Introduction to Python Tutorial

This tutorial is a general introduction to using Python in TouchDesigner. It doesn't require previous knowledge of Python, but it also doesn't go into "how to program". Resources for learning Python in general are here. Experienced programmers will also be able to glean the basics of working in TouchDesigner quickly from the information here.

Contents

Python in Parameter Expressions

One of the easiest and most common places to use Python is in parameter expressions. Parameter expressions allow you to write a one line Python program to set the value of a parameter. If the value of the expression changes, the parameter's value will update automatically.

This section will teach you all the basics of writing parameter expressions. For an extensive list of example TouchDesigner Python expressions, see the Expressions sections in Python Tips.

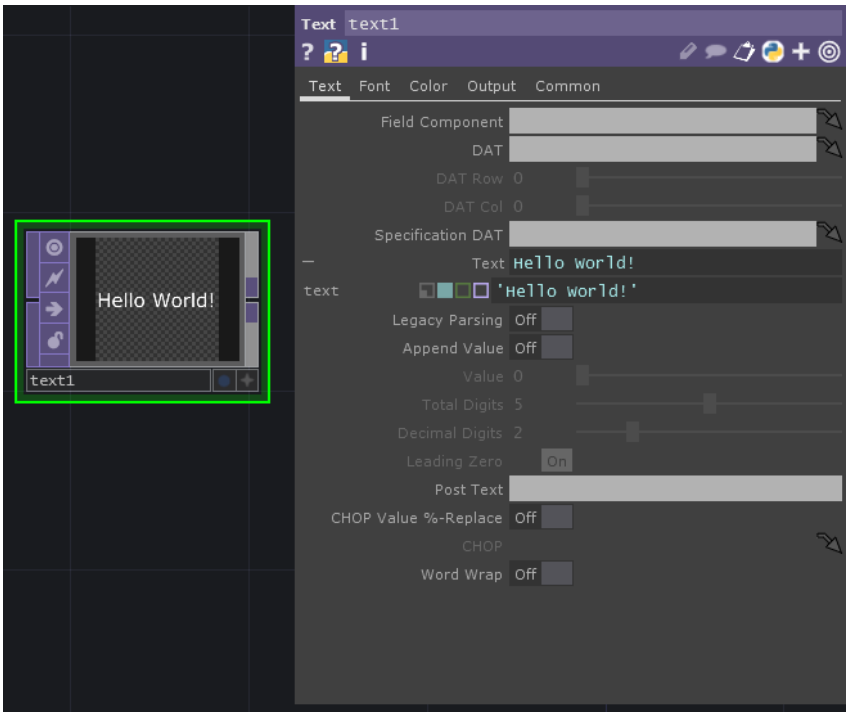
Writing a Simple Python Expression

To start with, create a Text TOP using the OP Create Dialog by selecting *Text* from the TOP family. If the Parameter Dialog is not open, press p keyboard shortcut to open it.

To enter an expression into the 'Text' parameter, expand the parameter by clicking on its name, then press the blue square. It is now in expression mode. In the entry area, type

```
'Hello World!'
```

and press <Enter> key. You should see this:



The quotes are important to let Python know that "Hello World!" is a string (a series of letters) and not a command. Let's try a command now. Replace your expression with

```
absTime.frame
```

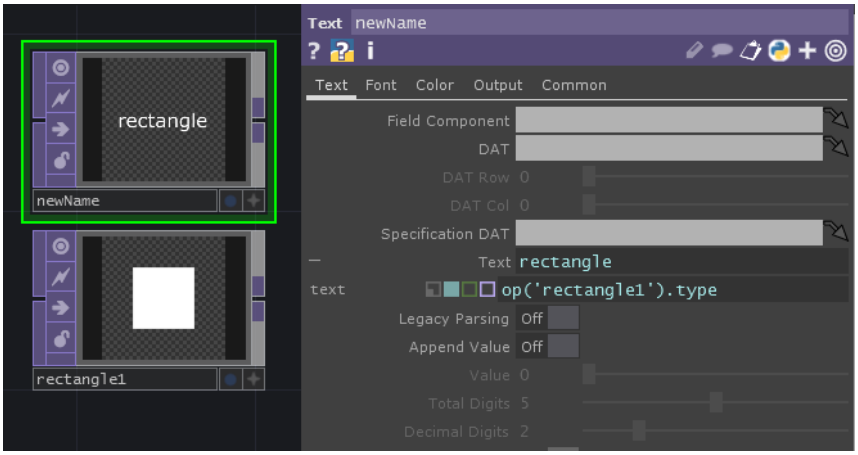
You can see the current frame changing now in both the parameter and the TOP display. Breaking down your short expression, there are two parts: `absTime` is TouchDesigner's absolute time object, the `.` tells Python you want information inside it (called a member), and `frame` is the specific thing you want from `absTime`.

This simple process will let you create powerful interactions in TouchDesigner. In general, you just have to write short expressions in Python code explaining the piece of data you want. Part of the power of TouchDesigner is that (in most cases) this information will automatically be updated in your parameter if it changes.

Accessing Operators in Expressions: `me`, `op`, and `parent`

While `absTime` is a standalone object in TouchDesigner, most of the time you will be getting information from operators in your network. To use information from the operator that the expression is located on, you use the `me` object. In your Text DAT's 'Text' parameter expression, write `me.name`. Change the name of the operator by clicking on it and you will see the change reflected in the viewer.

For the next example, put down a Rectangle TOP next to your Text TOP. The `op` command (op stands for "operator") lets you navigate to other operators and get information about them. In the Text TOP's 'Text' parameter, enter `op('rectangle1').type`. You should now have this:



As you probably figured out, the text now displays the operator type of *rectangle1*. Let's break down that expression a little more. The parentheses after `op` indicate that it is a "function". A function is a command that (usually) requires extra information, called arguments. The extra information in this case is `'rectangle1'`, which indicates which operator you want. After that, there is the familiar dot notation explaining that you want the `type` of the operator you named. The `'rectangle1'` is a Touchdesigner path and you will use these all the time. You're probably familiar with this sort of notation from file browsers. In the examples below, you'll notice that paths that begin with `/` are "absolute" paths, starting from the project's top network level (also known as root), while ones that don't start with `/` are "relative" paths, meaning the path starts from the operator where the expression is located.

Path Expression	Result
`op('rectangle1')`	operator named *rectangle1* next to me
`op('comp/rectangle1')`	operator named *rectangle1* inside an operator named *comp* next to me
`op('/project1')`	operator named *project1* at the top network level
`op('/project1/rectangle1')`	operator named *rectangle1* inside the operator named *project1*, which is at the top network level
`op('..')`	the operator I am inside, a.k.a. my "parent"
`op('.')`	the operator containing the expression, same as `me`
`op('./rectangle1')`	an operator named *rectangle1* inside me

Another common way of accessing operators is using `parent`. The `parent()` function returns the component one level up in the network hierarchy. You can give it a number as well, so `parent(2)` return the component two levels up in the network hierarchy.

Operator Shortcuts Using `parent` and `op`

The `parent` object can also be used to search upward in the hierarchy for a named **Parent Shortcut**. For example, in the 'Text' parameter of the Text TOP, enter this expression:

```
parent.Project
```

This returns the operator */project1* because in the tutorial file, the *project1* component has the Parent Shortcut `Project`. You can see this by opening */project1's* parameter box and looking at the 'Parent Shortcut' parameter on the 'Common' page. Using this `parent.<shortcut>` syntax will search upward until an operator with that Parent Shortcut is found.

Similarly, the `op` object can be used to search anywhere in your TouchDesigner file for a component with a Global OP Shortcut. As an example, open */project1's* parameters again and type `MyGlobal` into the 'Global OP Shortcut' parameter on the 'Common' page. You can now get that operator by entering `op.MyGlobal` into the Text TOP's 'Text' parameter expression.

A couple important notes here. **Global OP Shortcuts must be unique.** You can only ever have one of a given name in your file. They are best used only for the most important and unique components in your file. **Parent Shortcuts don't have to be unique and the first one found searching upward will be returned.** These shortcuts are used often; it is highly recommended to give your custom components a Parent Shortcut at their top level.

You can also combine the various ways of looking for operators. For example `parent.Project.op('text1')` will return the operator *text1* inside */project1*.

Accessing Parameters in Expressions: `par`

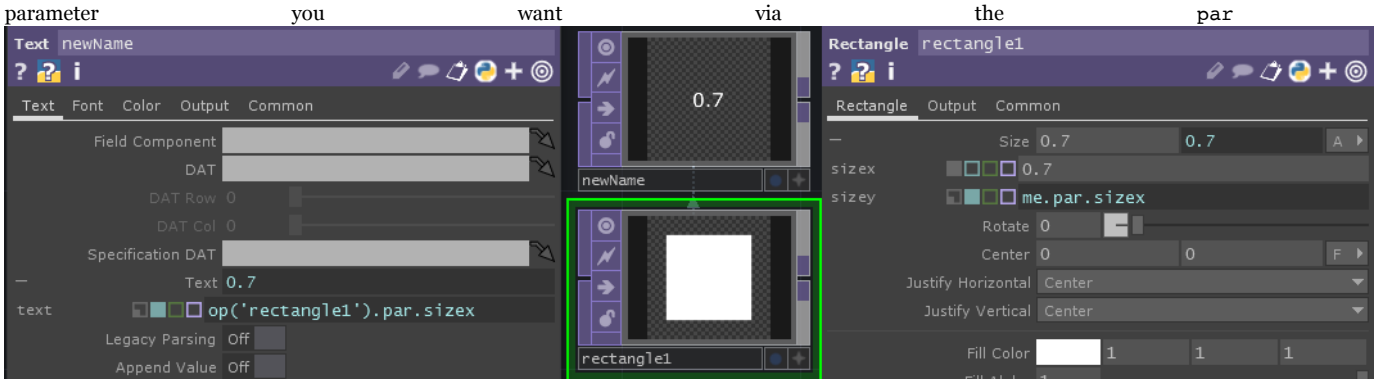
Parameters on operators can be accessed by using the par object. To start, expand the 'Size' parameter on *rectangle1*. You'll notice that there are two separate values, 'sizex' and 'sizey', which are the actual parameter names. Put 'sizey' into expression mode and enter

```
me.par.sizex
```

When you change 'sizex', 'sizey' now stays in sync, keeping the rectangle in a square shape. Now, in the 'Text' parameter of your Text TOP, enter this expression:

```
op('rectangle1').par.sizex
```

Now changing 'sizex' keeps the displayed text in sync as well. To review, the general process here is to use `me` or `op` to reach the operator you want, then access the parameter you want via the `par` object.



Accessing CHOP channels

Another common use of parameter expressions is to access CHOP channel data. For this example, put down a Beat CHOP. Rather than typing the expression this time, we are going to use drag/drop to create it. Follow these steps:

1. Press the + button on the bottom right of the Beat CHOP to make its viewer active.
2. Select *rectangle1* to see its parameter dialog.
3. Drag the 'ramp' channel and drop it into the left (*x*) entry box in the 'Size' parameter.
4. Select **CHOP Reference** from the popup menu.

You will now have the following expression in 'sizex':

```
op('beat2')['ramp']
```

In Python, the `[]` square bracket notation is a way to specify elements in a collection. In this case, because CHOPs can have multiple channels, you specify which channel you want by putting its name in the square brackets. While you will usually want to use the name, you also have the option of picking a channel by number. Change the 'sizex' expression to op('beat2')[0] and you will see the same results. Note that in Python, the first element of a sequential list is always zero, not one.

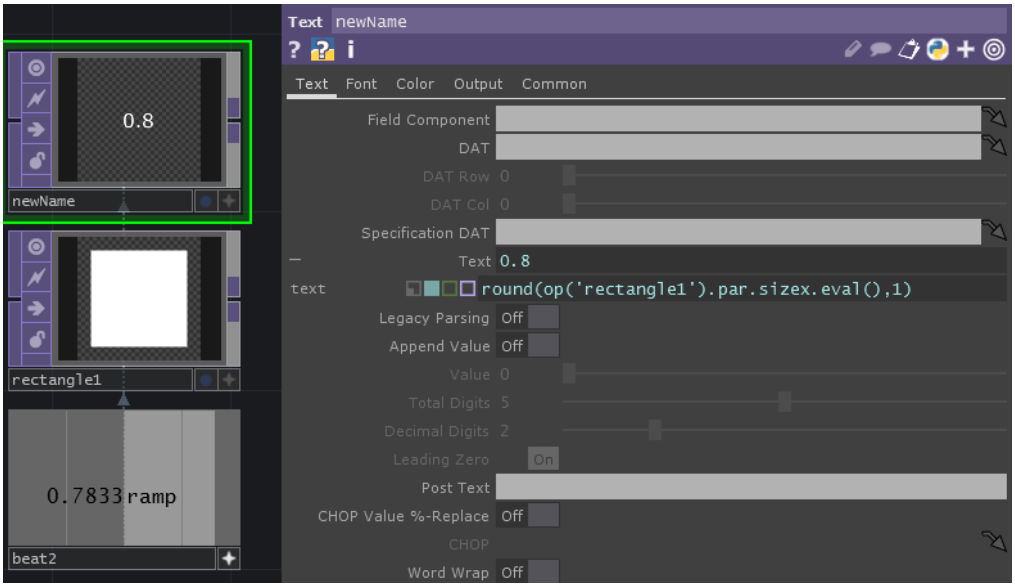
More Complex Expressions, Errors, and `eval`

You may have noticed that your Text TOP shows an unreadable jumble of numbers now, because the CHOP sends data that changes quickly and has many digits. To fix this, we will use Python's `round` function which takes two arguments: the number to round and the number of decimal places to show. Enter the following into the 'Text' parameter of the Text TOP:

```
round(op('rectangle1').par.sizex,1)
```

The parameter turns red and a red X marker is placed on the operator. This is what an error looks like. Don't worry, errors are an unavoidable part of scripting. Before we fix it, it's worth looking at a couple different ways to view the error message. One is to hover over the parameter or expression. Another is to click on the red X marker on the node. This specific error is *td.Par doesn't define __round__ method*. The problem is that technically `op('rectangle1').par.sizex` returns a parameter object and not an actual number, which is what the `round` function wants. Often using a parameter object will work, but occasionally you will run into this problem and need to convert from a parameter to a value. To do this, use the `eval` function. Change your expression to this:

```
round(op('rectangle1').par.sizex.eval(),1)
```

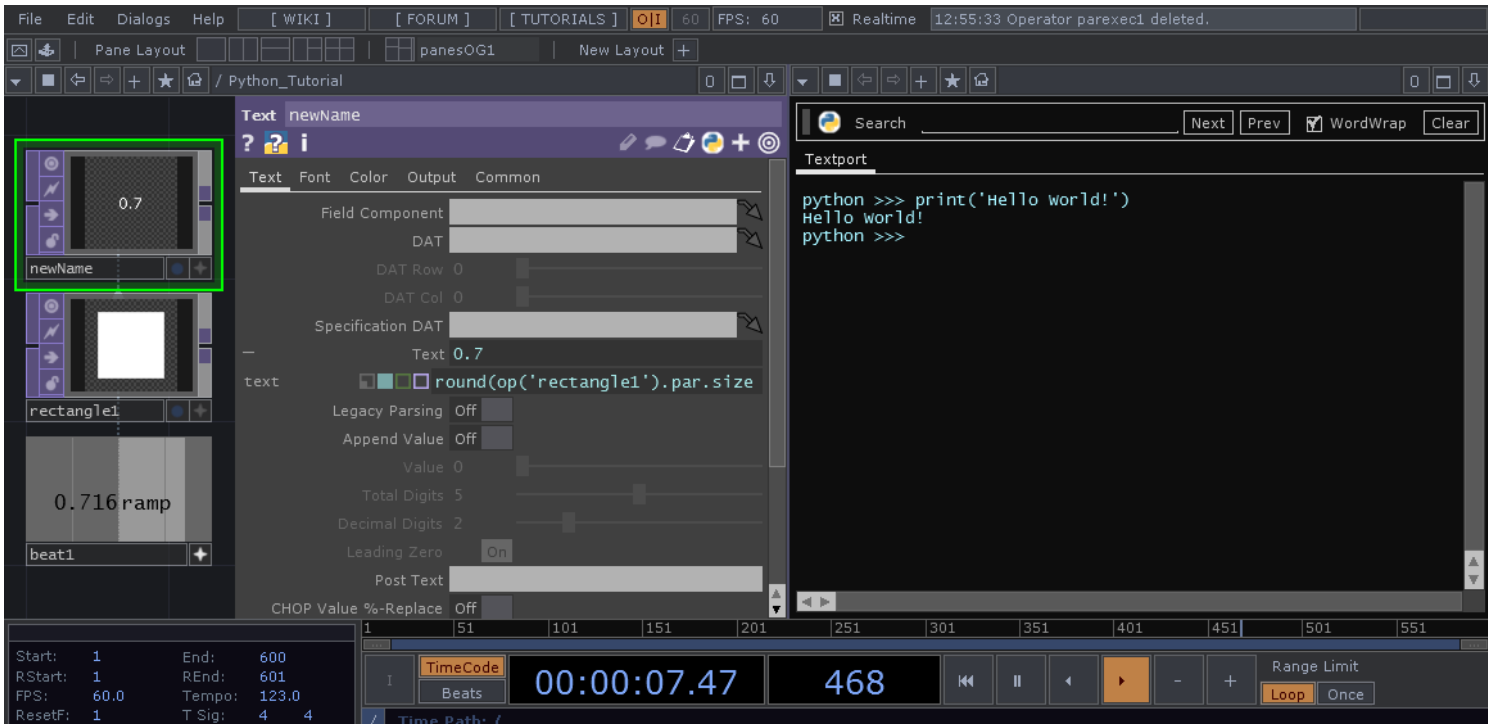


For an extensive list of example TouchDesigner Python expressions, see the Expressions sections in Python Tips.

Python in the Textport

When working with Python in TouchDesigner it is very helpful to have a Textport open. For quick tasks, you can open a floating textport by pressing **alt-t**, but for more extended Python work like this tutorial, opening a textport pane is recommended. To do this:

1. Click the downward facing arrow on the far right above the TouchDesigner network view and choose 'Split left/right'.
2. Click the downward facing triangle on the far left above your new pane and choose 'Textport and DATs'.



The textport performs these useful functions when working with Python:

- Displays output from `print` and `debug` statements in Python code.
- Displays all error messages from Python code.
- Lets you test simple Python.

As a simple test, type `print('Hello World!')` into the textport and press <Enter>.

Working With TouchDesigner Objects in the Textport

Accessing TouchDesigner objects from the textport is quite similar to in parameter expressions. Most commonly, you will start with an operator, which means you will need its path. There's a great trick for this:

1. Enter `r = op('` into the textport. Do not press <Enter>.
2. Drag and drop your *rectangle1* operator into the textport. The path is automatically appended to your command.
3. Finish the line by typing `')` and pressing <Enter>.

You have now assigned your rectangle operator to the variable `r`. Type `r` and press <Enter> and you will see information about the operator. Here's a few more things to try while you're at it:

Textport Command	Result
`r.name`	the operator name
`r.par.fillcolorr`	the value of the fill color parameter's red attribute
`r.par.fillcolorr = 0`	set the value of fill color red to 0
`dir(r)`	a list of the data and functions available in the operator
`help(r.resetViewer)`	show help for the operator's `resetViewer` method

You can use this technique to examine TouchDesigner objects, to test simple Python code, and to perform simple, one time Python tasks.

Python in DATs

Any Python code more extensive than expressions goes in [DAT|DATs]. There are four main places where you will find/create such code:

1. **Callbacks (callback DATs and execute DATs)** - react to various TouchDesigner events
2. **Script Operators** - actually a subset of callback DATs, this code creates the content in these special operators
3. **Extensions** - create a Python interface for custom components
4. **Standalone modules** - generalized, reusable Python code

Before we dive into working with Python in DATs, you may want to set up an external editor if you haven't already. DATs and the textport can be used for editing, but external editors provide more extensive tools including syntax highlighting, code folding, etc. To set this up, see Editing DAT Text in an External Editor.

The sections below will build on the simple examples described above in Python in Parameter Expressions.

Callbacks

Callbacks let you use Python to react to many different kinds of changes in TouchDesigner. Some operators will have callback DATs attached to them when you create them, such as Timer CHOP, while other callback DATs stand alone as their own operator types, such as Parameter Execute DAT.

For this example, put down a Parameter Execute DAT next to the operators you created in the Python in Parameter Expressions section. To set it up to watch the 'Text' parameter on the Text TOP, do the following:

1. Drag the Text TOP into *parexec1* DAT's 'OP' parameter. It should now contain the name of the Text TOP. We changed that name above, so yours might be different.
2. To pick which specific parameter to watch, type `text` in *parexec1* DAT's 'Parameters' parameter.

Now that we've set up the parameters, we can write some Python. Right-click on *parexec1* and select 'Edit Contents...'. We want to run code when the parameter value changes, so first let's make sure we're getting messages properly. Change the `onValueChange` callback to look like this:

```
def onValueChange(par, prev):
    debug(par)
```

If you look back at the textport, you'll see a debug statement showing up every tenth of a second, when the number in the parameter. Just to show something interesting you might do with this, now change the `onValueChange` callback (including the import line above) to:

```
import random

def onValueChange(par, prev):
    op('rectangle1').par.fillcolorr = random.random()
    op('rectangle1').par.fillcolorg = random.random()
    op('rectangle1').par.fillcolorb = random.random()
```

You will now see the rectangle's square changing to a random color every time the Text TOP's 'Text' parameter changes. This effect would be impossible to achieve with just parameter expressions.

For those unfamiliar with Python, the **`import`** statement is how you reference external Python code, called **modules**. Once you import the module `random`, you can use the function in it that happens also to be called `random`. To see a list of modules available with TouchDesigner, go here.

`print` vs. `debug`

An important thing to note in the above section is the `debug` command. It is a TouchDesigner utility that works much like a Python `print` statement but automatically adds exactly where in your network the `debug` command came from. In TouchDesigner, you will often find yourself needing to send information to the textport in order to fix problems in your scripts. You should always use the `debug` statement to do this instead of `print`.

Execute DATs

Execute DATs are DAT operators that provide callbacks for events in TouchDesigner. For example, the example above uses a Parameter Execute DAT to react to parameter value changes. Here are the execute DATs and what they react to:

- **CHOP Execute** - changes in CHOP channels
- **DAT Execute** - changes in a DAT table.
- **OP Execute** - general operator changes, such as name, children, wiring, etc.
- **Panel Execute** - interactions with a Panel Component (UI).
- **Parameter Execute** - changes to parameter values and settings
- **Execute** - system events, such as startup, file save, operator creation, etc.

See OP Snippets for examples of all these.

Script OPs

Script OPs are operators that provide callbacks that actually create the data of the operator itself. When you place one of these operators, there will be a callback DAT attached to it automatically. In the DAT, you can edit the `onCook` callback to define the operator's behavior when it cooks. Here is a list of the script OPs:

- **Script CHOP** - use Python to create a CHOP and its channels.
- **Script DAT** - use Python to create a DAT table.
- **Script SOP** - use Python to create a SOP using points, curves, meshes etc.
- **Script TOP** - use Python to create a TOP image.

See OP Snippets for examples of all these.

Extensions

Extensions are Python classes that allow you to build data and functionality into your custom components. There is a lot to extensions and we will only brush the surface in the tutorial below. For a more in-depth look at Extensions, look here. This simple technique for adding Python attributes and functions is incredibly powerful for giving custom components extended Python functionality.

Creating An Extension

To create a default extension, follow these steps:

1. Create a new Container COMP
2. Use the RMB Menu on the new component to select 'Customize Component...'

3. Open the 'Extension Code' section
4. Enter the name `TutorialExt` and press 'Add'
5. Press 'Edit' to see the contents

There's a lot going on in there that we aren't going to get into here, but to get an idea of what extensions are for, just note these lines of code:

```
        # attributes:
        self.a = 0 # attribute
        self.B = 1 # promoted attribute
```

Accessing An Extension From Inside a Component

To see the extension in action, go inside the container COMP. The first thing you'll notice is that there is a DAT in there called *TutorialExt*. This is where the actual extension text is located, and you can edit it through the Component Editor, as shown above, or directly by changing the DAT here.

For this example, create a Text TOP here in the container. Change the Text TOP's 'Text' parameter to expression mode (click the blue square) and enter the following expression:

```
ext.TutorialExt.a
```

This uses the `ext` object to search upward for `TutorialExt` and return it's `a` attribute. **Extensions can be accessed from anywhere inside their component using the `ext` object.** You can change this expression to `ext.TutorialExt.B` to see that value as well. You can use both extension attributes and extension functions in this way.

Accessing An Extension From Outside a Component

Promoted members of Extensions can also be accessed from outside their components. **Extension members are "promoted" when they start with a capital letter.** This applies to both attributes and functions.

To see this in action, navigate upward out of the component by pressing u keyboard shortcut. Create another Text TOP here, and again enter an expression into its 'Text' parameter. This time, the expression is: `op('container1').B`. As you can see, the extension attribute acts as a member of the container COMP itself. If you try the expression `op('container1').a` you'll get an error. This is because only capitalized attributes and functions of an extension are promoted to this level.

DATs as Modules

All DATs with Python code in them can be used as modules. Importing these modules is a little different than in most Python however, because they exist in TouchDesigner networks instead of in files. For the following examples, work inside `/project1`.

Place a Text DAT and rename it to *utils*. Enter the following function:

```
def my_adder(x,y):
    return 10*x+y
```

Add another text DAT and rename it to *test*. In this DAT enter the following:

```
import utils
a = utils.my_adder(1,3)
print(a)
```

Open a textport, then run the *test* module by right-clicking on its node and selecting 'Run Script'. **Note: if your node viewer is active on *test* you will have to right-click on its name bar.**

You will see the resulting `13` in the textport. The `my_utils` DAT has been treated as a module, using the import statement.

Component Modules

To create **Component Modules** that are accessible anywhere inside a component, you must place them in a specific child of that component, namely `local/modules`. You can create this path if it doesn't already exist by creating a Base COMP called *local* with another Base COMP called *modules* inside it.

The `import` statement in TouchDesigner

TouchDesigner uses a special version of the `import` statement which looks for modules in the following order:

1. The current component
2. The current component's Component Modules
3. The Component Modules of each parent until it reaches the root component
4. The component modules of /sys component (internal TouchDesigner modules)
5. The regular Python disk search

Note: When you attempt to import an external module, it will first look for DATs of that name in the same folder , so be sure to avoid name conflicts keeping the above search order in mind.

Module On Demand, the `mod` object

The import statement, though useful contains two disadvantages:

- Module names must be single words, so relative DAT paths cannot be used.
- They are unsuitable for use in a parameter expression.

To avoid this, use the Module On Demand or `mod` object.

In the DAT *test*, replace the contents with the following code:

```
a = mod.utils.my_adder(1,3)
print(a)
```

Notice how no import statement is needed, making this evaluation suitable for one-line parameter expressions.

String paths, including relative paths can also be used in the `mod` object. Simply pass in the string as a parameter to the mod object:

```
a = mod('utils').my_adder(1,3)
print(a)
```

Both absolute and relative paths may be used:

```
a = mod('/project1/utils').my_adder(1,3)
print(a)
```

All of the above will result in the same output.

Note: `mod` uses the same search order described above for `import`.

The `module` member

You can also access a DAT as a module via its `module` member. Using this method, the *test* script would be:

```
a = op('/project1/utils').module.my_adder(1,3)
print(a)
```

This method of access is useful if you know exactly where your module is located and want to avoid any of TouchDesigner's automated searching.
