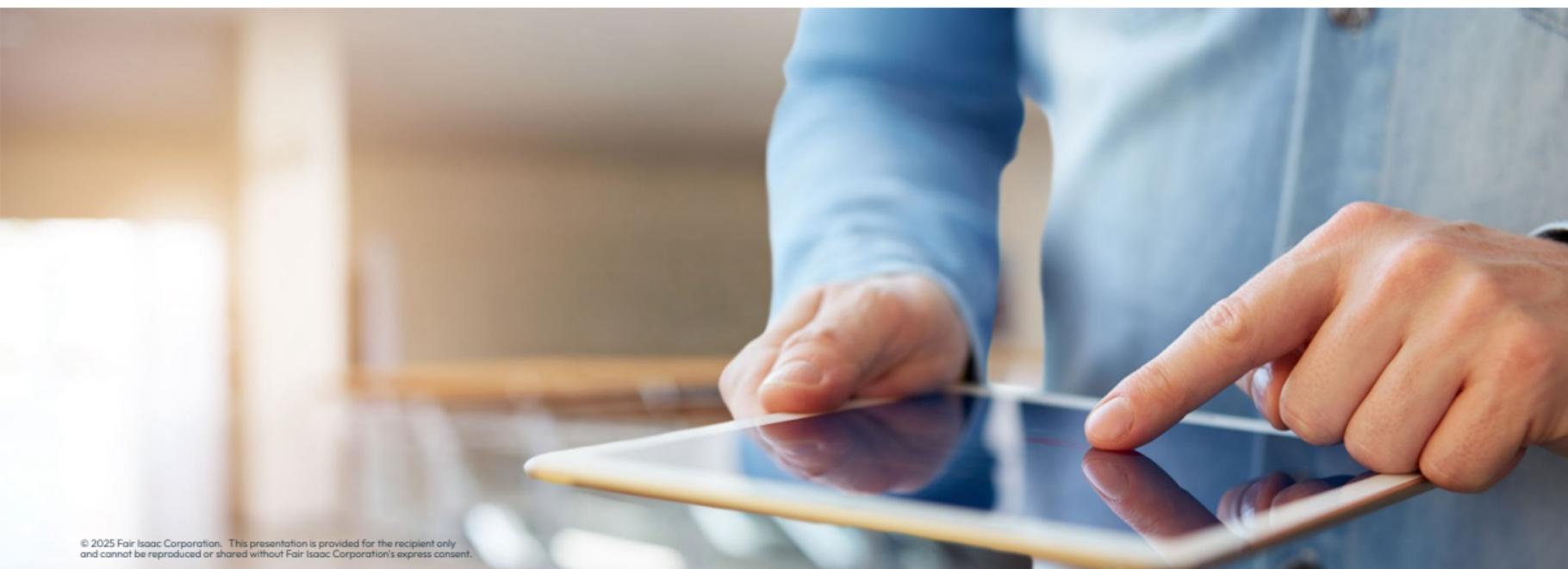




FICO® Xpress Optimizer - Python Interface

Xpress Optimization Training





Introduction to the course

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 Jupyter Notebooks
- 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 notebook examples and run them using Visual Studio Code
- 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



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 of size up to 5000 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 locate 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 3.9 to 3.13 are available, each package contains:
 - Xpress Solver libraries
 - Python interface module
 - Documentation in PDF format
 - Various examples of use
 - A copy of the community license (see <https://www.fico.com/fico-xpress-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 3.9 to 3.13, 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:
 - *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

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:
 - *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
 - *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 using `xpress.problem()`:

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

- A name can be assigned to a problem upon creation using the `name` argument:

```
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)
```

- 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`:
- 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/set, 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/set, 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 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 \leq , $=$, \geq :

```
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 `xpress.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:
 - Passing the body, type and rhs arguments instead of the constraint object:
 - Can be particularly useful to define range constraints by passing the type as `xp.rng` and lb, ub:

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

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

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

- This will add the range constraint $0 \leq x1 + 2*x2 \leq 3$

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 FICO® 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 FICO® 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 `problem.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 problem
```



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.getSlacks()`, 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
problem

Reading a problem

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

```
p.readProb(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.readProb()` method, which takes the file name as its argument:

```
p.readProb(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.readProb("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.writeProb()` method:

```
p.writeProb(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.writeProb("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 NumPy arrays of variables and creates the set of constraints $x[i] \leq y[i]$ 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 range(S)], dtype=xp.npvar)
y = np.array([p.addVariable(vartype=xp.binary) for i in range(S)], dtype=xp.npvar)
constr1 = x <= y
p.addConstraint(constr1)
```

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

Using NumPy arrays

- NumPy operations can be replicated on each element of an array, leveraging its *vectorization* and *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 $x[i] + y[i] \leq 1$ for all i in the set S :

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

Using NumPy arrays

- NumPy operations can be replicated on each element of an array, leveraging its *vectorization* and *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 $x[i] + y[i] \leq 1$ for all i in the set S :
- 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)
```

- Yields a 4×2 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 in
Python

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))
```

Using NumPy multidimensional arrays

- The `p.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 multidimensional arrays with one or more 0-based indices

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)
```

- Yields a 4×2 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

Use *SciPy* sparse arrays

- Sparse data is a data set where most elements have a value zero:
 - Can be an array like [1, 0, 2, 0, 0, 3, 0, 0, 0, 0, 0]
 - Sparse array formats allow building models more efficiently by avoiding iterating over all the elements (including the zeros) of a conventional array
- The *SciPy* package has a module, `scipy.sparse` that provides functions to deal with sparse data

Use *SciPy* sparse arrays

- Sparse data is a data set where most elements have a value zero:
 - Can be an array like [1, 0, 2, 0, 0, 3, 0, 0, 0, 0, 0]
 - Sparse array formats allow building models more efficiently by avoiding iterating over all the elements (including the zeros) of a conventional array
- The *SciPy* package has a module, `scipy.sparse` that provides functions to deal with sparse data
- The `xp.Dot()` operator supports the most common *SciPy* sparse matrix formats, allowing arrays of sparse expressions and constraints to be constructed efficiently:
 - Can compute the product of a 1-D NumPy array of variables or expressions with a sparse matrix of numbers in CSR or CSC format

```
import numpy as np
from scipy.sparse import csr_matrix

orig_array = np.array([1, 0, 2, 0, 0, 3, 0, 0, 0, 0, 0])      # sparse np array
scipy_array = csr_matrix(orig_array)          # convert to scipy sparse array form
p.addConstraint(xp.Dot(scipy_array, var) <= rhs)        # use with xp.Dot
```

Using the low-level API functions

- The `problem.load*`() functions provide a low-level interface to the FICO® Xpress Optimizer libraries:
 - Functions: `p.loadLP`, `p.loadMIP`, `p.loadQP`, `p.loadMIQP`, `p.loadQCQP`, `p.loadMIQCQP`
 - 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

Using the low-level API functions

- The `problem.load*`() functions provide a low-level interface to the FICO® Xpress Optimizer libraries:
 - Functions: `p.loadLP`, `p.loadMIP`, `p.loadQP`, `p.loadMIQP`, `p.loadQCQP`, `p.loadMIQCQP`
 - 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
- Consider the following MIQP 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)
```



Hint: Check other low-level API functions such as `problem.addrows()`, `problem.addcols()`, and `problem.addqmatrix()`

Using the low-level API functions

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

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



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)
- The constraint is only enforced when the value of the indicator variable matches a user-defined value (0 or 1)

Indicator constraints

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

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

- Example enforcing the constraint $y \leq 15$ 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)
```



Note: The `addIndicator()` method also accepts nonlinear expressions for the constraint to enforce



Special Ordered Set (SOS) constraints

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 FICO® 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 may be used in branching
 - `type`: type of the SOS constraint, can be 1 (default) or 2
 - `name`: name of the SOS constraint (optional)

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 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`:

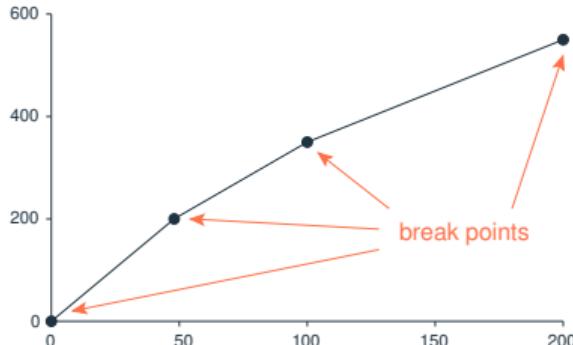
```
N = 20
p = xp.problem()
x = [p.addVariable() for i in range(N)]
s1 = p.addSOS([x[0], x[2]], [4,6])           # SOS type 1 with fixed weights
s2 = p.addSOS(x, [i+2 for i in range(N)], 2) # SOS type 2 with incremental weights
```



Piecewise linear (PWL) functions

Piecewise linear (PWL) functions

- Piecewise linear constraints define a variable as a **piecewise linear function of another variable**:
 - 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}))
```

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 *FICO® 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.addVariables(3, vartype=xp.integer, lb=-xp.infinity)
z = [p.addVariables(3,vartype=xp.binary)]
```

- 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 $(x[0] \text{ AND } x[1]) + (x[2] \text{ OR } x[3]) + 2*x[4] \geq 2$:

```
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.GenConsType.MAX` (0) and `xp.GenConsType.MIN` (1) indicate a maximum/minimum constraint, respectively
 - `xp.GenConsType.AND` (2) and `xp.GenConsType.OR` (3) indicates an and/or constraint
 - `xp.GenConsType.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 readability

General constraints

- Previous example where:
 - Variable y_1 is constrained to be the maximum among the set $\{x[0], x[1], 46\}$
 - Variable y_2 must be equal to the absolute value of $x[2]$
 - Variable y_3 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.GenConsType.MAX, xp.GenConsType.ABS, xp.GenConsType.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
objectives

Optimizing for different objectives sequentially

- The `problem.setObjective()` method allows users to add several `linear` objectives for solving a problem for different objectives sequentially:
 - Multiple calls to `p.setObjective()` are allowed
 - The user must define the `objidx` argument (with different integer values) to indicate the multi-objective context and the sequence of objectives to consider
 - The model is run for each objective sequentially, thus runs are independent of each other
 - The sense of the first objective (`objidx=0`) defines the default optimization sense for all objectives:
 - To reverse the optimization sense for secondary objectives, set the `weight` attribute to `-1`

Optimizing for different objectives sequentially

- The `problem.setObjective()` method allows users to add several `linear` objectives for solving a problem for different objectives sequentially:
 - Multiple calls to `p.setObjective()` are allowed
 - The user must define the `objidx` argument (with different integer values) to indicate the multi-objective context and the sequence of objectives to consider
 - The model is run for each objective sequentially, thus runs are independent of each other
 - The sense of the first objective (`objidx=0`) defines the default optimization sense for all objectives:
 - To reverse the optimization sense for secondary objectives, set the `weight` attribute to `-1`

```
p.setObjective(x1, objidx=0)          # minimize first objective  
p.setObjective(x2, objidx=1, weight=-1) # maximize second objective  
p.setObjective(...)                  # other objectives  
  
p.optimize()
```

- The Optimizer will print the logs for each sequential run and, in the end, a summary of the objective values found for each run:
 - This can be useful to assess the maximum possible value for each objective

Optimizing with multiple objectives

- The `problem.addObjective()` method allows users to add one or more linear objectives for solving **multi-objective optimization** problems:
 - Use `p.addObjective()`, possibly after an initial call to `p.setObjective()`, to create additional objectives (existing objectives will remain in the same problem):

```
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:
 - Use `p.addObjective()`, possibly after an initial call to `p.setObjective()`, to create additional objectives (existing objectives will remain in the same problem):

```
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 invert 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 in Python

- 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 in Python

- 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 [FICO® Xpress NonLinear reference manual](#)

User functions

- A **user function** enables the creation of an expression that is computed through external code:

- Any 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 below:

```
def myfunc(v1, v2, data):  
    model = MLmodel(v1, v2, data)      # MLmodel() defined elsewhere  
    return model.results  
  
data = readData()    # readData() defined elsewhere  
x, y = p.addVariable(), p.addVariable()  
p.setObjective(xp.user(myfunc, x, y, data))
```

- You can define user functions with a **simulation or machine learning** model!
 - Be aware of losses in **determinism** and performance
- User functions are not supported by FICO® Xpress Global



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 FICO® 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 FICO® 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 FICO® 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 = 10 * p2.controls.miprelstop # p1's miprelstop derived from p2
```



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 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 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 FICO® Xpress Optimizer during a MIP solve, explore [the reference manual](#)

Optimizer built-in Tuner

- The FICO® Xpress 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 FICO® Xpress 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

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 FICO® Xpress 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.add*Callback()**

Using callbacks

- The library **callbacks** are a collection of functions which allow **user-defined routines** to be specified to the FICO® Xpress 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.add*Callback()`
- 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.addMessageCallback()`
 - *LP callbacks*: functions associated with the search for an LP solution
 - The functions `p.addLplogCallback()` and `p.addBarlogCallback()` 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.addPreIntsolCallback()` 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.add*Callback()` method with `myfunction` as its argument

```
p.addPreIntsolCallback(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.add*Callback()` method with `myfunction` as its argument

```
p.addPreIntsolCallback(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.add*Callback()` function adds that function to a list of callback functions for that specific point of the BB algorithm:

```
p.addPreIntsolCallback(preint1, data, 3)  
p.addPreIntsolCallback(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.add*Callback()` function adds that function to a list of callback functions for that specific point of the BB algorithm:

```
p.addPreIntsolCallback(preint1, data, 3)  
p.addPreIntsolCallback(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 corresponding `problem.remove*Callback()` method:

```
p.remove*Callback(function, data)
```

- Deletes all elements of the list of callbacks that were added with the corresponding `add*Callback()` function that match the function and the data, for example `problem.removePreIntsolCallback()`
- 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.remove*Callback()`

Using callbacks

- Example for a callback function named `preintsolcb` that is called every time a new integer solution is found via the `p.addPreIntsolCallback()` 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.addPreIntsolCallback(preintsolcb, data)  # assume 'data' defined elsewhere
p.optimize()
```



Note: While the function argument is necessary for all `p.add*Callback()` 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