Parametric Lab

Thursday, September 17th, 8:00am - 5:10pm
Phil Ringsmuth, philr@sds2.com

Lab Outline
1. An Introduction to Parametrics
2. Some Python Basics
3. Basic Programming Terminology and Concepts
4. Interacting with Python via IDLE
5. Introduction to Parametric Modeling
6. Parametric Phase One: Parametric Modeling
7. Parametric Phase Two: Editing Code
Part One - An Introduction to Parametrics

A parametric is a program, sometimes called a script or macro, that can be run inside an SDS/2 model to accomplish any number of things. Parametrics have the ability to add members and material, as well as bolts, holes and welds. They can also be used to read and modify custom property information and status information.

Parametrics do not add “smart” objects to a model. Any objects added by a parametric such as members or materials are no different to the SDS/2 model than had you added them yourself. This differs from things like custom members and components. Custom members, once added, can be edited as an entire entity, and components exist as part of a member and can be edited via a member edit window.

One of the biggest advantages of using parametrics is that they can adapt themselves to members and framing situations in the model. This has a major advantage over assemblies. Consider the situation of a small “box” cap plate for an HSS column. If assemblies are used, one assembly must be created and saved for every applicable column used in the model. If a new HSS shape is added for which a properly sized assembly hasn't yet been created, one would have to be created and saved by hand. A single parametric can easily read the size and shape of any HSS column and design a box cap plate to fit the situation.

A well designed parametric can feel like a complete add-on to SDS/2, containing dialog boxes, user interaction, even report generation and email. Though the main focus of many parametrics is to save time by automating a tedious, repetitious processes in the model, they certainly are not limited to those types of situations. There are few limits to what you can accomplish with a well written parametric.
Part Two - Some Python Basics

Python is a programming language, much like C++, Java, Javascript, Perl and many others that you may have heard of. Each of these languages has its own set of strengths and weaknesses, and each one is designed with a particular set of goals or uses in mind. Python is in a class of languages commonly referred to as scripting languages. It can accomplish much on its own, but its real power lies in its ability to interact with other languages. In this way, Python can be thought of as a “glue” between parts of a software project where separate parts are written in different languages.

Python code, by design, tends to be easier to read than code in many other languages. It tries to keep the use of symbols such as braces {} to a minimum, making the code easier for us to read. To accomplish this readability, Python uses indentation as a way of separating blocks of code. In many languages where braces {} are used to identify blocks of code, indentation is used for readability but not enforced by the language. Python enforces an indentation structure, effectively forcing code to be easier to read.

Python, like many scripting languages, is interpreted instead of compiled. This means that the computer will read and execute Python code one line at a time, from the top of a file to the bottom. This forces objects to be declared before they can be used.

Many Python programs, especially parametrics, begin with a block of import statements. A statement such as import math tells Python to load the contents of a module called math, which contains dozens of common math-related operations. Many of the SDS/2 tools that are used when writing parametrics are defined in modules that must be imported into your code before they can be used. Operations in the math module can then be used as follows: math.sqrt().

When looking at parametric source code, you may see two other variants of the import statement. Programmers often use from module import object or from module import *. The first option imports a variable or function directly, so from math import sqrt would let the sqrt() function be used directly, instead of having to type math. in front of it. The * option imports all objects defined in a module so they can be used directly by name. You will see some of these variants in our parametric startup code.

Depending on the complexity of the program, the import statements (if any) may be followed by a few lines of code and nothing more. On the other hand, some programs have hundreds or even thousands of lines of code defining classes, methods, variables, loops and other logic. As programs grow, it’s important to keep structure and readability in mind.

Programs that get overly complex are often broken into several files and imported as needed. Parametrics written in Python can be as simple as a few lines of code, or complex to the point of needing several files, images and other resources. It’s entirely up to the programmer and the scope of the project.
Part Three - Basic Programming Terminology and Concepts

Variables
For a program to accomplish much of anything, we must create variables. Variables are chunks of memory to which we assign a particular name and can hold just about any value we want. Oftentimes we will use variables to hold the value of some counter, or to store which member or material a user has chosen from the model. Variables can be used to represent just about anything, from simple numbers or words, to menus and even entire dialog windows, or more abstract concepts such as an instance of modeling.

Conditions
Many programs perform operations based on some decision made by the user or by some surrounding circumstances. For example, the user may be given an option to choose the right end or left end of a member to perform some operation. The program will behave differently based on the result of this choice. Python uses the keywords if, elif and else to make these decisions.

Loops
Programs often need to repeat some operation many times. This operation could be quite simple, or consist of hundreds of lines of code. Rather than writing the same code over and over, we can use the concept of loops or looping. Loops allow a program to repeat a section of code multiple times. Imagine adding holes along the web of a beam. Instead of writing the code to add a hole repeatedly, we can write code that essentially does this: “Add a hole here, move down two inches and add another. Keep going until we run out of web.” There are two kinds of loops used in Python: the for loop and the while loop.

Functions
A function (also sometimes called a method) is a chunk of code that can be called (or invoked) to run whenever it is needed. A function can be thought of as a mini program that might be given some input, do some tasks, and possibly return some output. Imagine that adding a hole takes ten lines of code. Instead of writing these ten lines repeatedly whenever our parametric needs to add a hole, we can create a function that takes as input a material and hole coordinates and adds the hole for us. It might look something like this: def addHole(material, location):. Note that the code making up the body of this function is not shown. After creating this function, we can use it in our code, providing a valid material and location: addHole(somePlate, someCoordinate).

Data Type: String
Strings are a type of variable used to hold text. A string is a sequence of zero or more characters, enclosed in quotation marks ".". Any time a letter, word or phrase is shown to the user or input is taken from from the user, that data is likely stored in a string variable in the program. Strings can be as short as holding no characters at all to holding the text of an entire book, depending on the needs of the program. Parametrics typically use strings to hold section sizes, such as "W8x35".
**Data Type: Number**

There are two distinct types of numbers in most programming environments: integers and floating-point numbers. Integers are whole numbers that are either positive, negative or zero. Integers can be thought of as the elementary counting numbers. Numbers with a decimal point are called floating-point numbers (or floats, for short). In a parametric, for example, integers would be used to count how many members a user has selected, or how many plates are being added. Floats would be used whenever a dimension is involved, because dimensions are always stored in decimal inches or decimal millimeters.

**Container: List**

Most programming languages have some means of storing many things together in some kind of a collection or a container. In Python, a list is a *modifiable* sequence of zero or more items contained in brackets []. The list itself is given a name just like any other variable, and individual elements within the list can be accessed by their index, starting with zero for the first element. For example, given the list `piecemarks = ["B_1", "C_2", "C_7"]`, the piecemark for the beam could be accessed in this manner: `piecemarks[0]`. Because lists are modifiable (also called mutable), items can be added, removed and reordered.

**Container: Tuple**

A tuple is nearly identical to a list except for one fundamental difference: tuples cannot be modified once they have been created. They are *immutable*. A tuple can be created and assigned to a variable, but after that it is essentially a read-only list. Tuples are useful for storing a group of items that will never change, such as the options in a dialog menu. Visually, tuples can be distinguished from lists in that they are enclosed in parentheses () instead of brackets [].

**Mathematical Operations**

Python uses the four standard mathematical operators for addition +, subtraction -, multiplication * and division /. When more than one operator is present in a statement, Python operates on them with standard precedence: multiplication and division followed by addition and subtraction. When precedence is not a consideration, operations are performed left-to-right. It is strongly recommended that you write your code using parentheses to separate multiple operations that are on the same line. For example, the expression `x + y / b - a` is not clear and may not be interpreted by Python as the programmer intended. Enclosing pairs of operators in parentheses makes this easier to read and interpret: `(x + y) / (b - a)`.

**Boolean Operations**

Your Python code will use boolean operations to compare two values. Python support six standard comparison operators: less than <, greater than >, less than or equal to <=, greater than or equal to >=, equal to == and not equal to !=.

**Comments**

Nearly all programming languages have a way to leave comments in the code. Comments are pieces of text meant to be informative to a person reading the code but ignored by the computer. Comments can add clarity to code that might be confusing, or simply point out what a particular bit of code does. They can be used as notes to the programmer to help remind them why they did something a certain way, or things that still need to be worked on. In Python, a comment begins with a pound sign # and extends to the end of the line. It is very common to “comment out” portions of code that are being worked on, especially while debugging a program.
Part Four - Interacting with Python via IDLE

Because Python is an interpreted language, we can test out our code as we’re writing it instead of waiting for it to be compiled into something the computer can understand. When Python is installed on a computer (which happens automatically with an SDS/2 installation), it comes with a simple interactive editor called IDLE. Using IDLE allows us to test out our code interactively so we can get an understanding of how things work and test out ideas. In this section of the lab, we will spend some time interacting with Python using IDLE in order to get a better understanding of some of the concepts we have already discussed, before we begin to build our actual parametric.

Open Python IDLE and let’s try typing in some commands. IDLE will evaluate whatever you type and then respond with however it evaluated what you typed. If there are any errors detected in your code, the operation will likely fail and IDLE will report some kind of error message with a traceback (sometimes called backtrace). A traceback shows the last few steps that were taken by Python before the error was encountered, which can be helpful when you are debugging code.

Whenever you see the >>> symbol, this is the Python prompt. It means that Python is waiting for you to enter some code for it to evaluate. Press enter to send the code to Python to be evaluated. Let’s try some of this together.

```
>>> 4
4
>>> 4 * 3
12
>>> x = 17
>>> x
17
>>> x + 3
20
>>> x + y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
```

These are some pretty basic examples just to get your feet wet. Now let’s take a look at some of the other concepts we discussed earlier. In the code above, we defined a variable called \texttt{x} and assigned it the value \texttt{17}. Whenever we use \texttt{x} in our code, it is essentially replaced with whatever value it holds. Of course, being a \textit{variable}, we can always change \texttt{x} to be something else, another number or even another type of data altogether.
>>> x = 20
>>> x
20
>>> x = "Steve"
>>> x
'Steve'
>>> x = [1, 1, 2, 3, 5, 8]
>>> x
[1, 1, 2, 3, 5, 8]

Let's take a look at some of the other programming concepts we've discussed and see how they work in Python and IDLE. We will do this by making a simple “guess the number” game where the user will try to guess a number between 1 and 100, selected randomly by the computer. We’ll modify the game as we examine more topics.

In IDLE, choose File > New File. Save this file to your desktop, giving it the name game.py. This is the file we will use to create our program, and we will use IDLE to run the program. Creating our simple game program in this manner will help us get comfortable with the concepts of editing code.

Enter the code below into your new game.py file. You do not need to enter the comment lines, they are shown simply to explain what each of the parts of our code is doing. After you’ve entered the code, we will discuss what the program will do.

```python
# import a module used for random numbers
import random

# generate a random number between 1 and 100,
# and store it in the variable called target
target = random.randint(1, 100)

# print the random number, for debugging (or cheating?)
print(target)

# ask the user for a number,
# and store it in the variable called guess
guess = input("Guess a number between 1 and 100: ")

# convert guess from a string to an integer
guess = int(guess)

# check to see if the user guessed the correct number,
# and print "You win!" if they did, otherwise print "You lose!"
if guess == target:
    print("You win!")
else:
    print("You lose!")
```

comments, modules, variables, functions, data types, conditions, indentation
Save the file and run it by choosing Run > Run Module. If you get an error, we will discuss it and debug the problem together. Don’t feel bad if this happens to you, it’s a part of everyday programming, even for people who have been writing code for decades.

We have just written a very rudimentary “game” in which the computer picks a number and the player has one chance to guess it. If they guess correctly, they win, otherwise they lose. The game could be improved if it were to give the player more than one chance. Let’s make some changes that will give the user ten chances to guess the number. In the following code boxes, only new code and comments are colored, existing code is in grey.

```python
import random

target = random.randrange(100)
print target

# create a boolean variable called win, set it to False
win = False

# the function range(1, 11) returns a list, starting with 1, up to, but not
# including 11. This is equivalent to the following line of code:
# for turn in [1,2,3,4,5,6,7,8,9,10]

# each time through the for loop, turn becomes the next
# element in the list automatically.
for turn in range(1, 11):
    # print the current turn
    print str(turn) + " of 10"
    guess = raw_input("Guess a number between 1 and 100: ")
    guess = int(guess)
    if guess == target:
        # set win to True, because the player won the game
        win = True
        # use break to forcibly leave the loop
        break
    # if win is still False, the player lost the game
    if win:
        print "You win!"
    else:
        print "You lose!"
```

**boolean values, for loop, break statement, nested indentation**

This code demonstrates a few new concepts. First, we declare a variable called `win` and set it to `False`. We use this at the end of the program to see if the player has won the game or not. The big new concept here is the `for` loop. We use the `range()` function to generate a list of numbers from 1 to 10, and use a `for` loop to iterate through that list. Each time through the loop, the value of `turn` takes on the next value in the list of numbers.
The next change we will make is to modify the program to use a `while` loop instead of a `for` loop. The basic idea is the same: repeat the same code more than once. The fundamental difference is that a `for` loop iterates through a set of data, and a `while` loop repeats until some condition is false. Let's make the following changes to our code so that it uses a `while` loop instead.

```python
import random

target = random.randint(100)
print target

win = False

# create a variable called turn, to store the turn number
turn = 1

# repeat the loop body as long as turn is less than or equal to 10
while turn <= 10:
    print str(turn) + " of 10"
    guess = raw_input("Guess a number between 1 and 100: ")
    guess = int(guess)
    if guess == target:
        win = True
        break
    break

# increase (or increment) the value of turn by one
turn = turn + 1  # alternatively: turn += 1

if win:
    print "You win!"
else:
    print "You lose!"
```

The key difference here is that we repeat the body of the loop as long as its condition is met. The condition here is that `turn` must be less than or equal to 10. As long as this is true, the loop will continue to repeat. One common mistake with `while` loops is forgetting to change the variable involved in the condition. If we accidentally left out the line `turn = turn + 1`, then the value of `turn` would never change (it would always be 1) and the game would never end until the player guesses the correct value.
Part Five - Introduction to Parametric Modeling

Parametric modeling is a version of modeling that is used for designing and experimenting with creating parametrics or parts of parametrics. Any actions done in the parametric modeling environment do not affect the actual job model. You can think of parametric modeling as a sandbox or playground version of your main model, in which it is okay to experiment, knowing that the actual model will not be affected.

When you first open parametric modeling, you are asked to select which members you wish to keep. The purpose of this step is to eliminate parts of the model that you won't need, keeping only what is required to create your parametric. For our project, our main model consists of six columns and seven beams, but we will only be keeping a single beam.

The parametric modeling interface looks very similar to the standard modeling interface. One of the biggest differences to note is that what was the model tree is now the feature tree. The feature tree sits along the right-hand side of the screen and lists out individual features of your parametric as you create them.

Another notable difference is that parametric modeling, much like drawing editor, does not automatically save your work. Keep in mind that you're not creating a model, you're using the model to create a program. You will want to be sure to save your parametric at various points to make sure you have a working copy on disk.

Another common operation used in parametric modeling is testing your current project. This lets you run the parametric in its current state right inside of parametric modeling. This is useful for testing individual features as they are added or edited.

The final thing to point out about parametric modeling is that it allows you to record your actions and save them into the parametric code. In our project, we will be adding some stiffener plates to a beam. Instead of writing code by hand to accomplish this, we will use the record feature to let parametric modeling generate the code for us. Recording can be turned on and off as needed.
Part Six - Parametric Phase One: Parametric Modeling

We will now begin the process of building our parametric for this lab: a parametric for adding stiffener plates to a beam. This is a relatively simple parametric that will ask the user to select a beam. It will then show a dialog box prompting the user for a few pieces of information about the stiffener plates to be added. Finally, using the information from the dialog box, the plates will be added to the beam.

From the Main Menu, Miscellaneous Options, select Parametric Modeling.

Select Roof (12-0) from the list of erection views.

At the Select member(s) to keep prompt, choose beam B_7. Right-click and choose OK.

The feature tree along the right side of the screen is currently empty. Right-click and choose Add Functions -> ClearSelection(). A dialog box called Python Code Edit appears with the text `ClearSelection()` already filled out. This is the code we need to add, so go ahead and press OK.

This action adds two items to the feature tree. The first feature, Startup Code, is added automatically by parametric modeling as soon as you add your first feature. Startup code contains imports and other initialization code that SDS/2 will need in order to run most parametrics. In general, startup code should not be modified.

The second feature is the one we added, `ClearSelection()`. This calls a function which tells SDS/2 to clear any members from the current selection. We do this so that we can make sure no members are selected when our parametric starts running.

In the feature tree, right-click and add another function, this time choose MemberLocate... Another dialog for adding a line of Python Code appears, this time starting with the line of code

```
var = MemberLocate("Status Line")
```

This should look familiar to you from some of the code we saw while creating our game. Here are some similar lines of code that we have already seen:

```
target = random.randint(100)
guess = raw_input("Guess a number between 1 and 100: ")
guess = int(guess)
```
This line of code, `var = MemberLocate("Status Line")` will create a variable called `var`. It also calls a function called `MemberLocate()`, which will display the text "Status Line" in the status line while prompting the user to select a member. Once a member has been selected, it will be assigned to `var`.

Edit the code so that it looks like this: `beam = MemberLocate("Select Beam")`. When run, this will display the text "Select Beam" in the status line, and store the selected member to a variable called `beam`.

Upon hitting OK on the dialog, notice that parametric modeling actually runs the bit of code we just created. The status line says Select Beam, and SDS/2 is prompting you to select an item. Select B_7 with a single click, and the new code will be added to the feature tree.

Save your parametric with the name `StiffenerPlates.py`. We noted earlier that parametric modeling does not automatically save your work like the standard modeling interface does. It's best to save your work often, and even to keep multiple copies of your project as you go along. It's quite likely that, in the process of developing and adding features to a parametric, you will create bugs in the code that you can't figure out how to fix. Rather than having to start from scratch, it's better to be able to start from your “before lunch” copy, the one that was working earlier.

Be sure that the feature `beam = MemberLocate("Select Beam")` is highlighted in the feature tree. Right-click and add another `ClearSelection()` feature. It's important to do this with the `beam =` feature selected, because the feature tree always adds new features below the one that is currently selected.

Save your project again. At this point, we should be able to test our parametric to see if it's working properly. Choose File > Test to test the current project. If it works as expected, it should run the startup code (which we will never “see” happen), clear the selection, and prompt the user to select a beam. Clicking on the beam will store it in a variable called `beam` (again, something we can't actually see happen) and the selection will be cleared again.

We will now add a dialog box to our project. Dialog boxes can be fairly complex, and to create them by hand (by writing the code directly) takes a deep understanding of the SDS/2 dialog modules for Python. Luckily for us, parametric modeling has a tool that we can use to create dialog boxes in a visual manner, without writing any code directly.

Right-click in the feature tree, choose Add Functions, Dialog box... A window called Parametric Dialog Edit will be shown. This is the tool we use to create dialog boxes instead of writing them by hand in code. We will go through this window and point out the fields that will be used.

One important note before we create our dialog box: When using the Parametric Dialog Edit window, try to avoid pressing the Enter key on your keyboard when you’ve finished typing in a field. For many users it's almost an instinct to press Enter to finish an action, but doing so activates the OK button which will close out the dialog edit window. Instead of Enter, press the Tab key to move from one field to the next. This will make sure you don't accidentally exit your dialog editing before you are finished. Let’s go ahead and begin.
At the top is a section called General Information. Here we fill in the title of the dialog box, the Return dictionary name (not used in this lab) and the Dialog name. The Dialog name field is the variable name we will use in our code to access the dialog box. We will want to use the checkbox called Show Preview, which will show a preview of the dialog box as it is being built.

The main portion of the screen is a section called Dialog box entries. This will show a tree-like structure of our dialog’s fields. Every object created in a dialog box using this tool is considered a field. Fields can be nested, meaning fields can contain other fields. This is the case when a dialog box contains more than one tab, or multiple columns of information. Our dialog box will contain only four fields, none of which will be nested.

Below the list of entries are buttons for moving, cutting, copying, pasting and deleting fields. These can be used to rearrange fields within the dialog. Just below that is a dropdown menu called Field type. This is where we begin adding fields, by choosing the type to be added, clicking the Add button, then editing the field’s details.
In the Field type dropdown, choose “Menu” and click Add. This adds a generic menu to our dialog. Make sure the Show Preview checkbox is enabled so that you can see the menu appear. The bottom portion of the window, called Edit entry, now becomes active. We use these options to edit the properties of the selected field. Notice that our new field, Menu, is highlighted in blue automatically to indicate it is the feature being edited.

The Dictionary item name option lets us choose a variable name for this menu. Much like earlier, when we declared the variable beam to represent the selected beam, we want to choose a name that will represent this menu. The default is menuName, which is not very useful. Change this to side, because we will be using this menu to choose a side of the beam for adding plates.

Next is the List option. We use this to create a list (technically, a tuple) of strings that will be shown on the menu. Notice the default contains a tuple listing three generic strings, "Item 1", "Item 2" and "Item 3". Edit this tuple to contain these options: "N/S", "F/S", and "Both".

The next option, Default value, sets the default value for the menu. Notice again that this is a string, and it must match one of the options from the list we created in the previous step. Set the Default value field to "N/S".

Finally we have the Prompt string option. This is the text that shows up next to the drop down menu. We want to make sure to choose something which represents what the menu options contain, so let’s choose "Beam Side".

Verify that your dialog preview looks like the image above. Be aware that this preview is only a visual tool, and we cannot interact with it. Also notice that wherever we filled out a string value, it is shown on the preview with surrounding double quotes. This is only a side effect of the dialog preview, and won’t be present when the dialog is actually presented to the user.
Make sure that the Menu entry (the only entry) is highlighted in the list. Doing so will ensure our next entry appears below the menu in the dialog. In the Field type pulldown, choose Dimension and click Add. For the Dictionary item name, use `thickness`. The default value should be a string: "3/8". When we use this value in our code, we will convert it to a decimal. The Prompt string should read "Plate Thickness", and the Field type should be set to Dimension. The details of this field and the updated dialog preview are shown below:
Add two more fields with the following properties:

**Field type: Integer**

<table>
<thead>
<tr>
<th>Dictionary item name: <code>plateCount</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value: 3 (no quotes, this is a strict integer, not a string)</td>
</tr>
<tr>
<td>Prompt string: &quot;Number of Plates&quot;</td>
</tr>
<tr>
<td>Field type: Integer</td>
</tr>
</tbody>
</table>

**Field Type: Dimension**

<table>
<thead>
<tr>
<th>Dictionary item name: <code>endDistance</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Value: &quot;1-0&quot;</td>
</tr>
<tr>
<td>Prompt string: &quot;Distance to First Plate&quot;</td>
</tr>
<tr>
<td>Field type: Dimension</td>
</tr>
</tbody>
</table>
This will complete the entries for the dialog box. Before we continue, verify that your dialog box preview window and the dialog edit window match the following images:
Once you've verified that everything matches, click OK on the Parametric Dialog Edit window. It is important not to click OK until you're done, because doing so immediately activates the dialog box and will add it to your code. Interact with the new dialog box to make sure all options are there and work as expected. Press OK on this dialog and it will be added to the feature tree. Save your parametric.

At this point it's a good idea to test the parametric before we proceed any further. Run the Test command to test the script. It should prompt you to select a beam, then show the dialog. After pressing OK on the dialog, the program will end. If this all works without any errors, then we have successfully tested our parametric. If there are any issues, now is the time to fix them together.

The last step we need to do in Parametric Modeling is record the addition of a single plate. This will be used to generate some code to add a plate, code that we will edit later to adapt itself to the beam that the user selected and the dialog options they entered.

Use the Snap to Surface command to snap to the left end of the beam. Use Relative Depth to change the view depth by a value of -12. This moves the view plane into the beam one foot. Finally, adjust the depth check to go in and out 3 inches in each direction. This isolates a cross section of the beam with no other material so that we can focus on adding the plate.

Use Construction Line Add Material to add material construction lines to the beam. Use the Ruler tool to measure the distance from the face of the web to the tip of the flange. We will need to remember this value for the next step when we add our plate. It should be 1 3/8 inches, or 1.8875.

For the next step, we will be using the Record tool. This tool will "record" actions taken in the model and save them as code into the feature tree. Only physical additions, subtractions or alterations to the model will be recorded. Actions like zooming and panning, changing views, etc. will be ignored. Time is also ignored, so the amount of time between actions while recording will not matter, only the actions themselves are saved.

Make sure that the Dialog feature is highlighted in the feature tree before continuing. Select File > Record to turn on recording. Note that the only way to see if Record is enabled is to look for the checkmark next to the option in the File pulldown menu. With Record turned on, choose Model > Material > Add > Rectangular plate. Just like in modeling, you are prompted to select a member to add material to. Select the beam as you normally would.

Choose the two points for inputting the rectangular plate. The first point is on the inside edge of the top flange, and the second point is on the inside edge of the bottom flange:
We are now presented with a parametric version of the Rectangular Plate Material window which you are probably familiar with from modeling. The parametric version of this window (and likewise, most parametric modeling windows in which something is added to the model) is somewhat different from the standard modeling version. Notably, a new section called Recorded input is present at the bottom of the window. We will make a few changes to some of these fields before continuing.

Set the Material width to be 1.8875. This is the value we measured earlier. Change the Thickness reference point dropdown to be Center. Turn off the Prompt checkbox in the Recorded input area, and change its value to beam. Disabling the Prompt checkbox stops the parametric from asking the user to select a member to add this material to. Because our project begins with the user selecting a beam, which we store as a variable called beam, we can simply put that here and the plate will automatically be added to the beam.

Before clicking OK, be sure your Parametric Rectangular Plate Material window looks like the one shown above. If you see any differences, be sure to ask so we can fix them or discuss together what might be going on. Once you've ensured everything is okay, press OK and the plate will be added.
Just like in Modeling, the Rotate Material window will appear. We don’t need to make any changes here, but do make note of the X-axis rotation of 90 degrees. We will see this later when we edit our code. Press OK. Finally, we are prompted to locate the material’s dimension reference point. Choose the same point we used for the first point of the plate, the inside of the top flange.

You should now see your plate in the model. You will also see a new Add rectangular plate feature listed below the dialog in the feature tree. Turn off the Record feature and save your project.

Now is a great time to begin testing and iteratively making small changes to the project. To test the parametric, we first need to erase the plate we added from the model. Note that we are not erasing the plate from the feature tree, so the addition of the plate is still included in our code. Test the project to make sure everything works.

You’ll probably notice that the plate intersects with the beam in two places. The last thing we will do in Parametric Modeling before editing our code directly is putting in a temporary fix for this. Double click on the Add Rectangular Plate feature to bring up the Parametric Rectangular Plate Material edit window. For each of the left and right ends, change the Bottom Operation from None to Clip, and set the length and width to 1 and 1. Click OK to save these changes to the feature.

Notice that the plate did not change. This is because we weren’t editing the plate, we were editing the feature responsible for adding the plate. To see these changes in action, we need to run the parametric again. Delete the plate from the model and test the parametric. You should now see the plate being added with one inch clips on each of the inside corners.

At this point, we are finished using Parametric Modeling and we will continue working on our project by editing the code directly. Close Parametric Modeling, making sure to save any changes that you may not have saved up to this point. Before we start editing code, we will run our parametric in the actual model.
Open Modeling and choose Model > Parametric > Run... Browse to where your file is saved and choose it. When prompted to select a beam, choose B_11 from the center of the structure. Click OK on the dialog box and the plate will be added, but it almost certainly won't be what you might be expecting to see.

It's important to keep in mind that SDS/2 is going to run the code contained in a parametric exactly as it is written; it won't do anything else, even if you think it should. You might be thinking that this plate will adapt to the beam, but we need to edit the code ourselves to make that happen, which is what we will be doing in the final section of the lab.

Choose Model > Parametric > Erase. Select the parametric from the list and erase it from the model. This will get rid of all the material added by that particular parametric, so the plate we were just examining will be deleted from the model.
Part Seven - Parametric Phase Two: Editing Code

For the rest of this lab, we will be editing our code and testing it in the SDS/2 model repeatedly. To best accomplish this, we will open Sublime Text 2 (the text editor we will use to edit code) on one of our monitors and have the SDS/2 model open on the second screen. This makes it much easier to edit and test our code simultaneously. Open your parametric file using Sublime Text 2 and drag it to your secondary monitor. Keep SDS/2 open full screen on your main monitor.

The first time you look at the code for your parametric it will surely seem overwhelming, but parametric modeling does a good job of breaking the code into distinct sections. The code we have is arranged approximately as follows:

<table>
<thead>
<tr>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Startup Code: Generated automatically, and should not be altered.</td>
</tr>
<tr>
<td>Beam Selection: Includes two <code>ClearSelection()</code> function calls.</td>
</tr>
<tr>
<td>Dialog Imports: Located below the beam selection, in the order in which it was created.</td>
</tr>
<tr>
<td>Dialog Box: Create the dialog box and run it with the call to <code>dialog.done()</code>.</td>
</tr>
<tr>
<td>Rectangular Plate: Creates a plate object, sets its attributes, adds it and rotates it.</td>
</tr>
</tbody>
</table>

We want to modify our code to accomplish several things. We need the plate to size itself based on the size of the beam, we don’t want the values for the size of the plate set directly in the code. When a value is used directly in code in a situation like this, it is commonly referred to as **hardcoded**, because it is typed right in the code and cannot be changed. We also need the plates that are added to match the information entered in the dialog box.

What follows is an outline of what our modified code will look like:

<table>
<thead>
<tr>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Startup Code</td>
</tr>
<tr>
<td>Dialog Imports</td>
</tr>
<tr>
<td>Beam Selection</td>
</tr>
<tr>
<td>Dialog Box</td>
</tr>
<tr>
<td>Variable Creation</td>
</tr>
<tr>
<td>Function: Add Single Plate</td>
</tr>
<tr>
<td>Function: Add Plates to One Side</td>
</tr>
<tr>
<td>Call Appropriate Functions for Near and/or Far Sides</td>
</tr>
</tbody>
</table>

Move the dialog box import code from its current location to just below the `# startup code end` comment. This helps to keep all of the startup and other import code at the top of the file before any functional code is encountered.

The code for selecting the beam should follow:

```python
ClearSelection()
beam = MemberLocate("Select Beam")
ClearSelection()
```
We will now create several variables that will be used later on in the program. Some of these are simplified names for variables contained in the dialog box, the rest are values that are calculated based on other information. Each of these new variables is discussed below. When writing your own parametrics, consider creating a variable whenever there is a small bit of information you need to reuse, or a formula that you don’t want to calculate or type multiple times. Try to use good variable names that tend to describe what they are. We’ve seen some examples of this already when we changed **var to beam** and **menu_name to side**.

**beamLength**

We need to know the length of the beam when we calculate where our plates will be placed. We already have access to the beam itself via the **beam** variable, but member objects don’t have any data or functions that tell us directly what their length is. Because of this, we will calculate the distance between the beam ends using a method called **Distance()**. This method is part of a location object, which is part of a member end object, which is part of the beam. The **Distance()** method takes one object, a location, and returns the straight-line distance between the two points.

```
beamLength = beam.LeftEnd.Location.Distance(beam.RightEnd.Location)
```

**clipLength**

When we modeled our plate, we entered a value of **1.0** for the length of the corner clips. This value is now **hardcoded** into our script, which we don’t want. Instead we want to dynamically calculate this clip length based on the profile of the beam. The image below shows many of the variables that can be obtained from a wide flange member. Our clip length needs to be slightly larger than the k-radius. To calculate this, we subtract the flange thickness from the beam’s k-value, then add ¼ inch for clearance.

```
clipLength = beam.k - beam.tf + 0.25
```
plateLength
The length of the plate is the vertical distance between the first and second points that we chose when modeling the plate. To calculate this dynamically, we need to know the full depth of the beam and the thickness of the flanges. We subtract twice the flange thickness from the beam's depth and save the value.

\[
\text{plateLength} = \text{beam.depth} - (2 \times \text{beam.tf})
\]

plateWidth
The width of the plate is the distance from the edge of the beam's web to the tip of the flange. We hardcoded this value originally at 1.8875, specifically to work with the beam we chose. Here we calculate this distance as the width of the beam, less its flange thickness, cut in half.

\[
\text{plateWidth} = (\text{beam.bf} - \text{beam.tw}) / 2.0
\]

plateThickness
For the thickness of the plate, nothing needs to be calculated, but we do need to access the value entered by the user into the dialog box. Within the dialog object, the thickness is stored as a string, so we need to convert it to a floating point value. We use the function dim(), which is imported as part of the startup code, to accomplish this. The dim() function takes a string and evaluates it numerically, returning its floating point value. We store this value in the variable plateThickness for easier access.

\[
\text{plateThickness} = \text{dim(dialog.thickness)}
\]

plateCount
Very similar to plateThickness, plateCount is stored as a string in the dialog object. In order to make this easier to work with we convert it to an integer using the built-in int() function, saving it with the name plateCount.

\[
\text{plateCount} = \text{int(dialog.plateCount)}
\]

distance
Just like our last variable, plateCount, distance is a value entered by the user into the dialog box. We need to convert it to a floating point value and save it in our own variable for easier access.

\[
\text{distance} = \text{dim(dialog.distance)}
\]
Here we need to calculate the total distance that will be consumed by the plates. This is a situation where you should state to yourself in plain English what you need to know, maybe even write it down to help visualize what you need. We need to take the length of the beam and subtract the endDistance twice, once for each end. This value is stored in a variable called spread, to indicate the total spread of the plates. This distance will be used later to determine how much space to use in between each plate.

\[
\text{spread} = \text{beamLength} - (2 \times \text{endDistance})
\]

We already know how many plates we will be using, but it will also be helpful to know the number of spaces we have; that is, the gaps between the plates. There will always be one fewer space than there are plates, so we simply subtract 1 from plateCount and store it in a new variable called plateSpaces.

\[
\text{plateSpaces} = \text{plateCount} - 1
\]

After a plate has been added, we use a Rotate() function to rotate the plate into place. When we added the plate ourselves, the initial rotation values were correct, so we didn't make any changes. In our code, we need to alter the rotation based on side. Here we declare a variable called xRotation and set it to 0.0. A little later we will set this value to either 90 or -90, based on which side of the beam is getting plates.

\[
\text{xRotation} = 0.0
\]

We need to know the vertical distance of the plate's starting point along the beam's left end profile. This is the y-value of the first point we chose when we modeled the plate. A positive value moves the point up (from a starting position at the center of the top flange) and a negative value moves the point down. We set plateY equal to the negative value of the beam's flange thickness, which we can get by using beam.tf.

\[
\text{plateY} = -\text{beam.tf}
\]

We also need to know the horizontal distance of the plate's starting point along the beam's left end profile, which is the z-value of the point. From its starting point at the center of the top flange, a positive value moves the point to the right, a negative value moves it left. We get the width of the beam's flange with beam bf and divide it by 2.0, storing the result in plateZ.

\[
\text{plateZ} = \text{beam bf} / 2.0
\]
At this point we have created all of the variables that we will be using. The rest of our program contains the logic that will do the bulk of the work. The first thing we need to do is consider the special case where the user only wants a single plate. This kind of condition is known as an edge case, because it falls on the edge of what would be considered normal use, and may cause problems if we don’t take it into consideration.

If `plateCount` is greater than 1, we declare a new variable called `plateSpacing` to hold the distance between plates. We calculate this by dividing `spread` by `plateSpaces`. We use `else:` to cover the other case where `plateCount` would be equal to 1. If this happens, we need to change `endDistance` from its current value to be the halfway point along the beam. We divide `beamLength` by `2.0` and update `endDistance` to store the result.

```python
if plateCount > 1:
    plateSpacing = spread / plateSpaces
else:
    endDistance = beamLength / 2.0
```

We are now finished creating all of the variables we will need to add the plates to the model. The next thing we need to do is define two functions. Our first function, `addSinglePlate()` will have the job of adding one plate between two points that we provide. Recall from earlier that functions should be created whenever we need to do something repeatedly but don’t want to copy many lines of code over and over. Creating a function to add one plate makes sense because it isolates the code responsible for adding a plate, and we can call it whenever we need to.
The code below defines our `addSinglePlate()` function. It takes two pieces of input, called *arguments,* and uses those as the first and second points of the plate. To create this function, we type the line `def addSinglePlate(point1, point2):` before our plate-add code, then indent the plate-add code so that it becomes the body of the function. While doing this, we modify the plate-add code that parametric modeling generated so that it uses some of the variables we created earlier. In the code below, these alterations are *highlighted.*

```python
def addSinglePlate(point1, point2):
    p1 = RectPlate()
    p1.member = beam
    p1.Point1 = point1
    p1.Point2 = point2
    p1.MaterialGrade = "A36"
    p1.MaterialOriginPoint = "Center"
    p1.TopOperationTypeLeftEnd = "None"
    p1.TopOperationTypeRightEnd = "None"
    p1.BottomOperationTypeLeftEnd = "Clip"
    p1.BottomOperationTypeRightEnd = "Clip"
    p1.BottomLengthLeft = clipLength
    p1.BottomLengthRight = clipLength
    p1.BottomClipLeft = clipLength
    p1.BottomClipRight = clipLength
    p1.Width = plateWidth
    p1.Thickness = plateThickness
    p1.WorkpointSlopeDistance = p1.Point1.Distance(plate.Point2)
    p1.MaterialSetbackLeftEnd = 0
    p1.MaterialSetbackRightEnd = 0
    p1.WebCutLeftEnd = 0
    p1.WebCutRightEnd = 0
    p1.OrderLength = p1.WorkpointSlopeDistance
    p1.MaterialType = "Plate"
    p1.SurfaceFinish = "Red oxide"
    p1.MaterialColor3d = "Medium Beam"
    p1.ReferencePointOffset = (0, 0, 0)
    p1.add()
    p1.Rotate(p1.Member, (xRotation, 0.0, -90.0))
```
Our second function, called `addPlatesOneSide()` is responsible for adding all of the plates required to either the near side or far side side of the beam. We isolate this code in a function because we may need to do this twice, in the case that the user chose the *Both* option in the dialog box. The function does not take any arguments, because it uses the value stored in variables we already created. Because this function is all new code, we will go through it line by line.

```python
def addPlatesOneSide():
    platesAdded = 0
    distanceToPlate = endDistance
    while platesAdded < plateCount:
        point1 = beam.LeftEnd.Location + beam.TranslateToGlobal(distanceToPlate, plateY, plateZ)
        point2 = point1 + beam.TranslateToGlobal(0, -plateLength, 0)
        addSinglePlate(point1, point2)
        platesAdded = platesAdded + 1
        distanceToPlate = distanceToPlate + plateSpacing
```

**def addPlatesOneSide():**
This line is the function declaration. It defines the function’s name and shows that the function takes no arguments. The colon indicates that the body of the function is about to begin. The rest of the code inside the function is indented, this indicates to Python that it is the function body.

```python
platesAdded = 0
```
We create a new variable called `platesAdded` to keep track of how many plates we have added. This value will be incremented as we add each plate.

```python
distanceToPlate = endDistance
```
We create another new variable called `distanceToPlate`, which we initially set to the value stored in the `endDistance` variable. After we add a plate, we increase `distanceToPlate` so that the next plate will be added further along down the beam.

**while platesAdded < plateCount:**
Here we use a while loop to repeatedly add plates. Remember that a while loop is condition-based. In this case, our condition is `platesAdded < plateCount`. What we’re saying is that we want to repeat the body of the loop as long as the number of plates that have been added is less than the number of plates the user requested. The body of the while loop is indented again.

```python
point1 = beam.LeftEnd.Location + beam.TranslateToGlobal(distanceToPlate, plateY, plateZ)
```
We calculate the first point for a plate by translating along the beam from its origin point. We already have three variables for this: `distanceToPlate`, `plateY`, and `plateZ`. We begin at the beam’s origin point, `beam.LeftEnd.Location`, and use the beam’s `TranslateToGlobal()` function to move to the point where we will add the plate. This location is saved in a new variable called `point1`. 


point2 = point1 + beam.TranslateToGlobal(0, -plateLength, 0)

We calculate the second point by beginning with our newly-created point1 location and, again, using the beam's TranslateToGlobal() function, move down (using a negative value) along the y-axis the distance stored in our plateLength variable. This location is now the second point for this plate, so we save it as point2.

addSinglePlate(point1, point2)

Now we get to use the addSinglePlate() function we created earlier to add a plate to the model. Since we've calculated the first and second points, we simply pass them into our call to addSinglePlate() and it will do the hard work for us.

platesAdded = platesAdded + 1

In order for our while loop's condition to eventually not be met, we need to increment the number of plates we have added. Because we increment this value by 1 for each plate, eventually the statement platesAdded < plateCount will no longer be true, and the loop will stop executing. If we left this line out, the loop would continue forever and our program would never stop.

distanceToPlate = distanceToPlate + plateSpacing

To make sure our next plate does not get added in the same spot as the current plate, we need to increase distanceToPlate. We already have a variable we calculated to know how much distance should be in between each plate, we called it plateSpacing. We increment distanceToPlate by this value so that the next plate will appear further down the beam.

This completes the functions that we need to define in order to run our program. The only thing we have left is to actually call the functions based on which option for beam side the user chose. If the user chose N/S or Both, we set xRotation = 90.0 and call addPlatesOneSide(). If the user chose the F/S option (or, again, the Both option), we need to change the value of xRotation to be -90 instead. This will rotate each of the plates to the opposite side of the beam. We also need to negate the plateZ value, again, to make sure the plates get added where we want them. After making these two small changes, we again call addPlatesOneSide() to add plates to the left side of the beam.

```python
if dialog.side == "N/S" or dialog.side == "Both":
    xRotation = 90.0
    addPlatesOneSide()

if dialog.side == "F/S" or dialog.side == "Both":
    xRotation = -90.0
    plateZ = -plateZ
    addPlatesOneSide()
```

This completes the parametric for the lab, but we still need to test it to make sure it works and fix any bugs in the code. Save the file after making these modifications and switch to your Modeling window. Add a parametric and choose your file. It should prompt you to select a beam, then show the dialog, and add plates according to your input. If anything doesn't work, please speak up and we will work through the errors together. Be sure to try your parametric several times on different members and with different input to see how it reacts, and look for things that might need to be modified to make the script more robust.