FICO® Xpress Optimizer - Python Interface

FICO® Xpress Optimization Training

Introduction to the course

FICO

Format, aims and other materials

- Course split into modules, where each module comprises:
  - introduction to general concepts about a topic
  - code snippets with examples of application
  - video demonstration of an Xpress Python example using Xpress Workbench

Format, aims and other materials

- Course split into modules, where each module comprises:
  - introduction to general concepts about a topic
  - code snippets with examples of application
  - video demonstration of an Xpress Python example using Xpress Workbench
- At the end of the course you will:
  - be familiar about formulating optimization models using the Xpress Python interface
  - know how to use Xpress to model and solve problems and analyzing the solution
  - be able to navigate the Xpress Python examples and run them using Xpress Workbench

Format, aims and other materials

- Course split into modules, where each module comprises:
  - introduction to general concepts about a topic
  - code snippets with examples of application
  - video demonstration of an Xpress Python example using Xpress Workbench
- At the end of the course you will:
  - be familiar about formulating optimization models using the Xpress Python interface
  - know how to use Xpress to model and solve problems and analyzing the solution
  - be able to navigate the Xpress Python examples and run them using Xpress Workbench
- Other considerations:
  - not exhaustive, not a replacement for the reference manual
  - focuses on areas that are of practical importance
  - assumes the user is familiar with the mathematical optimization concepts involved

Hint: *Familiarize yourself with the Python interface reference manual by looking up the details for each topic*

Topics covered

- Basic topics:
  - Introduction and installation guidelines
  - Modeling a basic optimization problem
  - Solving and querying a problem
  - Reading and writing a problem
  - Using the numerical library *NumPy*
  - Building models efficiently
  - Exceptions

Topics covered

- Basic topics:
  - Introduction and installation guidelines
  - Modeling a basic optimization problem
  - Solving and querying a problem
  - Reading and writing a problem
  - Using the numerical library *NumPy*
  - Building models efficiently
  - Exceptions
- Advanced topics:
  - Indicator constraints
  - Special Ordered Set (SOS) constraints
  - Piecewise linear functions
  - General constraints
  - Optimizing with multiple objectives
  - Modeling and solving nonlinear problems
  - Controls and attributes
  - Using callbacks

Installing the Xpress Python module

Installing the Xpress Python module

- The Xpress Python module can be installed from the two main Python repositories: The Python Package Index (PyPI) and the Conda repository
  - installing the Xpress Python interface does not require one to install the whole Xpress suite, as all necessary libraries are provided

Installing the Xpress Python module

- The Xpress Python module can be installed from the two main Python repositories: The Python Package Index (PyPI) and the Conda repository
  - installing the Xpress Python interface does not require one to install the whole Xpress suite, as all necessary libraries are provided
- The install comes with a copy of the *community* license, which allows for solving problems with up to a total of 5000* between variables and constraints:
  - if you already have an Xpress license, please make sure to set the `XPAUTH_PATH` environment variable to the full path to the license file `xpauth.xpr`
    - for example, if the license file is `/home/brian/xpauth.xpr`, then `XPAUTH_PATH` should be set to `/home/brian/xpauth.xpr` in order for the module to pick the right license
  - for nonlinear problems, including non-quadratic and non-conic, a limit of 200 variables and constraints applies

Installation from the Python Package Index (PyPI)

- The Xpress Python interface is available on the PyPI server and can be installed with the following command:

```
pip install xpress
```

- Earlier versions of the module can be installed by appending a "`==VERSION`" string to the module name, for instance:

```
pip install xpress==9.2.5
```

Installation from the Python Package Index (PyPI)

- The Xpress Python interface is available on the PyPI server and can be installed with the following command:

```
pip install xpress
```

- Earlier versions of the module can be installed by appending a "`==VERSION`" string to the module name, for instance:

```
pip install xpress==9.2.5
```

- Packages for Python 3.8 to 3.12 are available, for Windows, Linux, and MacOS:
  - the package contains the Python interface module, its documentation in PDF format, the Xpress Solver libraries, various examples of use, and a copy of the community license (see http://subscribe.fico.com/xpress-optimization-community-license)

Installation from Conda

- A Conda package is available for download with the following command:

```
conda install -c fico-xpress xpress
```

- For installing earlier versions, follow the following example below:

```
conda install -c fico-xpress xpress=9.2.5
```

  - note that the Conda installer only uses a single "="

Installation from Conda

- A Conda package is available for download with the following command:

```
conda install -c fico-xpress xpress
```

- For installing earlier versions, follow the following example below:

```
conda install -c fico-xpress xpress=9.2.5
```

  - note that the Conda installer only uses a single "="
- The content of the Conda package is the same as that of the PyPI package:
  - Conda packages are available for Python 3.8 to 3.12, for Windows, Linux, and MacOS

Note: *The Xpress Conda package requires the 'intel-openmp' package on Intel platforms (available on the 'main' and 'intel' Conda channels)*

Important consideration

- If you installed the Xpress Optimization suite before downloading the Xpress Conda or PyPI package, the Xpress Python interface will try to use the license file in your Xpress installation automatically:
  - *on Windows*: the Xpress installer sets the `XPRESSDIR` environment variable to the installation directory, and the Xpress Python interface will look for a license file at `%XPRESSDIR%\bin\xpauth.xpr`

FICO

Important consideration

- If you installed the Xpress Optimization suite before downloading the Xpress Conda or PyPI package, the Xpress Python interface will try to use the license file in your Xpress installation automatically:
  - *on Windows*: the Xpress installer sets the `XPRESSDIR` environment variable to the installation directory, and the Xpress Python interface will look for a license file at `%XPRESSDIR%\bin\xpauth.xpr`
  - *on Linux and MacOS*: the Xpress installer creates a script named `xpvars.sh` in the `bin` folder of the Xpress installation
    - this script sets `XPRESSDIR` to the installation directory, and sets `XPAUTH_PATH` to the location of the license file
    - the Xpress Python interface will use the `XPAUTH_PATH` value to locate the license from your Xpress installation. If for some reason `XPAUTH_PATH` is not set, the Xpress Python interface will look for a license file at `$XPRESSDIR/bin/xpauth.xpr`

Modeling a basic optimization problem

Getting started and problem creation

- Importing the Xpress Python package:
  - the `xpress` Python module can be imported as follows:

    ```
    import xpress
    ```

  - since all types and methods must be called by prepending "xpress.", it is advisable to alias the module name upon import:

    ```
    import xpress as xp
    ```

  - a complete list of methods and constants available in the module is obtained by running the Python command `dir(xpress)`

Getting started and problem creation

- Importing the Xpress Python package:
  - the `xpress` Python module can be imported as follows:

    ```
    import xpress
    ```

  - since all types and methods must be called by prepending "xpress.", it is advisable to alias the module name upon import:

    ```
    import xpress as xp
    ```

  - a complete list of methods and constants available in the module is obtained by running the Python command `dir(xpress)`
- Problem creation:
  - create an empty optimization problem as follows:

    ```
    p = xp.problem()
    ```

  - a name can be assigned to a problem upon creation:

    ```
    p = xp.problem(name="My first problem")
    ```

Create and add decision variables

- Use the problem.addVariable() function to create decision variables and directly add them to the optimization problem:

```
p.addVariable(name, lb, ub, threshold, vartype)
```

Create and add decision variables

- Use the problem.addVariable() function to create decision variables and directly add them to the optimization problem:

```
p.addVariable(name, lb, ub, threshold, vartype)
```

- All parameters are optional:
  - `name`: string containing the name of the variable. A default name is assigned if not specified
  - `lb,ub`: lower bound (0 by default) and upper bound (+inf by default), respectively
  - `threshold`: must be defined for semi-continuous, semi-integer, and partially integer variables, with a value between their lower and upper bounds
  - `vartype`: the variable type, one of the six following types:
    - `xp.continuous` for continuous variables
    - `xp.binary` for binary variables (`lb,ub`: are further restricted to 0 and 1, respectively)
    - `xp.integer` for integer variables
    - `xp.semicontinuous` for semi-continuous variables
    - `xp.semiinteger` for semi-integer variables
    - `xp.partiallyinteger` for partially integer variables

Create and add decision variables

- Variables added to an Xpress problem are constrained to be nonnegative by default. To add a free variable, one must specify its lower bound as `-xp.infinity`:

```
x = p.addVariable(lb=-xp.infinity)
```

Create and add decision variables

- Variables added to an Xpress problem are constrained to be nonnegative by default. To add a free variable, one must specify its lower bound as `-xp.infinity`:

```
x = p.addVariable(lb=-xp.infinity)
```

- A set of variables can be created at once by using lists and dictionaries:

```
# with lists
L = range(20)
x = [p.addVariable(ub=1) for i in L]
y = [p.addVariable(vartype=xp.binary) for i in L]

# with dictionaries
LC = ['Seattle','Miami','Omaha','Charleston']
z = {i: p.addVariable(vartype=xp.integer) for i in LC}
```

Hint: *Dictionaries allow us to refer to such variables using the names in LC, for instance `z['Seattle']`, `z['Charleston']`.*

Create and add decision variables

- Variable names can be useful when saving a problem to a file and when querying the problem for the value of a variable in an optimal solution:
    - when querying for a variable or expression containing that variable, its name will be printed rather than the Python object used in programming:
        - this allows for querying a problem using both the variable object and its name

Create and add decision variables

- Variable names can be useful when saving a problem to a file and when querying the problem for the value of a variable in an optimal solution:
  - when querying for a variable or expression containing that variable, its name will be printed rather than the Python object used in programming:
    - this allows for querying a problem using both the variable object and its name
  - if a variable is not specified with a name by the user, it will be assigned a "C" followed by a sequence number:

```
v = p.addVariable(lb=-1, ub=2)
print(v)
>>> C1
```

  - if a variable name is explicitly specified:

```
x = p.addVariable(name='myvar')
print(v + 2 * x)
>>> C1 + 2 myvar
```

Create and add decision variables

- Use the function problem.addVariables() for creating an indexed set of variables:

```
p.addVariables(*indices, name, lb, ub, threshold, vartype)
```

  - parameter `*indices` stands for one or more arguments, each a list, a set, or a positive integer:
    - produces as many variables as can be indexed with all combinations from the lists/sets
  - if `*indices` consists of one list, a variable will be created for each element in the list:

```
myvar = p.addVariables(['a','b','c'], lb=-1, ub=+1)
```

    - yields myvar['a'], myvar['b'], and myvar['c']

Create and add decision variables

- Use the function problem.addVariables() for creating an indexed set of variables:

```
p.addVariables(*indices, name, lb, ub, threshold, vartype)
```

  - parameter `*indices` stands for one or more arguments, each a list, a set, or a positive integer:
    - produces as many variables as can be indexed with all combinations from the lists/sets
  - if `*indices` consists of one list, a variable will be created for each element in the list:

```
myvar = p.addVariables(['a','b','c'], lb=-1, ub=+1)
```

    - yields myvar['a'], myvar['b'], and myvar['c']
  - in case of more than one list/set, the Cartesian product of these lists/sets provides the indexing space of the result in the form of a dictionary indexed by tuples:

```
y = p.addVariables(['a','b','c','d'], [100, 120, 150], vartype=xp.integer)
```

    - results in 12 variables y['a',100], y['a',120],y['a',150],...,y['d',150]

Create and add decision variables

- Use the function problem.addVariables() for creating an indexed set of variables:

```
p.addVariables(*indices, name, lb, ub, threshold, vartype)
```

  - parameter `*indices` stands for one or more arguments, each a list, a set, or a positive integer:
    - produces as many variables as can be indexed with all combinations from the lists/sets
  - if `*indices` consists of one list, a variable will be created for each element in the list:

```
myvar = p.addVariables(['a','b','c'], lb=-1, ub=+1)
```

    - yields myvar['a'], myvar['b'], and myvar['c']
  - in case of more than one list/set, the Cartesian product of these lists/sets provides the indexing space of the result in the form of a dictionary indexed by tuples:

```
y = p.addVariables(['a','b','c','d'], [100, 120, 150], vartype=xp.integer)
```

    - results in 12 variables y['a',100], y['a',120],y['a',150],...,y['d',150]
  - for creating a large number of variables, one can obtain a *NumPy* array of any dimension by just specifying numbers as the index arguments. For example, to create 35 integer variables `x[0,0]`,`x[0,1]`,`...`,`x[4,6]`, one can simply use:

```
x = p.addVariables(5, 7, vartype=xp.integer)
```

Create and add constraints

- Constraints can be created in a natural way by overloading the operators <=, ==, >=:

```
myconstr = x1 + x2 * (x2 + 1) <= 4
myconstr2 = xp.exp(xp.sin(x1)) + x2 * (x2**5 + 1) <= 4
```

- Use the problem.addConstraint() method to add constraints to a problem:

```
p.addConstraint(c1, c2, ...)
```

  - where `c1,c2...` are constraints or list/tuples/array of constraints
  - can be added directly, for example:

```
p.addConstraint(v1 + xp.tan(v2) <= 3)
```

Create and add constraints

- Constraints can be created in a natural way by overloading the operators <=, ==, >=:

```
myconstr = x1 + x2 * (x2 + 1) <= 4
myconstr2 = xp.exp(xp.sin(x1)) + x2 * (x2**5 + 1) <= 4
```

- Use the problem.addConstraint() method to add constraints to a problem:

```
p.addConstraint(c1, c2, ...)
```

  - where `c1,c2...` are constraints or list/tuples/array of constraints
  - can be added directly, for example:

```
p.addConstraint(v1 + xp.tan(v2) <= 3)
```

- Several constraints (or lists of constraints) can be added at once:

```
p.addConstraint(myconstr, myconstr2)
p.addConstraint(x[i] + y[i] <= 2 for i in range(10))
```

Create and add constraints

- Lists and dictionaries can also be used to create constraints:

```
LC = ['Seattle','Miami','Omaha','Charleston']
constr = [x[i] <= y[i] for i in LC]
cliq = {(i,j): x[i] + x[j] <= 1 for i in LC for j in L if i != j}
p.addConstraint(constr, cliq)
```

Hint: *By using dictionaries, each constraint can be referred to with pairs of names, e.g.* `cliq['Seattle','Miami']`.

Create and add constraints

- Lists and dictionaries can also be used to create constraints:

```
LC = ['Seattle','Miami','Omaha','Charleston']
constr = [x[i] <= y[i] for i in LC]
cliq = {(i,j): x[i] + x[j] <= 1 for i in LC for j in L if i != j}
p.addConstraint(constr, cliq)
```

Hint: *By using dictionaries, each constraint can be referred to with pairs of names, e.g.* `cliq['Seattle','Miami']`.

- For compactness, formulate constraints with the `xp.Sum()` operator to define sums of variables or expressions:

```
p.addConstraint(xp.Sum(x) <= 1)
p.addConstraint(xp.Sum([y[i] for i in range(10)]) <= 1)
p.addConstraint(xp.Sum([x[i]**5 for i in range(9)]) <= x[9])
```

Create and add constraints

- Alternatively, use the method xpress.constraint() to be able to provide a name for the constraint:

```
xp.constraint(constraint, name)
xp.constraint(body, type, rhs, lb, ub, name)
```

  - can be passed a `constraint` object directly or defined via its members `body, type, rhs`
  - for the second case, `type` of constraint can be `xp.leq`, `xp.geq`, `xp.eq`, or `xp.rng`

Create and add constraints

- Alternatively, use the method xpress.constraint() to be able to provide a name for the constraint:

```
xp.constraint(constraint, name)
xp.constraint(body, type, rhs, lb, ub, name)
```

  - can be passed a `constraint` object directly or defined via its members `body, type, rhs`
  - for the second case, `type` of constraint can be `xp.leq`, `xp.geq`, `xp.eq`, or `xp.rng`
- Examples of use:
  - passing a constraint expression directly as an argument and defining a name:

```
c1 = xp.constraint(x1 + 2*x2 <= 3, name="myconstraint1")
```

  - passing the `body, type` and `rhs` arguments instead of the constraint object:

```
c2 = xp.constraint(body=x1 + 2*x2, type=xp.leq, rhs=3, name="myconstraint2")
```

  - can be particularly useful to define range constraints by passing the type as `xp.rng` and `lb`, `ub`:

```
c3 = xp.constraint(body=x1 + 2*x2, type=xp.rng, lb=0, ub=3, name="myconstraint3")
```

    - this will add the range constraint `0 <= x1 + 2*x2 <= 3`

Create and add the objective function

- The method problem.setObjective() sets the objective function of a problem:

```
p.setObjective(objective, sense=xp.minimize)
```

  - where `objective` is a required expression defining the objective, and the optional argument `sense` can be either `xp.minimize` or `xp.maximize`

Create and add the objective function

- The method problem.setObjective() sets the objective function of a problem:

```
p.setObjective(objective, sense=xp.minimize)
```

  - where `objective` is a required expression defining the objective, and the optional argument `sense` can be either `xp.minimize` or `xp.maximize`

- By default, the objective function is to be minimized:

```
p.setObjective(xp.Sum ([y[i]**2 for i in range (10)]))
```

- Define `sense=xp.maximize` to change the optimization sense to maximization:

```
obj = v1 + 3 * v2
p.setObjective (obj, sense=xp.maximize)
```

Solving and querying a problem

Solving a problem

- The method problem.optimize() is used to solve an optimization problem that was either built via Python functions or read from a file:

```
p.optimize(flag)
```

  - the algorithm is determined automatically as follows:
    - if all variables are continuous, the problem is solved as a continuous optimization problem
    - if at least one integer variable was declared, then the problem will be solved as a mixed integer (linear, quadratically constrained, or nonlinear) problem
    - if the problem contains nonlinear constraints that are non-quadratic and non-conic, then the appropriate nonlinear solver of the Xpress Optimization suite will be called: either Xpress Global or Xpress Nonlinear, depending on available licenses

Note: *Non-convex quadratic problems are included in the base offering of the Xpress Solver license and will by default be solved with the Xpress Global technology*

Solve and solution status

- The *solve* and *solution* statuses of a problem can be obtained via the *solvestatus* and *solstatus* attributes using problem.attributes.<*attribute*>, which are also returned by the p.optimize() function:

```
solvestatus, solstatus = p.optimize()
```

  - where the value of:
    - *solvestatus* can be {COMPLETED, STOPPED, FAILED, UNSTARTED}
    - *solstatus* can be {FEASIBLE, OPTIMAL, INFEASIBLE, UNBOUNDED, NOTFOUND}

Solve and solution status

- The *solve* and *solution* statuses of a problem can be obtained via the *solvestatus* and *solstatus* attributes using problem.attributes.<*attribute*>, which are also returned by the p.optimize() function:

```
solvestatus, solstatus = p.optimize()
```

  - where the value of:
    - *solvestatus* can be {COMPLETED, STOPPED, FAILED, UNSTARTED}
    - *solstatus* can be {FEASIBLE, OPTIMAL, INFEASIBLE, UNBOUNDED, NOTFOUND}
  - the statuses can then be conveniently queried as follows:

```
if solvestatus == xp.SolveStatus.COMPLETED:
  print("Solve completed with solution status: ", solstatus.name)
else:
  print("Solve status: ", solvestatus.name)
```

Querying a problem

- The method problem.getSolution() returns the optimal solution as a list:
  - an argument can be passed in the form of a list, dictionary, tuple, or any sequence (including *NumPy* arrays) of variables, indices, strings, expressions and other aggregate objects
  - if an optimal solution was not found but at least one feasible solution is available, data based on the best feasible solution will be returned

Querying a problem

- The method problem.getSolution() returns the optimal solution as a list:
  - an argument can be passed in the form of a list, dictionary, tuple, or any sequence (including *NumPy* arrays) of variables, indices, strings, expressions and other aggregate objects
  - if an optimal solution was not found but at least one feasible solution is available, data based on the best feasible solution will be returned
- Examples:

```
p.optimize()

print(p.getSolution())             # prints a list with an optimal solution
print("v1 is", p.getSolution(v1))  # only prints the value of v1

a = p.getSolution(x)               # gets the values of all variables in the list x
b = p.getSolution(range(4))        # gets the value of the first four variables
c = p.getSolution('Var1')          # gets the value of a variable by its name
d = p.getSolution(v1 + 3*x)        # gets the value of an expression for the solution
e = p.getSolution(np.array(x))     # gets a NumPy array with the solution of x
```

Querying a problem

- The method problem.getSlacks() retrieves the slack for one or more constraints of the problem w.r.t. the solution found:
  - works with indices, constraint names, constraint objects, and lists thereof

```
print(p.getSlacks())                    # prints a list of slacks for all constraints
print("slack_1 is", p.getSlacks(cons1)) # only prints the slack of cons1

a = p.getSlacks(conlist)         # gets the slacks of all constraints in list 'conlist'
b = p.getSlacks(range(2))        # gets the slacks of the first 2 constraints of the probl
```

Note: *Both methods `p.getSolution()` and `p.getSlacks()` work for continuous or mixed integer problems*

Querying a problem

- For problems that only have continuous variables, the two methods problem.getDuals() and problem.getRCosts() return the list of dual variables and reduced costs, respectively:
  - their usage is similar to that of problem.getSlacks()

```
print("Duals of last two constraints:", p.getDuals(constr[-2:]))
print("Reduced costs of first two variables:", p.getRCosts(x[:2]))
```

Querying a problem

- For problems that only have continuous variables, the two methods problem.getDuals() and problem.getRCosts() return the list of dual variables and reduced costs, respectively:
  - their usage is similar to that of problem.getSlacks()

```
print("Duals of last two constraints:", p.getDuals(constr[-2:]))
print("Reduced costs of first two variables:", p.getRCosts(x[:2]))
```

- The inner workings of the Python interface obtain a copy of the whole solution, slack, dual, or reduced cost vectors, even if only one element is requested:
  - instead of repeated calls to p.getSolution() or p.getSlack(), it is advisable to make one call and store the result in a list to be consulted in a loop:

```
sol = p.getSolution()
for i in N:
    if sol[i] > 1e-3:
        print(i)
```

Reading and writing a prob-lem

FICO

Reading a problem

- A problem can be read from a file via the problem.read() method, which takes the file name as its argument:

```
p.read(filename)
```

  - `filename` must be a string of up to 200 characters with the name of the file to be read
    - in case no file extension is passed, the method will search for the MPS and LP extensions of the file name

Reading a problem

- A problem can be read from a file via the problem.read() method, which takes the file name as its argument:

```
p.read(filename)
```

  - `filename` must be a string of up to 200 characters with the name of the file to be read
    - in case no file extension is passed, the method will search for the MPS and LP extensions of the file name
  - read problem in file `problem1.lp` and output an optimal solution:

```
p.read("problem1.lp")
p.optimize()
print("solution of problem1:", p.getSolution())
```

Writing a problem

- A user-built problem can be written to a file with the problem.write() method:

```
p.write(filename)
```

  - `filename` must be a string of up to 200 characters with the name of the file to which the problem is to be written
    - if extension is omitted, the default problem name is used with a `.mps` extension (recommended)
    - if the `.lp` extension is used, the problem is written in LP format
  - example writing a problem in LP format:

```
p.optimize()
p.write("problem2.lp")
```

Using the numerical library *NumPy*

Using *NumPy* arrays

- The *NumPy* library allows for creating and using arrays of any order and size for efficiency and compactness purposes:
  - *NumPy* arrays can be used when creating variables, expressions (linear and nonlinear) with variables, and constraints

Using *NumPy* arrays

- The *NumPy* library allows for creating and using arrays of any order and size for efficiency and compactness purposes:
  - *NumPy* arrays can be used when creating variables, expressions (linear and nonlinear) with variables, and constraints
  - the example below declares two arrays of variables and creates the set of constraints `x[i] <= y[i]` for all `i` in the set `S`:

```
import numpy as np
import xpress as xp
S = range(20)
p = xp.problem()
x = np.array([p.addVariable()                        for i in S], dtype=xp.npvar)
y = np.array([p.addVariable(vartype=xp.binary) for i in S], dtype=xp.npvar)
constr1 = x <= y
p.addConstraint(constr1)
```

Using *NumPy* multiarrays

- The problem.addVariables() function in its simplest usage directly returns a *NumPy* array of variables with one or more indices:
  - the array declarations:

```
x = np.array([p.addVariable(name='v({0})'.format(i)) for i in range(20)],
      dtype=xp.npvar).reshape(5,4)
y = np.array([p.addVariable(lb=-1, ub=1) for i in range(1000)], dtype=xp.npvar)
```

  - ...can be written equivalently in the compact form using p.addVariables() as:

```
x = p.addVariables(5, 4, name='v')
y = p.addVariables(1000, lb=-1, ub=1)
```

Hint: *NumPy allows for multiarrays with one or more 0-based indices*

Broadcasting features

- *NumPy* operations can be replicated on each element of an array, taking into account its *broadcasting* features:
  - these operations can be carried out on arrays of any number of dimensions, and can be aggregated at any level
  - to *broadcast* the right-hand side `1` to all elements of the array, creating the set of constraints `x[i] + y[i] <= 1` for all `i` in the set `S`:

```
constr2 = x + y <= 1
```

Broadcasting features

- *NumPy* operations can be replicated on each element of an array, taking into account its *broadcasting* features:
  - these operations can be carried out on arrays of any number of dimensions, and can be aggregated at any level
  - to *broadcast* the right-hand side `1` to all elements of the array, creating the set of constraints `x[i] + y[i] <= 1` for all `i` in the set `S`:

    ```
    constr2 = x + y <= 1
    ```

  - creating two three-dimensional arrays of variables involved in a set of constraints:

    ```
    z = p.addVariables(4, 5, 10)
    t = p.addVariables(4, 5, 10, vartype=xp.binary)
    p.addConstraint(z**2 <= 1 + t)
    ```

Products of *NumPy* arrays

- The xpress.Dot() operator is useful for carrying out aggregate operations on vectors and matrices in arrays containing Xpress variables and expressions:
  - when handling variables or expressions, use the xp.Dot() operator rather than *NumPy*'s *dot* operator
  - examples where `z` is one-dimensional:

```
p.addConstraint(xp.Dot(z, z) <= 1)     # restrict squared norm of z to at most 1
Q = np.random.random(20, 20)
p.addConstraint(xp.Dot((t-z), Q, (t-z)) <= 1) # bound quadratic expression by 1
```

Products of *NumPy* arrays

- The xpress.Dot() operator is useful for carrying out aggregate operations on vectors and matrices in arrays containing Xpress variables and expressions:
  - when handling variables or expressions, use the xp.Dot() operator rather than *NumPy*'s *dot* operator
  - examples where `z` is one-dimensional:

```
p.addConstraint(xp.Dot(z, z) <= 1)     # restrict squared norm of z to at most 1
Q = np.random.random(20, 20)
p.addConstraint(xp.Dot((t-z), Q, (t-z)) <= 1) # bound quadratic expression by 1
```

  - for multi-dimensional arrays, the size of the last dimension of the first array must match the size of the penultimate dimension of the second vector:

```
a = p.addVariables(4,6, name="a")
b = p.addVariables(6,2, name="b")
p.addConstraint(xp.Dot(a,b) <= 10)
```

    - is valid and yields a 4x2 matrix creating 8 new constraints
    - rules are the same as for the *NumPy* dot operator, except that there is no limit on the number of arguments

Building models efficiently

Avoid explicit loops

- The Xpress Python module facilitates the use of lists, dictionaries, and sets as arguments in most of its methods:
  - this ensures faster execution by avoiding using explicit loops which usually increase model building times
  - this is especially relevant in large optimization models with multiple calls to functions such as `p.addVariable()` and `p.addConstraint()`:

Avoid explicit loops

- The Xpress Python module facilitates the use of lists, dictionaries, and sets as arguments in most of its methods:
  - this ensures faster execution by avoiding using explicit loops which usually increase model building times
  - this is especially relevant in large optimization models with multiple calls to functions such as p.addVariable() and p.addConstraint():
    - consider a loop which makes N calls to `p.addConstraint`:

```
x = [p.addVariable()                      for i in range(N)]
y = [p.addVariable(vartype=xp.binary) for i in range(N)]
for i in range(N):
  p.addConstraint(x[i] <= y[i])
```

    - the external loop can be replaced by a single call to `p.addConstraint` with an inner loop:

```
p.addConstraint(x[i] <= y[i] for i in range(N))
```

FICO

Avoid explicit loops

- The Xpress Python module facilitates the use of lists, dictionaries, and sets as arguments in most of its methods:
  - this ensures faster execution by avoiding using explicit loops which usually increase model building times
  - this is especially relevant in large optimization models with multiple calls to functions such as p.addVariable() and p.addConstraint():
    - consider a loop which makes N calls to `p.addConstraint`:

```
x = [p.addVariable()                    for i in range(N)]
y = [p.addVariable(vartype=xp.binary) for i in range(N)]
for i in range(N):
  p.addConstraint(x[i] <= y[i])
```

    - the external loop can be replaced by a single call to `p.addConstraint` with an inner loop:

```
p.addConstraint(x[i] <= y[i] for i in range(N))
```

    - even more compact and efficient with the use of *NumPy* arrays and p.addVariables():

```
x = p.addVariables(N)
y = p.addVariables(N, vartype=xp.binary)
p.addConstraint(x <= y)
```

Using *loadproblem* for efficiency

- The problem.loadproblem() function provides a low-level interface to the Optimizer libraries:
  - preferable with very large problems and when efficiency in model creation is necessary
  - can be used to create problems with linear/quadratic constraints, a linear/quadratic objective function, and with continuous/discrete variables

Hint: *Check the reference page of the problem.loadproblem() function for detailed information and a full list of arguments*

Using *loadproblem* for efficiency

- The problem.loadproblem() function provides a low-level interface to the Optimizer libraries:
  - preferable with very large problems and when efficiency in model creation is necessary
  - can be used to create problems with linear/quadratic constraints, a linear/quadratic objective function, and with continuous/discrete variables

Hint: *Check the reference page of the problem.loadproblem() function for detailed information and a full list of arguments*

- Consider the following model built using the high-level functions:

```
import xpress as xp
p = xp.problem(name='myexample')
x = p.addVariable(vartype=xp.integer, name='x1', lb=-10, ub=10)
y = p.addVariable(name='x2')
p.setObjective(x**2 + 2*y)
p.addConstraint(x + 3*y <= 4)
p.addConstraint(7*x + 4*y >= 8)
```

Using *loadproblem* for efficiency

- The same problem can be created using problem.loadproblem(), including variable names and their types:

```
p = xp.problem()
p.loadproblem(probname='myexample',
              rowtype=['L', 'G'],      # constraint senses
              rhs=[4, 8],              # right-hand sides
              rng=None,                # no range rows
              objcoef=[0, 2],          # linear obj. coeff.
              start=[0, 2, 4],         # start pos. of all columns
              rowind=[0, 1, 0, 1],     # row index in each column
              rowcoef=[1, 7, 3, 4],    # coefficients
              lb=[-10,0],              # variable lower bounds
              ub=[10,xp.infinity],     #          upper bounds
              objqcol1=[0],            # quadratic obj. terms, column 1
              objqcol2=[0],            #                       column 2
              objqcoef=[2],            #                       coeff
              coltype=['I'],           # variable types
              entind=[0],              # index of integer variable
              colnames=['x1', 'x2']) # variable names
```

Exceptions

Exceptions

- The Xpress Python interface may raise the following exceptions in the event of a modeling, interface, or solver issue:
  - *xp.ModelError*: raised in case of an issue in modelling a problem, for instance if an incorrect constraint sign is given or if a problem is assigned an object that is neither a variable, a constraint, or a SOS
  - *xp.InterfaceError*: raised when the issue can be associated with the way the the API is used, for instance when not passing mandatory arguments or specifying incorrect ones in an API function
  - *xp.SolverError*: raised when the Xpress Solver returns an error that is given by the solver even though the model was specified correctly and the interface functions were used correctly

Exceptions

- The Xpress Python interface may raise the following exceptions in the event of a modeling, interface, or solver issue:
  - *xp.ModelError*: raised in case of an issue in modelling a problem, for instance if an incorrect constraint sign is given or if a problem is assigned an object that is neither a variable, a constraint, or a SOS
  - *xp.InterfaceError*: raised when the issue can be associated with the way the the API is used, for instance when not passing mandatory arguments or specifying incorrect ones in an API function
  - *xp.SolverError*: raised when the Xpress Solver returns an error that is given by the solver even though the model was specified correctly and the interface functions were used correctly
- Use the `try/except` construct in order to analyze the specific exception raised:

```
import xpress as xp
p = xp.problem()
c = makeConstraint() # assume makeConstraint is defined elsewhere
try:
  p.addConstraint(c)
except xp.ModelError as e:
  print ("Modeling error:", repr(e))
```

Advanced topics

Advanced topics

- This part follows from the basics module and covers the following advanced topics:
  - Indicator constraints
  - Special Ordered Set (SOS) constraints
  - Piecewise linear functions
  - General constraints
  - Optimizing with multiple objectives
  - Modeling and solving nonlinear problems
  - Controls and attributes
  - Using callbacks

Advanced topics

- This part follows from the basics module and covers the following advanced topics:
  - Indicator constraints
  - Special Ordered Set (SOS) constraints
  - Piecewise linear functions
  - General constraints
  - Optimizing with multiple objectives
  - Modeling and solving nonlinear problems
  - Controls and attributes
  - Using callbacks
- Course split into modules, where each module comprises:
  - introduction to general concepts about a topic
  - code snippets with examples of application
  - video demonstration of an Xpress Python example using Xpress Workbench
- Other considerations:
  - not exhaustive, not a replacement for the reference manual
  - focuses on areas that are of practical importance
  - assumes the user is familiar with the mathematical optimization concepts involved

Indicator constraints

Indicator constraints

- Indicator constraints are defined by using the problem.addIndicator() method:

```
p.addIndicator(c1, c2, ...)
```

  - an indicator constraint is a logic constraint that expresses the implication ’if indicator condition holds then apply the constraint’:
    - represented by a tuple containing a condition on a binary variable, called the indicator, and an expression representing a constraint: `(indicator condition, constraint)`
  - each argument `c1,c2,...` can be a single indicator constraint, or a list, tuple, or *NumPy* array of indicator constraints (tuples)
  - depending on a user-defined value (0 or 1) for the indicator, the constraint is enforced or relaxed

Indicator constraints

- Indicator constraints are defined by using the problem.addIndicator() method:

```
p.addIndicator(c1, c2, ...)
```

  - an indicator constraint is a logic constraint that expresses the implication 'if indicator condition holds then apply the constraint':
    - represented by a tuple containing a condition on a binary variable, called the indicator, and an expression representing a constraint: `(indicator condition, constraint)`
  - each argument `c1,c2,...` can be a single indicator constraint, or a list, tuple, or *NumPy* array of indicator constraints (tuples)
  - depending on a user-defined value (0 or 1) for the indicator, the constraint is enforced or relaxed
- Example enforcing the constraint `y <= 15` when binary variable `x = 1` for an optimization problem `p`:

```
x = p.addVariable(vartype=xp.binary)
y = p.addVariable(lb=10, ub=20)
ind1 = (x == 1, y <= 15)
p.addIndicator(ind1)
```

  - the p.addIndicator() method also accepts nonlinear expressions for the constraint to enforce

Special Ordered Set (SOS) constraints

FICO

Special Ordered Set (SOS) constraints

- Special Ordered Sets (SOSs) are ordered sets of variables, where only one/two contiguous variables in the set can assume non-zero values:
  - SOS type 1 (SOS1) are a set of variables, of which at most one can take a non-zero value with all others being at zero:
    - they most frequently apply for binary variables where at most one can take the value 1
    - for example, decide the location for a new facility amongst a set of candidate locations

Special Ordered Set (SOS) constraints

- Special Ordered Sets (SOSs) are ordered sets of variables, where only one/two contiguous variables in the set can assume non-zero values:
  - SOS type 1 (SOS1) are a set of variables, of which at most one can take a non-zero value with all others being at zero:
    - they most frequently apply for binary variables where at most one can take the value 1
    - for example, decide the location for a new facility amongst a set of candidate locations
  - SOS type 2 (SOS2) is an ordered set of non-negative variables, of which at most two can be non-zero:
    - if two variables are non-zero, these must be consecutive in their ordering
    - commonly used to model piecewise linear approximations of nonlinear functions

Note: *Special Ordered Sets are used by the Xpress Optimizer to improve the performance of the branch-and-bound algorithm*

Special Ordered Set (SOS) constraints

- The problem.addSOS() function can be used for creating and directly adding Special Ordered Set (SOS) constraints to a problem:

```
problem.addSOS(indices, weights, type, name)
```

  - SOS constraints enforce a small number of consecutive variables in a list to be nonzero
  - where the arguments correspond to:
    - `indices`: list of variables composing the SOS constraint
    - `weights`: list of floating-point weights (one per variable); these define the order for SOS2 constraints, must be sufficiently distinct and and may be used in branching
    - *`type`*: type of the SOS constraint, can be 1 (default) or 2
    - *`name`*: name of the SOS constraint (optional)

FICO

Special Ordered Set (SOS) constraints

- The problem.addSOS() function can be used for creating and directly adding Special Ordered Set (SOS) constraints to a problem:

  ```
  problem.addSOS(indices, weights, type, name)
  ```

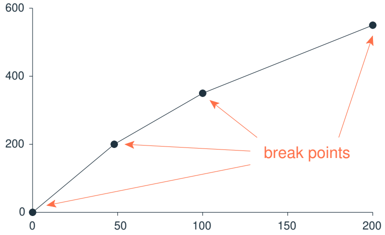
  - SOS constraints enforce a small number of consecutive variables in a list to be nonzero
  - where the arguments correspond to:
    - `indices`: list of variables composing the SOS constraint
    - `weights`: list of floating-point weights (one per variable); these define the order for SOS2 constraints, must be sufficiently distinct and and may be used in branching
    - *`type`*: type of the SOS constraint, can be 1 (default) or 2
    - *`name`*: name of the SOS constraint (optional)
- Examples including Python lists for specifying `indices` and `weights`:

```
N = 20
p = xp.problem()
x = [p.addVariable() for i in range(N)]
s1 = p.addSOS([x[0], x[2]], [4,6])          # SOS type 1 with fixed weights
s2 = p.addSOS(x, [i+2 for i in range(N)], 2) # SOS type 2 with incremental weights
```

Piecewise linear (PWL) functions

Piecewise linear (PWL) functions

- Piecewise linear constraints allow to define variables as piecewise linear functions of other variables:
  - also used to model stepwise functions or to approximate nonlinear functions
  - example for discounts on unit costs depending on the quantity of items bought:



- first 50 items: $COST_1$ = \$4 each
- next 50 items: $COST_2$ = \$3 each
- then, up to 200: $COST_3$ = \$2 each

- quantity break points x_i: 0, 50, 100, 200
- cost break points y_i (= total cost of buying quantity x_i): 0, 200, 350, 550

$$y_i = COST_i \cdot (x_i - x_{i-1}) + y_{i-1} \text{ for } i = 1, 2, 3$$

Piecewise linear (PWL) functions

- Piecewise linear functions can be intuitively added to a problem by using the xp.pwl(`dict`) method in constraints or objectives:
    - receives a dictionary as argument that associates intervals with linear functions:
        - dictionary has tuples of two elements as keys and linear expressions (or constants) as values
        - tuples specify the range of the input variable for which the expression is used as the function value

Piecewise linear (PWL) functions

- Piecewise linear functions can be intuitively added to a problem by using the xp.pwl(`dict`) method in constraints or objectives:
  - receives a dictionary as argument that associates intervals with linear functions:
    - dictionary has tuples of two elements as keys and linear expressions (or constants) as values
    - tuples specify the range of the input variable for which the expression is used as the function value
  - modeling the previous example where `y` is a piecewise linear function of `x`:

```
x = p.addVariable(vartype=xp.integer, ub=200)
y = p.addVariable()
p.addConstraint(xp.pwl({(0, 50): 4*x,
                        (50, 100): 3*(x-50) + 200,
                        (100, 200): 2*(x-100) + 350]}) == y)
```

Note: *The piecewise linear function is always univariate, i.e. there must always be only one input variable*

Piecewise linear (PWL) functions

- Piecewise linear functions can also be used as components of expressions in an optimization problem:

```
cons1 = y + 3*z**2 <= 3*xp.pwl({(0, 1): x + 4, (1, 3): 1})
p.addConstraint(cons1)
```

Piecewise linear (PWL) functions

- Piecewise linear functions can also be used as components of expressions in an optimization problem:

```
cons1 = y + 3*z**2 <= 3*xp.pwl({(0, 1): x + 4, (1, 3): 1})
p.addConstraint(cons1)
```

- Step functions need a further specification if a variable does not appear in the values; in this case we must specify an additional key-value pair as `None: x` for that variable:

```
p.setObjective(xp.pwl({(0, 1): 4, (1, 2): 1, (2, 3): 3, None: x}))
```

FICO

Piecewise linear (PWL) functions

- Piecewise linear functions can also be used as components of expressions in an optimization problem:

```
cons1 = y + 3*z**2 <= 3*xp.pwl({(0, 1): x + 4, (1, 3): 1})
p.addConstraint(cons1)
```

- Step functions need a further specification if a variable does not appear in the values; in this case we must specify an additional key-value pair as `None: x` for that variable:

```
p.setObjective(xp.pwl({(0, 1): 4, (1, 2): 1, (2, 3): 3, None: x})
```

- Discontinuities in the function are allowed, for example:

```
xp.pwl({(1, 2): 2*x + 4, (2, 3): x - 1})
```

  - which is discontinuous at 2, the function value for `x=2` will be either 8 or 1

Note: *Check the Xpress Optimizer reference manual for more information on how to deal with discontinuous functions*

General constraints

General constraints

- General constraints contain the mathematical operators `min`, `max`, `abs` and the logical operators `and`, `or`:
  - an intuitive way to create problems with these operators is by using the Xpress methods (xp.max,xp.min,xp.abs,xp.And,xp.Or) with p.addConstraint():
    - the Xpress Optimizer handles such operators as MIP constraints (if they contain only linear expressions), without having to explicitly introduce extra variables

General constraints

- General constraints contain the mathematical operators `min`, `max`, `abs` and the logical operators `and`, `or`:
  - an intuitive way to create problems with these operators is by using the Xpress methods (xp.max,xp.min,xp.abs,xp.And,xp.Or) with p.addConstraint():
    - the Xpress Optimizer handles such operators as MIP constraints (if they contain only linear expressions), without having to explicitly introduce extra variables
  - examples of use:

```
x = [p.addVariable(vartype=xp.integer, lb=-xp.infinity) for _ in range(3)]
z = [p.addVariable(vartype=xp.binary) for _ in range(3)]
```

    - integer variable `y1` is constrained to be the maximum among the set {`x[0],x[1],46`}:

```
p.addConstraint(y1 == xp.max(x[0], x[1], 46))
```

    - integer variable `y2` must be equal to the absolute value of `x[2]`:

```
p.addConstraint(y2 == xp.abs(x[2]))
```

    - binary variable `y3` is equal to the result of the logical `AND` for the set {`z[0],z[1],z[2]`}:

```
p.addConstraint(y3 == xp.And(z[0], z[1], z[2]))
```

General constraints

- The methods xp.And and xp.Or can be replaced by the corresponding Python binary operators & and |:
  - example for adding constraint `(x[0] AND x[1]) + (x[2] OR x[3]) + 2*x[4] >= 2`:

    ```
    x = [p.addVariable(vartype=xp.binary) for _ in range(5)]
    p.addConstraint((x[0] & x[1]) + (x[2] | x[3]) + 2*x[4] >= 2)
    ```

  - `And` and `Or` have a capital initial as the lower-case correspondents are reserved Python keywords
  - the & and | operators have a lower precedence than arithmetic operators +/− and should hence be used with parentheses

General constraints

- The methods xp.And and xp.Or can be replaced by the corresponding Python binary operators & and |:
  - example for adding constraint `(x[0] AND x[1]) + (x[2] OR x[3]) + 2*x[4] >= 2`:

```
x = [p.addVariable(vartype=xp.binary) for _ in range(5)]
p.addConstraint((x[0] & x[1]) + (x[2] | x[3]) + 2*x[4] >= 2)
```

  - `And` and `Or` have a capital initial as the lower-case correspondents are reserved Python keywords
  - the & and | operators have a lower precedence than arithmetic operators +/− and should hence be used with parentheses

Note: *General constraints must be set up before solving the problem, as they are converted into additional binary variables, indicator or linear constraints during presolve*

Keep in mind: *Using non-binary variables in `(AND, OR)` type constraints, or adding constant values to `(AND, OR, ABS)` type constraints will give an error at solve time*

General constraints

- The problem.addgencons() function allows for adding several general constraints more efficiently:

```
p.addgencons(ctrtype, resultant, colstart, colind, valstart, val)
```

  - `ctrtype`: list or array containing the Xpress types (value) of the general constraints:
    - *xp.gencons_max* (0) and *xp.gencons_min* (1) indicate a maximum/minimum constraint, respectively
    - *xp.gencons_and* (2) and *xp.gencons_or* (3) indicates an and/or constraint
    - *xp.gencons_abs* (4) indicates an absolute value constraint
  - `resultant`: array/list containing the output variables (or indices) of the general constraints
  - `colstart`: array/list containing the start index of each general constraint in the `colind` array
  - `colind`: array/list containing the input variables in all general constraints
  - `valstart`: array/list containing the start index of each general constraint in the `val` array
  - `val`: array/list containing the constant values in all general constraints

Note: *Using p.addgencons() allows for adding several general constraints more efficiently at the expense of modeling convenience and readibility*

General constraints

- Previous example where:
  - variable `y1` is constrained to be the maximum among the set {`x[0],x[1],46`}
  - variable `y2` must be equal to the absolute value of `x[2]`
  - variable `y3` must be the result of the logical `and` for the set {`z[0],z[1],z[2]`}

```
x = [p.addVariable(vartype=xp.integer, lb=-xp.infinity) for _ in range(3)]
z = [p.addVariable(vartype=xp.binary) for _ in range(3)]
y1 = p.addVariable(vartype=xp.integer)
y2 = p.addVariable(vartype=xp.integer)
y3 = p.addVariable(vartype=xp.binary)
type = [xp.gencons_max, xp.gencons_abs, xp.gencons_and]
resultant = [y1, y2, y3]
colstart = [0, 2, 3]
col = [x[0], x[1], x[2], z[0], z[1], z[2]]
valstart = [0,1,1]
val = [46]
p.addgencons(type, resultant, colstart, col, valstart, val)
```

Optimizing with multiple ob-jectives

Optimizing with multiple objectives

- The problem.addObjective() method allows users to add one or more linear objectives for solving multi-objective optimization problems:
  - multiple calls to p.setObjective() are allowed, but each replaces the objective function
  - use p.addObjective(), possibly after an initial call to p.setObjective(), to create additional objectives (existing objectives will remain):

```
p.addObjective(obj1,obj2,...,priority=None,weight=None,abstol=None,reltol=None)
```

Optimizing with multiple objectives

- The problem.addObjective() method allows users to add one or more linear objectives for solving multi-objective optimization problems:
  - multiple calls to p.setObjective() are allowed, but each replaces the objective function
  - use p.addObjective(), possibly after an initial call to p.setObjective(), to create additional objectives (existing objectives will remain):

```
p.addObjective(obj1,obj2,...,priority=None,weight=None,abstol=None,reltol=None)
```

  - with at least one objective expression and a set of *optional* arguments:
    - `obj1,obj2,...`: expression(s) for the objective(s) to be added to the problem
    - *priority*: priority for the new objective(s)
    - *weight*: weight for the new objective(s); negative values inverse the sense of the objective
    - *abstol*: absolute tolerance for the new objective(s)
    - *reltol*: relative tolerance for the new objective(s)

Note: *The sense of the first objective is applied to all objectives. The sense of an objective can be reversed by assigning it a negative weight*

Optimizing with multiple objectives

- Approaches followed by the Optimizer for solving multi-objective problems:
  - Blended (or Archimedian) approach:
    - applied when objectives have equal priority but different weights
    - weighted sum optimization, setting as objective function the linear combination of the added objectives and their weights

Optimizing with multiple objectives

- Approaches followed by the Optimizer for solving multi-objective problems:
  - Blended (or Archimedian) approach:
    - applied when objectives have equal priority but different weights
    - weighted sum optimization, setting as objective function the linear combination of the added objectives and their weights
  - Lexicographic (or preemptive) approach:
    - applied when each objective has a different priority and a unit weight
    - Xpress will solve the problem once for each distinct objective priority that is defined
    - all objectives from previous iterations are fixed to their optimal values within the tolerances:

```
objective <= optimal_value * (1 + reltol) + abstol  # for minimization obj.
objective >= optimal_value * (1 - reltol) - abstol  # for maximization obj.
```

Optimizing with multiple objectives

- Approaches followed by the Optimizer for solving multi-objective problems:
  - Blended (or Archimedian) approach:
    - applied when objectives have equal priority but different weights
    - weighted sum optimization, setting as objective function the linear combination of the added objectives and their weights
  - Lexicographic (or preemptive) approach:
    - applied when each objective has a different priority and a unit weight
    - Xpress will solve the problem once for each distinct objective priority that is defined
    - all objectives from previous iterations are fixed to their optimal values within the tolerances:

```
objective <= optimal_value * (1 + reltol) + abstol   # for minimization obj.
objective >= optimal_value * (1 - reltol) - abstol   # for maximization obj.
```

  - Hybrid approach:
    - applied when objectives have both different priorities and different weights
    - Xpress will solve the problem once for each distinct objective priority defined, optimizing in each iteration a linear combination of the objective functions with the same priority

Optimizing with multiple objectives

- Examples:

```
# Blended (weighted sum) approach with a negative weight
p.addObjective(2*x + y, weight=-0.7) # maximize, higher weight
p.addObjective(y, weight=0.3)        # minimize, lower weight

# Lexicographic approach with setObjective()
p.setObjective(xp.Dot(x, return), sense=xp.maximize, priority=1) # maximize return
p.addObjective(variance, priority=0, weight=-1)                  # minimize risk

# Hybrid approach with three objectives
p.addObjective(xp.Sum(x), priority=1, weight=0.5, reltol=0.1)
p.addObjective(xp.Dot(A,x), priority=1, weight=0.3)
p.addObjective(xp.Dot(B,x), priority=0, weight=-0.2)
```

Hint: *Check the MULTIOBJOPS control to configure the behaviour of the optimizer when solving multi-objective problems*

Modeling nonlinear problems

Modeling nonlinear problems

- Nonlinear problems, i.e. problems containing at least one nonlinear constraint or objective, can be modeled via the Xpress Python interface:
  - nonlinear expressions follow the same relational and arithmetic logic as linear expressions
  - available arithmetic operators: `+`,`−`, `*`, `/`, `**` (which is the Python equivalent for the power operator, "^")
  - univariate functions can be used from the following list: `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `exp`, `log`, `log10`,`abs`, `sign`, and `sqrt`
  - the multivariate functions `min` and `max` can receive an arbitrary number of arguments

Modeling nonlinear problems

- Nonlinear problems, i.e. problems containing at least one nonlinear constraint or objective, can be modeled via the Xpress Python interface:
  - nonlinear expressions follow the same relational and arithmetic logic as linear expressions
  - available arithmetic operators: `+`,`−`, `*`, `/`, `**` (which is the Python equivalent for the power operator, "^")
  - univariate functions can be used from the following list: `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `exp`, `log`, `log10`,`abs`, `sign`, and `sqrt`
  - the multivariate functions `min` and `max` can receive an arbitrary number of arguments
- Examples of nonlinear problem elements:

```
p.addConstraint(x**4 + 2 * x**2 - 5  >= 0)   # polynomial constraint
p.addConstraint(xp.sin(math.pi * x) == 0)   # terrible way to constrain x to be integer
p.addConstraint(x**2 * xp.sign(x) <= 4)     # signum function
p.setObjective((a-x)**2 + b*(y-x**2)**2)    # minimize Rosenbrock function
```

Finding help: *For more information about modeling nonlinear problems, browse the Xpress Nonlinear reference manual*

Modeling nonlinear problems

- A user function enables the creation of an expression that is computed by means of a user-specified function:
  - a user-defined function can be called within a problem by using the function xpress.user():

```
xp.user(f, a1, a2, ...)
```

    - where `f` represents the user-defined function name and `a1, a2, ...` the necessary arguments, as in the example that uses the *math* Python package:

```
def myfunc(v1, v2, v3):
  return v1 / v2 + math.cos(v3)

def mynorm(x1, x2):
  return (math.sqrt(x1**2 + x2**2), 2*x1, 2*x2)

x, y = p.addVariable(), p.addVariable()
p.addConstraint(xp.user(myfunc, x**2, x**3, 1/y) <= 3)
p.setObjective(xp.user(mynorm, x, y))
```

Controls and attributes

Controls and attributes

- The Xpress Python interface enables the user to set controls and query attributes of a problem:
  - a control is a parameter that can influence the behavior (and therefore the performance) of the Xpress Optimizer:
    - for example: the MIP gap target, the feasibility tolerance, or the type of root LP algorithms are controls that can be defined by the user
    - problem controls can both be read from and written to an optimization problem

Controls and attributes

- The Xpress Python interface enables the user to set controls and query attributes of a problem:
  - a control is a parameter that can influence the behavior (and therefore the performance) of the Xpress Optimizer:
    - for example: the MIP gap target, the feasibility tolerance, or the type of root LP algorithms are controls that can be defined by the user
    - problem controls can both be read from and written to an optimization problem
  - an attribute is a feature of an optimization problem, such as the number of rows and columns or the number of quadratic elements in the objective function:
    - they are read-only parameters, i.e. their value cannot be directly modified by the user
    - can be accessed in much the same manner as for the controls

Finding help: *For a full list of controls and attributes, explore the Controls and Attributes chapters of the Xpress Optimizer reference manual*

Accessing problem controls as object members

- Every problem has a problem.controls object that stores the controls related to the problem itself:

```
p.controls.<controlname>                      # read problem control
p.controls.<controlname> = <new value>  # set problem control
```

  - the functions p.getControl() and p.setControl() refer to this object

Accessing problem controls as object members

- Every problem has a problem.controls object that stores the controls related to the problem itself:

```
p.controls.<controlname>                # read problem control
p.controls.<controlname> = <new value>  # set problem control
```

  - the functions p.getControl() and p.setControl() refer to this object
  - examples:

```
print(p.controls.feastol)          # print feasibility tolerance
p.controls.presolve = 0            # disable presolve for this problem
p1.controls.miprelstop = 100 * p2.controls.miprelstop # p1's miprelstop derived from p
```

Note: *Control values are double precision and can be of three types:* *`integer`, `floating point`, `string`*

Heuristic emphasis control

- The problem.controls.heuremphasis control specifies an emphasis for the search w.r.t. primal heuristics and other procedures:

```
p.controls.heuremphasis = 1    # set heuremphasis to 1
p.optimize()
```

  - this control affects the speed of convergence of the primal-dual gap and can be assigned a value:
    - −1: applies the default strategy
    - 0: disables all heuristics
    - 1: focus on reducing the primal-dual gap in the early part of the search
    - 2: applies apply extremely aggressive search heuristics

Heuristic emphasis control

- The problem.controls.heuremphasis control specifies an emphasis for the search w.r.t. primal heuristics and other procedures:

```
p.controls.heuremphasis = 1    # set heuremphasis to 1
p.optimize()
```

  - this control affects the speed of convergence of the primal-dual gap and can be assigned a value:
    - −1: applies the default strategy
    - 0: disables all heuristics
    - 1: focus on reducing the primal-dual gap in the early part of the search
    - 2: applies apply extremely aggressive search heuristics
  - values 1 and 2 trigger many additional heuristic calls, aiming for reducing the gap at the beginning of the search, typically at the expense of an increased time for proving optimality

Finding help: *To learn more about the heuristics applied by the Xpress Optimizer during a MIP solve, explore the Optimizer reference manual*

Optimizer built-in Tuner

- The Optimizer Tuner is a tool intended to automate the process of discovering better control parameter settings:
  - systematically tests the problem against a range of different combinations of control settings
  - can be applied to either a single problem instance or a small collection of problem instances
  - a single tuning run will typically involve solving each problem at least 100-200 times:
    - can therefore become computationally very expensive for large problems

Optimizer built-in Tuner

- The Optimizer Tuner is a tool intended to automate the process of discovering better control parameter settings:
  - systematically tests the problem against a range of different combinations of control settings
  - can be applied to either a single problem instance or a small collection of problem instances
  - a single tuning run will typically involve solving each problem at least 100-200 times:
    - can therefore become computationally very expensive for large problems
  - examples of tuner-related controls and functions:

```
p.controls.tunermaxtime = 100     # set max time spent in tuning
p.controls.tunerthreads = 2       # set no. threads used by the tuner
p.tunerwritemethod('default.xtm') # export tuner options onto an XTM
p.tunerreadmethod('default.xtm')  # read tuner options from a file
p.tune('g')                       # tune the problem as a MIP
p.optimize()     # optimize the problem with best control settings found
```

Finding help: *Check the Xpress Optimizer tuning guide to learn more about the automatic built-in Tuner*

Accessing global controls as object members

- The Xpress module also has a controls object containing all controls of the Xpress Optimizer:
  - a "prompt-friendly" way to read and set controls of the Xpress module is by using the members of xpress.controls:

```
xp.controls.<controlname>                # read control
xp.controls.<controlname> = <new value> # set control
```

    - upon importing the Xpress module, these controls are initialized at their default value
    - when a new problem is created, its controls are copied from the global object

Accessing global controls as object members

- The Xpress module also has a controls object containing all controls of the Xpress Optimizer:
  - a "prompt-friendly" way to read and set controls of the Xpress module is by using the members of xpress.controls:

```
xp.controls.<controlname>                     # read control
xp.controls.<controlname> = <new value> # set control
```

    - upon importing the Xpress module, these controls are initialized at their default value
    - when a new problem is created, its controls are copied from the global object
    - examples:

```
if xp.controls.presolve: ...      # check if presolve is on or off
print(xp.controls.heuremphasis) # print heuristic emphasis control value
xp.controls.feastol = 1e-4        # set feasibility tolerance to 1e-4
```

Note: *Global controls are maintained throughout while the Xpress module is loaded and do not refer to any specific problem*

FICO

Accessing problem attributes as object members

- Every problem has its own attributes object that stores the attributes related to the problem itself:

```
p.attributes.<attributename> # read attribute
```

  - handled by its members the same way as with controls, with two exceptions:
    - there is no "global" attribute object, as a set of attributes only makes sense when associated with a problem
    - an attribute cannot be set, thus it can only be accessed for reading

Accessing problem attributes as object members

- Every problem has its own attributes object that stores the attributes related to the problem itself:

```
p.attributes.<attributename> # read attribute
```

  - handled by its members the same way as with controls, with two exceptions:
    - there is no "global" attribute object, as a set of attributes only makes sense when associated with a problem
    - an attribute cannot be set, thus it can only be accessed for reading
    - examples:

```
print(p.attributes.nodedepth)                # print node depth
number_infeas_sets = p.attributes.numiis     # get irreducible infeasible sets
print("MIPtol:",p.attributes.miprelstop)*100,"%") # print mip tolerance as %
```

Keep in mind: *Attributes are only available after a problem p has been created or read from a file*

Using callbacks

Using callbacks

- The library callbacks are a collection of functions which allow user–defined routines to be specified to the Optimizer:
  - called at various stages during the optimization process, prompting the Optimizer to return to the user's program before continuing with the solution algorithm
  - names of functions for defining callbacks are of the form problem.addcb*()

Using callbacks

- The library callbacks are a collection of functions which allow user–defined routines to be specified to the Optimizer:
  - called at various stages during the optimization process, prompting the Optimizer to return to the user's program before continuing with the solution algorithm
  - names of functions for defining callbacks are of the form problem.addcb*()
- Types of callbacks:
  - *Output callbacks*: called every time a text line is output by the Optimizer
    - the foremost use case, used for logging/reporting via the callback p.addcbmessage()
  - *LP callbacks*: functions associated with the search for an LP solution
    - the functions p.addcblplog() and p.addcbbarlog() allow the user to respond after each iteration of either the simplex or barrier algorithms, respectively
  - *MIP tree search callbacks*: called at various points of the MIP tree search process
    - for example, when a MIP solution is found at a node of the Branch-and-Bound, the Optimizer will call a routine set by p.addcbpreintsol() before saving the new solution

Finding help: *Check the Xpress Optimizer callbacks reference webpage to learn more about the most used callbacks*

Using callbacks

- Steps for using callbacks:
  1. define a callback function (say `myfunction`) that is to be run at certain points in time (i.e. every time the BB reaches a specific point)

     ```
     def myfunction(prob, data, ...):
      # user-defined routine here...
     ```

  2. call the corresponding problem.addcb*() method with `myfunction` as its argument

     ```
     p.addcbpreintsol(myfunction, data)  # assume data defined elsewhere
     ```

  3. run the p.optimize() command that launches the appropriate solver

Using callbacks

- Steps for using callbacks:
  1. define a callback function (say `myfunction`) that is to be run at certain points in time (i.e. every time the BB reaches a specific point)

     ```
     def myfunction(prob, data, ...):
      # user-defined routine here...
     ```

  2. call the corresponding problem.addcb*() method with `myfunction` as its argument

     ```
     p.addcbpreintsol(myfunction, data)   # assume data defined elsewhere
     ```

  3. run the p.optimize() command that launches the appropriate solver
- A callback function is passed once as an argument and used possibly many times while a solver is running, and receives:
  - a `problem object` declared with `p = xp.problem()`
  - a user-defined `data object` to read and/or modify information within the callback

Note: *The callbacks in the Python interface reflect as closely as possible the design of the callback functions in the C API*

Using callbacks

- Any call to a problem.addcb*() function adds that function to a list of callback functions for that specific point of the BB algorithm:

```
p.addcbpreintsol(preint1, data, 3)
p.addcbpreintsol(preint2, data, 5)
```

  - the two functions will be put in a list and called (`preint2` first since it has a higher priority) whenever the BB algorithm finds an integer solution

Using callbacks

- Any call to a problem.addcb*() function adds that function to a list of callback functions for that specific point of the BB algorithm:

```
p.addcbpreintsol(preint1, data, 3)
p.addcbpreintsol(preint2, data, 5)
```

  - the two functions will be put in a list and called (`preint2` first since it has a higher priority) whenever the BB algorithm finds an integer solution
- To remove a callback function, use the problem.removecb*() method:

```
p.removecb*(function,data)
```

  - deletes all elements of the list of callbacks that were added with the corresponding addcb* function that match the function and the data, for example problem.removecbpreintsol()
  - the `None` keyword acts as a wildcard that matches any function or data object:
    - if `None` is passed as the callback function, then all callbacks matching the `data` argument will be deleted
    - if data is also `None`, all callback functions of that type are deleted, this can also be obtained by passing no argument to p.removecb*()

Using callbacks

- Example for a callback function named `preintsolcb` that is called every time a new integer solution is found via the p.addcbpreintsol() method:

```
import xpress as xp

def preintsolcb(prob, data, soltype, cutoff):
 # callback to be used when an integer solution is found defined here
 ...
 return (reject, newcutoff)  # assume 'reject' and 'newcutoff' defined meanwhile

p = xp.problem()
p.read('myprob.lp')  # reads in a problem, let's say a MIP

p.addcbpreintsol(preintsolcb, data)  # assume 'data' defined elsewhere
p.optimize()
```

Note: *While the `function` argument is necessary for all p.addcb*() functions, the `data` object can be specified as `None`. In that case, the callback will be run with `None` as its data argument*

Thank You

www.fico.com/optimization