The following is a very rough concept of how to pass arguments into and out of a function:

**Step 1:** define the function by name:

1 def myFunctionName():

2 pass

In line 1 we use the “def” keyword to indicate that we are defining a function. We then follow it up with the function name (in this case it is myFunctionName). Then we add parenthesis as a placeholder for arguments of the function. The parenthesis must be there even if there are no arguments. Finally, we add a colon to indicate to the compiler that everything that follows is part of the function definition.

In line 2 we use the “pass” keyword as a temporary place holder for real commands. Python must have at least one command for each user defined function.

**Step 2**: add arguments to be passed in:

1 def myFunctionName(arg1, arg2):

2 pass

Notice we have updated line 1. In this example we will pass in two arguments. The first one is called arg1, the second one is called arg2

**Step 3**: add some logic to the code. Let’s assume we want this function to pass out the product of the two arguments and the result of raising arg1 to the power of arg2. First just add the logic described above:

1 def myFunctionName(arg1, arg2):

2 prod = arg1 \* arg2

3 exp = arg1 \*\* arg2

Notice that we have removed the “pass” keyword because we now have several commands in this user defined function.

In line 2 we assign the result of the two arguments multiplied together to a variable called “prod”.

In line 3 we assign the result of raising arg1 to the power of arg2.

The problem here is that all this hard work is unavailable to the calling routine. We have correctly defined the function, passed in two arguments, and written logic that correctly determines what we wanted. But due to variable scope, the variables “prod” and “exp” were declared in our small function and therefore that is where they reside. The variables and their values are not accessible outside of “myFunctionName”. As a result, if we were to type the following into python:

1 def myFunctionName(arg1, arg2):

2 prod = arg1 \* arg2

3 exp = arg1 \*\* arg2

4 a,b = myFunctionName(2,3)

5 print a, b

The result would be

>>> none, none

because nothing was assigned to a, b. So let’s fix it.

**Step 4**: return the values from the function

1 def myFunctionName(arg1, arg2):

2 prod = arg1 \* arg2

3 exp = arg1 \*\* arg2

4 return prod, exp

In line 4 we use the “return” keyword to indicate that once we reach this line, we are to terminate the function, return control to the calling routine, and return whatever follows back out to the calling function. In this case we are returning the product and the exponentiation. Now if wrote the following code:

1 def myFunctionName(arg1, arg2):

2 prod = arg1 \* arg2

3 exp = arg1 \*\* arg2

4 return prod, exp

5 a,b = myFunctionName(2,3)

6 print a, b

The result would be:

>>> 6, 8

You are allowed to pass out as many things as you want and they can be anything you want as long as they can be contained in a variable or advanced data structure.

Let’s extend this function concept to Gurobi’s library. Let’s say we want to write a function that creates a minimization model and provides it a name. We need to do the following:

1. Create the definition line
2. Give the function a meaningful name
3. Pass in the name of the model and its “sense”
4. Create the model
5. Pass the model out

Let’s do the first and second tasks together:

1 def createGurobiModel():

2 pass

Let’s do the third task

1 def createGurobiModel(modelName, sense):

2 pass

Let’s do the fourth task:

1 def createGurobiModel(modelName, sense):

2 myModel = Model(modelName)

3 myModel.modelSense = sense

4 myModel.update()

Finally, let’s do the fifth task:

1 def createGurobiModel(modelName, sense):

2 myModel = Model(modelName)

3 myModel.modelSense = sense

4 myModel.update()

5 return myModel

Now let’s call it in a routine and see what the entire module would look like:

1. from gurobipy import \*
2. def createGurobiModel(modelName, sense):
3. myModel = Model(modelName)
4. myModel.modelSense = sense
5. myModel.update()
6. return myModel
8. def main():
9. newModel = createGurobiModel(‘Model by Function Call’, GRB.MAXIMIZE)
10. if \_\_name\_\_ == ‘\_\_main\_\_’:
11. main()

In line 1 we call the Gurobi module at the global level.

In lines 3 through 7 we define the model building routine we developed earlier.

In lines 9 through 10 we define the main module (no arguments this time) and we call the createGurobiModule and “capture” the return argument with a variable called “newModel”.

In lines 13 and 14, it’s just the standard lines we see in Python that allows us to run our modules when they are run as the main module.

Now let’s investigate how to use this “functional” approach to make variables for our model. Assume this is a farming problem and we have 4 types of seeds we want to plant and we have determined our variables represent the amount of acres allocated to each crop. Furthermore, let’s limit the variables to be integer values. The table below provides us the information we need to make this model. Assume we are maximizing profit.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Seed | Wheat | Corn | Soy | Barley |
| Profit/Acre | $150 | $120 | $80 | $90 |

Let’s create the index of the seeds and the dictionary of the profit and we’ll do that in a function called getData. To do this we need to do the following:

1. Create the definition line
2. Give the function a meaningful name
3. There are no arguments for this model, so the parenthesis will be empty
4. Create the list of seeds
5. Create the dictionary of profits
6. Pass the two data structures out

Let’s do tasks 1, 2, and 3:

def getData():

pass

Now let’s do tasks 4 and 5:

def getData()

seeds = [‘wheat’, ‘corn’, ‘soy’, ‘barley’]

profit = {‘wheat’: $150, ‘corn’: $120, ‘soy’: $80, ‘barley’: $90}

Now let’s do task 6:

def getData()

seeds = [‘wheat’, ‘corn’, ‘soy’, ‘barley’]

profit = {‘wheat’: $150, ‘corn’: $120, ‘soy’: $80, ‘barley’: $90}

return seeds, profit

Notice how we are now passing out two arguments instead of one. This means when we call it, we will need to “capture” two arguments. Let’s continue adding to our code. Everything in blue is old code, everything in red is new code.

1. from gurobipy import \*
2. def createGurobiModel(modelName, sense):
3. myModel = Model(modelName)
4. myModel.modelSense = sense
5. myModel.update()
6. return myModel
8. def getData()
9. seeds = [‘wheat’, ‘corn’, ‘soy’, ‘barley’]
10. profit = {‘wheat’: $150, ‘corn’: $120, ‘soy’: $80, ‘barley’: $90}
11. return seeds, profit
13. def main():
14. newModel = createGurobiModel(‘Model by Function Call’, GRB.MAXIMIZE)
15. seedStock, revenue = getData()
16. if \_\_name\_\_ == ‘main’:
17. main()

Now let’s build our variables. We need to do the following:

1. define the function
2. give it a meaningful name
3. pass in the arguments – in this case it’s the model (because that is what we are adding the variables to), the seeds list (because that will be the index for our variables), and the profit dictionary (we need this for our objective function)
4. create the variables
5. pass out the necessary data elements (more on this later)

Let’s do the tasks 1, 2, and 3:

1. def makeVariables(myModel, seeds, profit):
2. pass

Now let’s do task 4:

1. def makeVariables(myModel, seeds, profit):
2. mySeeds = {}
3. for s in seeds:
4. mySeeds[s] = myModel.addVar(vtype = GRB.INTEGER, obj = profit[s], name = s)
5. myModel.update()

In line 2 we create the empty dictionary of our variables.

In line 3 we set up to iterate over our variables indexes.

In line 4 we create the variables. Note that we specify the variable type (integer), we give it a name, and we assign it its objective function value. This is one easy way to build your objective function. More specifically, it’s done implicitly.

In line 5 we update the model. Note how the model is updated AFTER all the variables have been created. If we had a very large model with lots of variables, it would take a very long time to update the model each time we add the variable. Instead we update after all of these variables have been added.

Now let’s do task 5 (pass out the variables). A couple of things to note: We don’t change “seed” and “profit”. They are the same when we are done using them as they were when we passed them in, so there is no sense in passing them out. The variable “myModel”, however, has changed. We have added variables to it. We have also created a new argument called mySeeds (which is the dictionary that contains the handle to each variable). We want access to that dictionary once we leave the function, so we need to pass it out.

1. def makeVariables(myModel, seeds, profit):
2. mySeeds = {}
3. for s in seeds:
4. mySeeds[s] = myModel.addVar(vtype = GRB.INTEGER, obj = profit[s], name = s)
5. myModel.update()
6. return myModel, mySeeds

Note there are two arguments being passed out so we have to “capture” them when we call the function. Let’s look at the code with this new function included. Again, all old code in blue, new code in red:

1. from gurobipy import \*
2. def createGurobiModel(modelName, sense):
3. myModel = Model(modelName)
4. myModel.modelSense = sense
5. myModel.update()
6. return myModel
8. def getData()
9. seeds = [‘wheat’, ‘corn’, ‘soy’, ‘barley’]
10. profit = {‘wheat’: $150, ‘corn’: $120, ‘soy’: $80, ‘barley’: $90}
11. return seeds, profit
12. def makeVariables(myModel, seeds, profit):
13. mySeeds = {}
14. for s in seeds:
15. mySeeds[s] = myModel.addVar(vtype = GRB.INTEGER, obj = profit[s], name = s)
16. myModel.update()
17. return myModel, mySeeds
18. def main():
19. newModel = createGurobiModel(‘Model by Function Call’, GRB.MAXIMIZE)
20. seedStock, revenue = getData()
21. newModel, myVars = makeVariables(seedStock, revenue)
22. if \_\_name\_\_ == ‘main’:
23. main()

Notice how in line 24 we “capture” the two arguments from the function we just created.

Now let’s add three constraints:

1. limit the number of total acres to 150;
2. limit the amount of labor available to plant to 40 hours, and
3. limit the amount of labor to harvest to 60 hours.

The table below provides the values (notice in this instance we only have one family of constraints – but we could add a new function for each family and call it in much the same way):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Resource | Wheat | Corn | Soy | Barley | Limit |
| Acres | 1 | 1 | 1 | 1 | 150 |
| Planting hours/acre | 2 | 1.5 | 1 | .75 | 40 |
| Harvest hours/acre | 1 | 1 | .75 | .25 | 60 |

First thing we have to do is update our getData function so it creates a dictionary with our terms in it. I will make a dictionary within a dictionary just so we can get fancy in the quicksum function when we get to it. We need to add a resources dictionary and a limits dictionary. The blue code is the original code, the red code is the new code added.

1. def getData()
2. seeds = [‘wheat’, ‘corn’, ‘soy’, ‘barley’]
3. profit = {‘wheat’: $150, ‘corn’: $120, ‘soy’: $80, ‘barley’: $90}
4. resources = {‘acres’:{ ‘wheat’: 1, ‘corn’: 1, ‘soy’: 1, ‘barley’: 1},
5. {‘plant’:{ ‘wheat’: 2, ‘corn’: 1.5, ‘soy’: 1, ‘barley’: .75},
6. {‘harvest’:{ ‘wheat’: 1, ‘corn’: 1, ‘soy’: .75, ‘barley’: .25}}
7. limits = {‘acres’: 150, ‘plant’: 40, ‘harvest’:60}
8. return seeds, profit, resources, limits

Notice how we have increased the number of arguments being passed out of our function. We will need to update the calling routine as well. Let’s look at our full set of code now. As always, blue is old code, red is new code.

1. from gurobipy import \*
2. def createGurobiModel(modelName, sense):
3. myModel = Model(modelName)
4. myModel.modelSense = sense
5. myModel.update()
6. return myModel
8. def getData()
9. seeds = [‘wheat’, ‘corn’, ‘soy’, ‘barley’]
10. resources = {‘acres’:{ ‘wheat’: 1, ‘corn’: 1, ‘soy’: 1, ‘barley’: 1},
11. {‘plant’:{ ‘wheat’: 2, ‘corn’: 1.5, ‘soy’: 1, ‘barley’: .75},
12. {‘harvest’:{ ‘wheat’: 1, ‘corn’: 1, ‘soy’: .75, ‘barley’: .25}}
13. limits = {‘acres’: 150, ‘plant’: 40, ‘harvest’:60}
14. return seeds, profit, resources, limits
15. def makeVariables(myModel, seeds, profit):
16. mySeeds = {}
17. for s in seeds:
18. mySeeds[s] = myModel.addVar(vtype = GRB.INTEGER, obj = profit[s], name = s)
19. myModel.update()
20. return myModel, mySeeds
21. def main():
22. newModel = createGurobiModel(‘Model by Function Call’, GRB.MAXIMIZE)
23. seedStock, revenue, resources, limits = getData()
24. newModel, myVars = makeVariables(seedStock, revenue)
25. if \_\_name\_\_ == ‘main’:
26. main()

In line 26 you see where we have increased the number of “capturing” variables for our function call.

Now let’s create a function that adds our constraints. We will need to do the following:

1. define the function
2. give it a descriptive name
3. pass in the arguments. We need to pass in the model since that is what the constraints will be added to, we need to pass in the variables since they are used in the constraints, we need to pass in the resource dictionary (it holds our constraint coefficients), seeds list (so we can iterate over the variables in the quicksum function), and the limits dictionary so we have a right hand side for our constraints.
4. build the constraints
5. pass out the necessary arguments (more on this later).

Let’s do tasks 1, 2, and 3:

1. def makeConstraints(myModel, myVars, seeds, resources, limits):
2. pass

Now let’s do task 4:

1. def makeConstraints(myModel, myVars, seeds, resources, limits):
2. myConstr = {}
3. for r in resources:
4. constrName = str(r) + ‘\_limits’
5. myConstr[constrName] = myModel.addConstr(quicksum(resources[r][s]\*  
    myVars[s]   
    for s in seeds) <= limits[r],   
    name = constrName)
6. myModel.update()

In line 1 we declare the function and pass in arguments.

In line 2 we create a dictionary that will “capture” the constraints so we can manipulate them later if we want (just like we did with the variables).

In line 3 we set up an iterator so we can create the three constraints that make up the family of constraints we are creating.

In line 4 we do a little bit of nice book keeping. We build a string that will be a unique name identifier for the constraint. We then use that string everywhere we want to use the constraint name.

In line 5 we actually build the constraint. We create a dictionary element that has the constraint name as its key and we assign to it the constraint. Remember that “resources” is a double dictionary so I have to use the double bracket notation to get to each of the elements I want. This is a little different than how we build the NFL dictionary. I did it this way for convenience. “resources[r]” gets us the element in the dictionary that is specific to the resource constraint while “resources[r][s]” gets us the coefficient that is specific to index value “s”. We multiply the constraint by the variable and iterate over all s in seeds. Finally, we apply the resource limitation that is specific to resource[r] and we assign a name to the constraint.

In line 6, we update the model AFTER all constraints have been made.

Now let’s do task 5. Again, we didn’t change any of the coefficients and constants (seeds, resources, and limits). What went in are the same things that would come out, so we don’t need to pass them out. We didn’t change the variables either, so there is no need to pass out the variable argument. The model has changed so we need to pass that out and we created the myConstr dictionary which we may want to use later. So we need to pass that out as well. The code for this function now looks as follows:

1. def makeConstraints(myModel, myVars, seeds, resources, limits):
2. myConstr = {}
3. for r in resources:
4. constrName = str(r) + ‘\_limits’
5. myConstr[constrName] = myModel.addConstr(quicksum(resources[r][s]\*  
    myVars[s]   
    for s in seeds) <= limits[r],   
    name = constrName)
6. myModel.update()
7. return myModel, myConstr

There are two arguments coming out, so the calling routine has to have two variables to “capture” the arguments. Let’s look at our updated code now. Again, all old code in blue, all new code in red.

1. from gurobipy import \*
2. def createGurobiModel(modelName, sense):
3. myModel = Model(modelName)
4. myModel.modelSense = sense
5. myModel.update()
6. return myModel
8. def getData()
9. seeds = [‘wheat’, ‘corn’, ‘soy’, ‘barley’]
10. resources = {‘acres’:{ ‘wheat’: 1, ‘corn’: 1, ‘soy’: 1, ‘barley’: 1},
11. {‘plant’:{ ‘wheat’: 2, ‘corn’: 1.5, ‘soy’: 1, ‘barley’: .75},
12. {‘harvest’:{ ‘wheat’: 1, ‘corn’: 1, ‘soy’: .75, ‘barley’: .25}}
13. limits = {‘acres’: 150, ‘plant’: 40, ‘harvest’:60}
14. return seeds, profit, resources, limits
15. def makeVariables(myModel, seeds, profit):
16. mySeeds = {}
17. for s in seeds:
18. mySeeds[s] = myModel.addVar(vtype = GRB.INTEGER, obj = profit[s], name = s)
19. myModel.update()
20. return myModel, mySeeds
22. def makeConstraints(myModel, myVars, seeds, resources, limits):
23. myConstr = {}
24. for r in resources:
25. constrName = str(r) + ‘\_limits’
26. myConstr[constrName] = myModel.addConstr(quicksum(resources[r][s]\*  
     myVars[s]   
     for s in seeds) <= limits[r],   
     name = constrName)
27. myModel.update()
28. return myModel, myConstr
30. def main():
31. newModel = createGurobiModel(‘Model by Function Call’, GRB.MAXIMIZE)
32. seedStock, revenue, resources, limits = getData()
33. newModel, myVars = makeVariables(seedStock, revenue)
34. newModel, myConstr = makeConstraints(newModel, myVars, seedStock, resources, limits)
36. if \_\_name\_\_ == ‘main’:
37. main()

Notice in line 36 our calling line accepts two returned arguments; one for the model and the other for the constraints.

Finally, we want to solve our model and output our results to the screen. We’ll just solve it in the main function. Since we have seen this before, I will not detail it. If you have questions, please let me know. Our completed code looks like the following (again blue code is old, red code is new)

1. from gurobipy import \*
2. def createGurobiModel(modelName, sense):
3. myModel = Model(modelName)
4. myModel.modelSense = sense
5. myModel.update()
6. return myModel
8. def getData()
9. seeds = [‘wheat’, ‘corn’, ‘soy’, ‘barley’]
10. resources = {‘acres’:{ ‘wheat’: 1, ‘corn’: 1, ‘soy’: 1, ‘barley’: 1},
11. {‘plant’:{ ‘wheat’: 2, ‘corn’: 1.5, ‘soy’: 1, ‘barley’: .75},
12. {‘harvest’:{ ‘wheat’: 1, ‘corn’: 1, ‘soy’: .75, ‘barley’: .25}}
13. limits = {‘acres’: 150, ‘plant’: 40, ‘harvest’:60}
14. return seeds, profit, resources, limits
15. def makeVariables(myModel, seeds, profit):
16. mySeeds = {}
17. for s in seeds:
18. mySeeds[s] = myModel.addVar(vtype = GRB.INTEGER, obj = profit[s], name = s)
19. myModel.update()
20. return myModel, mySeeds
22. def makeConstraints(myModel, myVars, seeds, resources, limits):
23. myConstr = {}
24. for r in resources:
25. constrName = str(r) + ‘\_limits’
26. myConstr[constrName] = myModel.addConstr(quicksum(resources[r][s]\*  
     myVars[s]   
     for s in seeds) <= limits[r],   
     name = constrName)
27. myModel.update()
28. return myModel, myConstr
30. def main():
31. newModel = createGurobiModel(‘Model by Function Call’, GRB.MAXIMIZE)
32. seedStock, revenue, resources, limits = getData()
33. newModel, myVars = makeVariables(seedStock, revenue)
34. newModel, myConstr = makeConstraints(newModel, myVars, seedStock, resources, limits)
35. newModel.optimize()
37. for var in myVars:
38. if myVars[var].x > 0:
39. print var, myVars[var].x
40. if \_\_name\_\_ == ‘main’:
41. main()

Disclaimer: None of this code has gone through an actual python interpreter (i.e., it was all created in Word without the benefit of compiling). This means there may be some small compiling errors in the code. In general, however, the code and concepts presented are correct. Let me know if there are issues.