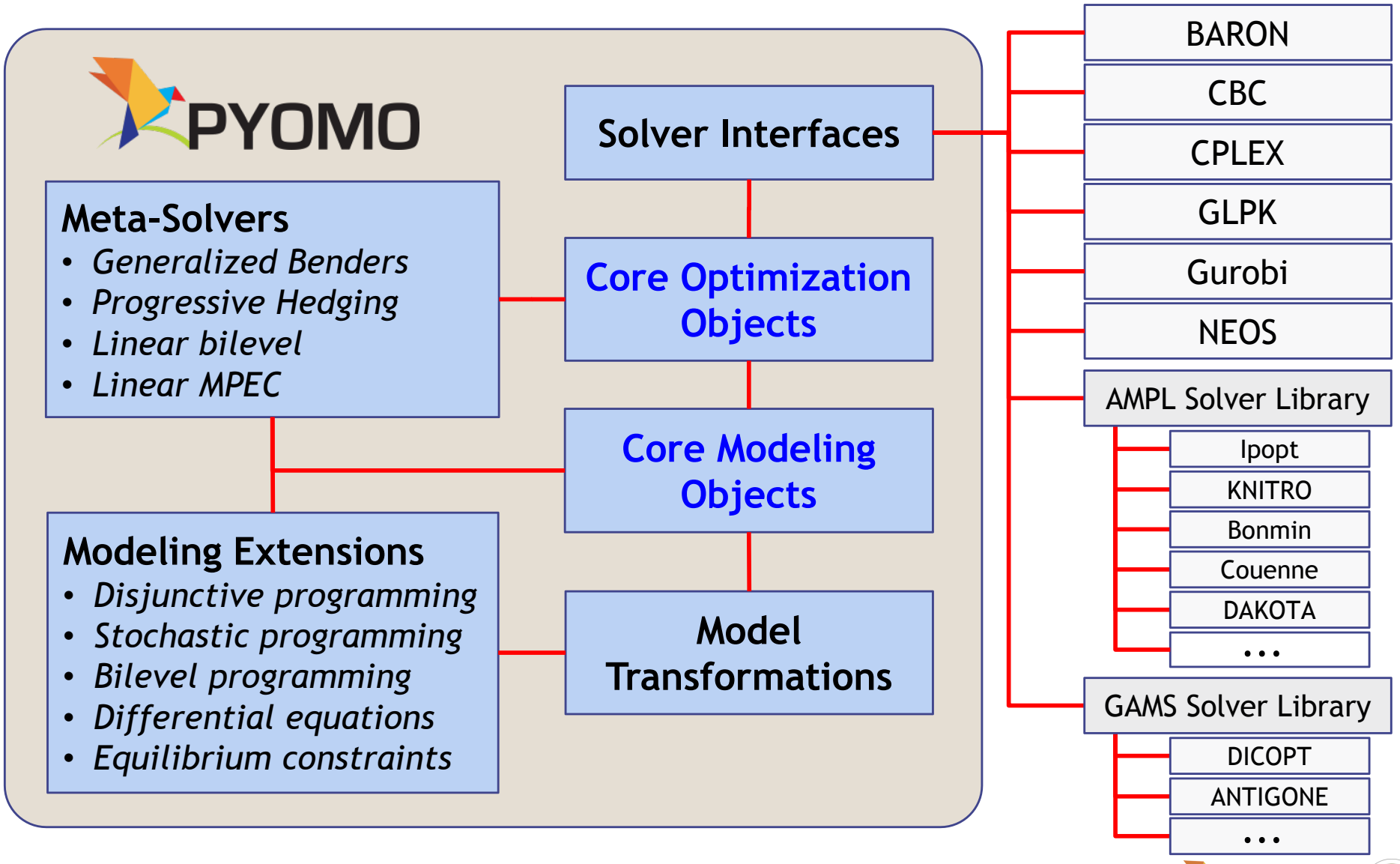
Meta-solvers

- Integrate scripting and/or transformations into optimization solver
- Leverage power of Python to build "generic" capabilities
- E.g.: progressive hedging, SP extensive form -> MIP

CCR Center for Computing Research

PYOMO COIN-OR

Pyomo at a Glance



Why Model in Python?

- Full-Featured Language
  - Language features includes functions, classes, looping, namespaces, etc
  - Introspection facilitates the development of generic algorithms
  - Python's clean syntax facilitates rapid prototyping
- Open Source License
  - No licensing issues w.r.t. the language itself
- Extensibility and Robustness
  - Highly stable and well-supported
- Support and Documentation
  - Extensive online documentation and several excellent books
  - Long-term support for the language is not a factor
- Standard Library
  - Includes a large number of useful modules
- Portability
  - Widely available on many platforms

Why Open Source?

- Transparency and reliability
- Foster community involvement
  - Extend the modeling language
  - Develop new solvers / algorithms
  - Interface with additional external utilities
  - “Stone Soup” model
- Flexible licensing
  - Pyomo released under 3-clause BSD license
  - No restrictions on deployment or commercial use

Who Uses Pyomo?

- Students
  - Rose-Hulman, UC Davis, Purdue, U Texas, Iowa State, Notre Dame, NPS, …
- Researchers
  - Government laboratories
    - Sandia National Labs, Lawrence Livermore National Lab, Los Alamos National Lab, National Energy Technology Lab, National Renewable Energy Lab, Federal Energy Regulation Commission
  - Universities
    - Carnegie Mellon, UC Davis, Texas A&M, Rose-Hulman, U. Texas, USC, GMU, Iowa State, NCSU, NTNU, Notre Dame, Imperial College, U Washington, NPS, U de Santiago de Chile, U Pisa, …
  - Companies… yes.
- Software Projects
  - IDAES – Optimization of energy production systems
  - Minpower – Power systems toolkit
  - SolverStudio – Excel plugin for optimization modeling
  - TEMOA – Energy economy optimization models
  - WNTR – Resilience analysis for water distribution systems

For More Information

Sandia National Laboratories

See the Pyomo homepage

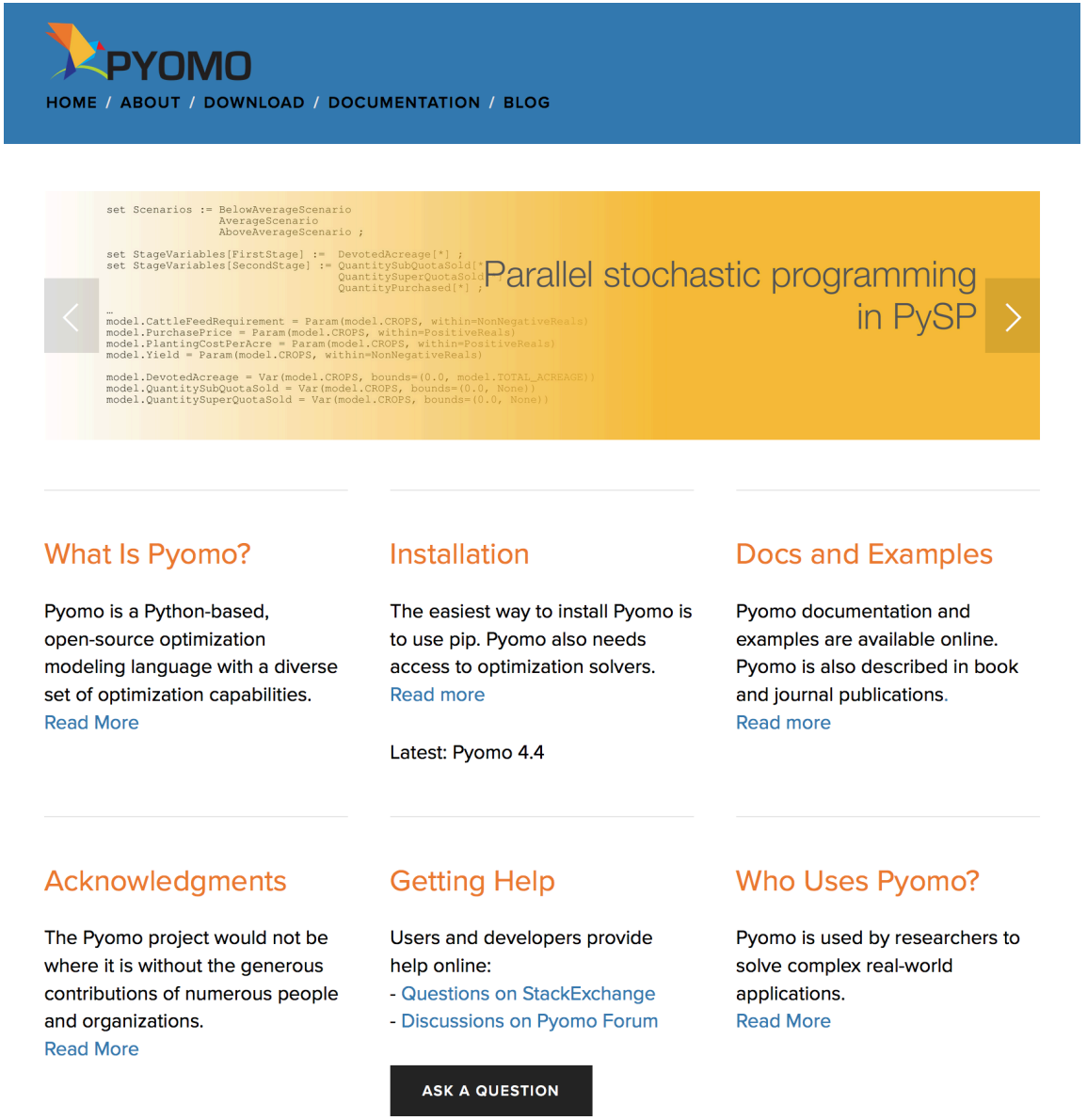
- www.pyomo.org

The Pyomo homepage provides a portal for:

- Online documentation
- Installation instructions
- Help information
- Developer links

GitHub:

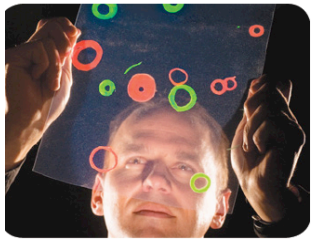
- github.com/Pyomo

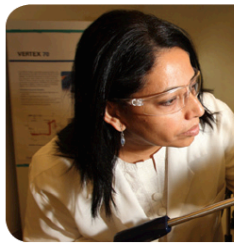


CCR Center for Computing Research

Acknowledgements

- William Hart (SNL)
- Jean-Paul Watson (SNL)
- John Siirola (SNL)
- Francisco Munoz (SNL, UAI)
- Bethany Nicholson (CMU, SNL)
- Prof. David L. Woodruff (UC Davis)
- Prof. Roger Wets (UC Davis)
- Prof. Carl D. Laird (Purdue, SNL)
- Gabe Hackebeil (U.Mich.)





Sandia National Laboratories

Exceptional service in the national interest

2. A Python Tutorial

PYOMO

U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

CCR Center for Computing Research

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Why Python?

Sandia National Laboratories

- Interpreted language
- Intuitive syntax
- Object-oriented
- Simple, but extremely powerful
- Lots of built-in libraries and third-party extensions
- Shallow learning curve and many resources
- Dynamic typing
- Integration with C/Java

CCR Center for Computing Research

PYOMO COIN-OR

Python Versions: 2.x *vs* 3.x

- Python 3.0 was released in 2008
  - Included significant backward incompatibilities
  - Initial adoption was slow
  - But, community is moving on to Python 3.0
- Pyomo support:
  - Python 2.7
    - Very stable; patches have included package updates to support Python 3.x compatibility
  - Python 3.5, 3.6
    - Very stable, but marginally slower than 2.7

We try to stick to "universal" syntax that will work in both 2.x and 3.x

https://pythonclock.org

Overview

- interactive "shell"
- basic types: numbers, strings
- container types: lists, dictionaries, tuples
- variables
- control structures
- functions & procedures
- classes & instances
- modules
- exceptions
- files & standard library

Interactive Shell

- Great for learning the language
- Great for experimenting with the library
- Great for testing your own modules
- Two variations:
  - IDLE (GUI)
  - python (command line)
- Type statements or expressions at prompt:

```
>>> print( "Hello, world" )
Hello, world
>>> x = 12**2
>>> x/2
72
>>> # this is a comment
```

Python Program

- To write a program, put commands in a file

```
# hello.py
print( "Hello, world" )
x = 12**2
print( x )
```

- Execute on the command line

```
C:\Users\me> python hello.py
Hello, world
144
```

Python Variables

- No need to declare
- Need to assign (initialize)
  - use of uninitialized variable raises exception
- Not typed

```
greeting = 34.2
if friendly:
    greeting = "hello world"
else:
    greeting = 12**2
print( greeting )
```

- ***Everything*** is a "variable":
  - Even functions, classes, modules

Control Structures

```
if condition:
    statements
elif condition:
    statements ...
else:
    statements
```

```
while condition:
    statements

for var in sequence:
    statements

break
continue
```

Note: *Spacing matters!*
Control structure scope dictated by indentation

Grouping Indentation

Sandia National Laboratories

In Python:

```
for i in range(20):
    if i % 3 == 0:
        print(i)
        if i % 5 == 0:
            print("Bingo!“)
    print("---")
```

In C:

```
for (i = 0; i < 20; i++)
{
   if (i % 3 == 0) {
      printf("%d\n", i);
      if (i % 5 == 0) {
         printf("Bingo!\n");
      }
   }
   printf("---\n");
}
```

Numbers

- The usual suspects
  - `12, 3.14, 0xFF, 0377, (-1+2)*3/4**5, abs(x), x <= 5`
- C-style shifting & masking
  - `1<<16, x&0xff, x|1, ~x, x^y`
- Integer division truncates
  - Python 2.x
    - `1/2 → 0, 1./2. → 0.5, float(1)/2 → 0.5`
    - `from __future__ import division`
      - `» 1/2 → 0.5`
  - Python 3.x
    - `1/2 → 0.5`
- Long (arbitrary precision), complex
  - `2L**100 → 1267650600228229401496703205376L`
    - In Python 2.2 and beyond, 2**100 does the same thing
  - `1j**2 → (-1+0j)`

Strings

- `"hello"+"world”` `"helloworld”` `# concatenation`
- `"hello"*3` `"hellohellohello"` `# repetition`
- `"hello"[0]` `"h"` `# indexing`
- `"hello"[-1]` `"o"` `# (from end)`
- `"hello"[1:4]` `"ell"` `# slicing`
- `len("hello")` `5` `# size`
- `"hello" < "jello”` `True` `# comparison`
- `"e" in "hello”` `True` `# search`

- `"escapes: \n etc, \033 etc, \if etc"`
- `'single quotes'  """triple quotes"""  r"raw strings"`

Lists

- Flexible arrays, *not* linked lists
  - `a = [99, "bottles of beer", ["on", "the", "wall"]]`
- Same operators as for strings
  - `a+b`, `a*3`, `a[0]`, `a[-1]`, `a[1:]`, `len(a)`
- Item and slice assignment
  - `a[0] = 98`
  - `a[1:2] = ["bottles", "of", "beer"]`

    `# -> [98, "bottles", "of", "beer", ["on", "the", "wall"]]`
  - `del a[-1]`

    `# -> [98, "bottles", "of", "beer"]`

List Operations

```
>>> a = range(5)                # [0,1,2,3,4]
>>> a.append(5)                 # [0,1,2,3,4,5]
>>> a.pop()                     # [0,1,2,3,4]
5
>>> a.insert(0, 42)             # [42,0,1,2,3,4]
>>> a.pop(0)                    # [0,1,2,3,4]
42
>>> a.reverse()                 # [4,3,2,1,0]
>>> a.sort()                    # [0,1,2,3,4]
```

Dictionaries

- Hash tables, "associative arrays"
  - `d = {"wood": "float", "witch": "nose"}`
- Lookup:
  - `d["wood"]` `# -> "float"`
  - `d["back"]` `# raises KeyError exception`
- Delete, insert, overwrite:
  - `del d["wood"]` `# {"witch": "nose"}`
  - `d["duck"] = "float"` `# {"duck": "float", "witch": "nose"}`
  - `d["witch"] = "burn"` `# {"duck": "float", "witch": "burn"}`

Dictionary Operations

- Keys, values, items:
  - `d.keys()  -> ["duck", "witch"]`
  - `d.values()  -> ["float", "burn"]`
  - `d.items()  -> [("duck","float"), ("witch","burn")]`

Note: These actually return lists in Python 2.x, and generators in Python 3.x.

- Presence check:
  - `d.has_key("duck")  # True`
  - `d.has_key("spam")  # False`
  - `"duck" in d  # True`
- Values of any type; keys almost any

```
{"age": 43,
  ("hello","world"): 1,
  42: "answer",
  "flag": ["red","white","blue"] }
```

Dictionary Details

- Keys must be **immutable**:
  - numbers, strings, tuples of immutables
    - these cannot be changed after creation
  - keys are hashed (to ensure fast lookup)
  - lists or dictionaries cannot be used as keys
    - these objects can be changed "in place"
  - no restrictions on values

- Keys will be listed in **arbitrary order**
  - key hash values are in an arbitrary order
  - that numeric keys are returned sorted is an artifact of the implementation and *is not guaranteed*

References vs.Copies

- Assignment manipulates references
  - `x = y` **does not make a copy** of y
  - `x = y` makes x **reference** the object y references
- Reference values can be modified!

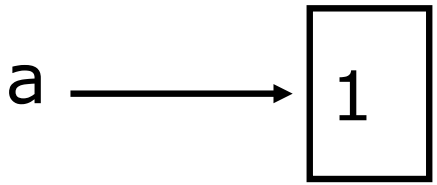
```
>>> a = [1, 2, 3]
>>> b = a
>>> a.append(4)
>>> print(b)
[1, 2, 3, 4]
```

- Copied objects are distinct

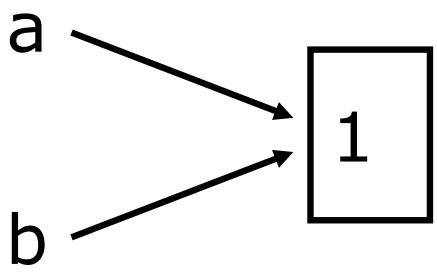
```
>>> import copy
>>> c = copy.copy(a)
>>> a.pop()
>>> print(c)
[1, 2, 3, 4]
```

Working with *immutable data*

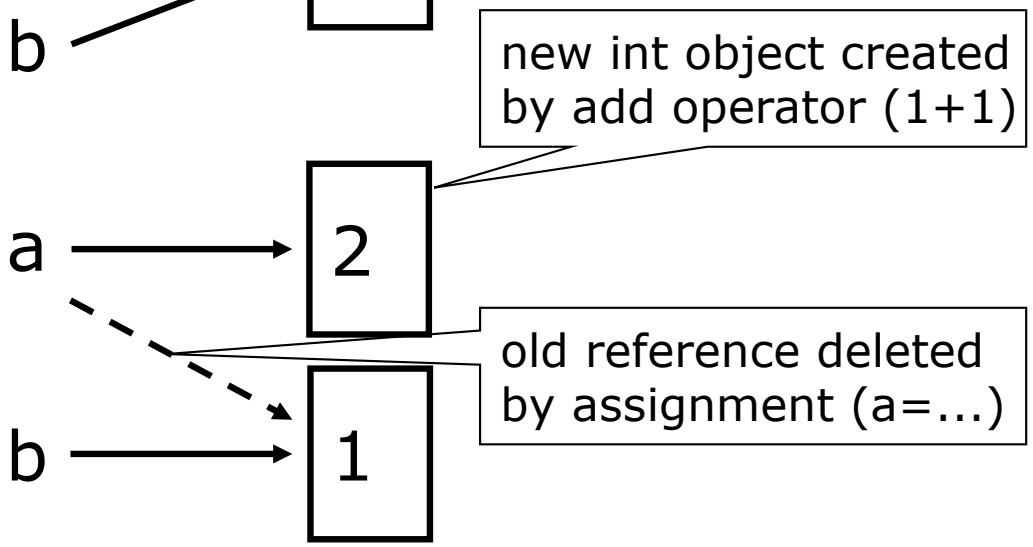
```
a = 1
```




```
b = a
```

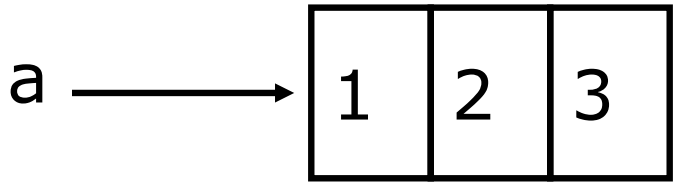



```
a += 1
```

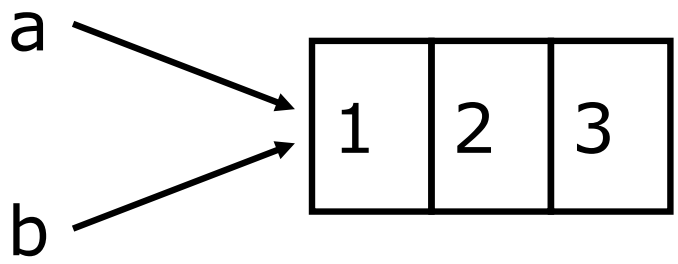


Working with objects

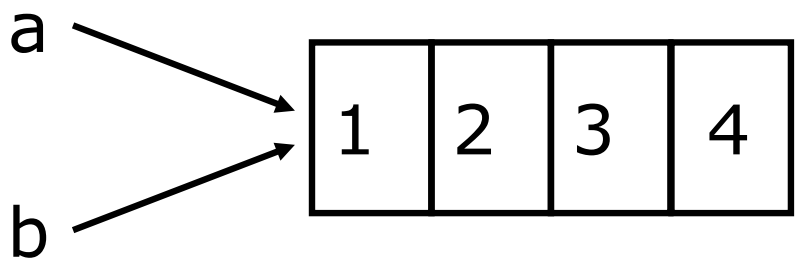
```
a = [1, 2, 3]
```




```
b = a
```




```
a.append(4)
```



Functions / Procedures

```
def name(arg1, arg2, ...):
    """documentation"""    # optional doc string
    statements

    return expression      # from function
    return                 # from procedure (returns None)
```

Example

```
def gcd(a, b):
    """greatest common divisor"""
    while a != 0:
        a, b = b%a, a   # parallel assignment
    return b

>>> gcd.__doc__
'greatest common divisor'

>>> gcd(12, 20)
4
```

Modules

- Collection of stuff in *foo*.py file
  - functions, classes, variables

- Importing modules:

```
import re
print( re.match("[a-z]+", s) )
from re import match
print( match("[a-z]+", s) )
```

- Import with rename:

```
import re as regex
from re import match as m
```

Major Python Packages

- Matplotlib
  - 2D plotting library
  - http://matplotlib.org/
- Pandas
  - Data structures and analysis
  - http://pandas.pydata.org/
- NetworkX, Plotly, …
- SciPy
  - Scientific Python for mathematics and engineering
  - http://www.scipy.org
- Numpy
  - Numeric array package
  - http://www.numpy.org/
- Ipython
  - Interactive Python shell
  - http://ipython.org/

Resources

- Software Carpentry
  - http://software-carpentry.org/
- Python webpage
  - http://www.python.org
- Books
  - Python Essential Reference (4th Edition), David Beazley, 2009
  - Python in a Nutshell, Alex Martelli, 2003
  - Python Pocket Reference, 4th Edition, Mark Lutz, 2009
  - …

Acknowledgements

- William Hart
- Ted Ralphs
- John Siirola
- Carl Laird
- Bethany Nicholson
- Dave Woodruff
- Guido van Rossum

Sandia National Laboratories

OTHER MATERIAL

CCR Center for Computing Research

PYOMO COIN-OR

Python Implementations

- Cpython
  - C Python interpreter
  - https://www.python.org/downloads/
  - SciPy Stack
    - http://www.scipy.org/install.html
    - Anaconda: Linux/MacOS/MS Windows
- PyPy
  - A Python interpreter written in Python
  - http://pypy.org/
- Jython
  - Java Python interpreter
  - http://www.jython.org/
- IronPython
  - .NET Python interpreter
  - http://ironpython.net/

Full Pyomo Support

Beta Pyomo Support

Pyomo Not Supported (yet)

Tuples

- `key = (lastname, firstname)`
- `point = x, y, z` `# parentheses optional`
- `x, y, z = point` `# unpack`
- `lastname = key[0]` `# index tuple values`
- `singleton = (1,)` `# trailing comma!!!`
  `#  (1) → integer!`
- `empty = ()` `# parentheses!`

- Tuples vs. lists
  - tuples immutable
  - lists mutable

Classes

```
class name(object):
    """documentation"""
    statements
```

Most, *statements* are method definitions:

```
    def name(self, arg1, arg2, ...):
        ...
```

May also be *class variable* assignments

Example

```
class Stack(object):
    """A well-known data structure..."""
    def __init__(self):                 # constructor
        self.items = []
    def push(self, x):
        self.items.append(x)            # the sky is the limit
    def pop(self):
        x = self.items[-1]              # what if it's empty?
        del self.items[-1]
        return x
    def empty(self):
        return len(self.items) == 0     # Boolean result
```

Example (cont'd)

- To create an instance, simply call the class object:

```
x = Stack()            # no 'new' operator!
```

- To use methods of the instance, call using dot notation:

```
x.empty()              # -> 1
x.push(1)              # [1]
x.empty()              # -> 0
x.push("hello")        # [1, "hello"]
x.pop()                # -> "hello"   # [1]
```

- To inspect instance variables, use dot notation:

```
x.items                # -> [1]
```

Class/Instance Variables

Sandia National Laboratories

```
class Connection(object):
    verbose = 0                 # class variable

    def __init__(self, host):
        self.host = host        # instance variable

    def debug(self, v):
        self.verbose = v        # make instance variable!

    def connect(self):
        if self.verbose:        # class or instance variable?
            print("connecting to %s" % (self.host,))
```

Instance Variable Rules

- On use via instance (`self.x`), search order:
  - (1) instance, (2) class, (3) base classes
  - this also works for method lookup
- On assignment via instance (`self.x = ...`):
  - always makes an instance variable
- Class variables "default" for instance variables
- But...!
  - mutable *class* variable: one copy *shared* by all
  - mutable *instance* variable: each instance its own

Catching Exceptions

```
def foo(x):
    return 1/x

def bar(x):
    try:
        print( foo(x) )
    except ZeroDivisionError as message:
        print("Can't divide by zero: %s" % message)

bar(0)
```

Try-Finally: Cleanup

```
f = open(file)
try:
    process_file(f)
finally:
    f.close()           # always executed
print("OK")             # executed on success only
```

Raising Exceptions

```
raise IndexError

raise IndexError("k out of range")

raise IndexError, "k out of range”
            # this only works in Python 2.x!

try:
    something
except:                          # catch everything
    print( "Oops" )
    raise                        # reraise
```

More on Exceptions

- User-defined exceptions
  - subclass `Exception` or any other standard exception
- Note: in older versions of Python exceptions can be strings
- Last caught exception info:
  - `sys.exc_info() == (exc_type, exc_value, exc_traceback)`
- Printing exceptions: traceback module

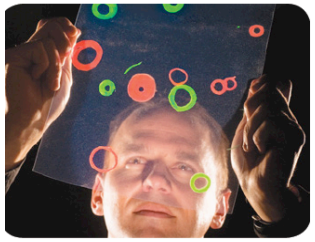
File Objects

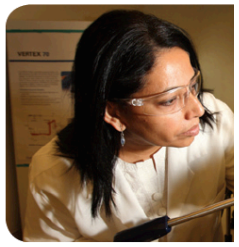
- `f = open(filename[, mode[, buffersize])`
  - mode can be `"r"`, `"w"`, `"a"` (like C stdio); default `"r"`
  - append `"b"` for text translation mode
  - append `"+"` for read/write open
  - `buffersize`: 0=unbuffered; 1=line-buffered; buffered

- methods:
  - `read([nbytes]), readline(), readlines()`
  - `write(string), writelines(list)`
  - `seek(pos[, how]), tell()`
  - `flush(), close()`
  - `fileno()`

Highlights from the Standard Library

- Core:
  - os, sys, string, getopt, StringIO, struct, pickle, json, csv, collections, math, tempfile, itertools, ...
- Regular expressions:
  - re module; Perl-5 style patterns and matching rules
- Internet:
  - socket, rfc822, httplib, htmllib, ftplib, smtplib, ...
- Miscellaneous:
  - pdb (debugger), profile+pstats
  - Tkinter (Tcl/Tk interface), audio, *dbm, ...





Sandia National Laboratories

Exceptional service in the national interest

3. Pyomo Fundamentals

PYOMO

U.S. DEPARTMENT OF ENERGY

National Nuclear Security Administration

CCR Center for Computing Research

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Learning Objectives

- Goals, benefits, drawbacks of Pyomo
- Basic understanding of Python
- Create and solve optimization problems within Pyomo
  - Vars, Constraints, Objectives
  - Sets, Parameters
- Basic scripting, workflow, and programming capabilities
  - Changing data, multiple solutions, importing data, plotting, reporting
  - Pieces to build your own workflows

Simple Modeling Example: Classic Knapsack Problem

Sandia National Laboratories

- Given a set of items A, with weight and value (benefit)
- Goal: select a subset of these items
  - Maximize the benefit
  - Under weight limit

Item (A)	Weight (w)	Benefit (b)
hammer	5	8
wrench	7	3
screwdriver	4	6
towel	3	11

Max weight: 14

Symbol	Meaning
A	set of items available for purchase
b_i	benefit of item i
w_i	weight of item i
$W_{\max}$	max weight that can be carried
x_i	discrete variable (select i or not)

Simple Modeling Example: Classic Knapsack Problem

- Given a set of items A, with weight and value (benefit)
- Goal: select a subset of these items
  - Maximize the benefit
  - Under weight limit

Item (A)	Weight (w)	Benefit (b)
hammer	5	8
wrench	7	3
screwdriver	4	6
towel	3	11

Max weight: 14

Symbol	Meaning
A	set of items available for purchase
b_i	benefit of item i
w_i	weight of item i
$W_{\max}$	max weight that can be carried
x_i	discrete variable (select i or not)

$$\max_x \quad \sum_{i \in A} b_i x_i$$

$$s.t. \quad \sum_{i \in A} w_i x_i \leq W_{\max}$$

$$x_i \in \{0,1\} \qquad \forall\, i \in A$$

Simple Modeling Example: Classic Knapsack Problem

- Variables
- Objectives
- Constraints

- Sets
- Parameters

Item (A)	Weight (w)	Benefit (b)
hammer	5	8
wrench	7	3
screwdriver	4	6
towel	3	11

Max weight: 14

Symbol	Meaning
A	set of items available for purchase
b_i	benefit of item i
w_i	weight of item i
$W_{\max}$	max weight that can be carried
x_i	discrete variable (select i or not)

$$\max_x \quad \sum_{i \in A} b_i x_i$$

$$s.t. \quad \sum_{i \in A} w_i x_i \leq W_{\max}$$

$$x_i \in \{0,1\} \qquad \forall\, i \in A$$

Anatomy of a Concrete Pyomo Model

Sandia National Laboratories

$$\max_x \quad \sum_{i \in A} b_i x_i$$

$$s.t. \quad \sum_{i \in A} w_i x_i \leq W_{\max}$$

$$x_i \in \{0,1\} \qquad \forall\, i \in A$$

Item	Weight	Value
hammer	5	8
wrench	7	3
screwdriver	4	6
towel	3	11

```
from pyomo.environ import *

A = ['hammer', 'wrench', 'screwdriver', 'towel']
b = {'hammer':8, 'wrench':3, 'screwdriver':6, 'towel':11}
w = {'hammer':5, 'wrench':7, 'screwdriver':4, 'towel':3}
W_max = 14

model = ConcreteModel()
model.x = Var( A, within=Binary )

model.value = Objective(
    expr = sum( b[i]*model.x[i] for i in A),
    sense = maximize )

model.weight = Constraint(
    expr = sum( w[i]*model.x[i] for i in A) <= W_max )

opt = SolverFactory('glpk')
result_obj = opt.solve(model, tee=True)

model.pprint()
```

Import packages

Specify/import data

Can be imported or even applied to model later

Define problem

Solve problem

Report results

Pyomo imports and *namespaces*

- Pyomo objects exist within the `pyomo.environ` namespace:

```
import pyomo.environ
model = pyomo.environ.ConcreteModel()
```

- …but this gets verbose. To save typing, we will import the core Pyomo classes into the main namespace:

```
from pyomo.environ import *
model = ConcreteModel()
```

- To clarify Pyomo-specific syntax in this tutorial, we will highlight Pyomo symbols in green

- Note: We violate good Python practice in this tutorial. One should:

```
import pyomo.environ as pe
model = pe.ConcreteModel()
```

Getting Started: the *Model*

```
from pyomo.environ import *

model = ConcreteModel()
```

Every Pyomo model starts with this; it tells Python to load the Pyomo Modeling Environment

Create an instance of a *Concrete* model

- Concrete models are immediately constructed
- Data must be present *at the time* components are defined

Local variable to hold the model we are about to construct

- While not required, by convention we use "`model`"

Populating the Model: *Variables*

Components can take a variety of keyword arguments when constructed.

```
model.a_variable = Var(within = NonNegativeReals)
```

The name you assign the object to becomes the object's name, and must be unique in any given model.

"`within`" is optional and sets the variable domain ("`domain`" is an alias for "`within`")

Several pre-defined domains, e.g., "`Binary`"

Populating the Model: *Variables*

Components can take a variety of keyword arguments when constructed.

```
model.a_variable = Var(within = NonNegativeReals)
```

The name you assign the object to becomes the object's name, and must be unique in any given model.

"`within`" is optional and sets the variable domain ("`domain`" is an alias for "`within`")

Several pre-defined domains, e.g., "`Binary`"

```
model.a_variable = Var(bounds = (0, None))

model.a_variable = Var(initialize = 42.0)

model.a_variable = Var(initialize=42.0, bounds=(0, None))
```

Defining the *Objective*

```
model.x = Var( initialize=-1.2, bounds=(-2, 2) )
model.y = Var( initialize= 1.0, bounds=(-2, 2) )

model.obj = Objective(
    expr= (1-model.x)**2 + 100*(model.y-model.x**2)**2,
    sense= minimize )
```

If "sense" is omitted, Pyomo assumes minimization

Note that the Objective expression is not a *relational expression*

"expr" can be an expression, or any function-like object that returns an expression

Defining the Problem: *Constraints*

```
model.a = Var()
model.b = Var()
model.c = Var()
model.c1 = Constraint(
    expr = model.b + 5 * model.c <= model.a )
```

$$b + 5c \leq a$$

"expr" can be an expression, or any function-like object that returns an expression

```
model.c2 = Constraint(expr = (None, model.a + model.b, 1))
```

$$a + b \leq 1$$

"expr" can also be a tuple:

- 3-tuple specifies (LB, expr, UB)
- 2-tuple specifies an equality constraint.

Higher-dimensional components

- (Almost) All Pyomo *components* can be *indexed*

```
A = [1, 2, 5]
model.x = Var(A)
```

  - *All* non-keyword arguments are assumed to be *indices*
  - Individual indices may be multi-dimensional (e.g., a list of pairs)

- E.g., Indexed variables

```
A = [1, 2, 5]
B = ['wrench', 'hammer']
model.x = Var(A)
model.y = Var(A, B)
```

The indexes are any iterable object, e.g., list or Set

- *Note:* while indexed variables look like matrices, they are not.
  - In particular, we do not support matrix algebra (*yet…)*

List Comprehensions

```
model.IDX = range(10)
model.a = Var()
model.b = Var(model.IDX)
model.c1 = Constraint(
    expr = sum(model.b[i] for i in model.IDX) <= model.a )
```

Python *list comprehensions* are very common for working over indexed variables and nicely parallel mathematical notation:

$$\sum_{i \in IDX} b_i \leq a$$

Putting it all together: Concrete Knapsack

Sandia National Laboratories

```
# knapsack.py
from pyomo.environ import *

A = ['hammer', 'wrench', 'screwdriver', 'towel']
b = {'hammer':8, 'wrench':3, 'screwdriver':6, 'towel':11}
w = {'hammer':5, 'wrench':7, 'screwdriver':4, 'towel':3}
W_max = 14

model = ConcreteModel()

model.x = Var( A, within=Binary )

model.value = Objective(
    expr = sum( b[i]*model.x[i] for i in A ),
    sense = maximize )

model.weight = Constraint(
    expr = sum( w[i]*model.x[i] for i in A) <= W_max )

opt = SolverFactory('glpk')

result_obj = opt.solve(model, tee=True)

model.pprint()
```

- Import packages
- Specify/import data
- Create model object
- Define variable x
- Define objective
- Define constraint
- Create solver
- Solve the problem
- Print results

Putting it all together: Concrete Knapsack

Sandia National Laboratories

```
# knapsack.py
from pyomo.environ import *
. . .
result_obj = opt.solve(model, tee=True)
model.pprint()
```

```
> python knapsack.py
```

```
1 Set Declarations
    x_index : Dim=0, Dimen=1, Size=4, Domain=None, Ordered=False, Bounds=None
        ['hammer', 'screwdriver', 'towel', 'wrench']
1 Var Declarations
    x : Size=4, Index=x_index
        Key         : Lower : Value : Upper : Fixed : Stale : Domain
             hammer :     0 :   1.0 :     1 : False : False : Binary
        screwdriver :     0 :   1.0 :     1 : False : False : Binary
              towel :     0 :   1.0 :     1 : False : False : Binary
             wrench :     0 :   0.0 :     1 : False : False : Binary
1 Objective Declarations
    value : Size=1, Index=None, Active=True
        Key  : Active : Sense    : Expression
        None :   True : maximize : 8*x[hammer] + 3*x[wrench] + 6*x[screwdriver] + 11*x[towel]
1 Constraint Declarations
    weight : Size=1, Index=None, Active=True
        Key  : Lower : Body                                                  : Upper : Active
        None :  -Inf : 5*x[hammer] + 7*x[wrench] + 4*x[screwdriver] + 3*x[towel] :  14.0 :   True
4 Declarations: x_index x value weight
```

Putting it all together: Concrete Knapsack

Sandia National Laboratories

```
# knapsack.py
from pyomo.environ import *
. . .
result_obj = opt.solve(model, tee=True)
model.display()
```

```
> python knapsack.py
```

```
Variables:
   x : Size=4, Index=x_index
       Key         : Lower : Value : Upper : Fixed : Stale : Domain
            hammer :     0 :   1.0 :     1 : False : False : Binary
       screwdriver :     0 :   1.0 :     1 : False : False : Binary
             towel :     0 :   1.0 :     1 : False : False : Binary
            wrench :     0 :   0.0 :     1 : False : False : Binary

Objectives:
   value : Size=1, Index=None, Active=True
       Key  : Active : Value
       None :   True :  25.0

Constraints:
   weight : Size=1
       Key  : Lower : Body : Upper
       None :  None : 12.0 :  14.0
```

Pyomo Fundamentals: Exercises #1

Sandia National Laboratories

1.1 Knapsack example: Solve the knapsack problem shown in the tutorial using your IDE (e.g., Spyder) or the command line: `> python knapsack.py`. Which items are acquired in the optimal solution? What is the value of the selected items?

1.2 Knapsack with improved printing: The `knapsack.py` example shown in the tutorial uses `model.pprint()` to see the value of the solution variables. Starting with the code in `knapsack_print_incomplete.py`, complete the missing lines to produce formatted output. Note that the Pyomo `value` function should be used to get the floating point value of Pyomo modeling components (e.g., `print(value(model.x[i]))`. Also print the value of the items selected (the objective), and the total weight. (A solution can be found in `knapsack_print_soln.py`).

1.3 Changing data: If we were to increase the value of the wrench, at what point would it become selected as part of the optimal solution? (A solution can be found in `knapsack_wrench_soln.py`.)

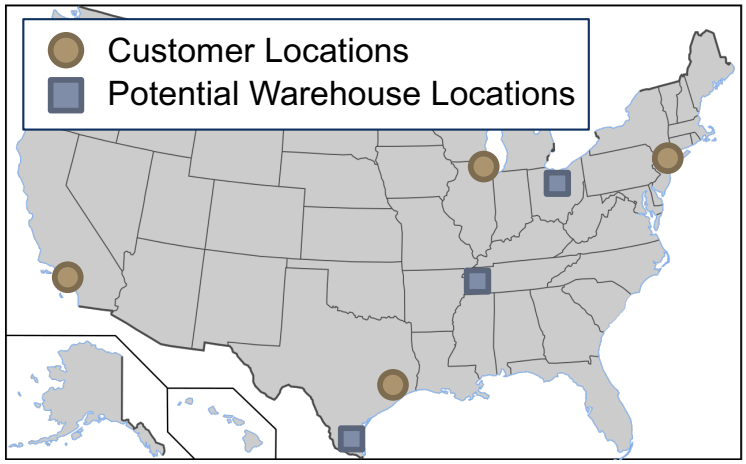
1.4 Loading data from Excel: In the knapsack example shown in the tutorial slides, the data is hardcoded at the top of the file. Instead of hard-coding the data, use Python to load the data from a different source. You can start from the file `knapsack_pandas_excel_incomplete.py`. (A solution that uses pandas to load the data from Excel is shown in `knapsack_pandas_excel_soln.py`.)

1.5 NLP vs MIP: Solve the knapsack problem with IPOPT instead of `glpk`. (Hint: switch `glpk` to `ipopt` in the call `SolverFactory`. Print the solution values for `model.x`. What happened? Why?

CCR Center for Computing Research

PYOMO COIN-OR

More Complex Example: Warehouse Location



- Determine the set of P warehouses chosen from the set W of candidates to minimize the cost of serving all customers C
- Here $d_{w,c}$ is the cost of serving customer c from warehouse at location w.

$$\min \sum_{w \in W, c \in C} d_{w,c} x_{w,c}$$

$$s.t. \quad \sum_{w \in W} x_{w,c} = 1 \qquad \forall\ c \in C$$

$$x_{w,c} \leq y_w \qquad \forall\ w \in W, c \in C$$

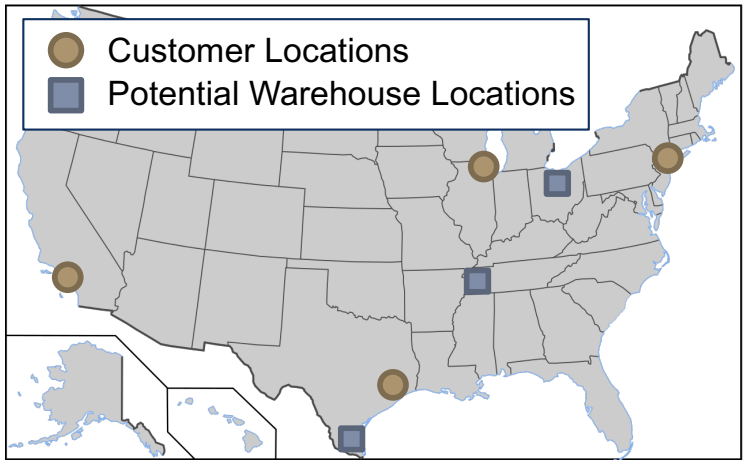
$$\sum_{w \in W} y_w = P$$

$$0 \leq x \leq 1 \qquad y \in \{0,1\}^{|W|}$$

y_w: discrete variable (location w selected or not)

$x_{w,c}$: fraction of demand of customer c served by warehouse w

More Complex Example: Warehouse Location



- Determine the set of P warehouses chosen from the set W of candidates to minimize the cost of serving all customers C
- Here $d_{w,c}$ is the cost of serving customer c from warehouse at location w.

$$\min \sum_{w \in W, c \in C} d_{w,c} x_{w,c} \qquad \text{(minimize total cost)}$$

$$s.t. \quad \sum_{w \in W} x_{w,c} = 1 \qquad \forall\ c \in C \qquad \text{(guarantee all customers served)}$$

$$x_{w,c} \leq y_w \qquad \forall\ w \in W, c \in C \qquad \text{(customer c can only be served from warehouse w if warehouse w is selected)}$$

$$\sum_{w \in W} y_w = P \qquad \text{(select } P \text{ warehouses)}$$

$$0 \leq x \leq 1 \qquad y \in \{0,1\}^{|W|}$$

Key difference?

Sandia National Laboratories

Knapsack

$$\max_{x} \quad \sum_{i \in A} v_i x_i$$

$$s.t. \quad \sum_{i \in A} w_i x_i \leq W_{\max}$$

$$x \in \{0,1\}^{|A|}$$

Warehouse Location

$$\min \quad \sum_{w \in W, c \in C} d_{w,c} x_{w,c}$$

$$s.t. \quad \sum_{w \in W} x_{w,c} = 1 \qquad \forall \ c \in C$$

$$x_{w,c} \leq y_w \qquad \forall \ w \in W, c \in C$$

$$\sum_{w \in W} y_w = P$$

$$0 \leq x \leq 1 \qquad y \in \{0,1\}^{|W|}$$

Key difference: Scalar vs Indexed Constraint

Sandia National Laboratories

Knapsack

$$\max_{x} \quad \sum_{i \in A} v_i x_i$$

$$s.t. \quad \sum_{i \in A} w_i x_i \leq W_{\max}$$

$$x \in \{0,1\}^{|A|}$$

$$w_{hammer} x_{hammer} + \cdots + w_{wrench} x_{wrench} \leq W_{\max}$$

Warehouse Location

$$\min \quad \sum_{w \in W, c \in C} d_{w,c} x_{w,c}$$

$$s.t. \quad \sum_{w \in W} x_{w,c} = 1 \qquad \forall \ c \in C$$

$$x_{w,c} \leq y_w \qquad \forall \ w \in W, c \in C$$

$$\sum_{w \in W} y_w = P$$

$$0 \leq x \leq 1 \qquad y \in \{0,1\}^{|W|}$$

Key difference: Scalar vs Indexed Constraint

Sandia National Laboratories

Knapsack

$$\max_x \sum_{i \in A} v_i x_i$$

$$s.t. \quad \sum_{i \in A} w_i x_i \leq W_{\max}$$

$$x \in \{0,1\}^{|A|}$$

$$w_{hammer} x_{hammer} + \cdots + w_{wrench} x_{wrench} \leq W_{\max}$$

Warehouse Location

$$\min \sum_{w \in W, c \in C} d_{w,c} x_{w,c}$$

$$s.t. \quad \sum_{w \in W} x_{w,c} = 1 \qquad \forall\ c \in C$$

$$x_{w,c} \leq y_w \qquad \forall\ w \in W, c \in C$$

$$\sum_{w \in W} y_w = P$$

$$0 \leq x \leq 1 \qquad y \in \{0,1\}^{|W|}$$

$$x_{Har,NYC} + x_{Mem,NYC} + x_{Ash,NYC} = 1$$

$$x_{Har,LA} + x_{Mem,LA} + x_{Ash,LA} = 1$$

$$x_{Har,Chi} + x_{Mem,Chi} + x_{Ash,Chi} = 1$$

$$x_{Har,Hou} + x_{Mem,Hou} + x_{Ash,Hou} = 1$$

W \ C	NYC	LA	Chicago	Houston
Harlingen	1956	1606	1410	330
Memphis	1096	1792	531	567
Ashland	485	2322	324	1236

CCR Center for Computing Research

PYOMO COIN-OR

Key difference: Scalar vs Indexed Constraint

Knapsack

$$\max_x \sum_{i \in A} v_i x_i$$

$$s.t. \quad \sum_{i \in A} w_i x_i \leq W_{\max}$$

$$x \in \{0,1\}^{|A|}$$

Warehouse Location

$$\min \sum_{w \in W, c \in C} d_{w,c} x_{w,c}$$

$$s.t. \quad \sum_{w \in W} x_{w,c} = 1 \qquad \forall\ c \in C$$

$$x_{w,c} \leq y_w \qquad \forall\ w \in W, c \in C$$

$$\sum_{w \in W} y_w = P$$

$$0 \leq x \leq 1 \qquad y \in \{0,1\}^{|W|}$$

- Knapsack problem: All constraints are singular (i.e., scalar)
- Warehouse location problem:
  - Multiple constraints defined over indices
- Pyomo used the concept of *construction rules* to specify indexed constraints

Indexed Constraints: Construction Rules

$$\sum_{w \in W} x_{w,c} = 1 \quad \forall \; c \in C$$

```
W = ['Harlingen', 'Memphis', 'Ashland']
C = ['NYC', 'LA', 'Chicago', 'Houston']
model.x = Var(W, C, bounds=(0,1))
model.y = Var(W, within=Binary)

def one_per_cust_rule(m, c):
    return sum(m.x[w,c] for w in W) == 1
model.one_per_cust = Constraint(C, rule=one_per_cust_rule)
```

Particular index is passed as argument to the rule

For indexed constraints, you provide a "rule" (function) that returns an expression (or tuple) for each index.

Rule is called once for every entry in C!

Construction Rules: Multiple indices

$$x_{w,c} \leq y_w \quad \forall\ w \in W, c \in C$$

```
W = ['Harlingen', 'Memphis', 'Ashland']
C = ['NYC', 'LA', 'Chicago', 'Houston']
model.x = Var(W, C, bounds=(0,1))
model.y = Var(W, within=Binary)

def warehouse_active_rule(m, w, c):
    return m.x[w,c] <= m.y[w]
model.warehouse_active = Constraint(W, C, rule=warehouse_active_rule)
```

Each dimension of the indices is a separate argument to the rule

Rule is called once for every entry w in W crossed with every entry c in C!

Construction Rules: Complex logic

$$x_i = \begin{cases} \cos(y_i), & i \text{ even} \\ \sin(y_i), & i \text{ odd} \end{cases} \qquad \forall \{i \in N : i <> 0\}$$

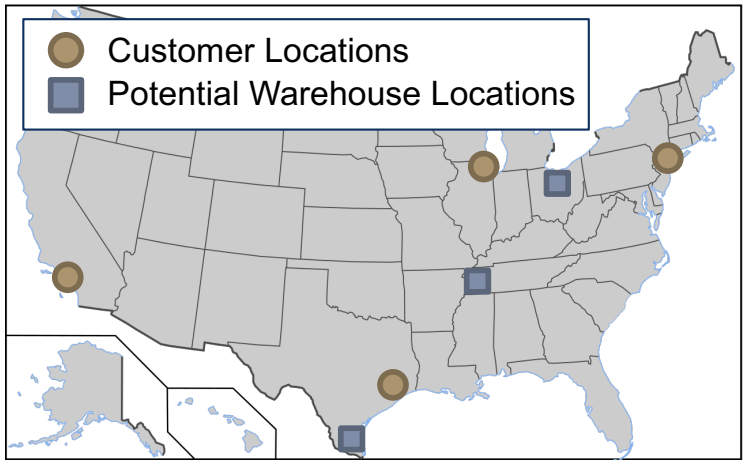
```
def complex_rule(model, i):
    if i == 0:
        return Constraint.Skip
    elif i % 2 == 0:
        return model.x[i] == cos(model.y[i])
    return model.x[i] == sin(model.y[i])
model.complex_constraint = Constraint(N, M, rule=complex_rule)
```

Constraint rules can contain complex logic on the indices and other parameters, but NOT on the value of model variables.

More on *Construction Rules*

- Components are constructed in declaration order
  - The instructions for *how* to construct the object are provided through a function, or *rule*
  - Pyomo calls the rule for each component index
  - *Rules* can be provided to virtually all Pyomo components
- Naming conventions
  - the component name prepended with "_" (`c4` → `_c4`)
  - the component name with "`_rule`" appended (`c4` → `c4_rule`)
  - each rule is called "`rule`" (Python implicitly overrides each declaration)
- Abstract models (discussed later) construct the model in two passes:
  - Python parses the model declaration
    - creating "empty" Pyomo components in the model
  - Pyomo loads and parses external data

Putting it all together: Warehouse Location



- Determine the set of P warehouses chosen from the set W of candidates to minimize the cost of serving all customers C
- Here $d_{w,c}$ is the cost of serving customer c from warehouse at location w.

$$\min \sum_{w \in W, c \in C} d_{w,c} x_{w,c}$$

$$s.t. \quad \sum_{w \in W} x_{w,c} = 1 \qquad \forall\ c \in C$$

$$x_{w,c} \leq y_w \qquad \forall\ w \in W, c \in C$$

$$\sum_{w \in W} y_w = P$$

$$0 \leq x \leq 1 \qquad y \in \{0,1\}^{|W|}$$

W C	NYC	LA	Chicago	Houston
Harlingen	1956	1606	1410	330
Memphis	1096	1792	531	567
Ashland	485	2322	324	1236

$P = 2$

Putting It All Together: Warehouse Location

Sandia National Laboratories

```
# warehouse_location.py: Warehouse location determination problem
from pyomo.environ import *

model = ConcreteModel(name="(WL)")

W = ['Harlingen', 'Memphis', 'Ashland']
C = ['NYC', 'LA', 'Chicago', 'Houston']
d = {('Harlingen', 'NYC'): 1956, \
    . . .
    ('Ashland', 'Houston'): 1236 }
P = 2

model.x = Var(W, C, bounds=(0,1))
model.y = Var(W, within=Binary)

def obj_rule(m):
    return sum(d[w,c]*m.x[w,c] for w in W for c in C)
model.obj = Objective(rule=obj_rule)

def one_per_cust_rule(m, c):
    return sum(m.x[w,c] for w in W) == 1
model.one_per_cust = Constraint(C, rule=one_per_cust_rule)

def warehouse_active_rule(m, w, c):
    return m.x[w,c] <= m.y[w]
model.warehouse_active = Constraint(W, C, rule=warehouse_active_rule)

def num_warehouses_rule(m):
    return sum(m.y[w] for w in W) <= P
model.num_warehouses = Constraint(rule=num_warehouses_rule)

SolverFactory('glpk').solve(model)

model.pprint()
```

Putting It All Together: Warehouse Location

Sandia National Laboratories

```
# warehouse_location.py: Warehouse location determination problem
from pyomo.environ import *

model = ConcreteModel(name="(WL)")

W = ['Harlingen', 'Memphis', 'Ashland']
C = ['NYC', 'LA', 'Chicago', 'Houston']
d = {('Harlingen', 'NYC'): 1956, \
    . . .
    ('Ashland', 'Houston'): 1236 }
P = 2

model.x = Var(W, C, bounds=(0,1))
model.y = Var(W, within=Binary)

def obj_rule(m):
    return sum(d[w,c]*m.x[w,c] for w in W for c in C)
model.obj = Objective(rule=obj_rule)

def one_per_cust_rule(m, c):
    return sum(m.x[w,c] for w in W) == 1
model.one_per_cust = Constraint(C, rule=one_per_cust_rule)

def warehouse_active_rule(m, w, c):
    return m.x[w,c] <= m.y[w]
model.warehouse_active = Constraint(W, C, rule=warehouse_active_rule)

def num_warehouses_rule(m):
    return sum(m.y[w] for w in W) <= P
model.num_warehouses = Constraint(rule=num_warehouses_rule)

SolverFactory('glpk').solve(model)

model.pprint()
```

Putting It All Together: Warehouse Location

Sandia National Laboratories

```
# warehouse_location.py: Warehouse location determination problem
from pyomo.environ import *

model = ConcreteModel(name="(WL)")

W = ['Harlingen', 'Memphis', 'Ashland']
C = ['NYC', 'LA', 'Chicago', 'Houston']
d = {('Harlingen', 'NYC'): 1956, \
    . . .
    ('Ashland', 'Houston'): 1236 }
P = 2

model.x = Var(W, C, bounds=(0,1))
model.y = Var(W, within=Binary)

def obj_rule(m):
    return sum(d[w,c]*m.x[w,c] for w in W for c in C)
model.obj = Objective(rule=obj_rule)

def one_per_cust_rule(m, c):
    return sum(m.x[w,c] for w in W) == 1
model.one_per_cust = Constraint(C, rule=one_per_cust_rule)

def warehouse_active_rule(m, w, c):
    return m.x[w,c] <= m.y[w]
model.warehouse_active = Constraint(W, C, rule=warehouse_active_rule)

def num_warehouses_rule(m):
    return sum(m.y[w] for w in W) <= P
model.num_warehouses = Constraint(rule=num_warehouses_rule)

SolverFactory('glpk').solve(model)

model.pprint()
```

Putting It All Together: Warehouse Location

Sandia National Laboratories

$x_{w,c} \quad \forall\, w \in W, c \in C$

$y_w \quad \forall\, w \in W$

```
# warehouse_location.py: Warehouse location determination problem
from pyomo.environ import *

model = ConcreteModel(name="(WL)")

W = ['Harlingen', 'Memphis', 'Ashland']
C = ['NYC', 'LA', 'Chicago', 'Houston']
d = {('Harlingen', 'NYC'): 1956, \
    . . .
    ('Ashland', 'Houston'): 1236 }
P = 2

model.x = Var(W, C, bounds=(0,1))
model.y = Var(W, within=Binary)

def obj_rule(m):
    return sum(d[w,c]*m.x[w,c] for w in W for c in C)
model.obj = Objective(rule=obj_rule)

def one_per_cust_rule(m, c):
    return sum(m.x[w,c] for w in W) == 1
model.one_per_cust = Constraint(C, rule=one_per_cust_rule)

def warehouse_active_rule(m, w, c):
    return m.x[w,c] <= m.y[w]
model.warehouse_active = Constraint(W, C, rule=warehouse_active_rule)

def num_warehouses_rule(m):
    return sum(m.y[w] for w in W) <= P
model.num_warehouses = Constraint(rule=num_warehouses_rule)

SolverFactory('glpk').solve(model)

model.pprint()
```

Putting It All Together: Warehouse Location

Sandia National Laboratories

$$\min_{x,y} \sum_{w \in W, c \in C} d_{w,c} x_{w,c}$$

```
# warehouse_location.py: Warehouse location determination problem
from pyomo.environ import *

model = ConcreteModel(name="(WL)")

W = ['Harlingen', 'Memphis', 'Ashland']
C = ['NYC', 'LA', 'Chicago', 'Houston']
d = {('Harlingen', 'NYC'): 1956, \
    . . .
    ('Ashland', 'Houston'): 1236 }
P = 2

model.x = Var(W, C, bounds=(0,1))
model.y = Var(W, within=Binary)

def obj_rule(m):
    return sum(d[w,c]*m.x[w,c] for w in W for c in C)
model.obj = Objective(rule=obj_rule)

def one_per_cust_rule(m, c):
    return sum(m.x[w,c] for w in W) == 1
model.one_per_cust = Constraint(C, rule=one_per_cust_rule)

def warehouse_active_rule(m, w, c):
    return m.x[w,c] <= m.y[w]
model.warehouse_active = Constraint(W, C, rule=warehouse_active_rule)

def num_warehouses_rule(m):
    return sum(m.y[w] for w in W) <= P
model.num_warehouses = Constraint(rule=num_warehouses_rule)

SolverFactory('glpk').solve(model)

model.pprint()
```

Putting It All Together: Warehouse Location

Sandia National Laboratories

$$\sum_{w \in W} x_{w,c} = 1 \quad \forall c \in C$$

```
# warehouse_location.py: Warehouse location determination problem
from pyomo.environ import *

model = ConcreteModel(name="(WL)")

W = ['Harlingen', 'Memphis', 'Ashland']
C = ['NYC', 'LA', 'Chicago', 'Houston']
d = {('Harlingen', 'NYC'): 1956, \
    . . .
    ('Ashland', 'Houston'): 1236 }
P = 2

model.x = Var(W, C, bounds=(0,1))
model.y = Var(W, within=Binary)

def obj_rule(m):
    return sum(d[w,c]*m.x[w,c] for w in W for c in C)
model.obj = Objective(rule=obj_rule)

def one_per_cust_rule(m, c):
    return sum(m.x[w,c] for w in W) == 1
model.one_per_cust = Constraint(C, rule=one_per_cust_rule)

def warehouse_active_rule(m, w, c):
    return m.x[w,c] <= m.y[w]
model.warehouse_active = Constraint(W, C, rule=warehouse_active_rule)

def num_warehouses_rule(m):
    return sum(m.y[w] for w in W) <= P
model.num_warehouses = Constraint(rule=num_warehouses_rule)

SolverFactory('glpk').solve(model)

model.pprint()
```

Putting It All Together: Warehouse Location

Sandia National Laboratories

$x_{w,c} \leq y_w \; \forall \, w \in W, c \in C$

```
# warehouse_location.py: Warehouse location determination problem
from pyomo.environ import *

model = ConcreteModel(name="(WL)")

W = ['Harlingen', 'Memphis', 'Ashland']
C = ['NYC', 'LA', 'Chicago', 'Houston']
d = {('Harlingen', 'NYC'): 1956, \
    . . .
    ('Ashland', 'Houston'): 1236 }
P = 2

model.x = Var(W, C, bounds=(0,1))
model.y = Var(W, within=Binary)

def obj_rule(m):
    return sum(d[w,c]*m.x[w,c] for w in W for c in C)
model.obj = Objective(rule=obj_rule)

def one_per_cust_rule(m, c):
    return sum(m.x[w,c] for w in W) == 1
model.one_per_cust = Constraint(C, rule=one_per_cust_rule)

def warehouse_active_rule(m, w, c):
    return m.x[w,c] <= m.y[w]
model.warehouse_active = Constraint(W, C, rule=warehouse_active_rule)

def num_warehouses_rule(m):
    return sum(m.y[w] for w in W) <= P
model.num_warehouses = Constraint(rule=num_warehouses_rule)

SolverFactory('glpk').solve(model)

model.pprint()
```

Putting It All Together: Warehouse Location

Sandia National Laboratories

$$\sum_{w \in W} y_w = P$$

```
# warehouse_location.py: Warehouse location determination problem
from pyomo.environ import *

model = ConcreteModel(name="(WL)")

W = ['Harlingen', 'Memphis', 'Ashland']
C = ['NYC', 'LA', 'Chicago', 'Houston']
d = {('Harlingen', 'NYC'): 1956, \
    . . .
    ('Ashland', 'Houston'): 1236 }
P = 2

model.x = Var(W, C, bounds=(0,1))
model.y = Var(W, within=Binary)

def obj_rule(m):
    return sum(d[w,c]*m.x[w,c] for w in W for c in C)
model.obj = Objective(rule=obj_rule)

def one_per_cust_rule(m, c):
    return sum(m.x[w,c] for w in W) == 1
model.one_per_cust = Constraint(C, rule=one_per_cust_rule)

def warehouse_active_rule(m, w, c):
    return m.x[w,c] <= m.y[w]
model.warehouse_active = Constraint(W, C, rule=warehouse_active_rule)

def num_warehouses_rule(m):
    return sum(m.y[w] for w in W) <= P
model.num_warehouses = Constraint(rule=num_warehouses_rule)

SolverFactory('glpk').solve(model)

model.pprint()
```

Putting It All Together: Warehouse Location

Sandia National Laboratories

```
# warehouse_location.py: Warehouse location determination problem
from pyomo.environ import *

model = ConcreteModel(name="(WL)")

W = ['Harlingen', 'Memphis', 'Ashland']
C = ['NYC', 'LA', 'Chicago', 'Houston']
d = {('Harlingen', 'NYC'): 1956, \
    . . .
    ('Ashland', 'Houston'): 1236 }
P = 2

model.x = Var(W, C, bounds=(0,1))
model.y = Var(W, within=Binary)

def obj_rule(m):
    return sum(d[w,c]*m.x[w,c] for w in W for c in C)
model.obj = Objective(rule=obj_rule)

def one_per_cust_rule(m, c):
    return sum(m.x[w,c] for w in W) == 1
model.one_per_cust = Constraint(C, rule=one_per_cust_rule)

def warehouse_active_rule(m, w, c):
    return m.x[w,c] <= m.y[w]
model.warehouse_active = Constraint(W, C, rule=warehouse_active_rule)

def num_warehouses_rule(m):
    return sum(m.y[w] for w in W) <= P
model.num_warehouses = Constraint(rule=num_warehouses_rule)

SolverFactory('glpk').solve(model)

model.pprint()
```

Putting It All Together: Warehouse Location

Sandia National Laboratories

```
# warehouse_location.py: Warehouse location determination problem
from pyomo.environ import *

model = ConcreteModel(name="(WL)")

W = ['Harlingen', 'Memphis', 'Ashland']
C = ['NYC', 'LA', 'Chicago', 'Houston']
d = {('Harlingen', 'NYC'): 1956, \
    . . .
    ('Ashland', 'Houston'): 1236 }
P = 2

model.x = Var(W, C, bounds=(0,1))
model.y = Var(W, within=Binary)

def obj_rule(m):
    return sum(d[w,c]*m.x[w,c] for w in W for c in C)
model.obj = Objective(rule=obj_rule)

def one_per_cust_rule(m, c):
    return sum(m.x[w,c] for w in W) == 1
model.one_per_cust = Constraint(C, rule=one_per_cust_rule)

def warehouse_active_rule(m, w, c):
    return m.x[w,c] <= m.y[w]
model.warehouse_active = Constraint(W, C, rule=warehouse_active_rule)

def num_warehouses_rule(m):
    return sum(m.y[w] for w in W) <= P
model.num_warehouses = Constraint(rule=num_warehouses_rule)

SolverFactory('glpk').solve(model)

model.pprint()
```

Pyomo Fundamentals: Exercises #2

2.1 Knapsack problem with rules: Rules are important for defining indexed constraints, however, they can also be used for single (i.e. scalar) constraints. Starting with `knapsack.py`, reimplement the model using rules for the objective and the constraints. (A solution can be found in `knapsack_rules_soln.py`.)

2.2 Integer formulation of the knapsack problem: Consider again, the knapsack problem. Assume now that we can acquire multiple items of the same type. In this new formulation, x_i is now an integer variable instead of a binary variable. One way to formulate this problem is as follows:

$$\max_{q,x} \sum_{i \in A} v_i x_i$$

$$\text{s.t.} \sum_{i \in A} w_i x_i \leq W_{\max}$$

$$x_i = \sum_{j=0}^{N} j q_{i,j} \qquad \forall\, i \in A$$

$$0 \leq x \leq N$$

$$q_{i,j} \in \{0, 1\} \qquad \forall i \in A, j \in \{0..N\}$$

Starting with `knapsack_rules.py`, implement this new formulation and solve. Is the solution surprising? (A solution can be found in `knapsack_integer_soln.py`.)

Decorator notation

- Reduce redundancy in *rule definitions* using *Decorators*

```
@model.Constraint(W, C)
def warehouse_active(m, w, c):
    return m.x[w,c] <= m.y[w]
```

- General notation

```
@<object>.<Component>(indices, …, keywords=…)
def my_thing(m, index, …):
    return # …
```

Is equivalent to

```
def my_thing(m, index, …):
    return # …
<object>.my_thing = <Component>(indices, …, rule=my_thing, keywords=…)
```

Note: this syntax *only* works for defining the *rule* keyword.

Parameters

- Pyomo models can be built using standard python numeric types (e.g., int, float) for all constants/data
  - This is how we have done examples so far.

- Pyomo also supports a parameter component (Param)
  - Keeps data documented on the model
  - Allows for validation of data, default values, and changes in data without the need to rebuild entire model
  - Allows Abstract model definitions (declare model, apply data later)

- Scalar numeric values

Provide an (initial) value of 42 for the parameter

```
model.a_parameter = Param( initialize = 42 )
```

Parameters

- Indexed numeric values

```
model.a_param_vec = Param( IDX,
                           initialize = data,
                           default = 0 )
```

Providing "default" allows the initialization data to only specify the "unusual" values

"data" *must* be a dictionary(*) of index keys to values because all sets are assumed to be *unordered*

(*) – actually, it must define __getitem__(), but that only really matters to Python geeks

- Mutable Parameters

```
model.a_parameter = Param( initialize = 42,
                           mutable = True )
```

Indicates to Pyomo that you may want to change this parameter later.

Generating and Managing Indices: *Sets*

- Any iterable object can be an index, e.g., lists:
  - `IDX_a = [1,2,5]`
  - ```
    DATA = {1: 10, 2: 21, 5:42};
    IDX_b = DATA.keys()
    ```

- Sets: objects for managing multidimensional indices
  - `model.IDX = Set( initialize = [1,2,5] )`

Note: capitalization matters:
Set = Pyomo class
set = native Python set

Like indices, Sets can be initialized from any iterable

  - `model.IDX = Set( [1,2,5] )`

Note: This does not mean what you think it does.
This creates a 3-member *indexed set*, where each set is *empty.*
    ```

Sequential Indices: *RangeSet*

- Sets of sequential integers are common
  - `model.IDX = Set( initialize=range(5) )`
  - `model.IDX = RangeSet( 5 )`

Note: RangeSet is 1-based.
This gives [1, 2, 3, 4, 5]

Note: Python range is 0-based.
This gives [0, 1, 2, 3, 4]

- You can provide lower and upper bounds to RangeSet
  - `model.IDX = RangeSet( 0, 4 )`

This gives [0, 1, 2, 3, 4]

Manipulating Sets

- Creating sparse sets

```
model.IDX = Set( initialize=[1,2,5] )
def lower_tri_filter(model, i, j):
      return j <= i
model.LTRI = Set( initialize = model.IDX * model.IDX,
                  filter = lower_tri_filter )
```

The `filter` should return *True* if the element is in the set; *False* otherwise.

- Sets support efficient higher-dimensional indices

```
model.IDX = Set( initialize=[1,2,5] )
model.IDX2 = model.IDX * model.IDX
```

This creates a *virtual* 2-D "matrix" Set

Sets also support union (&), intersection (|), difference (-), symmetric difference (^)

Higher-Dimensional Sets and Flattening

- Higher-dimensional sets contain multiple indices per element

```
model.IDX = Set( initialize=[1,2,5] )
model.IDX2 = model.IDX * model.IDX

tuples = list()
for i in model.IDX:
    for j in model.IDX:
        tuples.append( (i,j) )
model.IDXT = Set( initialize=tuples )
```

- IDX2 == IDXT == [(1,1), (1,2), (1,5), (2,1), (2,2), (2,5), (5,1), (5,2), (5,5)]

- Higher-dimensional sets and rules

```
def c5_rule(model, i, j, k):
    return model.a[i] + model.a[j] + model.a[k] <= 1
model.c5 = Constraint( model.IDX2, model.IDX, rule=c5_rule )
```

Each dimension of each index is a separate argument to the rule

Set ordering

- By default, Sets are *unordered* (just like Python sets and dictionaries)

```
model.IDX = Set( initialize=[1,2,5] )
for i in model.IDX:
    print(i)
```

No specific order guaranteed for this loop (e.g., 1,5,2 … 5,1,2 …)

```
model.x = Var(model.IDX)
for k in model.x:
    print(value(model.x[i]))
```

This is also true for variables indexed by unordered sets.

```
model.IDXO = Set( initialize=[1,2,5], ordered=True )
```

Use the keyword argument `ordered=True` to guarantee fixed order.

Other Modeling Components

- Pyomo supports “list”-like indexed components (useful for meta-algorithms and addition of cuts)

```
model.a = Var()
model.b = Var()
model.c = Var()
model.limits = ConstraintList()

model.limits.add(30*model.a + 15*model.b + 10*model.c <= 100)
model.limits.add(10*model.a + 25*model.b +  5*model.c <= 75)
model.limits.add(6*model.a  + 11*model.b +  3*model.c <= 30)
```

“add” adds a single new constraint to the list.
The constraints need not be related.

Expression performance tips

- For reasons that are beyond this tutorial, the following can be VERY slow in Pyomo:
  - ```
    ans = 0
    for i in m.INDEX:
        ans = ans + m.x[i]
    ```
- Recommended alternatives are
  - ```
    ans = 0
    for i in m.INDEX:
        ans += m.x[i]
    ```
  - ```
    sum(m.x[i] for i in m.INDEX)
    ```
- *Note that this is likely to change in Pyomo 5.6, where all three forms will be relatively equivalent.*

- To report the construction of individual Pyomo components, call:

  ```
  pyomo.util.timing.report_timing()
  ```

  before building your model.
  ```

Pyomo Fundamentals: Exercises #3

3.1 Changing Parameter values: In the tutorial slides, we saw that a parameter could be specified to be `mutable`. This tells Pyomo that the value of the parameter may change in the future, and allows the user to change the parameter value and resolve the problem without the need to rebuild the entire model each time. We will use this functionality to find a better solution to an earlier exercise. Considering again the knapsack problem, we would like to find when the wrench becomes valuable enough to be a part of the optimal solution. Create a Pyomo `Parameter` for the value of the items, make it mutable, and then write a loop that prints the solution for different wrench values. Start with the file `knapsack_mutable_parameter_incomplete.py`. (A solution for this problem can be found in `knapsack_mutable_parameter_soln.py`.)

3.2 Integer cuts: Often, it can be important to find not only the "best" solution, but a number of solutions that are equally optimal, or close to optimal. For discrete optimization problems, this can be done using something known as an integer cut. Consider again the knapsack problem where the choice of which items to select is a discrete variable $x_i \; \forall \; i \in A$. Let $x_i^\star$ be a particular set of x values we want to remove from the feasible solution space. We define an integer cut using two sets. The first set S_0 contains the indices for those variables whose current solution is 0, and the second set S_1 consists of indices for those variables whose current solution is 1. Given these two sets, an integer cut constraint that would prevent such a solution from appearing again is defined by,

$$\sum_{i \in S_0} x[i] + \sum_{i \in \in S_1} (1 - x[i]) \geq 1.$$

Starting with `knapsack_rules.py`, write a loop that solves the problem 5 times, adding an integer cut to remove the previous solution, and printing the value of the objective function and the solution at each iteration of the loop. (A solution for this problem can be found in `knapsack_integer_cut_soln.py`

Pyomo Fundamentals: Exercises #3

3.3 Putting it all together with the lot sizing example: We will now write a complete model from scratch using a well-known multi-period optimization problem for optimal lot-sizing adapted from Hagen et al. (2001) shown below.

$$\min \sum_{t \in T} c_t y_t + h_t^+ I_t^+ + h_t^- I_t^- \tag{1}$$

$$\text{s.t.} \quad I_t = I_{t-1} + X_t - d_t \qquad \forall t \in T \tag{2}$$

$$I_t = I_t^+ - I_t^- \qquad \forall t \in T \tag{3}$$

$$X_t \leq P y_t \qquad \forall t \in T \tag{4}$$

$$X_t,\ I_t^+,\ I_t^- \geq 0 \qquad \forall t \in T \tag{5}$$

$$y_t \in \{0, 1\} \qquad \forall t \in T \tag{6}$$

Our goal is to find the optimal production X_t given known demands d_t, fixed cost c_t associated with active production in a particular time period, an inventory holding cost h_t^+ and a shortage cost h_t^- (cost of keeping a backlog) of orders. The variable y_t (binary) determines if we produce in time t or not, and I_t^+ represents inventory that we are storing across time period t, while I_t^- represents the magnitude of the backlog. Note that equation (4) is a constraint that only allows production in time period t if the indicator variable y_t=1.

Write a Pyomo model for this problem and solve it using `glpk` using the data provided below. You can start with the file `lot_sizing_incomplete.py`. (A solution is provided in `lot_sizing_soln.py`.)

Parameter	Description	Value
c	fixed cost of production	4.6
I_0^+	initial value of positive inventory	5.0
I_0^-	initial value of backlogged orders	0.0
h^+	cost (per unit) of holding inventory	0.7
h^-	shortage cost (per unit)	1.2
P	maximum production amount (big-M value)	5
d	demand	$[5, 7, 6.2, 3.1, 1.7]$

Other Topics

Solving models: *the* pyomo *command*

- pyomo (`pyomo.exe` on Windows):
  - Constructs model and passes it to an (external) solver

```
pyomo solve <model_file> [<data_file> …] [options]
```

  - Installed to:
    - `[PYTHONHOME]\Scripts` [Windows; `C:\Python27\Scripts`]
    - `[PYTHONHOME]/bin` [Linux; `/usr/bin`]

- Key options (*many* others; see `--help`)

`--help`	Get list of all options
`--help-solvers`	Get the list of all recognized solvers
`--solver=<solver_name>`	Set the solver that Pyomo will invoke
`--solver-options="key=value[ …]"`	Specify options to pass to the solver as a space-separated list of keyword-value pairs
`--stream-solver`	Display the solver output during the solve
`--summary`	Display a summary of the optimization result
`--report-timing`	Report additional timing information, including construction time for each model component

Abstract Modeling

Sandia National Laboratories

CCR Center for Computing Research

PYOMO COIN-OR

Concrete *vs.* Abstract Models

- Concrete Models: data first, then model
  - 1-pass construction
  - All data must be present before Python starts processing the model
  - Pyomo will construct each component in order at the time it is declared
  - Straightforward logical process; easy to script.
  - Familiar to modelers with experience with GAMS
- Abstract Models: model first, then data
  - 2-pass construction
  - Pyomo stores the basic model declarations, but does not construct the actual objects
    - Details on how to construct the component hidden in functions, or *rules*
    - e.g., it will declare an indexed variable "x", but will not expand the indices or populate any of the individual variable values.
  - At "creation time", data is applied to the abstract declaration to create a concrete instance (components are still constructed in declaration order)
  - Encourages generic modeling and model reuse
    - e.g., model can be used for arbitrary-sized inputs
  - Familiar to modelers with experience with AMPL

Data Sources

- Data can be imported from "`.dat`" file
  - Format similar to AMPL style
  - Explicit data from "`param`" declarations
  - External data through "`load`" declarations:
    - Excel

```
load ABCD.xls range=ABCD : Z=[A, B, C] Y=D ;
```

    - Databases

```
load "DBQ=diet.mdb" using=pyodbc query="SELECT FOOD, cost,
  f_min, f_max from Food" : [FOOD] cost f_min f_max ;
```

- External data overrides "`initialize=`" declarations

Abstract p-Median (pmedian.py, 1)

```
from pyomo.environ import *

model = AbstractModel()

model.N = Param( within=PositiveIntegers )
model.P = Param( within=RangeSet( model.N ) )
model.M = Param( within=PositiveIntegers )

model.Locations = RangeSet( model.N )
model.Customers = RangeSet( model.M )

model.d = Param( model.Locations, model.Customers )

model.x = Var( model.Locations, model.Customers, bounds=(0.0, 1.0) )
model.y = Var( model.Locations, within=Binary )
```

Abstract p-Median (pmedian.py, 2)

```
def obj_rule(model):
    return sum( model.d[n,m]*model.x[n,m]
       for n in model.Locations for m in model.Customers )
model.obj = Objective( rule=obj_rule )

def single_x_rule(model, m):
    return sum( model.x[n,m] for n in model.Locations ) == 1.0
model.single_x = Constraint( model.Customers, rule=single_x_rule )

def bound_y_rule(model, n,m):
    return model.x[n,m] - model.y[n] <= 0.0
model.bound_y = Constraint( model.Locations, model.Customers,
                            rule=bound_y_rule )

def num_facilities_rule(model):
    return sum( model.y[n] for n in model.Locations ) == model.P
model.num_facilities = Constraint( rule=num_facilities_rule )
```

Abstract p-Median (pmedian.dat)

```
param N := 3;

param M := 4;

param P := 2;

param d:  1     2     3     4 :=
     1   1.7   7.2   9.0   8.3
     2   2.9   6.3   9.8   0.7
     3   4.5   4.8   4.2   9.3 ;
```

In Class Exercise: Abstract Knapsack

$$\max \quad \sum_{i=1}^{N} v_i x_i$$

$$s.t. \quad \sum_{i=1}^{N} w_i x_i \le W_{\max}$$

$$x_i \in \{0,1\}$$

Item	Weight	Value
hammer	5	8
wrench	7	3
screwdriver	4	6
towel	3	11
Max weight:	14	

Syntax reminders:

```
AbstractModel()
Set( [index, …], [initialize=list/function] )
Param( [index, …], [within=domain], [initialize=dict/function] )
Var( [index, …], [within=domain], [bounds=(lower,upper)] )
Constraint( [index, …], [expr=expression|rule=function] )
Objective( sense={maximize|minimize},
           expr=expression|rule=function )
```

Abstract Knapsack: *Solution*

```
from pyomo.environ import *

model       = AbstractModel()
model.ITEMS = Set()
model.v     = Param( model.ITEMS, within=PositiveReals )
model.w     = Param( model.ITEMS, within=PositiveReals )
model.W_max = Param( within=PositiveReals )
model.x     = Var( model.ITEMS, within=Binary )

def value_rule(model):
    return sum( model.v[i]*model.x[i] for i in model.ITEMS )
model.value = Objective( rule=value_rule, sense=maximize )

def weight_rule(model):
    return sum( model.w[i]*model.x[i] for i in model.ITEMS ) \
        <= model.W_max
model.weight = Constraint( rule=weight_rule )
```

Abstract Knapsack: *Solution Data*

```
set ITEMS := hammer wrench screwdriver towel ;

param:  v w :=
    hammer         8 5
    wrench         3 7
    screwdriver    6 4
    towel         11 3;

param W_max := 14;
```