Tools for Modeling Optimization Problems
A Short Course

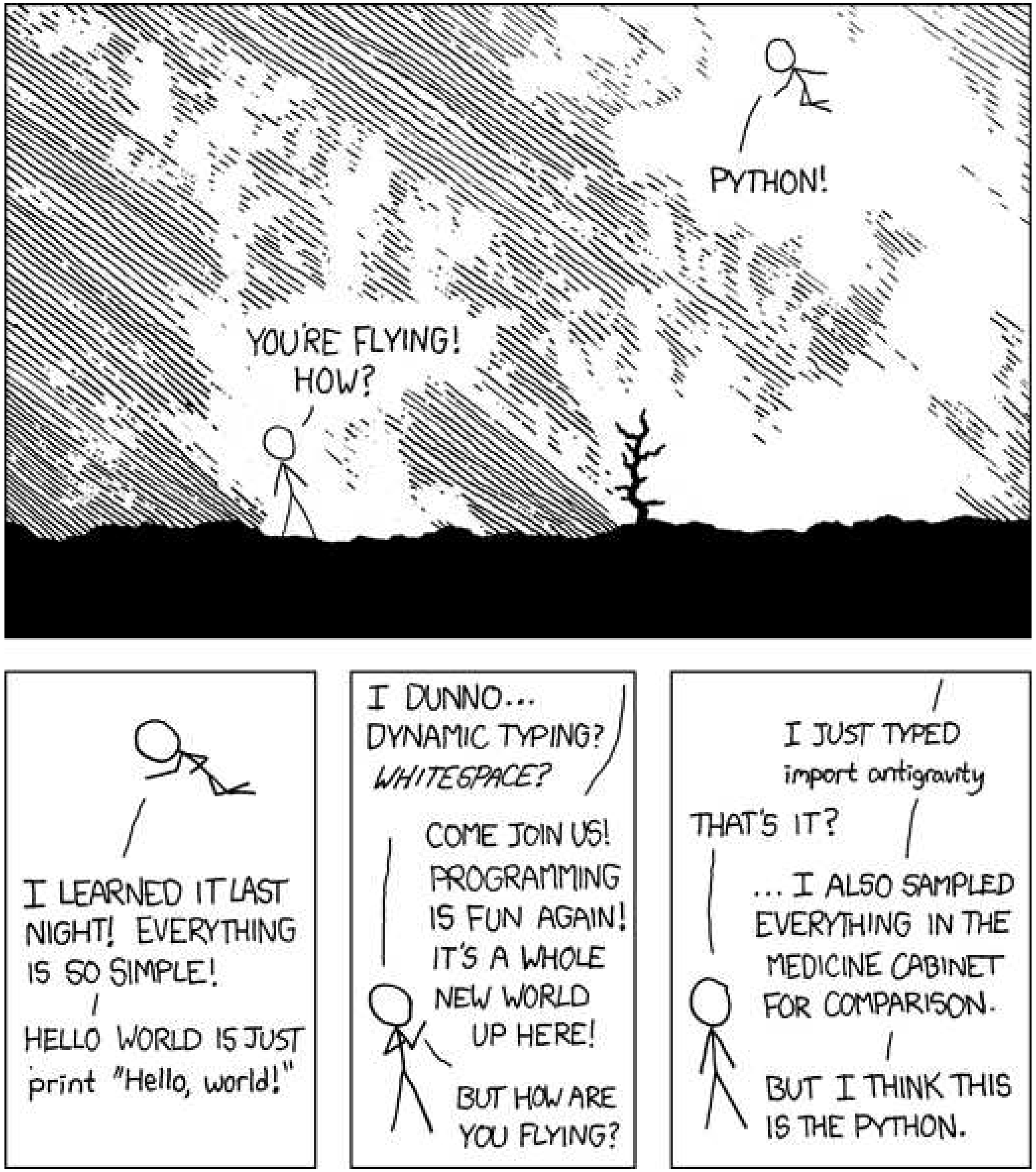
Modeling with Python

Dr. Ted Ralphs

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



Two-minute Python Primer

- Python is object-oriented with a light-weight class and inheritance mechanism.
- There is no explicit compilation; scripts are interpreted.
- Variables are dynamically typed with no declarations.
- Memory allocation and freeing all done automatically.
- Indentation has a syntactic meaning!
- Code is usually easy to read “in English” (keywords like `is`, `not`, and `in`).
- Everything can be “printed.”
- Important programming constructs
  - Functions/Classes
  - Looping
  - Conditionals
  - Comprehensions

Two-minute Python Primer (cont'd)

- Built-in data structures:
  - Lists (dynamic arrays)
  - Tuples (static arrays)
  - Dictionaries (hash tables)
  - Sets
- Class mechanism:
  - Classes are collections of *data* and associated *methods*.
  - Members of a class are called *attributes*.
  - Attributes are accessed using "." syntax.

Introduction to PuLP

- 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.

Bond Portfolio Example: Simple PuLP Model
(bonds_simple-PuLP.py)

```
from pulp import LpProblem, LpVariable, lpSum, LpMaximize, value

prob = LpProblem("Dedication Model", LpMaximize)

X1 = LpVariable("X1", 0, None)
X2 = LpVariable("X2", 0, None)

prob += 4*X1 + 3*X2
prob += X1 + X2 <= 100
prob += 2*X1 + X2 <= 150
prob += 3*X1 + 4*X2 <= 360

prob.solve()

print 'Optimal total cost is: ', value(prob.objective)

print "X1 :", X1.varValue
print "X2 :", X2.varValue
```

Notes About the Model

- Like the simple AMPL model, we are not using indexing or any sort of abstraction here.
- The syntax is very similar to AMPL.
- To achieve separation of data and model, we use Python's `import` mechanism.

Bond Portfolio Example: Abstract PuLP Model (`bonds-PuLP.py`)

```
from pulp import LpProblem, LpVariable, lpSum, LpMaximize, value

from bonds import bonds, max_rating, max_maturity, max_cash

prob = LpProblem("Bond Selection Model", LpMaximize)

buy = LpVariable.dicts('bonds', bonds.keys(), 0, None)

prob += lpSum(bonds[b]['yield'] * buy[b] for b in bonds)

prob += lpSum(buy[b] for b in bonds) <= max_cash, "cash"

prob += (lpSum(bonds[b]['rating'] * buy[b] for b in bonds)
         <= max_cash*max_rating, "ratings")

prob += (lpSum(bonds[b]['maturity'] * buy[b] for b in bonds)
         <= max_cash*max_maturity, "maturities")
```

Notes About the Model

- We can use Python's native `import` mechanism to get the data.
- Note, however, that the data is read and stored *before* the model.
- This means that we don't need to declare sets and parameters.
- Carriage returns are syntactic (parentheses imply line continuation).
- Constraints
  - Naming of constraints is optional and only necessary for certain kinds of post-solution analysis.
  - Constraints are added to the model using a very intuitive syntax.
  - Objectives are nothing more than expressions that are to be optimized rather than explicitly constrained.
- Indexing
  - Indexing in Python is done using the native dictionary data structure.
  - Note the extensive use of comprehensions, which have a syntax very similar to quantifiers in a mathematical model.

Bond Portfolio Example: Solution in PuLP

```
prob.solve()

epsilon = .001

print 'Optimal purchases:'
for i in bonds:
    if buy[i].varValue > epsilon:
        print 'Bond', i, ":", buy[i].varValue
```

Notes About the Data Import (`bonds_data.py`)

- We are storing the data about the bonds in a "dictionary of dictionaries."
- With this data structure, we don't need to separately construct the list of bonds.
- We can access the list of bonds as `bonds.keys()`.
- Note, however, that we still end up hard-coding the list of features and we must repeat this list of features for every bond.
- We can avoid this using some advanced Python programming techniques, but SolverStudio makes this easy.

PuLP Model in SolverStudio

(`FinancialModels.xlsx:Bonds-PuLP`)

- We've explicitly allowed the option of optimizing over one of the features, while constraining the others.
- Later, we'll see how to create tradeoff curves showing the tradeoffs among the constraints imposed on various features.

Portfolio Dedication

Definition 1. Dedication *or* cash flow matching *refers to the funding of known future liabilities through the purchase of a portfolio of risk-free non-callable bonds.*

Notes:

- Dedication is used to eliminate interest rate risk.
- Dedicated portfolios do not have to be managed.
- The goal is to construct such portfolio at a minimal price from a set of available bonds.
- This is a multi-period model.

Example: Portfolio Dedication

- A pension fund faces liabilities totalling ℓ_j for years $j = 1, ..., T$.
- The fund wishes to dedicate these liabilities via a portfolio comprised of n different types of bonds.
- Bond type i costs c_i, matures in year m_i, and yields a yearly coupon payment of d_i up to maturity.
- The principal paid out at maturity for bond i is p_i.

LP Formulation for Portfolio Dedication

We assume that for each year j there is at least one type of bond i with maturity $m_i = j$, and there are none with $m_i > T$.

Let x_i be the number of bonds of type i purchased, and let z_j be the cash on hand at the beginning of year j for $j = 0, \ldots, T$. Then the dedication problem is the following LP,

$$\min_{(x,z)} z_0 + \sum_i c_i x_i$$

$$\text{s.t. } z_{j-1} - z_j + \sum_{\{i:m_i \geq j\}} d_i x_i + \sum_{\{i:m_i = j\}} p_i x_i = \ell_j, \quad (j = 1, \ldots, T-1)$$

$$z_T + \sum_{\{i:m_i = T\}} (p_i + d_i) x_i = \ell_T.$$

$$z_j \geq 0, j = 1, \ldots, T$$

$$x_i \geq 0, i = 1, \ldots, n$$

AMPL Model for Dedication (`dedication.mod`)

- In multi-period models, we have to somehow represent the set of periods.
- Such a set is different from a generic set because it involves *ranged data*.
- We must somehow do arithmetic with elements of this set in order to express the model.
- In AMPL, a ranged set can be constructed using the syntax `1..T`.
- Both endpoints are included in the range.
- Another important feature of the above model is the use of conditionals in the limits of the sum.
- Conditionals can be used to choose a subset of the items in a given set satisfying some condition.

PuLP Model for Dedication (`dedication-PuLP.py`)

- We are parsing the AMPL data file with a custom-written function `read_data` to obtain the data.
- The data is stored in a two-dimensional table (dictionary with tuples as keys).
- The *range* operator is used to create ranged sets in Python.
- The upper endpoint is not included in the range and ranges start at 0 by default (`range(3) = [0, 1, 2]`).
- The `len` operator gets the number of elements in a given data structure.
- Python also supports conditions in comprehensions, so the model reads naturally in Python's native syntax.
- See also `FinancialModels.xlsx:Dedication-PuLP`.

Introduction to Pyomo

- Pyomo further generalizes the basic framework of PuLP.
  - Support for nonlinear functions.
  - Constraint are defined using Python functions.
  - Support for the construction of “true” abstract models.
  - Built-in support for reading AMPL-style data files.
- Primary classes
  - `ConcreteModel`, `AbstractModel`
  - `Set`, `Parameter`
  - `Var`, `Constraint`

Concrete Pyomo Model for Dedication (`dedication-PyomoConcrete.py`)

- This model is almost identical to the PuLP model.
- The only substantial difference is the way in which constraints are defined, using “rules.”
- Indexing is implemented by specifying additional arguments to the rule functions.
- When the rule function specifies an indexed set of constraints, the indices are passed through the arguments to the function.
- The model is constructed by looping over the index set, constructing each associated constraint.
- Note that if the name of a constraint is `xxx`, the rule function is assumed to be `xxx_rule` unless otherwise specified.
- Note the use of the Python slice operator to extract a subset of a ranged set.

Instantiating and Solving a Pyomo Model

- The easiest way to solve a Pyomo Model is from the command line.

```
pyomo --solver=cbc --summary dedication-PyomoConcrete.py
```

- It is instructive, however, to see what is going on under the hood.
  - Pyomo explicitly creates an “instance” in a solver-independent form.
  - The instance is then translated into a format that can be understood by the chosen solver.
  - After solution, the result is imported back into the instance class.
- We can explicitly invoke these steps in a script.
- This gives a bit more flexibility in post-solution analysis.

Abstract Pyomo Model for Dedication (`dedication-PyomoAbstract.py`)

- In an abstract model, we declare sets and parameters abstractly.
- After declaration, they can be used without instantiation, as in AMPL.
- When creating the instance, we explicitly pass the name of an AMPL-style data file, which is used to instantiate the concrete model.

```
instance = model.create('dedication.dat')
```

- See also `FinancialModels.xlsx:Dedication-Pyomo`.

Example: Short Term Financing

A company needs to make provisions for the following cash flows over the coming five months: $-150K$, $-100K$, $200K$, $-200K$, $300K$.

- The following options for obtaining/using funds are available,
  - The company can borrow up to $\$100K$ at 1% interest per month,
  - The company can issue a 2-month zero-coupon bond yielding 2% interest over the two months,
  - Excess funds can be invested at 0.3% monthly interest.
- How should the company finance these cash flows if no payment obligations are to remain at the end of the period?

Example (cont.)

- All investments are risk-free, so there is no stochasticity.
- What are the decision variables?
  - x_i, the amount drawn from the line of credit in month i,
  - y_i, the number of bonds issued in month i,
  - z_i, the amount invested in month i,
- What is the goal?
  - To maximize the the cash on hand at the end of the horizon.

Example (cont.)

The problem can then be modelled as the following linear program:

$$
\begin{aligned}
\max_{(x,y,z,v)\in\mathbb{R}^{12}} \quad & f(x,y,z,v) = v \\
\text{s.t.} \quad & x_1 + y_1 - z_1 = 150 \\
& x_2 - 1.01x_1 + y_2 - z_2 + 1.003z_1 = 100 \\
& x_3 - 1.01x_2 + y_3 - 1.02y_1 - z_3 + 1.003z_2 = -200 \\
& x_4 - 1.01x_3 - 1.02y_2 - z_4 + 1.003z_3 = 200 \\
& -1.01x_4 - 1.02y_3 - v + 1.003z_4 = -300 \\
& 100 - x_i \geq 0 \quad (i = 1,\ldots,4) \\
& x_i \geq 0 \quad (i = 1,\ldots,4) \\
& y_i \geq 0 \quad (i = 1,\ldots,3) \\
& z_i \geq 0 \quad (i = 1,\ldots,4) \\
& v \geq 0.
\end{aligned}
$$

AMPL Model for Short Term Financing (`short_term_financing.*`)

- Note that we've created some "dummy" variables for use of bonds and credit and investment before time zero.
- These are only for convenience to avoid edge cases when expressing the constraints.
- Again, we see the use of the parameter `T` to capture the number of periods.
- See also `FinancialModels.xlsx:Short-term-financing-AMPL`.