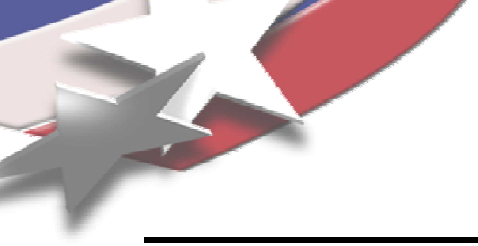


COOPR

SAND2012-9471P

Using Python and the Algebraic Modeling Language Pyomo to Specify, Solve, and Analyze Mathematical Programs

John D. Siirola and Jean-Paul Watson

Discrete Math and Complex Systems Department

Sandia National Laboratories

Albuquerque, NM USA

David Woodruff

Graduate School of Management

University of California Davis, USA

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

NNSA National Nuclear Security Administration

Sandia National Laboratories

0. Do You Have Pyomo Installed and Working?

- An interlude
  - But an important pre-requisite for what follows!

- If you don't:
  - Download & Install Python 2.7 (2.6 is also acceptable)
    - http://python.org/
    - If you are on Windows:
      - Use Python 2.7
  - Download & install Coopr
    - http://software.sandia.gov/trac/coopr/downloader
    - Linux / MacOS: coopr_install / coopr_votd
    - Windows: Coopr_*_setup.exe
  - Download & install "a solver":
    - glpk, cbc, ipopt are good starting points

1. Introduction to Pyomo

- Three really good questions:
  - *Why another Algebraic Modeling Language (AML)?*
  - *Why Python?*
  - *Why open-source?*

- Coopr: Software library infrastructure

- Coopr: Team overview and collaborators / users

- Where to find more information…

Why another AML: *Improve our productivity*

- Our initial objective was *not* to write another AML
  - However it ended up being a necessary prerequisite

- What we needed from a modeling environment:
  - Support rapid algorithm prototyping, development, and extension
  - Extensible to new modeling constructs (e.g., SP, GDP)
  - Facilitate hybrid approaches:
    - Hybrid models
    - Hybrid algorithms
    - Heuristic meta-algorithms
  - Transparent and accessible internal data structures
  - Interoperability (models, solvers, platforms, data sources)
  - Easily transferrable to non-expert user communities

Why Python: *it meets our needs*

- Python provides a full-featured object-oriented environment
  - Classes, inheritance, namespaces, exceptions, …
  - Interactive interpreter
- Python facilitates rapid prototyping and doesn't require a CS degree
  - Important for modelers and general productivity
- Python ships with a huge number of very useful libraries, including
  - Serialization, distributed computation, db/Excel interfaces, …
  - SciPy and NumPy
- Python has excellent support for dynamic loading
  - Critical for integrating 3rd-party extensions, custom user code
- Python introspection facilitates the development of generic algorithms

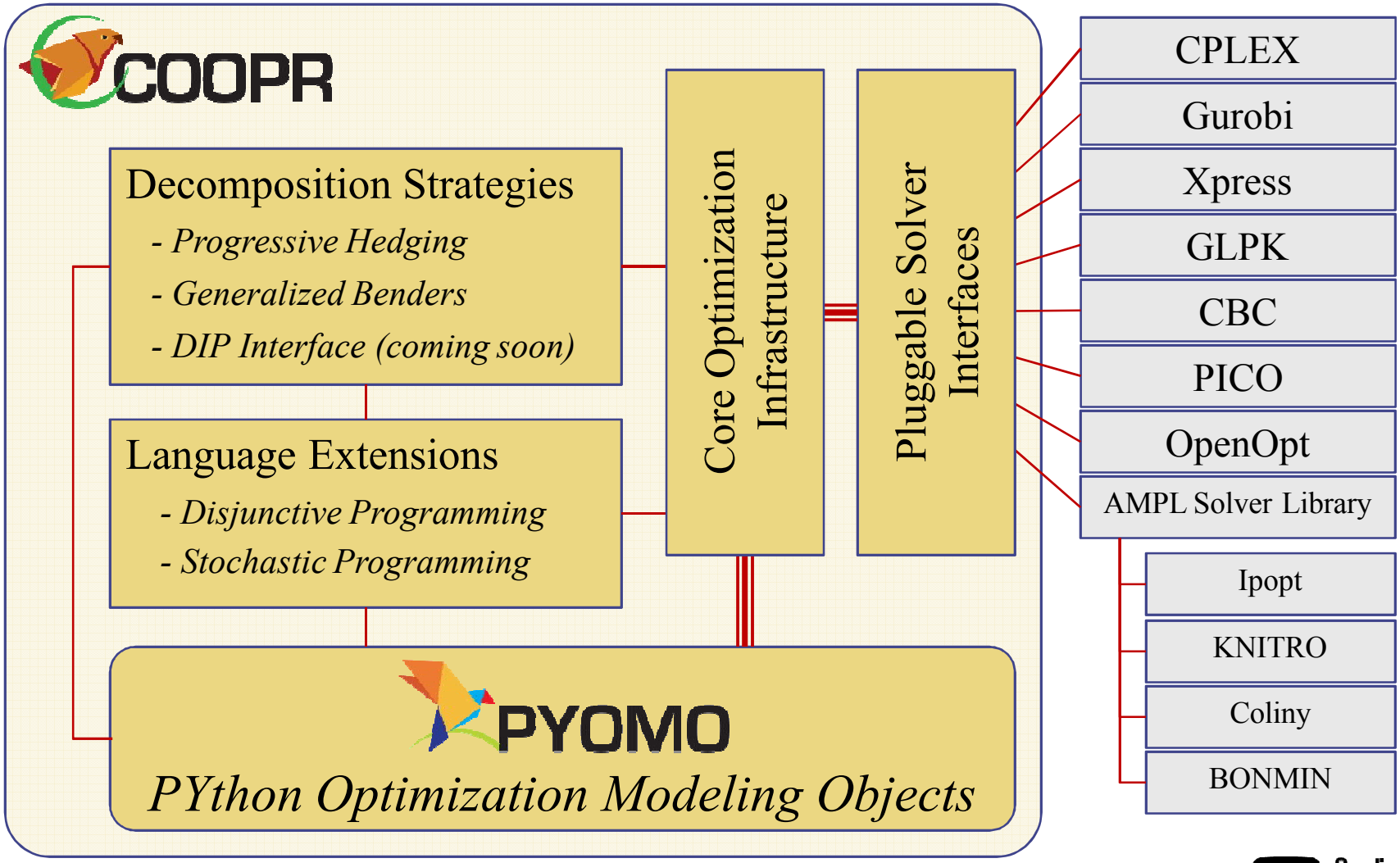
Why open source: *we want your involvement*

- Transparency and reliability

- Foster community involvement
  - Extend the modeling language
  - Develop new solvers / algorithms
  - Interface with additional external utilities
  - “Stone Soup” model

- Flexible licensing
  - Coopr/Pyomo released under 3-clause BSD license

Coopr: *a COmmon Optimization Python Repository*



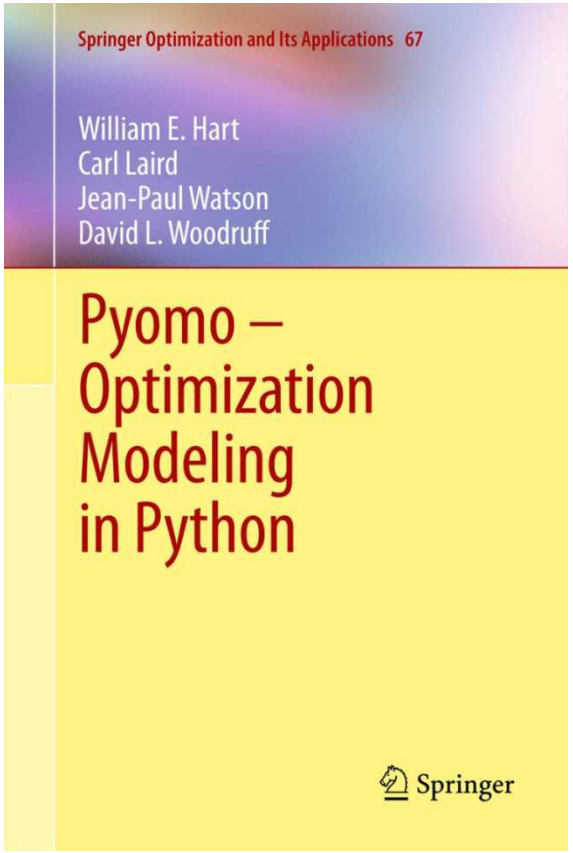
Team, Collaborators, (Known) Users

- Sandia National Laboratories
  - Bill Hart
  - Jean-Paul Watson
  - John Siirola
  - David Hart
  - Tom Brounstein
- University of California, Davis
  - Prof. David L. Woodruff
  - Prof. Roger Wets
- Texas A&M University
  - Prof. Carl D. Laird
  - Daniel Word
  - James Young
  - Gabe Hackebeil
- Carnegie Mellon University
  - Bethany Nicholson
- Texas Tech University
  - Zev Friedman
- Rose Hulman Institute
  - Tim Ekl
- William & Mary
  - Patrick Steele
- North Carolina State
  - Kevin Hunter

(Known) users:
- University of California, Davis
- Texas A&M University
- University of Texas
- Rose-Hulman Institute of Technology
- University of Southern California
- George Mason University
- Iowa State University
- N.C. State University
- University of Washington
- Naval Postgraduate School
- Universidad de Santiago de Chile
- University of Pisa
- Lawrence Livermore National Lab
- Los Alamos National Lab

For More Information…

- Project homepage
  - http://software.sandia.gov/coopr
- Mailing list
  - "coopr-forum" Google Group
  - "coopr-dev" Google Group
- "The Book"



- Mathematical Programming Computation papers
  - Pyomo: Modeling and Solving Mathematical Programs in Python (Vol. 3, No. 3, 2011)
  - PySP: Modeling and Solving Stochastic Programs in Python (To Appear)

2. An Introduction to Python

- Where credit is deserved…
  - Professor David Woodruff, University of California Davis

- Just enough Python to get you through the rest of the Pyomo tutorial

- For more information
  - www.python.org
  - Any of the great O'Reily books (www.ora.com)
  - Any of a zillion on-line tutorials

Python is a Calculator

- Python is often executed in an interactive mode
- But we don’t do this very often

Python is a Scripting Language

```
import sys
import os


for run in range(0, 3):
    print "Running forestfire.py in run
  number", run
    os.system(“myopt –r=”+str(run))
```

Python is a Programming Language

- Lists
- Dictionaries
- Sets and tuples, too
- First-class objects and functions
- (BTW: Python is an interpreted environment)

Lists (And Slicing)

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
>>> a[0]
'spam'
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
```

Dictionaries

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
 ['guido', 'irv', 'jack']
>>> 'guido' in tel
True
```

Functions

```
>>> def foo(x):
…       for I in range(1,x+1):
…           print i
>>> foo(2)
1
2
>>> def foo2(x):
…        return x*x
>>> foo2(4)
16
```

Python Provides an Application Development Framework

- Module definitions
- Class constructs
- And lots, lots more…

Python is Pretty Cool (Probably Cooler Than Ruby)

Many in the audience can probably say much more, but here are some items of note:

- List comprehensions
- Magic methods (lambda functions)
- Very extensive libraries (caveat: not everything is bullet proof; it is a wild world)
  - Exemplars: numpy, scipy, matplotlib
- Pickling (take the state of things and encode it as an ASCII string; almost (but still, it is pretty cool))
- Introspection
  - getattr and setattr

List Comprehensions

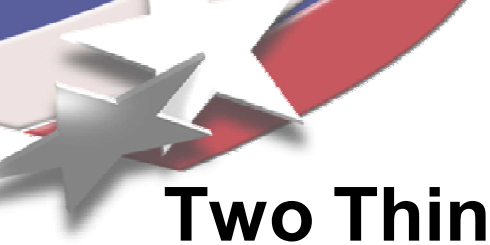
```
>>> vec = [2, 4, 6]

>>> [3*i for i in vec]
 [6, 12, 18]

>>> [3*i for i in vec if i > 3]
[12, 18]
```

Things You Might Find Odd and/or Useful

- Indentation is important
- Shallow vs. deep copy
- Variables are passed “by reference”
- (so) There are often many function return values
- First class functions (and objects)
- Mutable vs. immutable; defaults:
  - Strings: immutable
  - Integers: immutable
  - Objects: mutable



Two Things You Might Find Very Odd

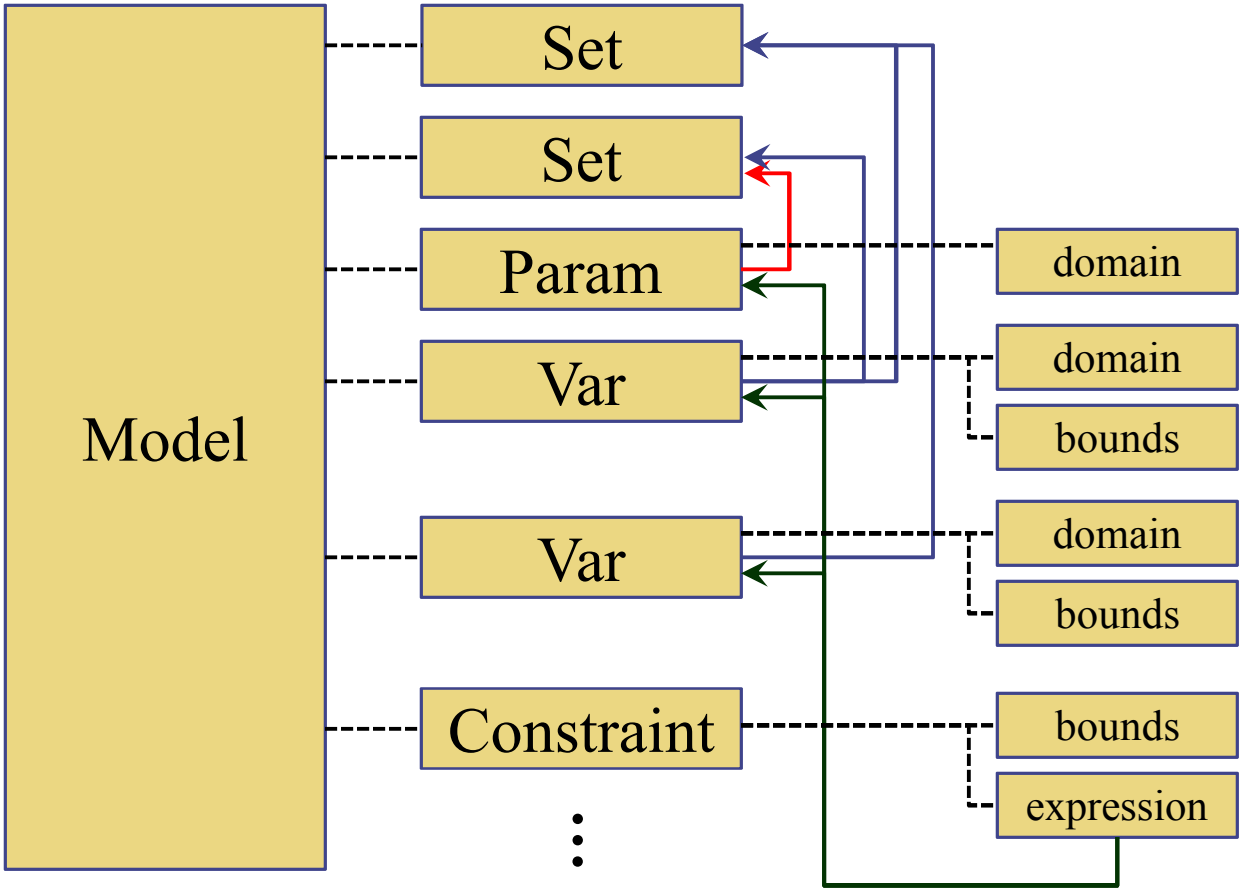
- Python is (very) weakly typed
- People get extremely excited about it

We Believe the Turing-Church Conjecture!!!!

- Everything algorithmically computable is also computable by a Turing machine
  - You probably don't want to program a Turing machine
- But we really value our time
  - Hence, our primary interest in Python
- Questions?

3. Fundamental Pyomo Components

- Pyomo is an *object model* for describing optimization problems



Getting Started: the *Model*

```
from coopr.pyomo 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`"
- If you choose to name your model something else, you will need to tell the Pyomo script the object name through the command line

Populating the Model: *Variables*

- Scalar variables

```
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))
```

Same as above: "domain" is assumed to be Reals if missing

- Indexed variables

```
model.a_vector = Var(IDX)
model.a_matrix = Var(IDX_A, IDX_B)
```

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

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])`

Like, indices, Sets can be initialized from any iterable

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

Note: This doesn't do what you want.
This creates a 3-member *indexed set*, where each set is *empty*.

Multidimensional Sets

- 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 index

- Creating sparse sets

```
model.IDX = Set(initialize=[1,2,5])
def lower_tri_filter(model, i, j):
    return j <= i
model.IDX2 = Set( initialize = model.IDX * model.IDX,
                  filter = lower_tri_filter )
```

The filter returns *True* if the element is in the set; *False* otherwise.

Defining the Problem: *Constraints*

```
model.IDX = Set(initialize=range(100))
model.a = Var()
model.b = Var(IDX)
model.c1 = Constraint(
     expr = sum(model.b[i] for i in model.IDX) <= model.a )
```

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

Python *list comprehensions* are very common for working over indexed variables

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

"`expr`" can also be a tuple:

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

Lists of Constraints

```
model.IDX = Set( initialize=range(10) )
model.b = Var(model.IDX)
model.c3 = ConstraintList()
for i in model.IDX:
    model.c3.add( (model.b[i] - i) ** 2 <= 1 )
```

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

Indexed Constraints and *rules*

```
model.IDX = Set(initialize=range(5))
model.a = Var(model.IDX)
model.b = Var()
def c4_rule(model, i):
    return model.a[i] + model.b <= 1
model.c4 = Constraint(model.IDX, rule = c4_rule)
```

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

NB: if you omit the "`rule`", Pyomo automatically looks for a function "<constraint name>`_rule`"

```
model.IDX2 = model.IDX * model.IDX
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

Defining the *Objective*

- Objectives are a special case of Constraint

```
model.IDX = Set(initialize=range(100))
model.b = Var(IDX)
model.obj = Objective(
     expr = sum(model.b[i] for i in model.IDX)
     sense = minimize )
```

If "`sense`" is omitted, Pyomo assumes minimization

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

4. Putting It All Together: *Concrete p-Median*

$$\min \quad \sum_{n \in N, m \in M} \mathrm{d}_{\mathrm{n,m}} x_{n,m}$$

$$s.t. \quad \sum_{n \in N} x_{n,m} = 1 \qquad \forall m \in M$$

$$x_{n,m} \le y_n \qquad \forall n \in N, m \in M$$

$$\sum_{n \in N} y_n = P$$

$$0 \le x \le 1 \qquad y \in \{0,1\}$$

Concrete p-Median (1)

```
from coopr.pyomo import *
import random
random.seed(1000)

N = 5
M = 6
P = 3
d = { (n,m): random.uniform(1.0,2.0)
      for n in range(N) for m in range(M) }

model = ConcreteModel()
model.Locations = RangeSet(1,N)
model.Customers = RangeSet(1,M)

model.x = Var( model.Locations, model.Customers,
               bounds=(0.0,1.0))

model.y = Var(model.Locations, within=Binary)
```

Concrete p-Median (2)

```
model.obj = Objective( expr = sum( d[n,m]*model.x[n,m]
    for n in model.Locations for m in model.Customers ) )

model.num_facilities = Constraint(
    expr=sum( model.y[n] for n in model.Locations) == P )

model.single_x = ConstraintList()
for m in model.Customers:
    model.single_x.add(
        sum( model.x[n,m] for n in model.Locations) == 1.0 )

model.bound_y = ConstraintList()
for n in model.Locations:
    for m in model.Customers:
        model.bound_y.add( model.x[n,m] <= model.y[n] )
```

In Class Exercise: Concrete Knapsack

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

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

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

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

Syntax reminders:

```
ConcreteModel()
Var( [index, …], [within=domain], [bounds=(lower,upper)] )
Constraint( [index, …], [expr=expression|rule=function] )
ConstraintList();    c.add( expression )
Objective( sense={maximize|minimize},
           expr=expression|rule=function )
```

Concrete Knapsack: *Solution*

```
from coopr.pyomo import *

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

model = ConcreteModel()
model.ITEMS = Set( initialize=v.keys() )
model.x = Var( model.ITEMS, within=Binary )

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

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

5: Abstract Modeling

Importing Data: *Parameters*

- Scalar numeric values

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

Provide an (initial) value of 42 for the parameter

- Indexed numeric values

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

"`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

Data Sources

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

      e.g., import ABCD.xls range=ABCD : Z=[A, B, C] Y=D ;

    - Databases

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

Abstract p-Median (1)

```
from coopr.pyomo import *
import random

random.seed(1000)

model = AbstractModel()

model.N = Param(within=PositiveIntegers)
model.Locations = RangeSet(1,model.N)
model.P = Param(within=RangeSet(1,model.N))
model.M = Param(within=PositiveIntegers)
model.Customers = RangeSet(1,model.M)

model.d = Param(model.Locations, model.Customers, rule=lambda
n, m, model : random.uniform(1.0,2.0), within=Reals)
```

Abstract p-Median (2)

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

def 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=rule)

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

def rule(model, n,m):
    return (None, model.x[n,m] - model.y[n], 0.0)
model.bound_y = Constraint(model.Locations, model.Customers, rule=rule)

def rule(model):
    return (sum( (model.y[n] for n in model.Locations) ) - model.P, 0.0)
model.num_facilities = Constraint(rule=rule)
```

Abstract p-Median (3)

```
param N := 10;
param M := 6;
param P := 3;
```

In Class Exercise: Abstract Knapsack

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

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

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

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

Syntax reminders:

```
AbstractModel()
Var( [index, …], [within=domain], [bounds=(lower,upper)] )
Constraint( [index, …], [expr=expression|rule=function] )
ConstraintList();    c.add( expression )
Objective( sense={maximize|minimize},
           expr=expression|rule=function )
```

Abstract Knapsack: *Solution*

```
from coopr.pyomo 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;
```

6: Pyomo and Python Efficiency

- Being embedded in a high-level (and interpreted) programming language can present challenges
  - Inability to constrain syntax => users have many guns
- Some of the blame can be placed on Python
  - But a lot can be blamed on Pyomo

What are *reasonable* performance expectations?

- Python is a *byte-compiled scripting language*
  - and Pyomo is pure Python
  - …so expectations were not high
  - *…and raw speed has never been a goal!*

- Early experiences bore this out… in November, 2010:
  - *p*-median facility location
    - AMPL model construction time: ~4 seconds
    - Pyomo model construction time: >2000 seconds
  - Logistics disruption modeling
    - GAMS solution time: ~20 seconds
    - Pyomo solution time: >200 seconds

- ...but the gap is closing… in Coopr 3.2:
  - *p*-median facility location: ~36 seconds
  - Logistics disruption modeling: ~25 seconds

Managing performance (how not to shoot yourself)

- Expression generation; consider:

$$\min \sum_{n} \sum_{m} d_{n,m} x_{n.m}, \quad n \in Locations, m \in Customers$$

Managing performance (how not to shoot yourself)

- Expression generation; consider:

$$\min \sum_n \sum_m d_{n,m} x_{n.m}, \quad n \in Locations, m \in Customers$$

```
def rule1(model):
    ans = 0
    for n in model.Locations:
        for m in model.Customers:
            ans = ans + model.d[n,m]*model.x[n,m]
    return ans
```

```
model.obj = Objective( rule=_ruleN )
```

Managing performance (how not to shoot yourself)

- Expression generation; consider:

$$\min \sum_n \sum_m d_{n,m} x_{n.m}, \quad n \in Locations, m \in Customers$$

```
def rule2(model):
    ans = 0
    for n in model.Locations:
        for m in model.Customers:
            ans += model.d[n,m]*model.x[n,m]
    return ans
```

```
model.obj = Objective( rule=_ruleN )
```

Managing performance (how not to shoot yourself)

- Expression generation; consider:

$$\min \sum_n \sum_m d_{n,m} x_{n.m}, \quad n \in Locations, m \in Customers$$

```
def rule3(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=_ruleN )
```

Managing performance (how not to shoot yourself)

- Expression generation; consider:

$$\min \sum_n \sum_m d_{n,m} x_{n.m}, \quad n \in Locations, m \in Customers$$

```
def rule4(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=_ruleN )
```

Managing performance (how not to shoot yourself)

- Expression generation; consider:

$$\min \sum_n \sum_m d_{n,m} x_{n.m}, \quad n \in Locations, m \in Customers$$

```
def rule1(model):
    ans = 0
    for n in model.Locations:
        for m in model.Customers:
            ans = ans + model.d[n,m]*model.x[n,m]
    return ans

def rule2(model):
    ans = 0
    for n in model.Locations:
        for m in model.Customers:
            ans += model.d[n,m]*model.x[n,m]
    return ans

def rule3(model):
    return sum( [ model.d[n,m]*model.x[n,m]
        for n in model.Locations for m in model.Customers ] )

def rule4(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=_ruleN )
```

[n = m = 1..640]

rule1:	>>10000 sec
rule2:	9.0 sec
rule3:	14.6 sec
rule4:	8.9 sec

Managing performance (how not to shoot yourself)

- Sparse data; consider:

$$\sum_{m \in M} a_{n,m} x_m \leq b_n \quad \forall\, n \in N$$

```
model.a = Param( model.N, model.M, default=0 )

def rule1(model,n):
     return sum( model.a[n,m] * model.x[m] for m in model.M) <= model.b[n] )
```

For n = 1..10, m = 1..1e5, 4% nonzero,

25.5 seconds to generate the constraint
1e5 terms in the constraint (dense!!)

```
model.C = Constraint( model.N, rule=_ruleN )
```

Managing performance (how not to shoot yourself)

- Sparse data; consider:

$$\sum_{m \in M} a_{n,m} x_m \leq b_n \quad \forall\, n \in N$$

```
model.a = Param( model.N, model.M, default=0 )
```

```
def rule2(model,n):
    return sum( model.a[n,m] * model.x[m] for m in model.M) <= model.b[n]
                if model.a[n,m] != 0 )
```

For n = 1..10, m = 1..1e5, 4% nonzero,

5 seconds *slower*, and still dense!

```
model.C = Constraint( model.N, rule=_ruleN )
```

Managing performance (how not to shoot yourself)

- Sparse data; consider:

$$\sum_{m \in M} a_{n,m} x_m \leq b_n \quad \forall\, n \in N$$

```
model.a = Param( model.N, model.M, default=0 )
```

```
def rule3(model,n):
    return sum( model.a[n,m] * model.x[m] for m in model.M) <= model.b[n]
                if value(model.a[n,m]) != 0 )

def rule4(model,n):
    return sum( model.a[n,m] * model.x[m] for m in model.M) <= model.b[n]
                if model.a[n,m].value != 0 )

model.C = Constraint( model.N, rule=_ruleN )
```

Managing performance (how not to shoot yourself)

- Sparse data; consider:

$$\sum_{m \in M} a_{n,m} x_m \le b_n \quad \forall\, n \in N$$

[n = 1..10, m = 1..1e5, 4% fill]

rule1: 25.5 sec
rule2: 30.5 sec
rule3: 7.6 sec
rule4: 5.7 sec

```
model.a = Param( model.N, model.M, default=0 )

def rule1(model,n):
    return sum( model.a[n,m] * model.x[m] for m in mo

def rule2(model,n):
    return sum( model.a[n,m] * model.x[m] for m in model.M) <= model.b[n]
                if model.a[n,m] != 0 )

def rule3(model,n):
    return sum( model.a[n,m] * model.x[m] for m in model.M) <= model.b[n]
                if value(model.a[n,m]) != 0 )

def rule4(model,n):
    return sum( model.a[n,m] * model.x[m] for m in model.M) <= model.b[n]
                if model.a[n,m].value != 0 )

model.C = Constraint( model.N, rule=_ruleN )
```

Managing performance (how not to shoot yourself)

- Sparse constraints; consider:

$$storage_{t,p,d} = storage_{t-1,p,d} + production_{t,p,d}$$

$$\forall\, t \in [dStart_d, dEnd_d],\ p \in Products,\ d \in Disruptions$$

Managing performance (how not to shoot yourself)

- Sparse data; consider:

$$\sum_{m \in M} a_{n,m} x_m \leq b_n \quad \forall\, n \in N$$

Managing performance (how not to shoot yourself)

- Sparse constraints; consider:

$$storage_{t,p,d} = storage_{t-1,p,d} + production_{t,p,d}$$

$$\forall\, t \in [dStart_d, dEnd_d],\ p \in Products,\ d \in Disruptions$$

```
def rule1(model,t,d,p):
    if t < model.dStart[d] or t > model.dEnd[d]:
        return Constraint.Skip
    return model.storage[t,p,d] == model.storage[t-1,p,d] + model.production[t,p,d]
model.C1 = Constraint( model.TIME, model.DISRUPTIONS, model.PRODUCTS, rule=rule1 )
```

Managing performance (how not to shoot yourself)

- Sparse constraints; consider:

$$storage_{t,p,d} = storage_{t-1,p,d} + production_{t,p,d}$$

$$\forall\, t \in [dStart_d, dEnd_d],\ p \in Products,\ d \in Disruptions$$

```
def rule2(model,t,d,p):
    return model.storage[t,p,d] == model.storage[t-1,p,d] + model.production[t,p,d]

def _filter2(model,t,d,p):
    return t >= model.dStart[d] and t <= model.dEnd[d]
model.ACTIVE_DISRUPTIONS = Set( model.TIME * model.DISRUPTIONS * model.PRODUCTS,
                                filter=_filter2 )
model.C2 = Constraint( model.ACTIVE_DISRUPTIONS, rule=rule2 )
```

Managing performance (how not to shoot yourself)

- Sparse constraints; consider:

$$storage_{t,p,d} = storage_{t-1,p,d} + production_{t,p,d}$$

$$\forall\, t \in [dStart_d, dEnd_d],\ p \in Products,\ d \in Disruptions$$

```
def rule2(model,t,d,p):
    return model.storage[t,p,d] == model.storage[t-1,p,d] + model.production[t,p,d]
```

```
def _filter3(model,t,d):
    return t >= model.dStart[d] and t <= model.dEnd[d]
model.ACTIVE_DISRUPTIONS = Set( model.TIME * model.DISRUPTIONS, filter=_filter3 )
model.C3 = Constraint( model.ACTIVE_DISRUPTIONS, model.PRODUCTS, rule=rule2 )
```

Managing performance (how not to shoot yourself)

- Sparse constraints; consider:

$$storage_{t,p,d} = storage_{t-1,p,d} + production_{t,p}$$

$$\forall\, t \in [dStart_d, dEnd_d],\ p \in Products$$

[t = d = 1..250,
p =1..10,
t*d = 2% fill]

C1: 19.8 sec
C2: 25.5 sec
C3: 3.2 sec

```
def rule1(model,t,d,p):
    if t < model.dStart[d] or t > model.dEnd[d]:
        return Constraint.Skip
    return model.storage[t,p,d] == model.storage[t-1,p
model.C1 = Constraint( model.TIME, model.DISRUPTIONS, m

def rule2(model,t,d,p):
    return model.storage[t,p,d] == model.storage[t-1,p,d] + model.production[t,p,d]

def _filter2(model,t,d,p):
    return t >= model.dStart[d] and t <= model.dEnd[d]
model.ACTIVE_DISRUPTIONS = Set( model.TIME * model.DISRUPTIONS * model.PRODUCTS,
                                filter=_filter2 )
model.C2 = Constraint( model.ACTIVE_DISRUPTIONS, rule=rule2 )

def _filter3(model,t,d):
    return t >= model.dStart[d] and t <= model.dEnd[d]
model.ACTIVE_DISRUPTIONS = Set( model.TIME * model.DISRUPTIONS, filter=_filter3 )
model.C3 = Constraint( model.ACTIVE_DISRUPTIONS, model.PRODUCTS, rule=rule2 )
```

The Performance "Elephant": Memory

- Known issue …
  - Python uses a fairly heavy-weight object model
  - "Significant" recent improvements in low-level core components
    - >50% over a year ago
  - But…
    - 640 x 640 *p*-median problem still consumes ~1.5 GB.

- Focus of current efforts, with several more enhancements on the horizon.

7. PySP: Stochastic Programming in Python

- Constructing the deterministic scenario model
- Specifying the scenario tree
- Specifying scenario instance data
- Creating and solving the extensive form

Constructing and Solving Block-Diagonal Models

- PySP: Stochastic Programming in Python
  - Begin with a deterministic (Pyomo) system model
  - Annotate model with data that defines the scenario(*) tree
  - Leverage automatic transformation, model decomposition

- Automated solution strategies
  - Solve the Extensive Form
    - Replicate deterministic model
    - Form nonanticipitivity constraints
    - Send to solver
  - Decompose (Progressive Hedging)
    - Replicate deterministic model
    - Duplicate complicating variables
    - Solve scenarios independently
    - Blend complicating variables
    - Weight scenarios to encourage convergence
    - Iterate

PySP: Formulating the System Model (1)

```
from coopr.pyomo import *
model = AbstractModel()

# Parameters
model.CROPS = Set()
model.TOTAL_ACREAGE = Param( within=PositiveReals )
model.PriceQuota = Param( model.CROPS, within=PositiveReals )
model.SubQuotaSellingPrice = Param( model.CROPS, within=PositiveReals )
model.SuperQuotaSellingPrice = Param( model.CROPS )
model.CattleFeedRequirement = Param( model.CROPS, within=NonNegativeReals )
model.PurchasePrice = Param( model.CROPS, within=PositiveReals )
model.PlantingCostPerAcre = Param( model.CROPS, within=PositiveReals )
model.Yield = Param( model.CROPS, within=NonNegativeReals )

# Variables
model.DevotedAcreage = Var( model.CROPS, bounds=(0.0, model.TOTAL_ACREAGE) )
model.QuantitySubQuotaSold = Var( model.CROPS, bounds=(0.0, None) )
model.QuantitySuperQuotaSold = Var( model.CROPS, bounds=(0.0, None) )
model.QuantityPurchased = Var( model.CROPS, bounds=(0.0, None) )
model.FirstStageCost = Var()
model.SecondStageCost = Var()
```


Birge and Louveaux's (1997) Farmer Example

PySP: Formulating the System Model (2)

```
# Constraints
def ConstrainTotalAcreage_rule(model):
    return summation(model.DevotedAcreage) <= model.TOTAL_ACREAGE

model.ConstrainTotalAcreage = Constraint()

def EnforceCattleFeedRequirement_rule(model, i):
    return model.CattleFeedRequirement[i] <= ( model.Yield[i] \
        * model.DevotedAcreage[i] ) + model.QuantityPurchased[i] \
        - model.QuantitySubQuotaSold[i] - model.QuantitySuperQuotaSold[i]

model.EnforceCattleFeedRequirement = Constraint( model.CROPS )

def LimitAmountSold_rule(model, i):
    return model.QuantitySubQuotaSold[i] + model.QuantitySuperQuotaSold[i] \
        - (model.Yield[i] * model.DevotedAcreage[i]) <= 0.0

model.LimitAmountSold = Constraint( model.CROPS )

def EnforceQuotas_rule(model, i):
    return(0.0, model.QuantitySubQuotaSold[i], model.PriceQuota[i])

model.EnforceQuotas = Constraint( model.CROPS )
```


Birge and Louveaux's (1997) Farmer Example

PySP: Formulating the System Model (3)

```
# Stage-specific cost computations
def ComputeFirstStageCost_rule(model):
    return 0.0 == model.FirstStageCost \
        - summation( model.PlantingCostPerAcre, model.DevotedAcreage )

model.ComputeFirstStageCost = Constraint()

def ComputeSecondStageCost_rule(model):
    expr = summation( model.PurchasePrice, model.QuantityPurchased )
    expr -= summation( model.SubQuotaSellingPrice, model.QuantitySubQuotaSold )
    expr -= summation( model.SuperQuotaSellingPrice, model.QuantitySuperQuotaSold )
    return(model.SecondStageCost - expr) == 0.0

model.ComputeSecondStageCost = Constraint()

# Objective
def Total_Cost_Objective_rule(model):
    return model.FirstStageCost + model.SecondStageCost

model.Total_Cost_Objective = Objective( sense=minimize )
```


Birge and Louveaux's (1997) Farmer Example

PySP: Specifying the Scenario Tree

```
set Stages := FirstStage SecondStage ;

set Nodes := RootNode
             BelowAverageNode
             AverageNode
             AboveAverageNode ;

param NodeStage := RootNode         FirstStage
                   BelowAverageNode SecondStage
                   AverageNode      SecondStage
                   AboveAverageNode SecondStage ;

set Children[RootNode] := BelowAverageNode
                          AverageNode
                          AboveAverageNode ;

param ConditionalProbability := RootNode          1.0
                                BelowAverageNode  0.33333333
                                AverageNode       0.33333334
                                AboveAverageNode  0.33333333 ;

set Scenarios := BelowAverageScenario
                 AverageScenario
                 AboveAverageScenario ;

param ScenarioLeafNode :=
                    BelowAverageScenario BelowAverageNode
                    AverageScenario      AverageNode
                    AboveAverageScenario AboveAverageNode ;

set StageVariables[FirstStage] :=  DevotedAcreage[*] ;

set StageVariables[SecondStage] := QuantitySubQuotaSold[*]
                                   QuantitySuperQuotaSold[*]
                                   QuantityPurchased[*] ;

param StageCostVariable := FirstStage  FirstStageCost
                           SecondStage SecondStageCost ;
```

PySP: Specifying the Scenario Instance Data

- Two methods are available to specify scenario data
  - Scenario-based
  - Node-based

- In the scenario-based approach, a single and complete .dat file is specified for each individual scenario
  - Redundant, but straightforward if computer-generated

- In the node-based approach, a single .dat file is specified for each node in the scenario tree
  - Maximally compact, but requires some book-keeping

Writing and Solving the Extensive Form (1)

- Now that you have a stochastic programming model in PySP…
- Step #1: Write the extensive form and hope that your favorite solver can actually solve it
  - Fantastic if it works
  - But often it doesn't
- In PySP, the ***runef*** script is provided to both write and solve the extensive form of a stochastic programming model
- The basic command-line:

```
runef --model-directory=models \
      --instance-directory=scenariodata \
      --solve
```

Writing and Solving the Extensive Form (2)

- After solution, you get:

```
Tree Nodes:

        Name=AboveAverageNode
        Stage=SecondStage
        Parent=RootNode
        Variables:
                QuantitySubQuotaSold[CORN]=48.0
                QuantitySubQuotaSold[SUGAR_BEETS]=6000.0
                QuantitySubQuotaSold[WHEAT]=310.0

        Name=AverageNode
        Stage=SecondStage
        Parent=RootNode
        Variables:
                QuantitySubQuotaSold[SUGAR_BEETS]=5000.0
                QuantitySubQuotaSold[WHEAT]=225.0

        Name=BelowAverageNode
        Stage=SecondStage
        Parent=RootNode
        Variables:
                QuantitySubQuotaSold[SUGAR_BEETS]=4000.0
                QuantitySubQuotaSold[WHEAT]=140.0
                QuantityPurchased[CORN]=48.0

        Name=RootNode
        Stage=FirstStage
        Parent=None
        Variables:
                DevotedAcreage[CORN]=80.0
                DevotedAcreage[SUGAR_BEETS]=250.0
                DevotedAcreage[WHEAT]=170.0
```

In-Class Exercise: Stochastic Knapsack

- Extend the (abstract) deterministic knapsack models to create and solve a stochastic knapsack problem

- Scenarios
  - Random values/profits – fixed weights!
  - 3 scenarios is sufficient for the exercise

- Scenario Tree

- Solve with **runef** script
  - `runef -m your-dir -i your-dir --solve –solver=glpk`

8. Advanced Topics

- Lots of things that we would like to talk about
  - But we assume we're already out of time

- Examples of advanced topics
  - Generalized Disjunctive Programming
  - Blocks and connectors
  - External data sources
  - Parallelization with Pyro
  - Non-linear programming and the AMPL solver library interface
  - Scripting for complex workflows
  - …

Thanks! Please Contribute!

- Now that you actually know something about Pyomo
- Please go off and do great things
- We welcome contributions
  - Bug reports
  - Extensions
  - Links to your projects that use Coopr/Pyomo
  - Source code
  - Documentation
  - …