The COIN-OR Optimization Suite: Open Source Tools for Optimization
Part 3: Python Tools

Ted Ralphs

INFORMS Computing Society Biennial Meeting
Richmond, VA, 10 January 2015
1. Introduction to Python

2. Python Tools in COIN-OR
   - CyLP
   - yaposib
   - PuLP and Dippy
   - Pyomo
   - GiMPy
   - GrUMPy
   - CuPPy
Outline

1. Introduction to Python

2. Python Tools in COIN-OR
   - CyLP
   - yaposib
   - PuLP and Dippy
   - Pyomo
   - GiMPy
   - GrUIMPy
   - CuPPy
Why Python?

- **Pros**
  - As with many high-level languages, development in Python is quick and painless (relative to C++!).
  - Python is popular in many disciplines and there is a dizzying array of packages available.
  - Python’s syntax is very clean and naturally adaptable to expressing mathematical programming models.
  - Python has the primary data structures necessary to build and manipulate models built in.
  - There has been a strong movement toward the adoption of Python as the high-level language of choice for (discrete) optimizers.
  - Sage is quickly emerging as a very capable open-source alternative to Matlab.

- **Cons**
  - Python’s one major downside is that it can be very slow.
  - Solution is to use Python as a front-end to call lower-level tools.
Drinking the Python Kool-Aid

I learned it last night! Everything is so simple! Hello world is just print "Hello, world!"

I dunno... dynamic typing? Whitespace?
Come join us! Programming is fun again! It's a whole new world up here!
But how are you flying?

I just typed import antigravity
That's it?
...I also sampled everything in the medicine cabinet for comparison.
But I think this is the Python.
Introduction to Python

Adapted from a Tutorial by Guido van Rossum
Director of PythonLabs at Zope Corporation

Presented at
LinuxWorld - New York City - January 2002
Why Python?

- Interpreted language
- Intuitive syntax
- Dynamic typing
- **Loads** of built-in libraries and available extensions
- Shallow learning curve
- Easy to call C/C++ for efficiency
- Object-oriented
- Simple, but extremely powerful
Tutorial Outline

- 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:
  ```python
  >>> print "Hello, world"
  Hello, world
  >>> x = 12**2
  >>> x/2
  72
  >>> # this is a comment
  ```
To write a program, put commands in a file

#hello.py
print "Hello, world"
x = 12**2
x/2
print x

Execute on the command line

~> python hello.py
Hello, world
72
Variables

- No need to declare
- Need to assign (initialize)
  - use of uninitialized variable raises exception
- Not typed
  ```python
  if friendly: greeting = "hello world"
  else: greeting = 12**2
  print greeting
  ```
- *Everything* is an "object":
  - Even functions, classes, modules
Control Structures

if condition:
    statements
[elif condition:
    statements] ...
else:
    statements

while condition:
    statements

for var in sequence:
    statements

break
continue
Grouping Indentation

In Python:

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

In C:

```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), 0<x<=5
- C-style shifting & masking
  - 1<<16, x&0xff, x|1, ~x, x^y
- Integer division truncates :-(
  - 1/2 -> 0 # 1./2. -> 0.5, float(1)/2 -> 0.5
  - Will be fixed in the future
- 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"  1 # comparison
- "e" in "hello"  1 # search
- "escapes: \n etc, \033 etc, \if etc"
- 'single quotes'  """"triple quotes""""  r"raw strings"
Lists

- Flexible arrays, not Lisp-like 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"]`
More List Operations

```python
>>> 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]
5.5
>>> a.reverse()  # [4,3,2,1,0]
>>> a.sort()  # [0,1,2,3,4]
```
Dictionaries

- Hash tables, "associative arrays"
  - d = {
    "duck": "eend",
    "water": "water"
  }

- Lookup:
  - d["duck"] -> "eend"
  - d["back"] # raises KeyError exception

- Delete, insert, overwrite:
  - del d["water"] # {'duck': 'eend', 'back': 'rug'}
  - d["back"] = "rug" # {'duck': 'eend', 'back': 'rug'}
  - d["duck"] = "duik" # {'duck': 'duik', 'back': 'rug'}
More Dictionary Ops

- Keys, values, items:
  - d.keys() -> ["duck", "back"]
  - d.values() -> ["duik", "rug"]
  - d.items() -> [("duck","duik"), ("back","rug")]

- Presence check:
  - d.has_key("duck") -> 1; d.has_key("spam") -> 0

- Values of any type; keys almost any
  - {"name":"Guido", "age":43, ("hello","world"):1, 42:"yes", "flag": ["red","white","blue"]}
Keys must be immutable:
- numbers, strings, tuples of immutables
  - these cannot be changed after creation
- reason is hashing (fast lookup technique)
- not lists or other dictionaries
  - these types of objects can be changed "in place"
- no restrictions on values

Keys will be listed in arbitrary order
- again, because of hashing
Tuples

- key = (lastname, firstname)
- point = x, y, z  # parentheses optional
- x, y, z = point  # unpack
- lastname = key[0]
- singleton = (1,)  # trailing comma!!!
- empty = ()  # parentheses!
- tuples vs. lists; tuples immutable
Reference Semantics

- Assignment manipulates references
  - \( x = y \) **does not make a copy** of \( y \)
  - \( x = y \) makes \( x \) **reference** the object \( y \) references

- Very useful; but beware!

- Example:
  ```
  >>> a = [1, 2, 3]
  >>> b = a
  >>> a.append(4)
  >>> print b
  [1, 2, 3, 4]
  ```
Changing a Shared List

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

\[ b = a \]

\[ a.append(4) \]
Changing an Integer

\[ a = 1 \]

\[ b = a \]

\[ a = a + 1 \]

- New int object created by add operator \((1+1)\)
- Old reference deleted by assignment \((a=\ldots)\)
Functions, Procedures

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

    return  # from procedure
    return expression  # from function
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
Classes

class name:
   "documentation"
   statements
-or-
class name(base1, base2, ...):
   ...
Most, statements are method definitions:
   def name(self, arg1, arg2, ...):
       ...
   ...
May also be class variable assignments
Example Class

class Stack:
    "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 happens if it’s empty?
        del self.items[-1]
        return x
    def empty(self):
        return len(self.items) == 0  # Boolean result
Using Classes

- To create an instance, simply call the class object:
  ```python
  x = Stack()  # no 'new' operator!
  ```

- To use methods of the instance, call using dot notation:
  ```python
  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:
  ```python
  x.items  # -> [1]
  ```
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
Getting Python

- There are many different flavors of Python, all of which support the same basic API, but have different backends and performance.
- The “original flavor” is CPython, but there is also Jython, Iron Python, Pyjs, PyPy, RubyPython, and others.
- If you are going to use a package with a C extensions, you probably need to get CPython.
- For numerical computational, some additional packages are almost certainly required, NumPy and SciPy being the most obvious.
  - On Linux, Python and the most important packages will be pre-installed, with additional ones installed easily via a package manager.
  - On OS X, Python comes pre-installed, but it is easier to install Python and any additional packages via Homebrew.
  - On Windows, it’s easiest to install a distribution that includes the scientific software, such as PythonXY or Portable Python.
- Another option is to use Sage, a Matlab-like collection of Python packages (including COIN).
In Class Exercise: Install Python!
Getting an IDE

- An additional requirement for doing development is an IDE.
- My personal choice is Eclipse with the PyDev plug-in.
- This has the advantage of being portable and cross-platform, as well as supporting most major languages.
It is possible to implement extensions to the basic language in C/C++.
Calls into these extensions libraries are then executed efficiently as native C/C++ code.
Although it is possible in theory to provide binary packages for these extensions, this is a headache on OS X and Linux.
It is likely you will have to build your own versions, but this is relatively easy.
On Windows, building extensions is harder, but working binaries are usually easier to obtain.
Basic Build Steps

- First, build and install the relevant project using the autotools.
  - You can avoid some potential complications by configuring with
    `--enable-static --disable-shared`.
  - Otherwise, you need to set either `LD_LIBRARY_PATH` (Linux) or `DYLD_LIBRARY_PATH` (OS X) to point to `{$prefix}/lib`.

- Next, set some environment variables.
  - For `yaposib`, you need to have `pkg-config` installed and set
    `PKG_CONFIG_PATH={$prefix}/lib/pkgconfig`.
  - For `CyLP` and `DipPy`, you need to set `COIN_INSTALL_DIR={$prefix}`.

- Finally, just execute `python setup.py install`. 
Outline

1. Introduction to Python

2. Python Tools in COIN-OR
   - CyLP
   - yaposib
   - PuLP and Dippy
   - Pyomo
   - GiMPy
   - GrUMPy
   - CuPPy
1 Introduction to Python

2 Python Tools in COIN-OR
   - CyLP
     - yaposib
     - PuLP and Dippy
   - Pyomo
   - GiMPy
   - GrUOMPy
   - CuPPy
CyLP: Low-level Modeling and API for Cbc/Clp/Cgl

- CyLP provides a low-level modeling language for accessing details of the algorithms and low-level parts of the API.
- The included modeling language is “close to the metal”, works directly with numerical data with access to low-level data structures.

- Clp
  - Pivot-level control of algorithm in Clp.
  - Access to fine-grained results of solve.

- Cbc
  - Python classes for customization

- Cgl
  - Python class for building cut generators wrapped around Cgl.

**Developers**: Mehdi Towhidi and Dominique Orban
lp = CyClpSimplex()
x = lp.addVariable('x', numVars)
lp += x_u >= x >= 0

lp += A * x <= b if cons_sense == '<=' else A * x >= b

lp.objective = -c * x if obj_sense == 'Max' else c * x
lp.primal(startFinishOptions = 1)
numCons = len(b)
print 'Current solution is', lp.primalVariableSolution['x']
print 'Current tableaux is', lp.tableaux
for row in range(lp.nConstraints):
    print 'Variables basic in row', row, 'is', lp.basicVariables[row],
    print 'and has value' lp.rhs[row]
Introduction to Python

Python Tools in COIN-OR

- CyLP
- yaposib
- PuLP and Dippy
- Pyomo
- GiMPy
- GrUUMPy
- CuPPy
yaposib: Python Bindings for OSI

Provides Python bindings to any solver with an OSI interface

```python
solver = yaposib.available_solvers()[0]

for filename in sys.argv[1:]:
    problem = yaposib.Problem(solver)
    print("Will now solve %s" % filename)
    err = problem.readMps(filename)
    if not err:
        problem.solve()
        if problem.status == 'optimal':
            print("Optimal value: %f" % problem.obj.value)
            for var in problem.cols:
                print("\t%s = %f" % (var.name, var.solution))
        else:
            print("No optimal solution could be found.")
```

T.K. Ralphs (Lehigh University)

COIN-OR

January 10, 2015
Introduction to Python

Python Tools in COIN-OR

- CyLP
- yaposib
- PuLP and Dippy
- Pyomo
- GiMPy
- GrUmpy
- CuPPy
PuLP is a modeling language in COIN-OR that provides data types for Python that support algebraic modeling.

PuLP only supports development of linear models.

Main classes

- LpProblem
- LpVariable

Variables can be declared individually or as “dictionaries” (variables indexed on another set).

We do not need an explicit notion of a parameter or set here because Python provides data structures we can use.

In PuLP, models are technically “concrete,” since the model is always created with knowledge of the data.

However, it is still possible to maintain a separation between model and data.
from products import REQUIREMENT, PRODUCTS
from facilities import FIXED_CHARGE, LOCATIONS, CAPACITY

prob = LpProblem("Facility_Location")

ASSIGNMENTS = [(i, j) for i in LOCATIONS for j in PRODUCTS]
assign_vars = LpVariable.dicts("x", ASSIGNMENTS, 0, 1, LpBinary)
use_vars = LpVariable.dicts("y", LOCATIONS, 0, 1, LpBinary)

prob += lpSum(use_vars[i] * FIXED_COST[i] for i in LOCATIONS)

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

for i in LOCATIONS:
    prob += lpSum(assign_vars[(i, j)] * REQUIREMENT[j]
               for j in PRODUCTS) <= CAPACITY * use_vars[i]

prob.solve()

for i in LOCATIONS:
    if use_vars[i].varValue > 0:
        print "Location ", i, " is assigned: ",
        print [j for j in PRODUCTS if assign_vars[(i, j)].varValue > 0]
DIP Framework

DIP is a software framework and stand-alone solver for implementation and use of a variety of decomposition-based algorithms.

- Decomposition-based algorithms have traditionally been extremely difficult to implement and compare.
- DIP abstracts the common, generic elements of these methods.
  - Key: API is in terms of the compact formulation.
  - The framework takes care of reformulation and implementation.
  - DIP is now a fully generic decomposition-based parallel MILP solver.

Methods

- Column generation (Dantzig-Wolfe)
- Cutting plane method
- Lagrangian relaxation (not complete)
- Hybrid methods
from products import REQUIREMENT, PRODUCTS
from facilities import FIXED_CHARGE, LOCATIONS, CAPACITY

prob = dippy.DipProblem("Facility_Location")

ASSIGNMENTS = [(i, j) for i in LOCATIONS for j in PRODUCTS]
assign_vars = LpVariable.dicts("x", ASSIGNMENTS, 0, 1, LpBinary)
use_vars = LpVariable.dicts("y", LOCATIONS, 0, 1, LpBinary)

prob += lpSum(use_vars[i] * FIXED_COST[i] for i in LOCATIONS)

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

for i in LOCATIONS:
    prob.relaxation[i] += lpSum(assign_vars[(i, j)] * REQUIREMENT[j]
    for j in PRODUCTS) <= CAPACITY * use_vars[i]

dippy.Solve(prob, doPriceCut:1)

for i in LOCATIONS:
    if use_vars[i].varValue > 0:
        print "Location ", i, " is assigned: ",
        print [j for j in PRODUCTS if assign_vars[(i, j)].varValue > 0]
In Class Exercise: Install DipPy!
1 Introduction to Python

2 Python Tools in COIN-OR
   - CyLP
   - yaposib
   - PuLP and Dippy
   - Pyomo
   - GiMPy
   - GrUMPy
   - CuPPy
Pyomo

- An algebraic modeling language in Python similar to PuLP.
- Can import data from many sources, including AMPL data files.
- More powerful, includes support for nonlinear modeling.
- Allows development of both concrete models (like PuLP) and abstract models (like AMPL).
- Also include PySP for stochastic Programming.

Primary classes
- `ConcreteModel`, `AbstractModel`
- `Set`, `Parameter`
- `Var`, `Constraint`

**Developers**: Bill Hart, John Siirola, Jean-Paul Watson, David Woodruff, and others...
model = ConcreteModel()

Bonds, Features, BondData, Liabilities = read_data('ded.dat')

Periods = range(len(Liabilities))

model.buy = Var(Bonds, within=NonNegativeReals)
model.cash = Var(Periods, within=NonNegativeReals)
model.obj = Objective(expr=model.cash[0] +
    sum(BondData[b, 'Price'] * model.buy[b] for b in Bonds),
    sense=minimize)

def cash_balance_rule(model, t):
    return (model.cash[t-1] - model.cash[t] +
        sum(BondData[b, 'Coupon'] * model.buy[b]
            for b in Bonds if BondData[b, 'Maturity'] >= t) +
        sum(BondData[b, 'Principal'] * model.buy[b]
            for b in Bonds if BondData[b, 'Maturity'] == t) == Liabilities[t])
model.cash_balance = Constraint(Periods[1:], rule=cash_balance_rule)
In Class Exercise: Install Pyomo!

```
pip install pyomo
```
Introduction to Python

Python Tools in COIN-OR
- CyLP
- yaposib
- PuLP and Dippy
- Pyomo
- GiMPy
- GrUOMPy
- CuPPy
GiMPy (with Aykut Bulut)

- A graph class for Python 2.*.
- Builds, displays, and saves graphs (many options)
- Focus is on visualization of well-known graph algorithms.
  - Priority in implementation is on clarity of the algorithms.
  - Efficiency is not the goal (though we try to be as efficient as we can).

easy_install install coinor.grumpy

g = Graph(display='xdot')
g.add_edge(0,1)
g.add_edge(1,2)
g.add_edge(3,4)
g.display()
g.search(0)
GIMPy Example
GiMPy: Graph Methods in Python

The following problem/algorithm pairs with similar visualization options exist.

- **Graph Search:**
  - BFS
  - DFS
  - Prim’s
  - Component Labeling,
  - Dijkstra’s
  - Topological Sort

- **Shortest path:** Dijkstra’s, Label Correcting
- **Maximum flow:** Augmenting Path, Preflow Push
- **Minimum spanning tree:** Prim’s Algorithm, Kruskal Algorithm
- **Minimum Cost Flow:** Network Simplex, Cycle Canceling
- **Data structures:** Union-Find (quick union, quick find), Binary Search Tree, Heap
Tree class derived from Graph class.

BinaryTree class derived from Tree class.

Has binary tree specific API and attributes.
Introduction to Python

Python Tools in COIN-OR
- CyLP
- yaposib
- PuLP and Dippy
- Pyomo
- GiMPy
- GrUMPy
- CuPPy
GrUMPy Overview

- Visualizations for solution methods for linear models.
  - Branch and bound
  - Cutting plane method
- **BBTree derived from GiMPy Tree.**
  - Reads branch-and-bound data either dynamically or statically.
  - Builds dynamic visualizations of solution process.
  - Includes a pure Python branch and bound implementation.
- **Polyhedron2D derived from pypolyhedron.**
  - Can construct 2D polyhedra defined by generators or inequalities.
  - Displays convex hull of integer points.
  - Can produce animations of the cutting plane method.
- GrUMPy is an expansion and continuation of the BAK project (Brady Hunsaker and Osman Ozaltin).

```
easy_install coinor.grumpy
```
GrUMPy: BBTree Branch and Bound Implementation

T = BBTree()
#T.set_layout('dot2tex')
#T.set_display_mode('file')
T.set_display_mode('xdot')
CONSTRAINTS, VARIABLES, OBJ, MAT, RHS = \
    T.GenerateRandomMIP(rand_seed = 19)
T.BranchAndBound(CONSTRAINTS, VARIABLES, OBJ, MAT, RHS, \
    branch_strategy = PSEUDOCOST_BRANCHING, \
    search_strategy = BEST_FIRST, \
    display_interval = 1)
GrUMPy: BBTree Branch and Bound Implementation
GrUMPy: Dynamic Branch and Bound Visualizations

- GrUMPy provides four visualizations of the branch and bound process.
- Can be used dynamically or statically with any instrumented solver.
  - BB tree
  - Histogram
  - Scatter plot
  - Incumbent path
Figure: BB tree generated by GrUMPy
Figure: BB histogram generated by GrUMPy
Figure: Scatter plot generated by GrUMPy
GrUMPy Incumbent Path

Figure: Incumbent path generated by GrUMPy
GrUMPy: Polyhedron2D

```python
f = Figure()
p = Polyhedron2D(A = [[4, 1], [1, 4], [1, -1], [-1, 0], [0, -1]],
    b = [28, 27, 1, 0, 0])
#p = Polyhedron2D(points = [[0, 0], [2, 2], [3.75, 2.75], [3, 1]])
f.add_polyhedron(p, color = 'blue', linestyle = 'solid', label = 'p',
    show_int_points = True)
f.set_xlim(p.plot_min[0], p.plot_max[0])
f.set_ylim(p.plot_min[1], p.plot_max[1])
pI = p.make_integer_hull()
f.add_polyhedron(pI, color = 'red', linestyle = 'dashed', label = 'pI')
f.add_point((5.666,5.333), 0.02, 'red')
f.add_text(5.7, 5.4, r'$(17/3, 16/3)$')
f.add_line([3, 2], 27, p.plot_max, p.plot_min,
    color = 'green', linestyle = 'dashed')
f.show()
```
Figure: Convex hull of $\mathcal{P}$
Introduction to Python

Python Tools in COIN-OR
- CyLP
- yaposib
- PuLP and Dippy
- Pyomo
- GiMPy
- GrUMPy
- CuPPy
CuPPy: Cutting Planes in Python

- Simple implementations and visualizations of cutting plane procedures.
- Uses CyLP to access the tableaux of the underlying Clp model.
- Currently has visualizations for GMI and split cuts.

```python
f0 = getFraction(sol[basicVarInd])
f = [getFraction(lp.tableau[row, i]) for i in range(lp.nVariables)]
pi = np.array([f[j]/f0 if f[j] <= f0
               else (1-f[j])/(1-f0) for j in range(lp.nVariables)])
pi_slacks = np.array([x/f0 if x > 0 else -x/(1-f0)
                      for x in lp.tableau[row, lp.nVariables:]])
pi -= pi_slacks * lp.coefMatrix
pi0 = (1 - np.dot(pi_slacks, lp.constraintsUpper) if sense == '<='
      else 1 + np.dot(pi_slacks, lp.constraintsUpper))
```
The GMI cut from the first row is

\[
\frac{1}{10}s_1 + \frac{8}{10}s_2 \geq 1, \quad (1)
\]

In terms of \(x_1\) and \(x_2\), we have

\[
12x_1 + 33x_2 \leq 234, \quad \text{(GMI-C1)}
\]
The GMI cut from the third row is

\[
\frac{4}{10} s_1 + \frac{2}{10} s_2 \geq 1,
\]  

(2)

In terms of \(x_1\) and \(x_2\), we have

\[
3x_1 + 2x_2 \leq 26,
\]  

(GMI-C3)
Figure: GMI Cut from row 2 as an intersection cut
End of Part 3!

Questions?