APLEpy: An Open Source Algebraic Programming Language Extension in Python

Suleyman Karabuk

University of Oklahoma, School of Industrial Engineering,
202 West Boyd, Norman, OK, 73019, karabuk@ou.edu

1 Modeling in Python with APLEpy

1.1 An example formulation

$$
\begin{aligned}
& minimize \sum_{i\in N, t\in T} c^Z Z_{it} + \sum_{i\in N, t\in T} c^I I_{it} && && (1)\\
& subject\ to && && \\
& \sum_{i\in N} X_{it} = m && \forall t \in T && (2)\\
& X_{it} - X_{it-1} - Z_{it} \leq 0 && \forall i \in N, t \in T && (3)\\
& Y_{it} \leq X_{it} && \forall i \in N, t \in T && (4)\\
& Y_{it} r_i + I_{it-1} = d_{it} + I_{it} && \forall i \in N, t \in T && (5)\\
& \mathbf{X}, \mathbf{Y}, \mathbf{Z} \in \mathbb{Z}^+, \mathbf{I} \in \Re^+ && &&
\end{aligned}
$$

1.2 The model in APLEpy

Modeling in Python with APLEpy relies mostly on built in Python language constructs and native data types. On top of that, the APLEpy package provides classes to represent model data, decision variables, objective function and constraints. Python has two major complex data types: list and dictionary. A Python list is an ordered set of objects indexed numerically and sequentially with the following syntax: `listobject`[*index*]=*value*. It resembles an array data type in other languages. A Python dictionary is an unordered collection of key and value pairs, where keys are unique identifiers and values are any Python object. It follows a similar syntax as a list: `dictobject`[*key*]=*value*.

Python is a dynamic language, therefore objects of these complex data types grow and shrink dynamically and memory management is done by the Python interpreter transparently. These data types form a suitable basis to represent index sets, collection of decision variables and constraints of a mathematical programming model.

An algebraic modeling language (AML) relies heavily on iterating over one or more index sets. This is accomplished by the Python generators which provide syntax for describing nested loops and creating compound index sets in a single and compact statement as follows:

```
((i1, i2..)  for i1 in set1 for i2 in set2 ..  if i1 = val1 if i2 = val2 ..)
```

The `for, in` and `if` are Python keywords, and `set1` and `set2` represent any iterable Python object. The `for` loops return elements of the iterable objects one by one and store it in `(i1, i2...)`. Any number of `for` loops and `if` statements can be nested. When the above statement is executed, it returns a collection of (`i1, i2..`) combinations that pass the if conditions.

Another commonly used element of an AML is a sum operator which creates a linear expression in a compact form. This is provided by the built in `sum` function in Python with the following syntax:

```
sum(f(i1,i2..) for i1 in set1 for i2 in set2 ..  if i1 = val1 if i2 = val2 ..)
```

The expression `f(i1,i2..)` represents any function that receives parameters created as a result of the nested loops. In this document we do not go beyond these Python constructs, which is all that the reader needs, together with the APLEpy class descriptions, to understand the example code of this section. Interested readers should consult a Python language documentation for more details.

The APLEpy package is organized into modules. The `LP` module provides the base classes for core functionality and a foundation for AML extensions. It implements mainly the following classes: `data, var, objective` and `constraint`. Index sets are stored in list objects. A `data` object is initiated with a collection of index sets and it is derived from built-in type dictionary, which means all the operators that Python provides for manipulating dictionary objects are available for `data` type objects as well. A `var` object is initiated with a collection of index sets and a set of optional (`keyword`=*value*) pairs to set the type of the variable and upper and lower bounds. An `objective` object holds a linear expression and the sense of optimization. A `constraint` object is initialized with a collection of index sets and it is derived from type dictionary.

Next, we go over the Python/APLEpy code that describes the formulation given in the previous section. Python keywords are underlined and APLEpy namespace is differentiated with the prefix `LP`. We analyze the code in the following consecutive parts shown in the respectively numbered listings: (1) model data declaration and input, (2) declaration of decision variables, objective function and constraints, (3) description of objective function and (4) description of constraints.

Listing 1: APLEpy code: data decleration and input

```
1 from APLEpy import LP
2 from APLEpy.LP_Constants import LP_integer
```

```
 4 total_weeks = 0
 5 total_styles = 0

 7 execfile("Size.dat")

 9 WEEKS =  range(total_weeks)
10 STYLES = range(total_styles)
11 total_looms = 0
12 change_cost = 0
13 init_inv = LP.data(STYLES)
14 init_alloc = LP.data(STYLES)
15 inv_cost = LP.data(STYLES)
16 rate = LP.data(STYLES)
17 dem = LP.data(STYLES, WEEKS)

19 execfile("Input.dat")
```

Lines 1-2 import the `LP` module and the `LP_Constants` module respectively. The first `import` statement imposes the requirement that any class in module `LP` referred to with prefix `LP` followed by a dot, whereas the second `import` statement provides the flexibility to use the constants as they appear in the originating module. All model data except scalar values are represented as `LP.data` objects. Python variables are not declared explicitly, instead they spring into existence the first time they are assigned a value. In lines 4-5, two input parameters that determine the size of the problem are assigned an initial value for the purpose of documenting these parameters. Next, in line 7, the actual values of these parameters are read from file "Size.dat", the contents of which are also Python statements that assign values to the two parameters. These values are needed to further determine the size of some of the multi-dimensional input parameters. The index sets are declared and created at lines 9-10 as list objects, followed by two more scalar parameters. The `range` function returns a list of consecutive integers starting from zero up to the number in the argument. Lines 13-17 declare multidimensional input data with index specifications. Finally, line 19 reads the rest of the input data from file "Input.dat" and initializes all the problem parameters declared so far. An `LP.data` object provides a simple way of initializing its data, which is not shown here for brevity. Otherwise, it can be initialized as a regular Python dictionary.

Listing 2: APLEpy code: constraint, variable and obj.fn. declerations

```
20 Alloc  = LP.var(STYLES, WEEKS, type = LP_integer, UB = total_looms)
21 Change = LP.var(STYLES, WEEKS, type = LP_integer, UB = total_looms)
22 Used  = LP.var(STYLES, WEEKS, type = LP_integer, UB = total_looms)
23 Inv = LP.var(STYLES, WEEKS)

25 Total_Costs = LP.objective()

27 availability = LP.constraint(WEEKS)
28 setup_balance = LP.constraint(STYLES, WEEKS)
29 run_limit = LP.constraint(STYLES, WEEKS)
```

```
30 demand = LP.constraint(STYLES, WEEKS)
```

Lines 20-23 declare the decision variables with appropriate index, type, and upper bound specifications. Both `type` and `UB` are APLEpy keywords. Line 25 declares the objective function. Finally lines 27-30 declare the constraint sets with appropriate index sets.

Listing 3: APLEpy code: objective function description

```
31 Total_Costs.Minimize = \
32   sum(change_cost*Change[i,t] for i in STYLES for t in WEEKS) \
33     + sum(Inv[i,t]*inv_cost[i] for i in STYLES for t in WEEKS)
```

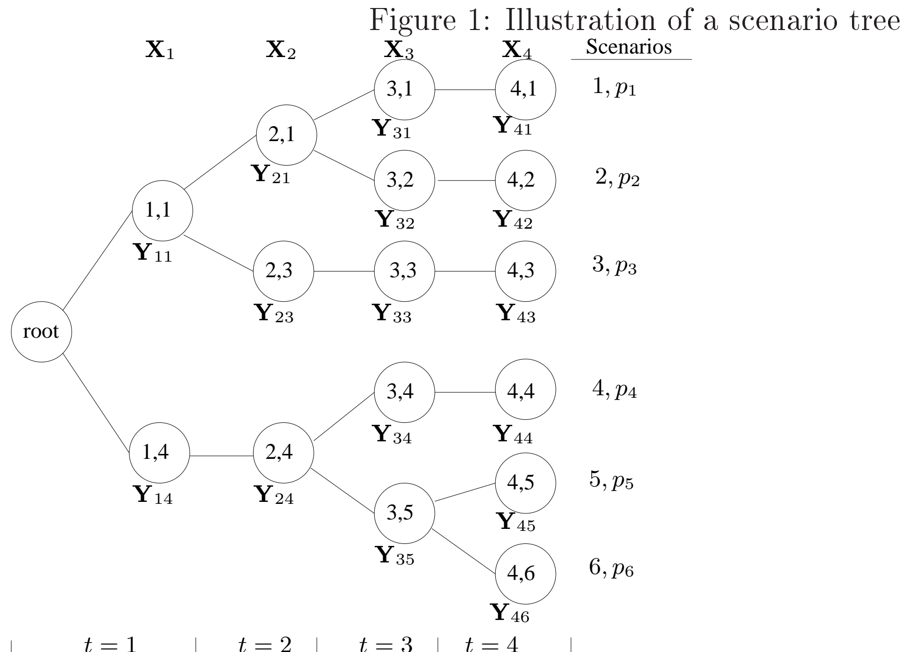
Lines 31-33 create the objective function and set the sense of optimization by assigning to the `Minimize` property of the objective function object. The `Maximize` property would similarly set the sense to be maximization. It also demonstrates the use of the `sum` statement. The backslashes are needed to tell the Python interpreter that all three lines make up a single program statement. In Python, indentations are used to determine code blocks, therefore all the indentations in this sample code are significant, except when a long statement spans several lines as in the above listing.

Listing 4: APLEpy code: description of constraints

```
34 for t in WEEKS:
35   availability[t] = sum (Alloc[i,t] for i in STYLES) == total_looms

37 for (i,t) in ((i,t) for i in STYLES for t in WEEKS):
38   if t == 1:
39     setup_balance[i,t] = Alloc[i,t] - init_alloc[i] - Change[i,t] <= 0
40   else:
41     setup_balance[i,t] = Alloc[i,t] - Alloc[i,t-1]  - Change[i,t] <= 0

43 for (i,t) in ((i,t) for i in STYLES for t in WEEKS):
44   run_limit[i,t] = Used[i,t] <= Alloc[i,t]

46 for (i,t) in ((i,t) for i in STYLES for t in WEEKS if t == 1):
47   demand[i,t] = Used[i,t]*rate[i] + init_inv[i] == dem[i,t] + Inv[i,t]
48 for (i,t) in ((i,t) for i in STYLES for t in WEEKS if t > 1):
49   demand[i,t] = Used[i,t]*rate[i] + Inv[i, t-1] == dem[i,t] + Inv[i,t]

51 LP.Solve()
```

The final listing, shown above, creates the model constraints. It illustrates the use of `for` loops for creating a set of constraints, the use of `if` statement in creating a conditional constraint and the use of generators for creating arbitrary sets of composite indices. A double column marks the start of an `if` or a `for` code block. The indentation of all statements within the same code block has to be the same. The `setup_balance` and the `demand` constraints, at lines 37-41, and 46-49 respectively, both require the first period's constraint to be different than the rest in the set. This can be achieved either by an explicit `if` statement as in the `setup_balance` constraints or by adding an `if` in the index generating part as in the `demand`

Figure 1: Illustration of a scenario tree



constraints. Finally, line 51 passes the model to a solver, invokes it, retrieves the solution and updates the `LP.var` objects, which then show the solution when requested.

2 An Extension for Stochastic Programming

2.1 Background

We consider the multistage recourse models with discrete scenarios. Central to the formulation of a multistage recourse model is a data structure referred to as a scenario tree that shows stages in which random events unfold and decisions are made. Figure 1 shows an example scenario tree for a problem that spans four stages and includes six scenarios. A node in the tree is associated with outcome of an event that leads to a particular realization of the random parameters of the model; hence they are referred to as event nodes (nodes for short). The root node corresponds to the starting point of the planning horizon at which time no random event has cleared yet. As seen in the tree, nodes have unique labels (t,s) used as reference indices, which are utilized in the mathematical formulation of the problem. A scenario is a complete path of nodes starting from the root node of the tree and terminating at a leaf node. It defines a sequence of events that make up one particular realization of the future through the planning horizon. Path probability of a node is represented by p_{ts} which we do not show in the figure for brevity. The decisions that are to be made independent of random outcomes (stage-one decisions) are referred to by $\mathbf{X}_t$ and they are usually associated with the root node. The decisions that can be deferred until the outcome of a random event becomes known (recourse decisions) are referred to by $\mathbf{Y}_{ts}$ which are indexed by the associated event node (t,s).

The node labeling scheme we describe above is adapted from [1]. The representation uses two indices t, and s: the first index t indicates the stage that the node is in, whereas, the second index s indicates the lowest numbered scenario that shares the node. For example, consider scenario 2 and its path of nodes [(1,1), (2,1), (3,2), (4,2)]. It is the only scenario with path that goes through nodes (3,2) and (4,2) in periods 3 and 4 respectively, hence s=2 for both nodes. On the other hand, in t=2, both scenario 1 and scenario 2 share node (2,1), hence s index takes the value of the lowest numbered scenario. Similarly, in t=1, scenarios 1, 2, and 3 goes through the same node (1,1), whose index s is set to 1. This representation ensures that the recourse decisions associated with a set of scenarios that share the same node are equal. This is referred to as the nonanticipativity (NA) constraints. For example, consider node (2,4) which is shared by scenarios 4, 5 and 6 in stage 2. The scheme implicitly enforces that $\mathbf{Y}_{24} = \mathbf{Y}_{25} = \mathbf{Y}_{26}$.

2.2 Stochastic formulations

A node based stochastic programming formulation built on the deterministic model of the previous section is given below.
Model SP1:

$$\begin{aligned}
&\mathit{minimize} \sum_{i\in N, t\in T} c^Z Z_{it} + \sum_{i\in N,(t,s)\in E} p_{ts} c^I I_{its} && (6)\\
&\mathit{subject\ to}\\
&\quad (2),(3)\\
&\quad Y_{its} \le X_{it} && \forall i\in N, (t,s)\in E \quad (7)\\
&Y_{its} r_i + I_{it-1\mathtt{prev}(t,s)} = d_{its} + I_{its} && \forall i\in N, (t,s)\in E \quad (8)\\
&\mathbf{X},\mathbf{Y},\mathbf{Z} \in \mathbb{Z}^+,\ \mathbf{I}\in \Re^+
\end{aligned}$$

Model SP2 below provides a scenario based formulation for the same problem, where NA_{ts} represent the set of scenarios that share node (t,s).
Model SP2:

$$\begin{aligned}
&\mathit{minimize} \sum_{i\in N, t\in T} c^Z Z_{it} + \sum_{i\in N, t\in T, s\in S} p_s c^I I_{its} && (9)\\
&\mathit{subject\ to}\\
&\quad (2),(3)\\
&\quad Y_{its} \le X_{it} && \forall i\in N, t\in T, s\in S \quad (10)\\
&Y_{its} r_i + I_{it-1s} = d_{its} + I_{its} && \forall i\in N, t\in T, s\in S \quad (11)\\
&\quad Y_{its} = Y_{its'} && \forall s, s' \in NA_{ts} \quad (12)\\
&\quad I_{its} = I_{its'} && \forall s, s' \in NA_{ts} \quad (13)\\
&\mathbf{X},\mathbf{Y},\mathbf{Z} \in \mathbb{Z}^+,\ \mathbf{I}\in \Re^+
\end{aligned}$$

In the above formulation (12) and (13) explicitly impose NA constraints that have been taken care of implicitly in formulation SP1.

2.3 APLEpy implementation

In the heart of APLEpy's implementation lies the `scenario_tree` class which reads a specially formatted data file that describes a scenario tree. It creates all the required data structures and makes them available for use with APLEpy objects. The scenario data file constitutes one line of description for each node in the tree in the format shown in the first row of Table 1 with an example shown in row 2. Every node is to be labeled uniquely within a stage. The label could be either a number or an informative string that describes the significance of the associated event node. Parent node field refers to the label of the parent node at the previous stage. Node probability is the probability of the associated event occurring. Note that the total of probabilities over all siblings of a parent node has to sum to 1.0. The rest of the data fields constitute values of a random vector ($n > 1$) or a scalar ($n = 1$) associated with the node.

Table 1: Data input format for scenario tree

Stage	Node label	Parent node	Node probability	Rate1	Rate2	...	Rate$n$
2	Low_demand	High_demand	0.25	-0.2	0.05	...	0.4

Table 2 below shows the methods provided by the `scenario_tree` class.

Table 2: Methods provided by `scenario_tree` class.

Property/Method	Return value
`event_nodes`	a list that contains event nodes E: $\{(t, s)\|(t, s) \in E\}$
`cprob`	a dictionary of conditional probabilities for each node (t,s)
`sprob`	a list of scenario probabilities for each scenario s
`prev`	a dictionary with key, value pairs as (t,s):`prev`(t,s)
`scenarios`	an object that has a listing of scenarios
`stages`	an object that has a listing stages
`rate[i]`, $i \in R$	the ith dictionary of values associated with each node (t,s)

Formulation SP1 can be implemented easily with the properties described above. Listing (5) below shows the differences with respect to the code for the deterministic formulation.

Listing 5: APLEpy code: Formulation SP1

```
1  from APLEpy import SLP

3  Tree = SLP.scenario_tree('tree.dat')
4  EVNODES = Tree.event_nodes
5  prob = Tree.cprob
6  prev = Tree.prev
```

```
7  ch = Tree.rates[0]

9  ...

11 Used  = LP.var(STYLES, EVNODES, type=LP_integer, UB=total_looms)
12 Inv = LP.var(STYLES, EVNODES)

14 ...

16 run_limit = LP.constraint(STYLES, EVNODES)
17 demand = LP.constraint(STYLES, EVNODES)

19 ...

21 Total_Costs.Minimize = \
22   sum(change_cost*Change[i,t] for i in STYLES for t in WEEKS) \
23     + sum(Inv[i,t,s]*inv_cost[i]*prob[t,s] for i in STYLES for (t,s)
          in EVNODES)

25 ...

27 for (i,t,s) in ((i,t,s) for i in STYLES for (t,s) in EVNODES):
28   run_limit[i,t,s] = Used[i,t,s] <= Alloc[i,t]

30 for (i,t,s) in ((i,t,s) for i in STYLES for (t,s) in EVNODES if t==1):
31   demand[i,t,s] = Used[i,t,s]*rate[i] + init_inv[i] ==
        dem[i,t]*(1+ch[t,s]) + Inv[i,t,s]
32 for (i,t,s) in ((i,t,s) for i in STYLES for (t,s) in EVNODES if t > 1):
33   demand[i,t,s] = Used[i,t,s]*rate[i] + Inv[i, t-1, prev[t,s]] ==
        dem[i,t]*(1+ch[t,s]) + Inv[i,t,s]
```

Line 1 imports the stochastic linear programming (SLP) extensions module of APLEpy. An object of type `SLP.scenario_tree` is created from input file 'tree.dat' in line 3. The data structures that are required for the formulation are copied to local variables in lines 4-7. This is only for brevity and clarity of code. The tree object could also be used, with appropriate properties, in place of the local variables. In this example, the `rates` values represent fractional changes applied to the whole demand vector (which is input separately) at each node (t,s). Another approach would be to input as many rates fields as the total number of styles in which case the rate of change with respect to the base demand would be different for all styles. It is convenient to input a base demand separately in case a deterministic model is to be solved, for example for comparison purposes. Alternatively, `rates` fields could keep the absolute value for demand rather than the deviation from a base.

The recourse variables and the recourse constraints are declared with nodes as part of their index set in lines 10-11 and 16-17 respectively. The objective function is described in lines 21-23. Notice that each entry in the index set EVNODES is a tuple (t,s) same as in our formulation of SP1. Lines 27-33 describe the recourse constraints each of which have an element for every index in EVNODES. It is critical to use the `prev` variable to refer to the

parent node of an (t,s) element as in line 33. The rest of the code is exactly the same as in the deterministic version given in listings 1-4 in the previous section.

In order to implement formulation SP2 without explicit NA constraints, we provide classes `SLP.data`, `SLP.var` and `SLP.constraint`. These classes are the stochastic counterparts of the corresponding `LP` classes and they are derived from their counterparts. The `SLP` objects are initialized with an object of type `SLP.scenarios` and optionally with an object of type `SLP.stages` both of which are returned by an `SLP.scenario_tree` object. We illustrate their use in the following listing which implements formulation SP2 without explicit NA constraints.

Listing 6: APLEpy code: Formulation SP2

```
1  from APLEpy import SLP

3  Tree = SLP.scenario_tree('tree.dat')
4  prob = Tree.sprob
5  SCENARIOS = Tree.scenarios
6  WEEKS = Tree.stages
7  rate1 = Tree.rates[0]

9  ch = SLP.data(WEEKS, SCENARIOS)
10 ch(rate1)

12 ...

14 Used  = SLP.var(STYLES, WEEKS, SCENARIOS, type=LP_integer, UB =
       total_looms)
15 Inv = SLP.var(STYLES, WEEKS, SCENARIOS)

17 ...

19 run_limit = SLP.constraint(STYLES, WEEKS, SCENARIOS)
20 demand = SLP.constraint(STYLES, WEEKS, SCENARIOS)

22 ...

24 Total_Costs.Minimize = \
25   sum(change_cost*Change[i,t] for i in STYLES for t in WEEKS) \
26     + sum(Inv[i,t,s]*inv_cost[i]*prob[s] for i in STYLES for t in WEEKS
           for s in SCENARIOS)

28 ...

30 for (i,t,s) in ((i,t,s) for i in STYLES for t in WEEKS for s in
       SCENARIOS):
31   run_limit[i,t,s] = Used[i,t,s] <= Alloc[i,t]

33 for (i,t,s) in ((i,t,s) for i in STYLES for t in WEEKS for s in
```

```
        SCENARIOS if t == 1):
34    demand[i,t,s] = Used[i,t,s]*rate[i] + init_inv[i] ==
          dem[i,t]*(1+ch[t,s]) + Inv[i,t,s]
35  for (i,t,s) in ((i,t,s) for i in STYLES for t in WEEKS for s in
        SCENARIOS if t > 1):
36    demand[i,t,s] = Used[i,t,s]*rate[i] + Inv[i, t-1, s] ==
          dem[i,t]*(1+ch[t,s]) + Inv[i,t,s]
```

Line 1 imports SLP extensions and line 3 creates and initializes the same `scenario_tree` object as before. In line 4 the scenario probabilities are acquired (compared to path probabilities in Listing 5.) Scenario and stage objects are also captured in lines 5-6 respectively. Notice that in this formulation weeks have a one to one correspondence with stages. The first `rate` vector is contained in a local variable in line 7. Lines 9 and 10 declare and initialize an object from `SLP.data` class respectively. Notice that the index sets of the `SLP.data` type objects include a scenario object which is required. Line 10 shows how an object of type (`S`)`LP.data` is initialized. Alternatively an (`S`)`LP.data` type object can be initialized as any dictionary in Python. Remember that `LP.data` class is derived from a Python dictionary type and `SLP.data` is in turn derived from `LP.data` and as such it inherits every feature of its parent class.

Lines 14-15 and 19-20 declare the recourse variables and constraints respectively. The objective function is described in lines 24-26. Notice that the scenario probabilities are used with no explicit reference to node probabilities. Lines 30-36 describe the recourse constraints in the same way as in formulation SP2. Again notice that parent node relationship is not needed to be imposed when referring to the previous week's inventory in the demand constraints. The recourse constraints are simply straightforward extensions of their deterministic counterparts with an additional scenario index.

In the APLEpy code of Listing 6 there is no mention of any of the node based data structures. Everything in the formulation is based on scenarios. The NA constraints are handled behind the scenes by APLEpy classes. The implementation is based on intercepting the scenario indices before they are used, in any manner, by the respective `SLP` objects and map them to the corresponding node index in the scenario tree. The result is that the formulation that is passed on to the solver is the same as the formulation of SP1 which is much more efficient than formulation SP2 in terms of total number of variables and constraints.

References

[1] H. I. Gassman and A. M. Ireland. Scenario formulation in an algebraic modeling language. *Annals of Operations Research*, 59:45–75, 1995.